CABNfs: CMSC621 Final Project Report

Christian Badolato*, Andre Nguyen†
Department of Computer Science, University of Maryland, Baltimore County
Email: *cbad1@umbc.edu, †tv28617@umbc.edu

***Abstract*—We present CABNfs, a distributed file system architecture and implementation that provides standard file system functionality while also managing aspects unique to distributed systems such as concurrent writes and fault tolerance using replication. CABNfs is a high read and create availability system that preserves file integrity and consistency in the presence of multiple node failures and allows for non-conflicting concurrent writes. We run experiments to show the expected behavior of our system at work, as well as to uncover the limitations of our implementation.**

***Index Terms*—Distributed computing, file systems.**

I. Introduction: Distributed File Systems

When it comes to modern computing, there isn't a one-size fits all solution to file management; many different types of modern file systems exist which offer varying supported file sizes, management attributes, journaling, and protections. Distributed file systems are a special subset of these systems that distribute the files they store across multiple storage nodes which can provide fault tolerance, redundancy, consistent access times across geographically diverse locations, and scalability that it impossible with single-server based solutions. However, there cannot be a single 'silver bullet' distributed file system which satisfies all possible needs — guaranteeing complete reliability and consistency will end up sacrificing availability, guaranteeing availability of all operations if even a single node is active will sacrifice consistency, and other situations such as these will always result in a sacrifice of at least one attribute.

For this project, we were tasked with designing and implementing a simple distributed file system which guarantees consistency even when multiple concurrent writes are requested. In this paper, we propose CABNfs, a high read and create availability system designed to preserve file integrity and consistency in the face of multiple node failures and to allow non-conflicting concurrent writes with no chance of file corruption. We then continue to show CABNfs' file operation metrics gathered from empirical experimentation, reflect on the lessons learned from creating this system, and propose next steps to improving this system in the future.

II. System Architecture

CABNfs was designed to be a high read and create availability, low write availability system which provides fault tolerance through file replication and consistency through a primary-server-per-file architecture. The availability traits of the system lend itself well to applications where data is frequently created and read while being rarely updated; laboratories in which test results are frequently generated and used as inputs into other experiments would be an ideal use-case for CABNfs as the redundant storage and fault tolerant nature of the system would allow for nearly uninterrupted service short of total power failure of all nodes present within the system.

The Linux Filesystem in Userspace (FUSE) API was utilized to abstract the internals of file operations within the Linux kernel and to allow us to focus our effort on the design of the filesystem itself. Through FUSE, custom code can be run on top of standard Linux operations such as `unlink` and `open` to provide users with a completely Unix-like interface with the system capable of handling shell commands such as `echo`, `ls`, or `touch`.

RabbitMQ was chosen as the messaging broker for CABNfs to support communication between the nodes; RabbitMQ is an open-source message broker which implements the Advanced Message Queuing Protocol (AMQP). This service allows us to define both broadcast and direct messaging queues which accept arbitrary JSON statements over the network, perform actions based on these messages, and reply back to the sending entity with custom response data.

CABNfs is a fully distributed system which doesn't rely on any centralized server or set of servers for operation. Similarly to many modern database solutions such as Scylla [Scy15], any server can run the full application without relying on any overarching management nodes. This allows quick recovery of a completely down system as only a single node needs to be brought back online for total operation. We note that we included a RabbitMQ monitoring management node in our implementation to monitor RabbitMQ. This extra node is not needed for operation of our file system and can be excluded from deployments if desired. The monitoring node is included purely for the convenience of a dashboard.

CABNfs mounts a physical directory containing the replicated files onto a virtual Linux file system. A user interacts with the virtual file system within the Linux terminal. When the user performs an operation, the virtual file system calls into the FUSE API and hands the request off to our code which allows us to perform any logic required with that operation including messaging between nodes if required. Once the request is completed, either the operation is performed on a physical file system — which may or may not be the actual server with which the user interacted — or an error is returned to the user. Figure 1 shows an illustration of the system architecture.

