Simple Distributed File System (SDFS)

Abhishek Lingwal
aslingwa@ncsu.edu

Anas Siddiqui
asiddiq@ncsu.edu

Raunaq Saxena
rsaxena@ncsu.edu

***Abstract*— Project aims to create a client-server based distributed file system. Distributed file systems have various advantages major of which are storage scalability, increased availability, load sharing, reliability and easier maintenance. Our main objective is to implement multiple server nodes (server cluster) and present it as a single coherent storage to the end user. A centralized server is a single point of failure and has poor scalability and low availability. Mitigating these issues was the main motivation behind adopting multiple servers to store data. In our solution, we try to achieve all the basic advantages mentioned in order to make a robust and scalable file system. We decided to adopt a model where the server cluster contains a master node, whose responsibility is to maintain track of data location on the basis of the server, global metadata and optimize the efficiency of the underlying system.**

***Index Terms*— storage system, distributed file system, clustered storage, fault tolerance**

I. INTRODUCTION

A distributed file system (DFS) is a file system which is shared by being simultaneously mounted on multiple servers. The major design goal is to achieve "transparency" in a lot of aspects. This implies being "invisible" to the clients using the system in a way that they "see" a system very similar to a local file system [1]. A few of the common goals for a DFS could be:

1) *Access Transparency*
   The client should not perceive that the files are not stored on a local file system
2) *Location Transparency*
   The name of a file has no implication on its location. It could be stored on any server, but appear as a global namespace to the client.
3) *Failure Transparency*
   The client programs should operate even after a server failure.
4) *Scalability*
   The file system should scale well for a large number of clients
5) *Replication Transparency*
   To support scalability, files could be replicated to multiple servers. Clients should be oblivious to the replication or failure recovery.
6) *Migration Transparency*
   If files are moved around, clients should be unaware of this.

Sharing many of the goals as listed, our main motive behind this project was to understand the intricacies of building a DFS from scratch. In this paper, we present our assumptions, design decisions and performance benchmarks of our implementation.

A. Problem Motivation

The main motivation behind designing a DFS are the limitations offered by a local file system. Our main focus were to mitigate the following:

1) *Reliability*
   This implies that failure of data nodes should not impact the ability of the client to fetch data. In addition to handling data node failure, the metadata from the main node is also handled by continuously backing up to a passive secondary main node.
2) *Avoiding Single Node Bandwidth Bottleneck*
   To let data flow through a single Master node would prove detrimental to the performance of the system. Hence, we made it mandatory for the client to send data directly to data nodes to avoid bottleneck and single point of failure.
3) *Priority Replication*
   Replication is done not based on a strict policy (e.g. RAID-1) but on priority as defined by the user. This ensures that storage explosion does not take place for files that are not very important for the clients.

II. DESIGN OVERVIEW

A. Assumptions

As is the case with any system design, there were a few assumptions and tradeoff decisions to keep in mind while implementing SDFS. These assumptions were guided by the challenges and opportunities offered while keeping in mind the priorities and feasibility of the project.

1) Stable Network Connection

It seemed feasible for us to consider a stable internet connection, more importantly between the main node and the slave data nodes. It could be argued that these components are connected reliably with fault tolerance. In most systems, such an assumption would be reasonable.

2) File size restriction

File size is currently restricted to the maximum limit of the RAM, as it is a Java program running on client and server machines. It is currently not built to support large datasets.

3) Reliability of intra-data node communication

Although we aim to achieve reliability issues between client and master node or when one of the data nodes goes down, it is assumed that during replication, data nodes are reliably connected to each other.

B. System Components

SDFS consists of 3 primary components:

1) Client Program

The program that connects to the server to access/store files while running at the client side.

2) Master Node

This is the main controller node that the client initially connects to. It is responsible for storing, maintaining and updating the metadata of the client nodes.

3) Storage Node

These are the "slave" nodes that are controlled by the master node. The data is stored and replicated on these nodes.

C. Architecture

The architecture is similar to that of a traditional client-server, albeit with a little modification. There are a set of data nodes operated and controlled by the master node. All client requests pass through the master node and are responded to. If necessary, the client nodes then talk to data nodes independently, without the intervention of master node. This major design decision gives us the confidence to avoid bottleneck as well as single point of failure.

