A Distributed File System

Ambrose Tuscano
COEIT
University of Maryland Baltimore County
Baltimore, USA
ambrosetuscano@gmail.com

Jayalakshmi Mangalagiri
COEIT
University of Maryland Baltimore County
Baltimore, USA
jmangal1@umbc.edu

***Abstract*—The file system that is distributed on various file servers or in multiple locations is basically referred to as a distributed file system that enables programs to access or store isolated files similar to doing the same in local that allows programmers to access files from any machine or network. In this project, our aim is to design a simple distributed file system with replication and disconnection management features to improve the reliability and availability of our designed system. We have designed our system to accomplish features like replication transparency, structure transparency, high availability, resilience in presence of node failures, making our system more feasible and user friendly and with reasonable performance rate.**

***Index Terms*—Distributed, replication, resilience, reliability, availability, performance, feasible, user friendly.**

I. Introduction

The importance of a Distributed File System is very well known as it allows users of physically distributed systems to share their data and resources by making use of a common file system.

The configuration on a Distributed File System is considered to be the collection of workstations and mainframes that are connected through a Network. In a Distributed File System, creation of namespace and making the process transparent for the clients is vital.

The Distributed File System should be designed in such a way that it consists of two major components : Location Transparency which achieves via namespace component and Redundancy which can happen through file replication component. Whenever there comes a scenario of any failures or heavy loads, these components come together for enhancing the availability of data by sharing the data on multiple locations to be logically grouped under one folder referred to as Distributed File System root.

The remaining of our paper is structured as follows, section 2 presents the survey of previous works, section 3 presents the details of our system, section 4 presents the semantics our system, section 5 presents the assumptions made while building our Distributed File System, section 6 presents the commands that a user can give on the client, section 7 presents the evaluation of our system, section 8 presents the instructions on how to access and run our Distributed File System and finally the conclusion of our work.

II. Literature

In [3], the authors aim in this paper is to build the Coda File system which is considered as a descendant of Andrew File System (AFS) that basically retains the positive features of AFS while offering considerably enhanced availability. The authors of this paper are expertise in using AFS over the past five years and are very happy about the functionality, performance and feasibility in administering AFS. However, they were equally concerned about the vulnerability of AFS in server failures and network components which can bother many users for times ranging from a few minutes to many hours. So, it is important to build a system like CODA which is resilient to failures which strives for constant availability of data that enables users to continue working irrespective of any failures in the system.

The main idea behind this paper is to offer users with benefits of a shared data repository while enabling them to depend completely on the local resources when that repository is partially or completely not accessible. The authors have described the design and implementation of CODA with well motivated design choices and elaborated key data structures and protocols. The authors have achieved all the desired expectations from the CODA design by two complementary mechanisms called server replication and disconnected operation which is quite similar to what we are trying to achieve in our project of building a simple distributed file system. The authors in this paper have also presented on how disconnection operation could be used for supporting portable workstations. The authors state that the well tuned version of CODA system would meet the goal of providing high availability without seriously affecting the performance, scalability and security of the system. However the questions on optimistic replication scheme is left open to the users for experiencing it.

In [6] , the authors aimed to present a comparative study on the two distributed file systems namely Google File System (GFS) and Hadoop Distributed File System (HDFS) which is an open source implementation of GFS. The comparison is made by considering various uses of parameters like design goals, processes, file management, scalability, protection, security, cache management, replication etc.

The authors have made clear comparisons on the design goals of GFS and HDFS where the main aim of GFS is to support large files. GFS is built depending upon the assumption that terabytes of datasets will be distributed across thousands of disks attached to commodity compute nodes where GFS is used for data intensive computations. Also, GFS stores data reliably even when any chunk servers, master or network partitions fail. It is very clear that GFS is specially designed for batch processing rather than users to have interactive use of the environment.

On the other hand, the design goals of HDFS are somewhat similar to GFS but it's just that the reliability with data storage can happen even when failures occur within name nodes, data nodes or network partitions.

The processes for GFS are the Master and chunk server whereas for HDFS are the Name node and Data node. And in terms of scalability for GFS and HDFS where both hold cluster based architecture. The authors also compared the replication strategy of GFS and HDFS where in GFS, chunk replicas are spread across the racks and the master automatically replicates the chunks. GFS provides the user the ease of specifying the number of replicas to be maintained. In HDFS, the replication is an automatic rack based system where by default two copies of each block are stored by different data nodes in the same rack and a third copy is kept on a data node in a different rack which holds much greater reliability.

