# The Redundant File System: Designing a Resilient and Scalable File System

Devon Slonaker
University of Maryland, Baltimore County
devons1@umbc.edu

Ben Lagnese
University of Maryland, Baltimore County
blagnese1@umbc.edu

***Abstract*—File systems are constantly facing the challenge of trying to be the most secure and reliable that they can be. Redundancy is one of the greatest ways to ensure that a file system does not face too many adverse effects when something goes wrong in the system. Redundancy is also one of the greatest challenges in file systems with people constantly trying to find the best and most efficient ways to handle how data is stored, how to load-balance the system, and how to make the system feel transparent to an end user.**

## I. INTRODUCTION

The Redundant File System (RFS) leverages various techniques to ensure that user data is securely stored and isn't lost. Utilizing a Linux-based environment, RFS is able to built-in functionality to easily copy, read, and write data from one node to another. RFS also ensures that data is replicated across nodes in an optimistic manner using CODA-like protocols. User data is partitioned into their own sectors so only users who are owners of that data can access, read, modify, write, and delete said data. RFS also ensures resilience in a catastrophic scenario such as when a storage or master node goes down. RFS will react in a downed node scenario to reroute the appropriate data from a failing node to another node that is still currently alive. In doing this, there is also a focus on attempting to give the user a seamless experience with logging in and accessing their data.

## II. ENVIRONMENT

The Redundant File System resides in a Linux-based environment. This is to ensure less operating system overhead from a system like Windows Server which, on average, uses more resources than Linux. When running a file system, the environment the system runs on determines how expensive your cost of operation will be. Using an environment with more overhead, and thus using more resources, will not be able to support as many clients or as much data as a system with less resource overhead.

The Linux operating system was also chosen for its built-in tools. Tools such as sshpass, ssh, touch, and rsync allow for a quick and easy way for systems to transport files. Instead of creating our own file sharing protocol, which may have numerous bugs, vulnerabilities, and exploits, we have opted to use rsync for file sharing which is a highly matured file sharing protocol. Rsync was also chosen over something like SCP for the way it copies files. Rsync will append new and changed data on copy instead of outright completely overwriting previous files and data to make way for an updated version. We feel that this approach will overall reduce the chance of corruption of user data as only changed and new data is copied instead of all the data being overwritten with new data, even if most of that data is largely the same as the previous data. The approach of using rsync also allows for less network overhead since less things are being copied with rsync as opposed to something like SCP. Using sshpass, we are able to run rsync between servers without being prompted for a password by the ssh handler every time a server wants to pass data or a file to another server. In this fashion, we can still have system passwords for security purposes and still have them enabled for standard rsync and ssh commands for things such as administrative purposes and maintenance, but RFS can internally bypass them to ensure a smooth file exchanging experience.

The other major tools used in RFS are Python, Flask, and Python's OS library. Python is a very powerful language with many great built-in

tools that make the development of our file system extremely simple and intuitive. Flask is a web framework and is the thing that gives our file system life. It enables us to develop a user-friendly environment in which anyone is able to use our system so long as they have a compatible browser. Flask also allows us to easily transfer data over HTTP, allows us to take advantage of reading user input from the web interface, and allows us to list out files to the users on the web interface. Flask also makes the upload (write) and download (read of files very simple for both the user and us as developers. A user can upload a file and from there we can then do things with the file in the back end such as versioning the file and replicating it. Additionally, Flask itself helps with load balancing through the use of multi-threading. So not only does this prevent traffic backup in the system by eliminating the use of just one single thread, but it also utilizes a more balanced approach to accessing the system resources. Using Python's OS library is another thing that plays a huge part in the communication in RFS. While Flask handles front-end communication, Python's OS library handles back-end communication between master-master and storage-storage communication. The Python OS library allows us to run system commands within python such as sshpass and rsync which make communication between servers seamless for both developers and for the users.

## III. ARCHITECTURE

The Redundant File System is set up in a way in which there are numerous "master" servers and numerous "storage" servers. The job of the master server is to keep track of what storage servers a single user belongs to, monitor the session data of the user, and ensure that a user is routed to a storage server in a given session. No file data is ever stored on or handled by the master server. With the storage servers, their job is to keep track of all user data including who the owner or "primary" storage server is of any given file, allows the user to upload (write/create), download (read) and delete files, and also replicates user data among other storage servers called "replicas". Because the master server does not handle any form of user file data, the storage nodes communicate with one another directly in a form of peer-to-peer basis. This eliminates any overhead and bottlenecks from having to constantly communicate with the master server for file communication, and also ensures faster file transfer since there is one less hop a storage server needs to make when it is working with any given file.