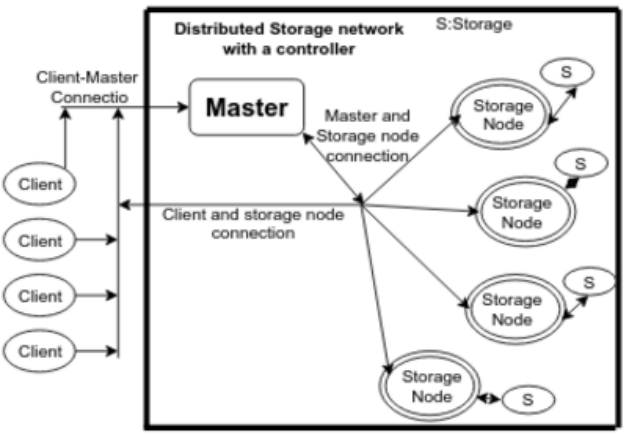

Fig. 1. System architecture of SDFS

III. Implementation

A. Design decisions

For the sake of simplicity, storage and replication are handled at the file-level instead of block level. These are the design decisions taken into consideration

1) Push-Pull model

The design is based on a push-pull model, where any changes made to the files takes place by active client requests. Since the file system is not mounted on the local system, local changes are not reflected in real-time, but need to be pushed to DFS. Similarly, before accessing a file to read/modify, it needs to be pulled from DFS.

2) Centralized metadata management

The metadata is managed by the master node. Since all client requests first go through the master node, the data nodes are oblivious to the hierarchical structure of the clients.

3) Client-Data node direct transfer

The master node returns a set of data node(s) for the client to contact for putting a file. The client only contacts one of the data node. This ensures that client is not a victim of high latency, experienced due to network congestion at the data nodes. The client program waits for only one data node to accept the file.

4) Replication handling by data nodes

The replication is handled in background by the data nodes, after the transfer through client node is successful. This ensures that the client node is not held at a blocking call. This would result in a low latency and better network bandwidth utilization. The underlying assumption being data nodes would be connected by dedicated and efficient networks.

5) Flat file structure for data nodes

The data nodes act only as a "store" of data and hence are not required to maintain the hierarchical file structure requested by the clients. The files are stored in a directory maintained by each client in a flat hierarchy, by appending a suffix to ensure duplicate file names do not create a problem (when they belong to different folders in the client hierarchy).

6) Load balancing

The load balancing is based on a dynamic feedback from the data nodes about the amount of data stored on them. The data nodes are handled as a priority queue, where the best contenders are chosen for putting data as well as replication.

7) TCP Connection

We rely on TCP Socket APIs provided by Java for data transfer mechanisms. This ensures reliable data transfer.

B. Client command mechanisms

We shall describe the commands that are used by clients to make use of DFS. The structure of the commands is inspired from the interface provided by HDFS [3]. The path given for all operations are global path, similar to the interface provided by HDFS.

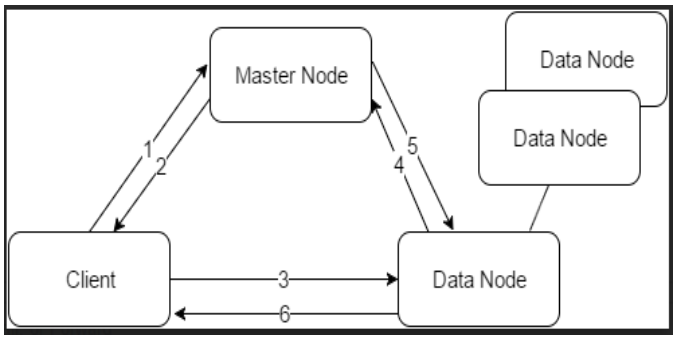

Fig. 2. Flow of data

Message number	Data Sent in the message
1	Client command request
2	Command verification and metadata response
3	Client data transfer request
4	Metadata update request
5	Metadata update acknowledgment
6	Client operations acknowledgment

Table. 1. Legend for Fig 2.

1) Register

This command is used by the client to register itself with the master node. This creates a hidden local file which is subsequently used for authentication purposes with each command implicitly.

2) Login

When a pre-registered client wants to interact with the master node from another machine, it uses the login to authenticate its validity with the master node. The hidden file is created again on the new machine, for further authentication purposes.

3) Ls

The command is invoked as "sdfs ls <remote path>". The command is used to get the metadata from the main node to present a logical view of the hierarchical file structure. When the master node receives the request, it validates the authenticity of the client and replies with the metadata requested.

4) Mkdir

The command is invoked as "sdfs mkdir <remote path>". It is used to create directories on the server side. When the client request is received, the master node validates the path given, and creates an entry in the metadata for a valid path. Since we maintain a flat file structure on data nodes, they are not involved in this process. This makes sure that directory is only a logical entity, and not created on physical storage.

