Client-Server Fuse File System

Vishwas Patel, Saad Afzal, Vijeth Rai
Department of Electrical & Computer Engineering,
University of Florida, Gainesville, FL – 32608
Email: {vishwas.s.patel, saadafzal, vijethrai }@ufl.edu

***Abstract*— We have designed and implemented a Client-Server Architecture on Fuse File system which uses server replication for performance and fault tolerance. This technique helps in providing fault tolerance in case of a power failure or data corruption in a single server. The main focus revolves around details about the implementation and performance evaluation of the applied architecture.**

Keywords—DataNode; NameNodeClient; Server; FUSE

I. INTRODUCTION

There are a number of problems with a conventional file system using a Client-Server Architecture. It could range from a power failure during a transaction to data corruption due to component failure which could fail at the entire server level. These kinds of errors not only hamper the performance of the system but also introduce the data corruption which could, if not detected or checked, lead to corruption of the entire system. Thus, there arises a need to contain and rectify these errors. We studied and took references from few of the popular file systems used widely today. Few of them are as follows:

A. Google File System

Google File system [1] consists of a single master and multiple chunk-servers as shown in Fig.1[1] below that is accessed by multiple clients. Files are organized in directories and accessed via pathnames, just the way our file system works. Most files are modified by appending the data instead of over-writing it on the already existing data. Data is appended upon an update by a record-update operation. This operation allows multiple clients to append the data to the same file, yet ensuring atomicity for each of the client's append operation on the data being accessed simultaneously. Thus appending the data and guarantee of atomicity become the main focus of performance optimization.

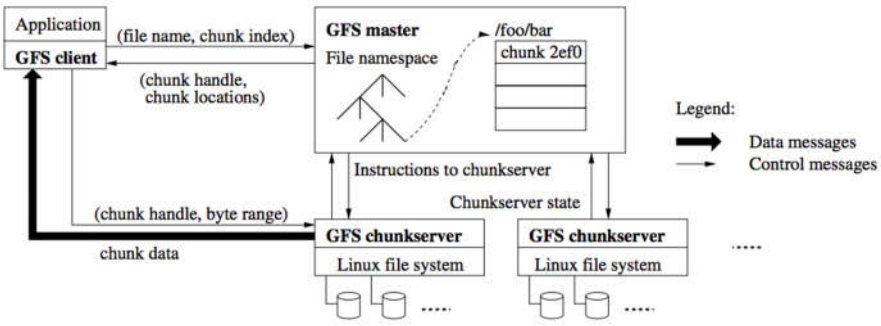

Figure 1: GFS Architecture

The master maintains the metadata for the entire file system. This includes the name space, the control information along with other mapping details required for the file-chunk servers. It performs varied range of operations like migration of data between servers, garbage collection and mapping of data files. It also periodically sends out a 'heartbeat' signal to its servers to check the health of the server or detect a failure.

Error checksum is used to correct any errors at the disk to prevent failures arising of any software or hardware failures. The system performs fault tolerance by replication of data across various data servers, restoring the faulty server quickly and also constantly monitoring the health of the servers as mentioned above.

B. *Hadoop File System*

Hadoop provides a distributed file system which uses the MapReduce algorithm. The most important characteristic of this file is the partitioning of data across thousands of its servers to execute the data in parallel which accounts for its performance enhancement.

The architecture consists primarily of a name node, data nodes and an HDFS (Hadoop Distributed File System) client [2]. The HDFS namespace is a hierarchy of files and directories. NameNode represent files and directories with the help of inodes and is responsible for capturing attributes like access time, permissions, modification time, quota of disk space and namespace. The content of each file is split and stored across multiple DataNodes, where the number of data nodes holding the contents of the file can be selected by the user on a per file basis. The NameNode does, thus, holds the mapping of these files to the DataNodes and also the namespace tree.
Every block of a file is represented by two separate files on the DataNode. One of them contains the metadata including the checksum for the block of data while the other ones is the data itself. DataNode has to verify the software version and namespace ID every time it starts up. For this purpose, the DataNode performs a handshake with the NameNode upon booting up. In case of a mismatch of the namespace ID or version, the DataNode shuts down on its own. All the DataNodes of a cluster have a unique namespace ID. So a node with a namespace ID cannot join a cluster with another ID. This preserves the integrity of the system. Also, whenever a new DataNode is initialized, it can join any cluster and obtain its namespace ID.

