Distributed File System

Repository: https://github.com/roblee04/Distributed_File_System.git

Jordan Randleman
Computer Science and Engineering
W1524026
jrandleman@scu.edu

Robin Lee
Computer Science and Engineering
W1597796
rblee@scu.edu

Venkata Yalavarthi
Computer Science and Engineering
W1652195
vyalavarthi@scu.edu

***Abstract*—In the contemporary landscape of data management, Distributed File Systems (DFS) play a pivotal role in efficiently handling substantial data volumes across distributed environments. They offer scalability by distributing data storage across multiple machines while ensuring fault tolerance and high availability through redundancy mechanisms. In this project, we present a distributed file system to tackle the challenges of managing files while accounting for scalability, fault tolerance, and concurrency within a distributed context. Central to our architecture are three components: a Python client library, a central server for routing requests (our middleware), and File System Nodes consisting of User Virtual Machines (UVMs) and Replacement Virtual Machines (RVMs) for operational execution and fault recovery. Our endeavor focuses on crafting a DFS tailored to enable users to seamlessly execute idempotent file system operations concurrently on a distributed network of storage devices that clients interface with as one logical machine.**

I. Project Goals and Motivation

In the contemporary digital era, the capacity to manage vast quantities of data efficiently across geographically dispersed systems has become a cornerstone of information technology infrastructure. At the heart of addressing this critical need are Distributed File Systems (DFS), which offer a sophisticated mechanism for handling data across multiple nodes in a network. This project embarks on the ambitious goal of designing and implementing a DFS that not only adheres to the foundational principles of distributed computing, but also explores innovative solutions to enhance scalability, fault tolerance, and data integrity.

The primary objective of our project is to develop a DFS that allows users to perform a variety of idempotent file operations concurrently (such as reading, writing, and deleting files) with the system presenting a unified, coherent view to each user as though they were operating a single logical machine. This system transparency is pivotal to the user experience, as it masks the underlying complexity of the distributed system, and makes interfacing with our system all the more appealing. The scope of this endeavor extends to ensuring the system's scalability, enabling it to manage increasing amounts of data seamlessly as the network expands. Additionally, we aim to fortify the system with robust fault-tolerance mechanisms, ensuring high availability and reliability even in the face of node failures or network disruptions.

The intrigue of this project lies in the complex challenge of constructing a DFS from the ground up, navigating the intricacies of distributed algorithms and technologies along the journey. The task demands a deep dive into the design and implementation of distributed systems, offering an opportunity for exploration, innovation, and learning. One of the key challenges we face is the development of an effective resource allocation algorithm. This algorithm must dynamically manage the distribution of files across the system's nodes, ensuring optimal load balancing and efficient data retrieval, while also integrating new system nodes as client file needs exceed the network's capacity. Furthermore, the system must be adept at detecting failures and swiftly reallocating resources to maintain service continuity, thereby demonstrating resilience and fault tolerance via replication.

Another pivotal aspect of our project is its concurrency control mechanisms, which are essential for maintaining data consistency and integrity. In a distributed environment where multiple users may attempt to perform operations on the same file simultaneously, ensuring that each operation is executed in an idempotent manner without adverse effects on the system's consistency is paramount. Furthermore, with all of our servers spawning a variety of background threads in order to execute periodic operations, our distributed file system has an extensive series of locking mechanisms in place to ensure data race safety. See Section IV-A for locking details.

II. Distributed System Challenges

Implementing a distributed file system introduces a set of complex challenges in order to make sure that the system supports concurrent file operations, while also maintaining system's reliability, scalability, and availability. Our project is focused on addressing key issues inherent to distributed systems, which includes performing efficient leader election algorithms for coordinated events, implementing effective resource allocation schemes to bring in new storage devices as needed, discovering file resources to route client requests across our distributed network, providing replication and replacement subroutines to account for machine failures, and controlling concurrency by ensuring mutual exclusion and maintaining file consistency through idempotent client operations. These challenges are critical to the system's ability to provide transparent and consistent file operations across distributed machines, and

must be faced while providing as much system availability to as many concurrent clients as possible.

See Section IV-B for specific details on how we addressed each of the aforementioned challenges in our system!