In [7], the authors presented the survey of peer-to-peer (P2P) storage techniques for distributed file systems. The paper explores the design models and vital problems related to P2P systems along with some discussion on various research activities in the related field. The inherent properties of the P2P system and the analysis of characteristics of some major distributed P2P file systems have been explored in this paper. Several benefits of using P2P systems over other distributed storage mechanisms have been discussed along with presenting their analysis of open problems on the same.

The survey in this paper details CFS, Freenet, PAST, Ivy, OceanStore and Farsite systems that are built on the basic P2P philosophy i.e., a network of independent hosts that is structured loosely. Because of decentralization, symmetry, robustness, availability and persistence in data, P2P distributed file systems became a vital part of the research related to file systems. The authors of this paper claim that even after this kind of system is having limitations, it is quoted to satisfy the ultimate needs of the future computing environments.

III. System Details

A. Server Details

Our server process begins by creating two threads one for Server connections and the other for client operations. It is then that the main thread attempts to get connected with the other servers where the IP addresses of those servers are hard coded and provided previously. The socket is ran with default values, as in INET and TCP protocol. This guarantees that the connection is established and the two parties have a conversation until the connection is terminated by one of the parties or by a network error, a much needed feature for our work.

The wait time taken by the process after sending a connection request to each server to go to the next one is 3 seconds where this helps in opposing the blocking state that the connection thread seems to be entering as the sockets connect call is in a blocking state. Before the connection, the sockets are kept in a non-blocking state so that whenever a connection is established, they can get back to a non-blocking state.

The state of our Thread would be running in a non-daemon mode and the listener threads remain to run whereas the main thread exits when no connections are established. For example, if server 'A' is set up and now let's say another server 'B' is booted up, it will then create its own listener threads and then it will attempt to establish connections with other servers using the IP addresses that were stored in the server previously. Server A's Connection thread will be connected with the Server B's server listener thread and a socket will be created between the two servers. Once the servers are started, connections can be established automatically and once the connection is established, both the servers will create server connection threads for the purpose of communicating with one another.

Client connection thread in the server receives the client requests where it initially validates the client's commands and then takes action accordingly. The operations like create, read, download/write, refresh and ls can be performed on the files that are stored locally on the server and can also request the other servers to carry out the same operations. For suppose, a request to upload a new version of an existing file is received by the client connection thread, then the client connection thread will make the upload on the current server where the previous version of the file exists and then sends the request to other servers that contain the same file. Then the server connection thread helps the other servers to listen to any requests made.

A Global directory list is preserved by each server which consists of names of all the files, version numbers, file descriptors of the sockets and the IP address of the servers. At first, the Global Directory list is updated during the initial stage when the servers boot up and keeps on updating continuously when the operations are executing on the file.

The File name has the version number which is appended to the file name whenever any updates are made on the files where the version number represents the number of updates made to that particular file. Below is the fig.1 which illustrates our server design.

B. Client Details

The client process is used by the users to request files from the respective servers. It is a sequential process where the client waits for the commands given by the users and then performs the specified tasks i.e., it performs one task at a time. In our system, the client can connect to random servers which is basically achieving something called as load balancing whenever a server is busy.

The user can see the file when the client requests the file from the server. Once the user gets access to the file, they can