Files and directories are referenced by paths present in the namespace. From a user point of view, there is no knowledge about the replication of data across multiple servers or, if the data and metadata are on same server or different ones. Whenever a user wants to read a data file, the application will send a request to the NameNode to obtain the list of DataNodes holding the contents of the replicated data. Upon response, it then directly connects to the DataNodes holding the file replicas. For a write request, the application first connects to the NameNode and asks it to choose the DataNodes where the file needs to be replicated. Upon the response, the client sends the data to these nodes. The process flow is depicted in the figure 2 below.

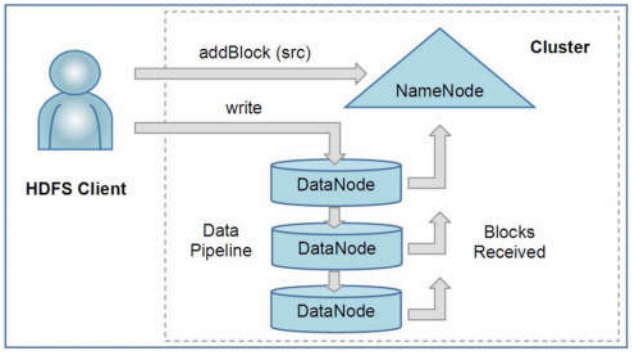

Figure 2: Process Flow

One functionality of HDFS is that it gives information about the locations of the file blocks by proving an API. The application such as MapReduce uses it to its advantage by mean of task scheduling. This improves the performance of reads. Also, the default file replication factor is three but this can be modified for critical files or for the ones which are accessed more often than other files. Increasing the replication factor means increasing the bandwidth for reads and also the fault tolerance.

Along with the above three main components of HDFS architecture, there are additional three functional components.

For a persistent record of image that is written is called a checkpoint. A log is maintained for such records which is act as a commit-log for any of such changes. Every action by the client is logged in the journal. Whenever the computation of the transaction is complete, the previous non-committed versions or copies are then cleared and synched. The change is then committed to the client and is reflected outside the thread of execution.
The NameNode performs two more functions namely CheckpointNode and BackupNode. The NameNode cannot behave as both, but is chosen to execute as either of one at the startup. The function of CheckpointNode is to combine the existing checkpoint and journal to create a new checkpoint. This is done to protect the metadata in case of a system fail. The system then starts from the most recently created checkpoint. The BackupNode functions to maintain the most recent copy of the system namespace which is always up to date with the state of the NameNode.

The data flow of a write example is depicted below in figure 3.

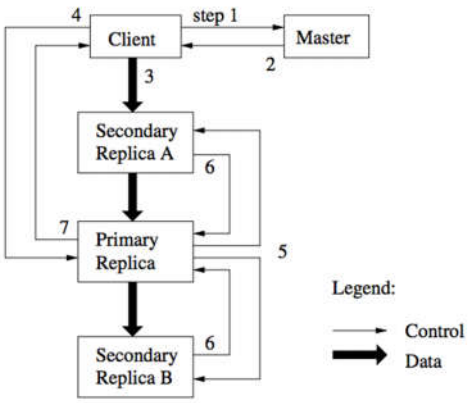

Figure 3

The client sends a request to the master to know which chunk server holds the data and also to know the location of the data.
The master then replies to the client with the identities of the servers which hold the data. There can be two kinds of servers named primary and secondary. The purpose is to have a backup of secondary servers in case the primary ones fail.

The client upon receiving this response starts to write the data onto the server replicas in any order. It starts off by writing the data to the primary servers. Once the write operation is complete, the primary servers send back and acknowledgment to the client. The client now

starts with writing the data to the secondary servers. Upon successful completion, the secondary then send an acknowledgement to the primary servers stating completion of write. Now the primary sends the confirmation to the client and the write request is complete. In case of any errors or corruption during the operation, the client is notified.