We have put careful thought into the file hierarchy of the file system to ensure the most efficient way of handling user files and data. We eventually settled on a hierarchy in which the file is not directly stored in a username directory, but instead a directory with the same name as an uploaded file is placed into a username directory of a user and the file is then placed into the corresponding directory with the same name. We have decided on this course of action for the file hierarchy so then the file system can keep track of file versioning. Stored alongside the file is a version file containing the current version of the file and the primary storage server that the file belongs to. The file hierarchy of a storage server is as followed: FILES (directory) -> FILE NAME (directory) -> FILE NAME (file) / VERSION FILE (file).

## IV. USER ACTIVITY

When a user initially logs into the master server, they are greeted to enter in a username. If the user does not have a username, they can just enter a new one and a profile will be created for them. If the user does have a username, they will be given their files. After the user has submitted their username, before having access to their files, the master server will order random storage servers (up to five storage servers) to ping the user, and the master server will then sort the round-trip-times (RTTs) of the pings to find the lowest latency storage server. The random selection allows for around equal distribution of users belonging to the storage servers without any one given storage server being saturated with users and potentially becoming slow. Because the selection of nodes is random, we realized that in a real world scenario with nodes thousands of miles away, this could potentially create an issue where a user may possibly be stuck with a slow storage server for their session. We attempt to remedy this by selecting the lowest RTT storage server for the user in that user's given session. These two techniques just for user login are great strides towards scalability in the real world. After sorting the RTTs of the storage servers to the users and selecting the lowest RTT for the user in their session, a session is created for the user on

the master server and that session is replicated on the storage server. After session creation occurs, the user is then rerouted to said storage server where a session is created. RFS uses sessions to reroute a user back to their storage node without having to constantly ping and sort ips for the user every time the user closes and opens a new tab. As long as the user is in the same session, they will be rerouted to the same storage server as they were connected to at the beginning of the session. This allows for faster connection on the user's end and less overhead on the server end as there is no need for constant pinging and sorting which takes up time and resources. Any servers that the user currently does not have in their list of known storage servers will be appended to their list of servers.

When the user is rerouted to their primary storage server, that storage server is now considered the "primary" server for any and all files created by the user in that server. At this point, the user can begin uploading (writing/creating), downloading (reading), and deleting files. When a user downloads a file, the download is handled locally by the current session storage server. Because the storage server holds replicas of user files, the user will directly download the file from the storage server they are currently connected to. This peer-to-peer download communication eliminates even more overhead by preventing the need of going through the master server to download the file, or the need of making a connection to the primary storage server of the file (assuming the current storage server is merely server a replica of the file) to alleviate another connection hop both the user and current session storage node need to take. The specifics about how user write and delete are handled will be discussed later in this paper.

## V. STORAGE SERVER RESILIENCE

In the event a storage server crashes or shuts down, we have a protocol that ensures that user data remains accessible and safe. For all intents and purposes, we will always assume that a storage server shuts down gracefully without interruption. When a storage server is given the shutdown signal, it will gather information of all the users and append its ip to an array which it will then send to the current master server. The master server will receive the request from the downed storage server, read the ip of the storage server from the data that was sent over, and then immediately remove the presence of the storage server from the list of nodes to prevent anyone from connecting to it. The master server will then read every user's list of storage servers to find if anyone has the downed storage server in their list of servers. If so, the master server will remove the storage server from their list of servers. After that, the master server will change the session data of the users currently connected to the downed storage server to the next available storage server in their list of servers. We use the next available storage server in the user's list of storage servers because since the storage nodes for the user are sorted by RTT, the next available storage server is also the new fastest primary server. The master server also removes all references of the downed storage server from all users' list of storage nodes. After that, the master server will send data back to the storage server going down with data of every user regarding their username, new primary server, and replica servers. After receiving this information, the downed storage server will then read the server data of all the users and then check to see if itself is the primary server of users' files. Any file it is the primary server for, it will change the version file to have the primary server ip to be the new primary server provided by the master server, essentially changing ownership of the file from itself to a replica server that is now the primary server for its files. After that, it will replicate the version file to all of that user's replica servers for that file to keep everyone up to date on the new storage server owner. After all of this has happened, the downed storage server can then be safely shut off.