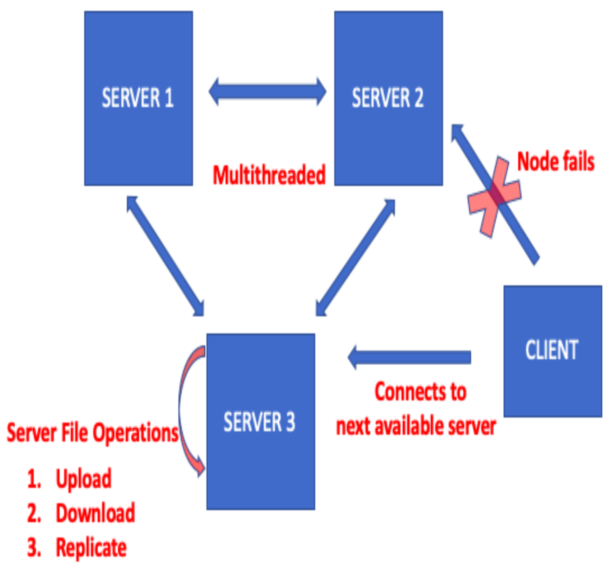

Fig. 1. Multithreaded Servers

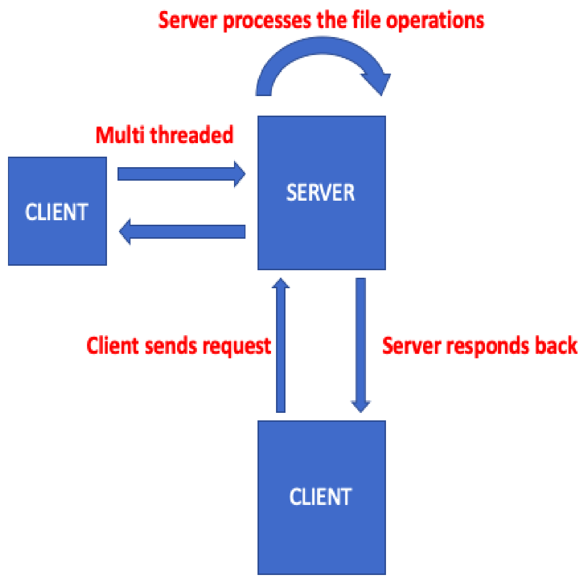

Fig. 2. Multithreaded Client

make necessary changes on the file and the updated file will be sent back to the server from where it has come. The clients can be started by connecting to all the servers where the IP addresses of the servers have been stored beforehand in the client. When the client tries to connect to the servers which are all down, then appropriate message will be presented on the users screen, else the client will prompt the user to proceed with the commands for getting access to the distributed file system.

If suppose a client is connected to one server and requests a file which resides on the same server, then it will sent back to the client but when a situation occurs where a client requests a file which resides on some other server, then the server which doesn't have that particular file will redirect the client to the server which has that file by returning the IP address of that server so that it can initiate another connection with that server to request the required file. This process will help in saving the time of sending files from one server to another and then again sending it back to the client. Also, this process will enhance the network traffic because less number of messages are being passed from here and there. Below is the fig.2 that illustrates our client design.

IV. Semantics

We have used the Upload Download model which is illustrated in fig.3 for accessing the files that are on the servers. We have used this model by referring to the NFS distributed file system where in theory NFS uses the remote access model, most of the implementation uses local caches, so in practice they use the upload/download model [1].

The file is requested by the client by sending the download command to the server which is when the whole file is downloaded on the client's machine and the client can have

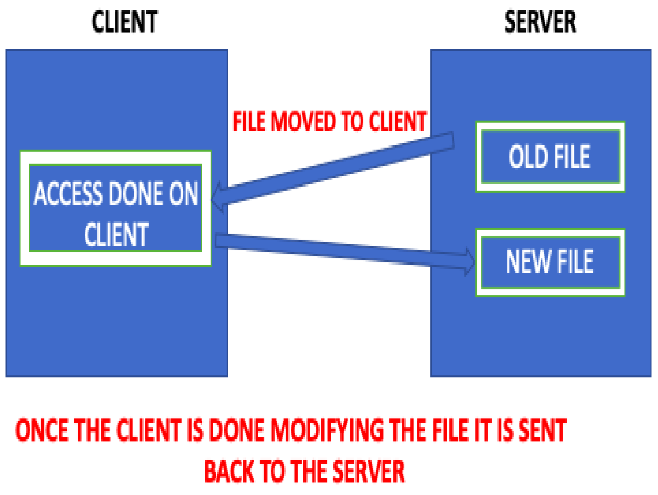
CLIENT
SERVER
FILE MOVED TO CLIENT
OLD FILE
ACCESS DONE ON CLIENT
NEW FILE
ONCE THE CLIENT IS DONE MODIFYING THE FILE IT IS SENT BACK TO THE SERVER

Fig. 3. Access Sementics

access to make necessary changes and sends back the file where it is uploaded on the server where the file originally came from. If in the case where the download file didn't go through any changes, then it is not uploaded onto the server due to no updates. Also, to prevent inconsistency in data, the other replicas of this file are also updated whenever any changes have been made.