II. IMPLEMENTATION

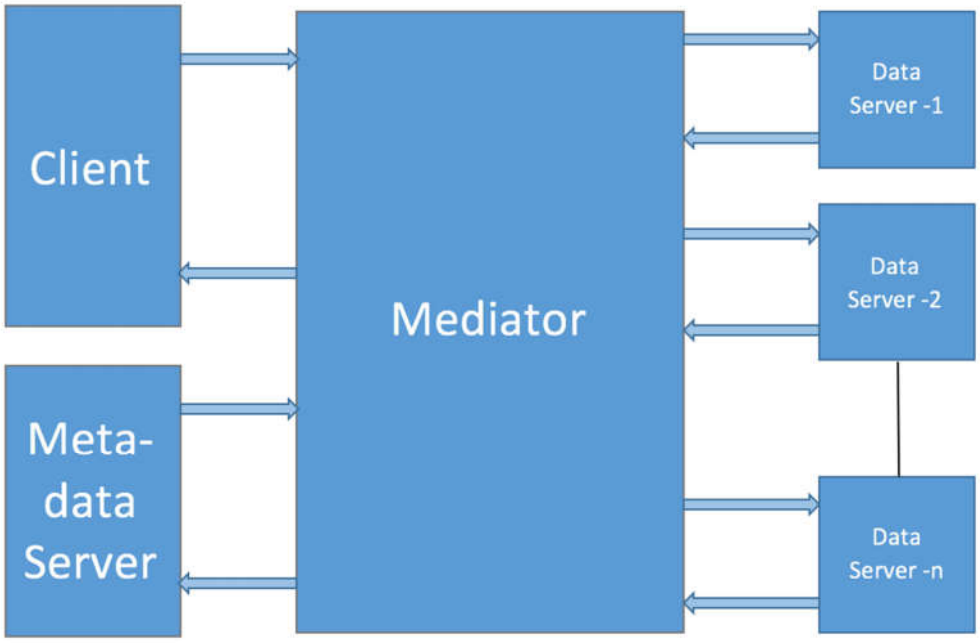

Figure 3: Architecture of the implementation

We implemented the client-server architecture taking cues from the above mentioned file systems. The architecture is comprised of four elements namely client, metadata server, mediator and a number of data servers. The implementation architecture is shown above in figure 3. The details and significance of each component is explained below.

There are a few assumptions that have been made in the implementation of this module. Firstly, there is a single server client whose requests would be handled by our system. Moreover, we assume that the MetaData server and the Mediator never go offline.

A. *Client*

The client in the implementation is assumed to be a single client which is the source of origination of read and write requests. The user uses the application at the client server to initiate the request. The XML-RPC protocol is used to send requests across to the Mediator to read and write the files in the form of put and get.

B. Mediator

Code implementation for mediator module:

mediator.py acts as mediator between client (remote_tree.py) and metaServer.py and dataServers.py servers.

repair_server function repairs the list of servers which is in the global variable **repair_restarted_server_list**. The list gets appended by get and put function whenever there is a server fail or corrupted data in the server. This function retries for 6 times with every 5 second interval. After maximum number of retries, the repair function do nothing and exit the function.

Quorum_voter function ensures if data from multiple servers are same and the count of same data is equal to or greater than Qr(read Quorum), then read of data is said to be successful.

put function puts the data into multiple servers. If the server has crashed, it retries for 6 number of times within an interval of 5 secs each. It calls **repair_server** function at the end of all writes.

get function gets the data from multiple servers as requested by the client. It calls **Quorum_voter** function to get the majority voted copy of the data. It calls **repair_server** function after the **Quorum_voter** function.

This module is the most important part of the system design. It acts as a resolver between the client and the Data servers. It is responsible for a variety of tasks which involves catering to the requests originating from the client. The functions performed by the mediator are listed below:

i) Replication across DataNodes

The number of servers that can be initialized to decide the replication factor is programmable in the mediator. The programmer can decide the level of replication of data across DataNodes. The mediator then initializes the DataNodes on different ports which are all listening to the client.

ii) Quorum Criteria

The read and writes from the client have to satisfy the quorum criteria to support consistency and fault tolerance in the system. The quorum for read and write is also programmable which depends on the number of data servers being initialized.