## VI. MASTER SERVER RESILIENCE

In the event a master server goes down, we have a protocol that ensures that both master server data and user storage server data remains accessible and safe. Again, for all intents and purposes, we will always assume that a master server shuts down gracefully without interruption. As a definition, a "shadow" master server is a master server that is not the current active primary master server. When the master server is given the shutdown signal, it will read its list of available shadow master servers and ping them. The master server that is going down will copy all of its information to every available shadow

master server for the sake of redundancy. For the sake of being quick, it will pick the first available responsive shadow master server to be the new primary master server. After selecting the new primary master server, the downed master server will then send the news to all the available storage servers that they now have a new master server. When all of this is completed, the downed master server can safely be shut off.

As sort of a side effect to the way RFS implements the use of shadow masters, a shadow master can still route a user to a storage node. Although this is unintentional on the end of development, there is no foreseeable negative impact in allowing this functionality. If anything, after ironing out any bugs or issues that could be caused in the current implementation, this can help with load balancing on the primary master server.

## VII. Optimistic Replication

The Redundant File System uses a robust optimistic replication approach. Files are replicated immediately on creation, and afterwards are managed by version-vector controlled logic. Periodically, each storage node will loop over all the files it owns to sync with its replicas. For performance, this occurs on a fairly slow timer (with some randomness), which allows for concurrent writes to occur easily. To manage this, storage nodes maintain a version file for every file in the system. These version files list each other replica for that file alongside their respective version number. The version number for a node increments for every change it initiates. Each version number need not match when replication is complete. Instead, we utilize the causal relationship between version vectors. If the respective version numbers between two vectors are equivalent, the file is synced. When all elements of one vector are greater or smaller than the other, the smaller vector can be replaced safely by the larger vector. When none of these conditions are true, however, one of the nodes attempts a merge between the two files.

To facilitate the merge, we utilize the linux diff command. If diff can merge the files without any conflicting sections, the version vector is updated to hold the maximum values between the two vectors. The resulting vector and file are synced between the two servers and it is finalized as a successful merge. If diff fails to merge cleanly, however, the file is marked as a merge conflict. Utilizing a the -D flag on diff, merge conflicted files are marked with directives to show which parts of the file conflict. The version vector and file are then synced between the two nodes as in a successful merge. To improve the speed of detecting merge conflicted files, however, an additional file is created similar to the version file to indicate the conflict. The conflict must be fixed manually by uploading a new version. The directives are used to detect merge conflicts within the file, so if the file undergoes another merge without removing those, it will be flagged again as under conflict.

Optimistic replication is also used to handle deletions. When a file is deleted, it is initially moved to a deleted directory on the primary server. As part of its periodic syncing, the node will then ask all replicas to delete the node as well, moving it into their deleted directories. Replicas that do this successfully will also ask replicas to delete the nodes as well, to ensure no server repeatedly fails to heed the message from the primary due to conflicting sync timings. Once all replicas acknowledge a successful deletion, they delete the file from memory.

## VIII. DIAGRAMS

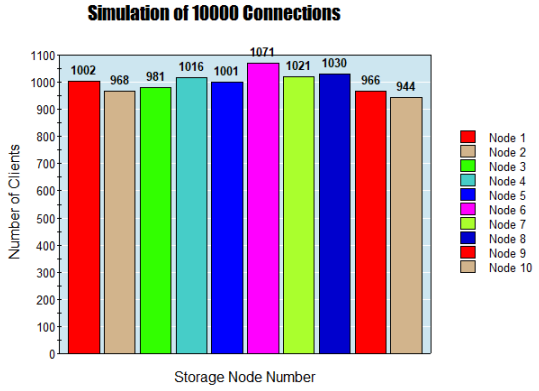


Fig 1. A diagram of 10,000 initial connections being distributed among their primary storage servers.

## IX. REFERENCES

[1] "Repairing Conflicts," *www.coda.cs.cmu.edu*. http://www.coda.cs.cmu.edu/doc/html/manual/x367.html (accessed Dec. 11, 2023).