Also, we have obtained the idea of using session semantics from the NFS implementation [1] to implement the distributed file system where no other user can see the changes made within the files except the one who is editing the file. Once, the user finishes updating the file, then only it is sent to the servers where any other users who attempt to download that file can see the updated information on the file. So, with this approach one can assume that how would the files be updated when multiple users use the same file to make modifications and close at the same time. In this scenario, whichever file reaches last on the server will be considered as the final updated version of the file. Once all the updating states are done by the servers, the global directories get updates.

Our server is designed to be stateless just like the NFS distributed file system version before 4[1][2] where they don't preserve any data about the clients. As there are no files that are opened on the server, there are no file tables to retain. Therefore, we can say that there is no limit on the number of files that a client could access. The file is uploaded on the server when it is created and then the server selects another server randomly to upload the same file which is basically considered to be a replication mechanism just like the CODA distributed file system[3].

Replication helps in promoting reliability and availability as it can have backups and also can enable file access even when a server consisting of the file is down. Any server may randomly fail or get disconnected. In order to overcome this we have also implemented replication of the file across servers where resilience is present when the node fails. When a node having the primary copy of the file is failed, the client can get connected to the random node which may or may not have the replicated copy as the replication is done on random servers. However, if the client is connected to the node which has the replication file, then the user can continue to perform the operations else the user is recommended to use the download command to maintain data consistency.

We have referred through a python package [4] and also referred through some of the file systems like Lisp Machine File system (LMFS) [5] which can be used for versioning the updated files. And so we have implemented versioning by appending the version number to the file names that exist on the servers.

V. Assumptions

The files stored are less than 20kBs. But still, we can create larger files that can be shared between servers due to which upload download model should work well for transferring complete files between the systems.

File is read more frequently than writing which is why we considered session semantics to be the appropriate choice for updating the files on the servers.

Replication is done at least on 2 random servers to increase the scope of availability.Only the downloaded files that have been modified by the client are uploaded back to the servers and in the case of two users working on the same file, whoever finishes last would be considered as the final updated file.

VI. Commands to run on client

- Ls: used to list down the current files that are available.
- Refresh: used to refresh the contents of the global directory.
- Download: used along with the name of the file to be downloaded that is requested from the server to make necessary modifications to the file and then uploaded back to the server from where it is originally taken. However, if no updates are made to the file then it does not upload anything on the server.
- Upload: upload command is used to upload the modified file to all the servers where the file exists i.e., on all the relevant replicated files.
- Create: used to create new files on the client in a text editor which helps the user to write into it. Once the user finishes writing and closes the file, it automatically gets uploaded onto the servers.
- Read: used to read the files that are created and updated. But, read is done locally on the client memory. When the file that we want to read exists in the local memory of the client, then it reads directly from the file, else it displays a message to the user to go and download the file.
- Write: used along with the name of the file to be downloaded that is requested from the server, If file is not on the current connected server, it checks for the file in global directory and if the file exists it initiates a connection with the server where the file is located and then downloads the file to make necessary modifications and then uploads back to the server from where it is originally taken and also will update all the replicated files on other servers. However, if no updates are made to the file then it does not upload anything on the server.
- Append: this command works just like the write command where we can enter new data to the end of an existing file. Once the file is appended with the data, it can be read from the same client. However, to maintain consistency it is always a best practice to use the upload ¡file¿ command after every append to share the file over the system.

VII. Evaluation

- Scalability: We have evaluated our system incrementally with 4 servers and 4 clients. It is evident for us that our system is supporting scalability at this time as required by the set guidelines. Also, all the clients are properly managed across the servers as we implemented load balancing.
- Resilience: Our system is tolerant to node failures where one node is failed due to any interruption or network connectivity it gets connected to another node. When a node crashes the client automatically searches for the next node and randomly connects to a server. We then run the file operations to confirm that they work and operations like create, download and ls work perfectly for the system.
- Performance: We have also tried to evaluate our system for the performance under different load size by recording the average of timestamps over multiple runs after each

operation which includes replication mechanism, download, write, update, upload.However, the system currently doesn't support operations on files over 1mb.Time mentioned is in seconds and rounded to the nearest whole value.