The concept of quorum is such that it ensures the consistency of data across the system data servers, that is, any read request will not be returned two different values for the data. Similarly, upon a write or put request, all the data servers need to have the same written value. In case of a failure while writing to a disk, the request needs to retry to write the data, else, fail the put request completely. There cannot be a partial write to the disks.

Our implementation involves a function *quorum_voter* which caters to the quorum requirement for reads from the client. The write quorum needs to be same as the number of servers initialized. The read quorum however can be programmed in the mediator depending on the value of 'N' replicated servers. Typically, it needs to be greater than (N+1)/2 servers to achieve consistency [3].

Whenever a read request originates from the client, it calls the above mentioned function. The mediator would then request the DataNodes to reply with the values for the key passed. Let's assume the number of data servers initialized is N=5, and hence the read quorum would be 3. So upon a matching response from 3 data servers, the requested value would be sent back to the client by the mediator.

For write requests, all the 'N' number of data servers need to be written to as the write quorum would be the number of servers, which is N in this case. Even if write to a single server fails, the entire write operation fails. But our implementation has a relaxed consistency model for this approach which is taken care by the error correction module as explained in subsequent parts of the report.

iii) Error Detection and Correction

The mediator is also responsible for the detection and correction of error in spite of the measures taken for fault tolerance as few programming errors may cause a system to fail. The *quorum_voter* function also takes care of any 'read failure' upon a read request. For instance, if data reply from one of the five servers does not match, the mediator marks it as corrupted and adds it on the list of servers that need to be restarted. It then refreshes these serves and replicates the data from the healthy servers.

As mentioned in the previous sub-section, in case of data corruption during a write operation to one of the five servers, the write operation still succeeds. And this error is corrected upon a read read operation as mentioned above. For example, if newly written data resides in four of five servers and a read happens to be on the same data, the mediator identifies that one server returns a different value, it adds this server to the repair list and refreshes in the same mechanism as done for read.

C. *Data Servers*

Code implementation for data server module*:*

dataServers.py starts multiple processes listening through multiple ports simultaneously.

Any server can be terminated by calling terminate function from any client using XMLRPC protocol.

dataServers.py also list the contents, which are keys.

dataServers.py also has the function corrupt which can be used to corrupt the data in the server.

The data servers are the DataNodes which are initialized in the mediator and are programmable to match the criticality of the data. All of these nodes are continuously listening to the mediator for read and write requests. These DataNodes store all the data of the file system with data replicated across them. On startup, the DataNode connects to the mediator in a spinning fashion until the service comes up. It then caters to the read and write requests of the client parsed through the mediator.

Unlike the conventional distributed file systems, where the client application interacts directly with the data servers, our implementation involves the mediator to be the buffer between the server and client in every transaction.

D. *MetaData Servers*

Code implementation meta server module:

metaServer.py uses almost the same functionalities of dataServer.

The MetaData servers are also assumed to be always up and running like the mediator. The implementation is built on these two assumption. As the name suggests, the metadata server contains all the list nodes and the metadata of the file system.

III. TESTING MECHANISMS

We have created a test corruption file to test data corruption in a single server. The data will be corrupted by the test file to one of the server. The mediator will repair the corrupted server. The mediator will call the repair_server function after every put and get operations.

For example, in 5 number of data server replicas, say if we kill a server and also corrupt one of the server. There are still 3 valid data out of 5 servers. Hence by considering the majority Quorum approach, read of the data is still successful.

The mediator will repair the server by writing right data to the corrupted server. The mediator in repair_server function replicates all the data present in the working data server to the fresh server which has started working again to maintain the consistency. If the server which has gone down has not come up again, then it retries for 6 times with 5 seconds interval. After 6 maximum retries, the repair_server function will exit the function.

IV. EVALUATION MECHANISM

The benefits of having multiple data servers which are being replicated helps not to lose the data even if 1 or more servers are not working.

If the bits are flipped in any of the server due to some technical glitches or through environmental faults. We will be able to recover the data through the other replicated server.

This system has provided the enhancement of providing the client to not easily lose the user data, even if the server crashes or get corrupted.

V. EVALUATION

The percentage of error in the case of a single data server is maximum if the server fails or corrupts. When a server is down, read requests do not get a response and in case of corruption, incorrect data might be handed back to the client which is inacceptable.