III. Literature Review

The development of distributed file systems has significantly evolved to address the challenges of data management across distributed computing environments. This literature review delves into important contributions to distributed systems, analyzing their implementations and methodologies. We have divided the literature we have gone over into two categories at a high level.

The first category explores a broad spectrum of distributed systems literature, providing a foundation upon which our project's design principles are built. This segment delves into how these foundational theories and practices have influenced our architectural decisions, highlighting the direct impact on our DFS design. The second category focuses on examining various distributed file system implementations, each addressing distinct challenges within their operational contexts. Through this examination, we gained a nuanced understanding of the diverse approaches to solving the inherent problems of distributed file management. This section aims to articulate how the lessons learned from these implementations have informed the development of our DFS, guiding our adaptation of solutions to meet our specific objectives.

Reading through the *Survey of Distributed File System Design Choices*[3], provided us with an outlook on the design decisions that make up the distributed system. The literature also gave a high-level architecture overview for the distributed file system whilst also discussing particular critical design choices for the distributed file systems in detail. Although this system was way more complex than the scope of our project, learning about its algorithms and goals gave us a clear understanding of how to implement a distributed file system.

Examination of *Distributed Database Design Methodologies*[1] offers a comprehensive survey of various approaches for distributed database design. The paper goes over the top-down and bottom-up approaches for distributed designs, it also highlights the necessity for a dedicated phase in the database design process specifically for distribution design. Although it is more inclined towards having a specific design phase for the distributed database, on a high level this mirrors the implementation of distributed file systems.

Another important system design inspiration was found in *Gossiping with Multiple Messages*[2], which explores the application of the Gossip Protocol in distributed systems for failure detection and data propagation. Although the paper discusses data propagation in large networks through peer-to-peer interactions, the exposure to its implementation assisted us into implementing our health monitoring and propagation algorithms for our system.

Frangipani[6] offers a look at a DFS designed for scalability and fault tolerance. Frangipani's implementation consists of a virtual disk system, Petal, to distribute the file system's namespace across multiple servers while maintaining a transparent view of the file system as a unified resource. This layered approach, which separates file storage from file system logic, significantly influenced our DFS design. By abstracting the storage layer, we are able to enhance our system's scalability and simplify maintenance procedures, simulating Frangipani's successes in these areas. Our implementation is similar to the two-layer structure: we have an upper layer, consisting of the user library and middleware, to manage direct file access in the second layer, which itself consists of distributed virtual machines that serve as storage devices.

Google File System (GFS)[5] is designed and implemented for large-scale, data-heavy applications. The Google File System emphasizes scalability, fault tolerance, and efficient processing of large data sets through a master and chunkserver architecture. However, while this architecture in particular proved too complex to directly implement in our system, we still strived to target each of the same principles as GFS in our own way: we made scalable design choices via our resource allocation and concurrency control algorithms, with fault-tolerance baked into the same systems that underlie our consistency.

Ivy[4] is a peer-to-peer file system that addresses the decentralized distributed designs. Ivy operates without central coordination, leveraging a network of nodes that independently store file versions and log updates. This decentralized approach ensures system scalability and robustness, facilitating efficient data replication and versioning to maintain system integrity across all the nodes. We have also emulated such a decentralized architecture abstraction to implement redundancy in our distributed file system.

IV. PROJECT DESIGN

A. Design Goals

Our system's design goals revolved around ensuring that we addressed the core challenges inherent to any distributed system, namely: heterogeneity, openness, security, failure handling, concurrency, quality of service, scalability, and transparency.

Heterogeneity and failure handling are simultaneously accomplished by our system's backup-server leader election protocol, wherein a backup server is elected to manage the system health of a subnetwork of nodes. Each subnetwork consists of a pool of backup servers replicating the actions of a primary server, which in turn executes client operations routed to it by the middleware. The leader's copy of the file system determines what gets transferred to newly allocated nodes as they join the subnetwork. The leader also handles subnetwork failures by constantly pinging the other nodes to ensure that they are awake, as well as by pinging the primary server to ensure that it remains active. Should any of the backup servers or the primary server go down, the leader has a pair of protocols it can execute to begin the replacement process. Downed backup servers are replaced with freshly allocated machines returned by our middleware, and a

downed primary server has its associated leader escalate its privileges to become the subnetwork's new primary server (after replacing itself with a new backup server to keep the same number of redundant machines)!