5) Put

The command is invoked as "sdfs put [replication_flag] <local path> <remote path>". This command enables the user to transfer a file from local storage to DFS. The flow of messages is as follows:

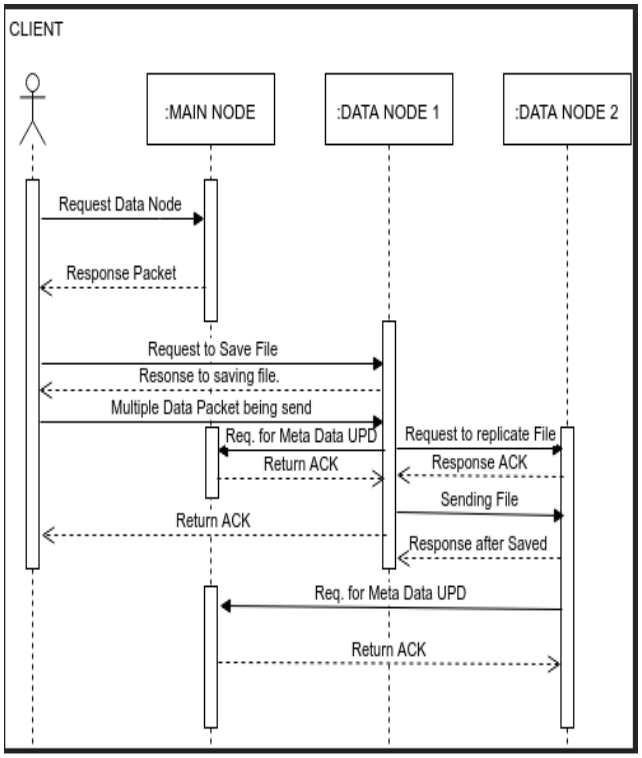

Fig. 3. Sequence diagram of Put Command with replication

6) Get

The command is invoked as "sdfs get <remote path>". This stores the requested file in the current directory of the client. When the data is unavailable on the one of the data nodes, the client can access it from the replicated nodes if the client had requested for a replication during file creation. Since steps 4 and 5 (from Table 1) are not required for this operation, latency would be lesser than the "Put" command.

7) Remove

The command is invoked as "sdfs rm <remote path>". It removes the file from DFS, if the path specified is valid. The file is also removed from all replica nodes, if requested during file creation.

C. Master and Data Node operations

The master executes all namespace operations. Additionally, it runs the load-balancing algorithm, along with replica management. It also co-ordinates with data nodes about live status. We discuss in detail the topics below.

1) In-Memory Data Structures

The in-memory data structures are a careful design decision for faster access and modifications. They are made persistent sporadically for backup purposes. This limits the size of metadata to the main memory of the master server, which can be reasonably assumed to be large.

2) Logging

All the requests made by each client are logged on to the master node as well as the data nodes (including request type, time, exception handling, etc.). It is our way of implementing provenance in SDFS. These logs prove to be useful for the administrator to identify any anomalous activities as well as for data analysis.

3) Heartbeat mechanism

To check the status of data nodes, we implement a heartbeat mechanism, where the master node continuously pings the data nodes before assigning them to the clients (this is done only on a "put" request to reduce system utilization). This ensures that failed nodes are actively removed from the data node list, and added when they go live again.

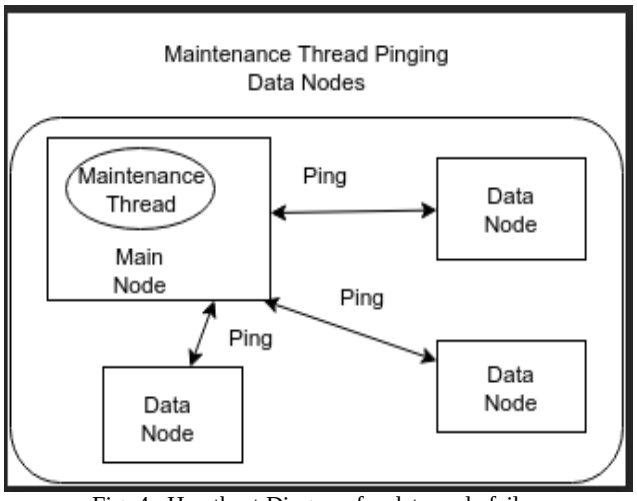

Fig. 4. Heartbeat Diagram for data node failure

4) File encryption

Before the transfer of file takes place, they are encrypted using symmetric-key based techniques. This achieves data confidentiality.

5) Data Node Addition (Scalable)

In order to make the system scalable, the addition of data nodes is simplified through a setup program run. This is currently done manually by an administrator.

IV. EVALUATION

A. Criteria

We used 3 MB files as the sample data for our benchmarking operations. The environment used for testing were Ubuntu instances on the Amazon AWS Cloud (Oregon). The application was tested with an active master node, backup master node, and three data nodes (all being separate AWS instances). The client programs were run on local machines connected to NCSU network. The results are documented below.