The replication and primary-server-per-file architecture were inspired by the Google File System [GGL03]. For every file within the system, a server which contains a replica of that file is chosen to be the 'primary' for that file. The primary server is

responsible for tracking and managing the replicas as well as de-conflicting and authorizing writes to existing files. A server is promoted to primary when one of the following conditions is true:

i No other server is listed as the primary server for a given file on startup
ii A file is created on the server
iii The server is randomly chosen to be promoted for a file by the file's current primary which is shutting down

Once a server is promoted to primary it finds all replicas of the file within the distributed system and re-balances the replicas so only a configurable number of replicas exist within the system. CABNfs uses an 'at least' replication factor — since every server requesting to read a file must have a copy of that file locally (as given within the project specification), it would be impossible to ensure only a configurable number of replicas and also let every node read the same file at the same time. Because of this, CABNfs only guarantees to maintain at least the replication factor of replicas. The re-balancing operation will attempt to delete excess replicas, but will not delete files currently open for reading or writing. These replicas ensure that the file can still be read from regardless of the current online server count and that total disk loss of any one server will not cause a data loss of any file.

During server startup, CABNfs loads the local file replicas already present on the node into memory and learns the primary server for each file from the other nodes; if no primary is defined for the file, the server will attempt to promote itself to primary. The server also loads file versioning data and a list of all files present in the system which it keeps within memory.

The high availability of read and create operations is made possible due to restrictions placed on writes within the system. While reads and file creations can occur at any time if one or more nodes are active, writes will be declined unless every node is online. This ensures that every server which contains a replica of the file is aware of the change and no incompatible changes can be made without being caught by the file's primary server as detailed in the next paragraph. Since all replicas are mandatory updated on every change and no node can be offline and not accept the change, it is guaranteed that every server always has the most up to date copy of every file. Therefore we can have complete consistency and not need to worry about voting algorithms or achieving a quorum like in more write-available system such as the Galera cluster for SQL databases [Clu14].

Write and read requests are sent to the primary server for that file. Writes are de-conflicted similarly to the Cassandra database [LM10] except by using the version ID for a file (which is maintained in memory during operation and written to disk on shutdown) instead of the timestamp. Since writes are disallowed if every node is offline, every server has the same version ID for a file. If a proposed change contains the same or a dominate version ID as the primary's copy, it will accept the change and update the replicas accordingly, if a proposed change contains a lesser version ID than the primary's copy, the primary will accept the change only if the area of the file written to does not conflict with changes from a greater version ID. In this way, concurrent writes can be accepted as long as writes de-conflicted after committing a change do not conflict with that change. Reads are accepted in all cases, during a read, the primary replicates the requested file to the server requesting a read; since every read is performed on a local file, we don't have any network latency beyond the first read for this operation.

Delete requests are also sent to the primary server for that file. The primary determines whether or not the delete can be processed; if so, it deletes its local file and sends a request for all replica holders to delete their replicas for that file.

III. Experiments and Results

Experiments were performed using client simulation scripts, and each cluster node was run in a Docker container. Performance of the system was evaluated using two metrics: time taken to complete a request and request success rate. We varied the frequency of client requests to be one of 1 request/second and 10 requests/second, the data sizes involved (for writes) to be one of 1 kB and 1 MB, and the number of server nodes in the cluster. Operation types were also varied to demonstrate requests of different natures such as replication (create), transferring data (write), communication with replicas (delete), and communication with all nodes in the cluster (list files). Writes were also evaluated in conflicting and non-conflicting scenarios.

Table I describes means and standard deviations for time taken (in seconds) for various file system operations, as well as associated operation success rates. A replication factor of three was used in all experiments. Time between operations (in seconds) as well as file sizes were varied as well. Conflicts were simulated by having two nodes write to the same section of a file.

An analysis of the means and standard deviations in the results show that we see expected behaviors and trends. In particular, operations that do require communication with all nodes (such as list files) do see an increase in time needed to complete an operation when the cluster size increased. Operations such as create that require communication between a number of nodes that is a function of the replication factor do not see an as meaningful (when the standard deviation is taken into account) difference in time needed as the cluster

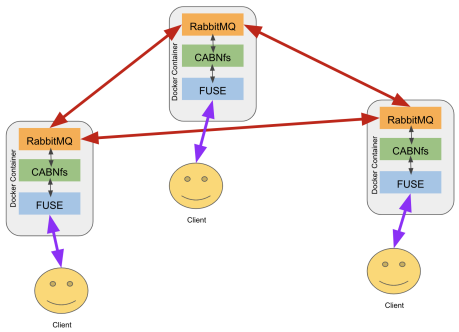

Fig. 1. An illustration of the system architecture.