With respect to concurrency, consistency is managed by ensuring we only support idempotent client file operations. However, our servers also execute a broad variety of threads within their processes in order to be able to concurrently execute background tasks that monitor the state of the system during execution. These threads manipulate shared state: including, but not limited to, a logged history of mutating file operations, the current set of active primary servers managed by our middleware, which primary server allocation request is allowed to access the "ips/" subdirectory to register newly allocated storage devices on the middleware, and more! This shared state is manipulated by locking mutexes around each atomic data structure operation, while also keeping each critical section as small as possible in order maximize parallelism.

By leveraging this concurrent design, we provide scalability for the number of concurrent clients that we can connect to our distributed file system at any given time. Furthermore, we also support horizontal scalability by allowing the middleware to dynamically allocate new storage devices in order to serve new file creation requests, as they exceed to capacity of the current storage devices. That same middleware also provides system transparency: by having the client library API act like a regular file library that could operate on the native system, wrapping HTTP GET requests to the middleware within the library functions allows the middleware to route our request throughout the distributed storage devices.

This concurrent, scalable, and transparent system makes for a wonderful end user experience, with excellent quality of file operation service. Part of this quality comes in the ease with which our supports synchronizing access to the shared file system across clients. This support allows for different teams across the world to synchronize data access through a single convenient interface!

For security, we anchor our file creation operations to only occur in a dedicated directory, so as to make user-file identification trivial against our system's source code files. For the sake of our live system demonstration, we allowed for requests to come in from any machine to simplify our Amazon Web Services EC2 instance set up, however, the code could be trivially extended in order to only permit requests from authorized sources.

B. Key Components and Algorithms

Our system's key components are the client library, middleware, and the distributed file system nodes (aka the storage devices). The client library communicates user file operation requests (read, write, delete, copy, rename, or exists) to the middleware, which is a centralized server that routes the file operation request through our distributed storage devices. The file system nodes themselves are split into two classes: User Virtual Machines (UVMs) and Replacement Virtual Machines (RVMs). UVMs are the primary servers pinged by the middleware to execute user file operations, and RVMs are the backup servers to replace UVMs in the event of UVM failure. Each UVM has a pool of RVMs associated to it to maximize fault tolerance through redundancy, and the RVM pool elects a leader RVM among its members to coordinate health monitoring for the UVM/RVM subnetwork.

See Section VI-A to learn more about how our system implemented each of these components, as well as how they interact with one another!

Our distributed file system's key algorithms include leadership election algorithms to coordinate synchronized activities, resource allocation protocols to onboard new storage devices as needed, file resource discovery to route client requests across our distributed machines, replication and replacement subroutines to account for machine failures, and concurrency control to ensure mutual exclusion and maintain file consistency via idempotent file operations. See the following subsections for more details on how our system approached each algorithm.

1) Leader Election: The leader election process is crucial for maintaining the operational continuity of the system, especially in the face of device failures. Our distributed file system uses a leader election protocol to dynamically select an RVM responsible for executing coordinated activities (including system health monitoring and replacement algorithms) among a UVM's RVM pool. If 3 seconds pass without a leader pinging an RVM, that RVM will assume that its pool has no leader, and hence will initiate the leadership election protocol. Since each RVM has the IP address of the other RVMs in its pool, the RVM simply conducts the leadership election protocol by pinging the RVM with the highest IP address. If that RVM confirms that it can support the leadership role, that RVM is elected to become the leader, and begins executing coordinated activities! Otherwise, that RVM is presumed inactive, and the RVM with the next-highest IP address is pinged, etc.

This design makes for an extremely efficient protocol: the leadership election is a constant-time O(1) operation, since no intra-RVM broadcasting needs to take place to acquire the system properties required for a consistent leader election result to be produced by any RVM node. This constant time leadership election means that our protocol scales excellently with an increasing number of concurrent clients!

2) Resource Allocation: Our system addresses resource allocation by having its middleware allocate device resources as needed by the DFS' components at runtime. If a client attempts to create a file that our distributed file system's storage devices do not have capacity for, a new virtual machine is pulled from a pool of virtual machines preallocated during initialization, in order to horizontally scale our network and integrate the allocated resource into the preexisting network. The new file creation request is then routed to that freshly allocated storage device.