	Time	
File Operations	1kb	less than 1mb
Create and replicate	0.4	0.7
Download, write and update	1	5
Upload to all replicas	0.7	4

Table.1

VIII. Instructions to access our system

- Download the server code and the client code.
- Create folders named Servermemory and Clientmemory on the same place where the server code and client code is stored.
- For multiple servers, since we have coded our system to run function over LAN, we need to run the server code on Multiple machines. Client code can be run along with it too.
- Create a file names test 1 which should be empty. The reason we need this file is that when the servers interchange their lists when they boot up, we would need to have at least one file in our local folder.
- Enter the appropriate IP addresses of the servers on the client code and server code.
- Change the directory paths of Clientmemory in client code and directory paths of Servermemory in the servers code.
- Finally, to run our code, open the terminal and go to the appropriate folder where the client and server code is stored using the cd command and then type : python client.py to run client code, python server.py to run the server code.
- In this manner, Client code can be copied along with Clientmemory folder at different locations to run as a separate client.

IX. Discussion

In our work, we have achieved replication and disconnection management features like Pessimistic replication and resilience in presence of node failures where a client can get connected to another server when the current server connection disconnects. In addition, we have also achieved a functionality which lets the client redirect to another server by returning the IP address of that server when the replicated files are not present in the previous server and the users can perform the same file operations on the client with the replicated files just as they do with the original files on the servers.

The system we created is an upload download model which makes changes at every upload. Each change is documented as a version change and similarly replicated throughout the system. We also assume that our system has more downloads, rather than uploads which make changes to the file and the version. So at every download, the newest version of the file is sent to the client. A client can do file operations on its own memory without affecting the server till it uploads the file. Hence before every operation we recommend the user to download, just to get the latest version of the file. Keep in mind that we have minimal operations that make changes to the file, so mostly consistency is achieved. Here we wanted to achieve optimistic replication by designing our system to be working similar to a version control system that basically uses the copy-modify-merge model [8]. Our system ensures that the users are allowed to edit the local versions of the files, users can modify the files by downloading and then uploading them back to the server from where they are originally taken, all the file operations can be done as scheduled in the order they are received, the changes made to the files by the users are committed permanently. Also, conflicts are managed at the time of uploading the file to the system for the user to resolve it. Our initial approach was that the version of file before client access and after client access (just before saving the file) will be checked, and the operation will only go through if the version at both times is the same. However, the limitation that we could see in our system at this point is that we were not able to achieve the same due to inconsistencies and shortage of time , and hence our system works to a point that whichever file saved at the last is the finalized version.

Furthermore, our system is considered to be horizontally scalable and is certainly possible to increase the number of servers and clients upto an extent but then we also couldn't reach a point where we could evaluate on getting a fixed number for increasing the servers and clients as our system is currently not using the broadcast mechanism of connecting servers.

The technical challenges that we faced were with respect to version control which didn't work as expected earlier but then we somehow tried to fix the issue. In addition, we had to explore to a great extent to decide on the architecture that would suffice our requirements. Since many of the version control systems are centralized and not peer-to-peer, it became challenging for us to understand how we can achieve it for peer-to-peer. However, we have found our way and progressed with the aforementioned architecture to accomplish our goals.

X. Conclusion

Our goal in designing a simple Distributed File System is to have a system which is completely transparent, supports replication to be highly reliable and available, and also to be resilient in presence of node failures. We have accomplished our desired system by adding on many operational features like ls, refresh, create, download, upload, compare, read, append, which held our system to be more efficient in terms of its performance.

References

[1] http://si.deis.unical.it/ talia/aa0304/dis/lezione7-2p.pdf

[2] https://www.cs.swarthmore.edu/ newhall/cs85/s08/trilok.pdf

[3] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere (1990). Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on computers, 39(4), 447-459.

[4] acidfile · PyPI

[5] Versioning file system — Wikipedia Republished // WIKI 2

[6] R. Vijayakumari, R. Kirankumar, K. G. Rao (2014). Comparative analysis of google file system and hadoop distributed file system. International Journal of Advanced Trends in Computer Science and Engineering, 3(1), 553-558.

[7] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, R. Campbell (2005, April). A survey of peer-to-peer storage techniques for distributed file systems. In International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II (Vol. 2, pp. 205-213). IEEE.

[8] https://en.wikipedia.org/wiki/Optimistic$_replication$