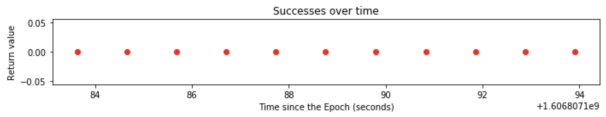

Fig. 2. An example plot of absolute completion timestamps for the list files operation, at one operation initiated per second.

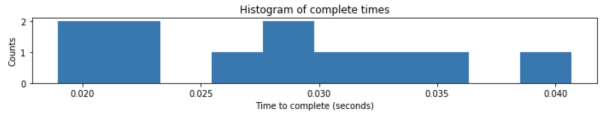

Fig. 3. An example plot of a histogram of the distribution of complete times for the list files operation, at one operation initiated per second.

size increased. As expected, data transfer is the most costly action, as shown by the results for the write operations.

While our system is able to successfully complete write operations when the data size is small at 1 kB, when the data size increases to 1 MB, the backend messaging fails, causing an inability to complete the write operations at the faster operation frequencies tested.

Also, when testing conflicting writes for the 1 kB data size, we observe the expected 0.5 success rate as exactly half of the proposed writes should be completed.

Our experimental data and results can also be visualized using graphs and histograms. We show two examples for the list files operation in figure 2 and figure 3. Figure 2 shows a consistent pattern of success for the operation over the course of ten seconds. Figure 3 shows a somewhat uniform distribution over complete times and no outliers. We omit plots for every experiment as they did not contain any interesting trends outside of what was already captured in table I.

IV. Discussion and Conclusions

All in all, we have introduced a distributed file system architecture and implementation that provides standard file system functionality while also managing aspects unique to distributed systems. We also illustrated the expected behavior of our system through experimental results.

Challenges included the learning curves for the tools used, in particular FUSE and RabbitMQ. Another pain point was the slow messaging and data transfer between nodes, and related messaging timeouts and TTL configuration.

While debugging in the context of a distributed system was tricky, our use of Docker greatly simplified our development process and iteration speed.

Next steps would focus on improving the design and configuration of the data transfer procedure to allow for a higher success rate of larger data size operations.

Acknowledgments

The authors would like to thank the teaching staff.

References

[Clu14] G. Cluster, “Quorum components,” 2014.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[LM10] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[Scy15] ScyllaDB, “Building the real-time big data database: Seven design principles behind scylla,” 2015.

TABLE I
MEANS AND STANDARD DEVIATIONS FOR TIME TAKEN (IN SECONDS) FOR VARIOUS FILE SYSTEM OPERATIONS, AS WELL AS ASSOCIATED OPERATION SUCCESS RATES. A REPLICATION FACTOR OF THREE WAS USED IN ALL EXPERIMENTS. TIME BETWEEN OPERATIONS (IN SECONDS) AS WELL AS FILE SIZES WERE VARIED AS WELL. CONFLICTS WERE SIMULATED BY HAVING TWO NODES WRITE TO THE SAME FILE.

Cluster Size	3 Nodes		5 Nodes	
	Time (sec)	Success	Time (sec)	Success
Create (1 sec)	0.04657 (0.008741)	1	0.05681 (0.01190)	1
Write (no conflict) (1 sec, 1 kB)	1.225 (0.3853)	1	1.203 (0.3817)	1
Write (no conflict) (1 sec, 1 MB)	NA (failure)	0	NA (failure)	0
Write (w/ conflict) (1 sec, 1 kB)	1.152 (0.4100)	0.5	1.143 (0.4223)	0.5
Write (w/ conflict) (1 sec, 1 MB)	NA (failure)	0	NA (failure)	0
Delete (1 sec)	0.02850 (0.01144)	1	0.04500 (0.02267)	1
List Files (1 sec)	0.02177 (0.01007)	1	0.02787 (0.006519)	1
Create (0.1 sec)	0.04744 (0.01193)	1	0.04779 (0.01551)	1
Write (no conflict) (0.1 sec, 1 kB)	1.218 (0.3862)	1	1.149 (0.4159)	1
Write (no conflict) (0.1 sec, 1 MB)	NA (failure)	0	NA (failure)	0
Write (w/ conflict) (0.1 sec, 1 kB)	1.074 (0.3893)	0.5	1.209 (0.4400)	0.5
Write (w/ conflict) (0.1 sec, 1 MB)	NA (failure)	0	NA (failure)	0
Delete (0.1 sec)	0.02866 (0.01104)	1	0.04514 (0.008158)	1
List Files (0.1 sec)	0.02089 (0.008058)	1	0.02849 (0.008410)	1