It is worth mentioning that, realistically, Amazon's EC2 instances would never naturally run out of disk while implementing our DFS tests. Hence, we have implemented a hard limit of 3 files per storage device in our submitted source

code, so as to artificially constrain the capacity of our system to allow for device allocations to take place during testing.

Furthermore, while this preallocation of resources during initialization works very well for the scope of our final project, it is important to note that a production-ready tool ought to use Amazon's "boto3" API instead: a popular library for Amazon Web Services (AWS) developers, boto3 allows for AWS resources to be dynamically and programmatically allocated.

3) Resource Discovery: Our system discovers new file resources by having the middleware ping every node in the network to see which machine can support the requested file operation. Machines can respond to the middleware with one of three options:

1) Ideal: this machine is the perfect candidate for the file operation. This will be returned if the machine has the file in question.
2) Possible: this machine can support the operation, but other machines should be checked first. This is the case when we want to write a new file or check if a file exists, but we do not have the file in question.
3) Invalid: this machine cannot support the operation. This can occur if the machine does not have the file but the client wants to read, copy, rename, or delete, as well as when the client wants to write but the machine already reached its file system capacity.

The middleware will immediately route the request to a node if it returns "Ideal", or any machine that returns "Possible" if none are ideal. If no ideal nor possible machines are available, a new resource is allocated, and that new device is considered to be the "discovered resource".

4) Replication and Replacement: In any distributed file system, ensuring continuous operation despite system failures requires a reliable mechanism for the quick replacement of primary and backup servers. Our system employs an automated protocol that monitors the health of primary servers (known as User Virtual Machines), as well as of backup servers (called Replacement Virtual Machines).

Every time a UVM receives a client request from the middleware that can mutate the file system (e.g. a write, delete, rename, or copy), that command is forwarded from the UVM to its associated set of RVMs. This ensures that the RVM file systems consistently mirror the contents of the UVM file system, with this replication allowing for quick RVM replacement of the UVM upon UVM failure.

In order to coordinate that RVM replacement of the failed UVM, the RVMs elect a leader amongst one another, which is then in turn responsible for managing all coordinated RVM activities. This includes not only the aforementioned UVM monitoring, but also intra-RVM monitoring so as to ensure that downed backup servers are swiftly replaced with freshly allocated storage device resources.

Should there ever be the unlikely circumstance that all of the RVMs in a subsystem goes down, and only the UVM remains, that UVM will still be able to allocate a new RVM to trigger the leadership election and RVM replacement protocols required resurrect the subnetwork. This process helps our distributed file system ensure file data integrity and system resilience in the face of sudden failures, leading to a fault-tolerant and highly available end user experience.

5) Controlling Concurrency: The concurrency model of our distributed file system mandates synchronized file access to prevent data inconsistencies, and to ensure reliable operations across multiple clients. We addressed this challenge by only supporting idempotent file operations, as well as by integrating several locking mechanisms.

Idempotent client file operations are supported in order to ensure that the system remains in a consistent state across parallel users running multiple operations. Idempotent file operations are so-called by virtue of those operations leaving the system in a consistent state after repeated execution. We show that each of our operations are idempotent below:

1) Read: repeatedly reading a file does not mutate the file system.
2) Write: repeatedly writing a file always wipes the prior file if it existed, then replaces the contents with the written data. Hence repeated runs always yield the same file with the same contents.
3) Delete: repeatedly deleting a file always guarantees the system is left without that file on board.
4) Copy: repeatedly copying a file (to the same copied name!) always ends up with the original version existing, as well as the same named copy.
5) Rename: repeatedly renaming a file always leaves the system without the original file, and with the renamed version.
6) Exists: repeatedly checking if a file exists does not mutate the file system.

For a concrete example of a non-idempotent file operation, think of "append": each time you execute append to a file, the system is left with a different (newly appended to) version of the file! Hence "append" is not an idempotent operation.