With the introduction of replication and migration of data across multiple data servers, the feature of fault tolerance is introduced where the dependence on one single server for response and consistency is reduced. Also, this approach would reduce the latency of the client request and increase dependability but at the cost of increased hardware.

The failure chances due to error in single data server compared to replicated service system is depicted below below graphically:

For a single DataServer

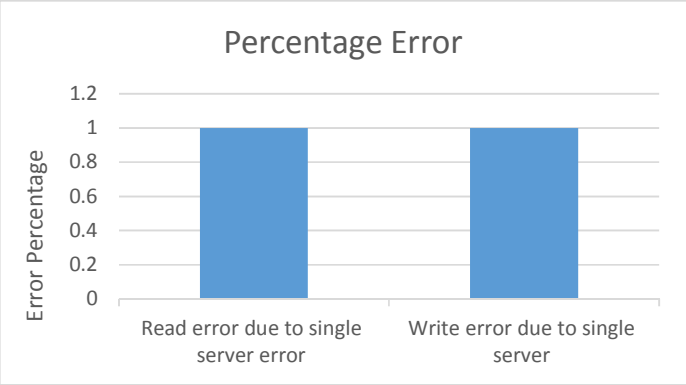

Figure 4

For a replicated DataServer with N = 5

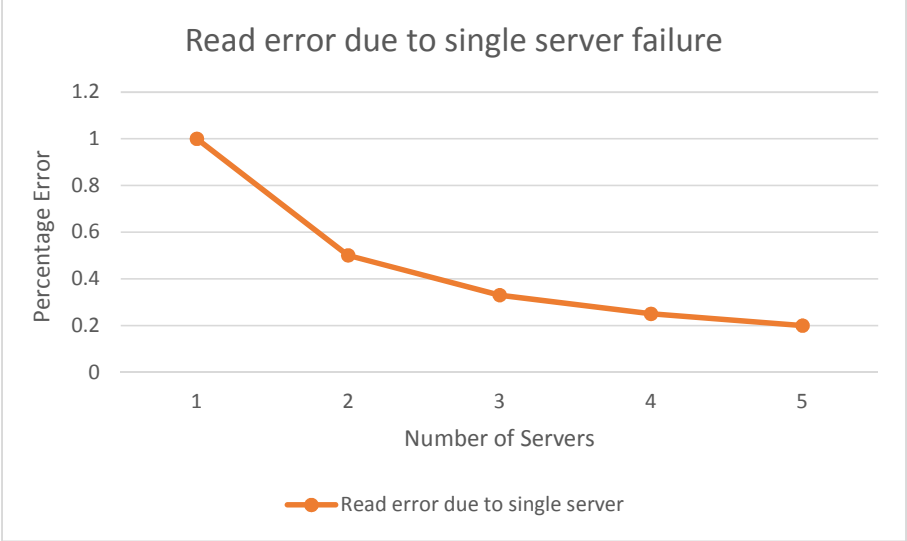

Figure 5

VI. POTENTIAL ISSUES

There are a number of potential issues still existing in our implementation. We have listed them out as follows:

1. According to Quorum approach, Nreplicas must be equal to Qw. The write to the server is said to be successful if write to Qw number of servers are success. If any server fails, write to the server should not be successful. As per our implementation, if any server fails, we will still the write the data to remaining Qw-1 servers, leading to read to be successful.
2. Quorum ensures read-write consistency. It doesn't provide before-or-after atomicity or all-or-nothing atomicity, the both of which provide protection to read-write consistency when the server crashes.
3. If Qr, number of server fails in the data replicas. The read fails.

4. Our implementation calls the repair_server function for every put and get, and this will increase the latency to the client. Repair_server function is called every time even if there is no corrupted or the server which restarted after a fail.
5. There is a possibility that during certain time instants when the mediator is updating the servers, there will may some data complication. This results in a replica which is not identical to each other.

VII. REFERENCES

[1] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *ACM SIGOPS operating systems review*. Vol. 37. No. 5. ACM, 2003.

[2] Shvachko, Konstantin, et al. "The hadoop distributed file system." *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010.

[3] Saltzer, Jerome H., and M. Frans Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.