B. Technology Used

The entire system was developed in Java, using TCP Sockets and multithreaded programming. There are 3 jar files deployed for each component: client, master node and data node. We tested the system on Ubuntu, but it can be ported to any system that has JDK 1.7 or above.

C. Operation Tests

Operation	SDFS	Client-server program
Put	1534 ms	1219ms

Table. 2. Time comparison for Put Command

Table 2 shows the time comparison while running "Put" command on SDFS versus executing a simple java client-server file transfer. This gives us a rough estimate of the overhead occurred due to SDFS master node operations (metadata management, load balancing, etc.). As can be inferred, there is a negligible overhead for the system designed and would remain constant regardless of file size.

When running the Put command with replication, we observed consistent timing results, owing to our design for replication handled as a background process.

"Ls" was consistently timed at 300-350ms, which achieves the goal of "access transparency" in the confines of practicality. "Mkdir" gave the same performance of ~300ms, as only the metadata is being updated at the master nodes.

D. Reliability Tests

We checked for the master node failing, and the recovery was successful. The master node was programmed to backup its operations every 1 second. The clients have previous knowledge of the secondary master node via a configuration file. When the main master node crashed, clients switched to the secondary master after a timeout period. The backup time of 1 second is a tradeoff decision between consistency and system utilization.

E. Availability Tests

While testing for availability, we crashed a data node. When the client tried to connect to the crashed data node, a timeout occurs. Subsequently, the client connects to the replica copy, passed by the master node.

F. Storage Load Balancing

While executing the "put" requests by clients, we witnessed an even distribution of data across all data nodes. This verifies the accuracy of the load balancer module.

V. FUTURE WORK

Implementing a distributed file system has definitely been a humbling experience. After careful study of other distributed file systems [4] and [5], we propose some features that could make our implementation robust and better to be used in practice. Some of them are listed below:

A. File Consistency

Consistency is the classic problem while designing a distributed file system. Some implementations use a stateless architecture while some try to maintain state and validate client caches actively through interrupts. While the file is being concurrently accessed by the same client account from two different machines, we could implement a flush on close to all replicas.

B. Data Compression

Currently, the data is being sent as a stream of bytes, without using any compression algorithms. Such a module could reduce our network bandwidth as well as storage utilization significantly.

C. De-duplication on data nodes

Data is not stored on nodes "intelligently". The byte streams are stored without any processing. De-duplication on individual data nodes would help us exploit more storage. This would come with a cost of sporadic garbage collection of files if done in a lazy fashion, or increased overhead on the "put" command if done eagerly.

D. Client-side caching

In our current implementation, data is requested from the master or data nodes on-demand and in real time. Thus, no copies are made on the local file system for future access. Either files (to an extent) or metadata could be stored as hidden files for future use, again reducing the network bandwidth. In addition to client-side caching, block level division of files could enable replica nodes to transfer files in parallel, increasing the throughput significantly.

E. Hot and Cold Storage

The data being stored is judged by a single parameter – priority. This does not give us a lot of control, beyond load balancing. If the type of data node is known (SSD or HDD), we can incorporate movement of data using another parameter, or based on usage statistics.

F. Access Control

File level access privileges could be implemented as parameters while creating the file, which would mimic a local file system.

VI. CONCLUSION

In the ever increasing demand for ubiquitous access to shared files, distributed file systems play a pivotal role in helping us achieve this. The global namespace provided by DFS as well as the transparencies help us overcome the limitations of a local file system. We chose to implement the system from scratch to gain a deeper understanding of the complexities of a DFS. We were successful in building an application which met our design goals. The biggest learning from this experience is the appreciation of design tradeoffs. The system was designed in a modular fashion, which would make it easily extensible to include more functionalities in the future.

VII. REFERENCES

[1] "Clustered File System", https://en.wikipedia.org/wiki/Clustered_file_system

[2] S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc.of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.

[3] K. Shvachko et al. "The Hadoop Distributed File System", California, 2010

[4] R. Sandberg et al. "Design and Implementation of the Sun Network File System", In Proc. of USENIX Conference and Exhibition, Portland, Oregon, 1985

[5] Remzi H. Arpaci-Dusseau & Andrea C. Arpaci-Dusseau,"Andrew File System", *Operating Systems:Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[6] "Declustered RAID", https://www.ibm.com/support/knowledgecenter/SSFKCN_4.1.0/com.ibm.cluster.gpfs.v4r1.gpfs200.doc/bl1adv_introdeclustered.htm

[7] "Stateful vs Stateless Servers", https://www.cs.swarthmore.edu/~newhall/cs85/s08/trilok.pdf