In addition to file operations, our system has to deal with shared state across a process' threads in a variety of ways. Threads are regularly spawned by our system in order to execute background tasks that manipulate shared data: the middleware spawns a thread to asynchronously allocate a new storage device while the client library waits for the operation to complete outside of a single TCP session, UVMs spawn a daemon thread to consistently verify that at least one RVM is alive, RVMs spawn a daemon thread to elect a leader RVM if they do not receive a ping from a leader within a 3 second timeout, and the leader RVM spawns daemon threads to become a UVM if it does not detect a UVM ping within 0.5 seconds, as well as to replace downed RVMs. These threads require locking to prevent data races and disk corruption for their shared values: the middleware locks to track the number of UVMs in the system, storage devices lock to access and update the RVM and UVM IP address text files, and RVMs lock to track how long it has been since they pinged by a leader. This ensures that we can expect consistent behavior from our system's atomic operations in concurrent contexts,

as we do not have to worry about data being left in an unpredictable state.

C. System Architecture

The system architecture on a high level consists of three main components: a user library to perform file system operations, a central server(middle ware), and the file system node. 1 shows how our DFS operates, illustrating how the user library communicates to the central server to have its file operations routed to the appropriate file system node. File system node executes file system operations and handle fault tolerance using replacement, health monitoring, and leader election in a distributed manner.

The use case diagram in 2 displays the overall functionality of our Distributed File System. It illustrates how the client interacts with the DFS through a series of file operations such as read, write, delete, copy, rename, and exists. These operations are communicated to the DFS via the middleware, which is responsible for directing requests and returning responses.

Two failure scenarios are discussed below using the sequence diagrams. UVM node failure sequence diagram 3 goes over the sequence of actions that take place whenever an UVM Node Failure occurs within the DFS. It showcases how the leader RVM from detecting the UVM Failure, also spawns a new UVM process on its own machine. Then the leader pings the middleware for a new RVM to replace itself with, after doing so, the leader updates all the RVMs with the newly replaced RVM IP address. Then the Leader terminates its own process.

The sequence diagram going over the sequence of actions taken in case of a RVM leader node failure is illustrated in the 4. It depicts that the RVM detects failure of the RVM leader. As soon as it recognizes this failure, the RVM pings all other RVMs in decreasing order to become the new leader. The new leader RVM listens to ping and confirms as the leader with the RVM that pinged it. Leader RVM is elected through these actions described.

V. Evaluation

Upon evaluating our system, we find that it performs fault tolerance operations flawlessly by spawning replacement UVMs, RVMs, and leader RVMs in a timely manner as needed to dynamically heal the network upon node failure. The system mitigates file consistency issues through its idempotent client file operations, as well as by ensuring that all UVMs have up-to-date RVM pools at the ready to act as drop-in replacements upon UVM failure. Our system handles scalability both with respect to the number of concurrent clients it can support, as well as in terms of how many files it can allocate: resource allocation algorithms and concurrent controls ensure we can support an arbitrarily high number of users with arbitrarily high file creation needs (so long as we have the physical disks available to store such!).

Please see Section VIII for an in-depth analysis of the specific metrics that we used in order to evaluate our implementation, as well as how those metrics grew with the number

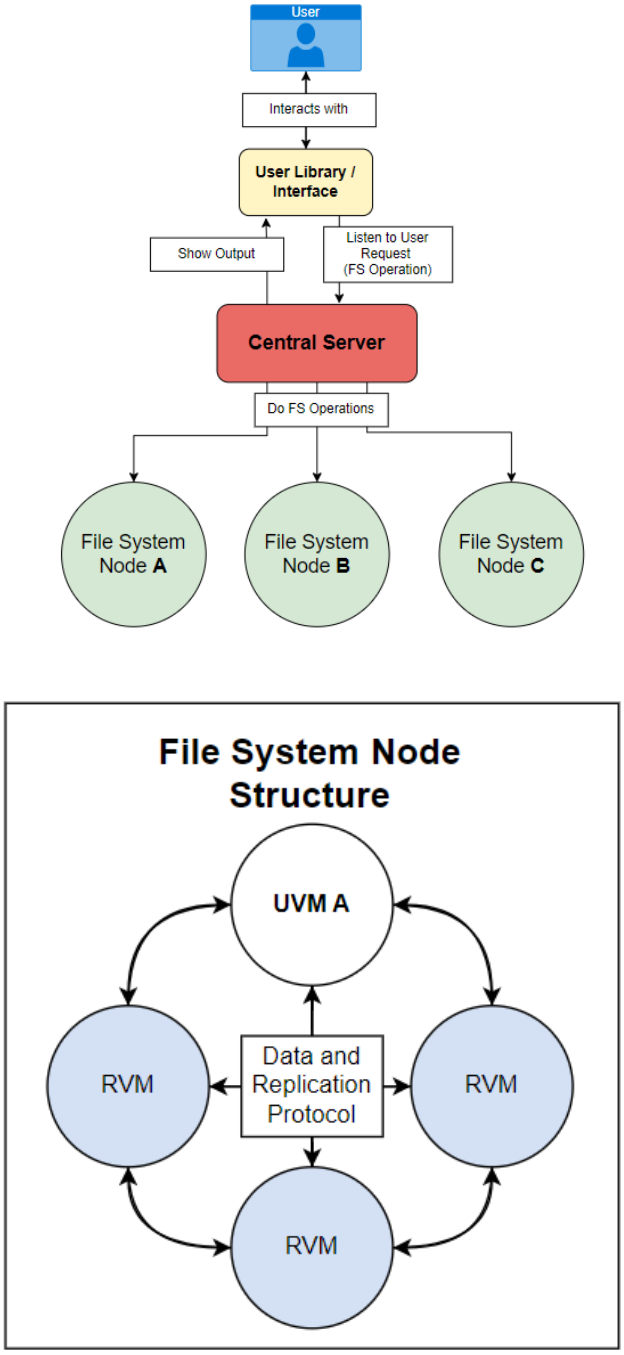

Fig. 1. High Level Architecture

of concurrent users. In summary though, it scaled extremely well across file operations even with increased concurrent user counts: with sublinear growth rates proving leveraged concurrent operations!

With regards to system challenges, we do have one component in particular that could be improved upon: our middleware. As the only centralized component in our system, it suffers the risk of congestion by being bombarding with

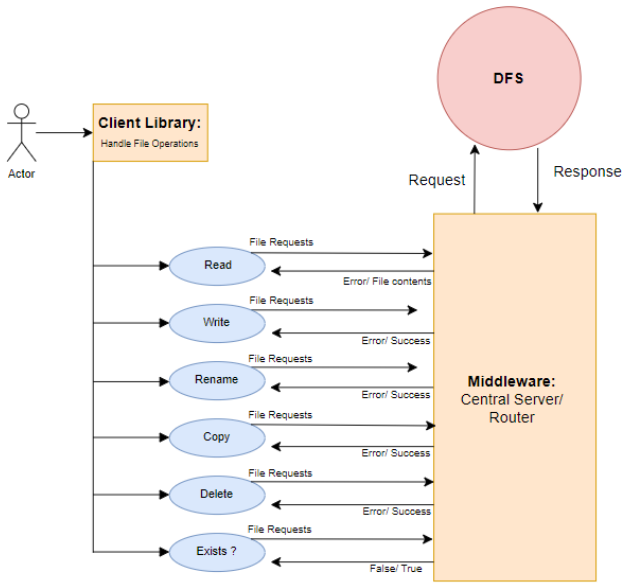

Fig. 2. Use Case Diagram: High level DFS Operations

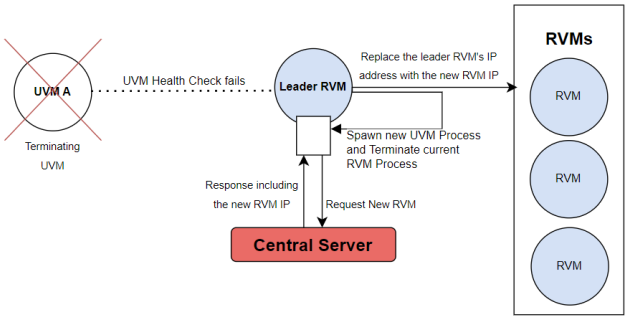

Fig. 3. Sequence Diagram: UVM Node Failure

client requests, though realistically the routing operation itself is quick enough such that each request can be quickly satisfied.

VI. Implementation Details

A. Architectural Style and Technologies

Our distributed file system is implemented as a Python client library that interfaces with a series of EC2 Amazon Web Services (AWS) servers. Each server runs a Flask application that listens for HTTP GET requests along different routes to communicate state and distribute data. The EC2 instances each fall under one of three categories: the middleware, User Virtual Machines, or Replacement Virtual Machines.

1) Client Library: Our client library is implemented in Python, and supports ”read”, ”write”, ”delete”, ”rename”, ”copy”, and ”exists?” operations on our DFS’ files. Each function makes an HTTP GET request to our system’s middleware. Should the middleware return a request for the client

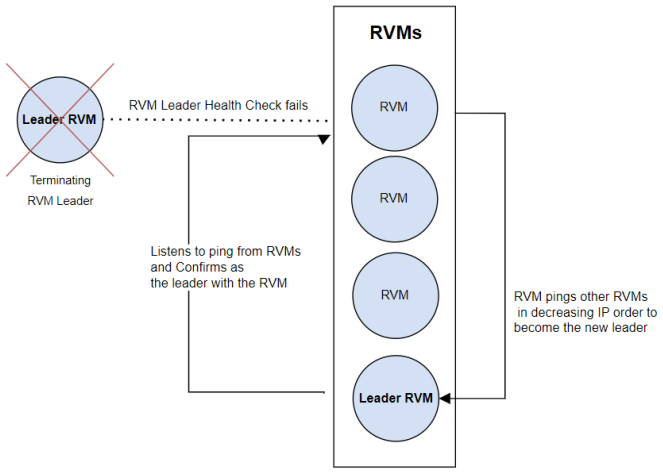

Fig. 4. Sequence Diagram: RVM Leader Node Failure

to stall due to the completion of an operation exceeding TCP’s timeout (e.g. when the middleware is allocating new storage devices), the client library will use a token system to repeatedly sleep until the middleware confirms that the token’s associated operation has completed and the client can proceed.

2) Middleware: The middleware (aka ”Central Server” or ”Router”) serves as a single, consistent point of communication that the client library can reliably ping in order to propagate file operations to our distributed storage devices. The middleware chooses which storage device to forward the client request to based on whether the device has the capacity to create the file as needed, or whether the device has the file at all. Should the middleware be asked to create a file that exceeds the capacity of the network’s current storage devices, the middleware will allocate a new set of EC2 instances in order to accommodate the request.

3) User Virtual Machines: User Virtual Machines (UVMs) serve as the primary storage device pinged by the middleware to execute a file operation, and they forward any request neither ”read” nor ”exists?” to an associated pool of Replacement Virtual Machines (RVMs). These RVMs are designed to serve as quick replacement machines in event of UVM failure. Furthermore, if the UVM detects that all of its associated RVMs have failed, it will allocate a new RVM to resurrect the network.

4) Replacement Virtual Machines: Replacement Virtual Machines are found in pools associated to a single UVM, as part of a logical storage device. RVMs receive mutating file operations from the UVMs to replicate on their own file system, so as to be able to immediately replace the UVM in the system upon its failure.

That replacement requires coordination however, in order to determine which RVM is responsible for replacing the UVM in event of failure, the RVM with the highest IP address is elected as the ”RVM leader”. Each RVM can independently conduct the election protocol to come up with the same election result

without pinging other nodes, since each UVM/RVM storage device subnetwork has a list of every one of its IP addresses shared amongst its nodes.

The RVM leader is also responsible for pinging the health of all the other RVMs, and if any of the RVMs do not respond, they are presumed dead then replaced with a freshly allocated replacement EC2. If any of the RVMs detect that they have not been pinged for their health by a leader passed some time threshold—3 seconds in our implementation—the leader is presumed dead, and a new leader election protocol is initiated.

5) Dynamic EC2 Allocation: Throughout this discussion, we have referred to allocating EC2 instances freely as if such were a given operation. However, in order to be able to do so dynamically at run-time, a proper production-ready tool would want to use Amazon's "Boto3" API: a mechanism by which to allocate AWS resources on the fly! And yet, as a team, we decided that since the focus of the project was intended to be on its distributed system components (e.g. its leader election protocols, resource allocation scheme, concurrency control, broadcasting, etc.), we decided to compromise on a simplistic allocation scheme.

Rather than truly allocating new VMs at runtime, we spawn a series of pooled RVM servers waiting on standby to be called into service by the middleware. Then, whenever one of our system's components needs to allocate a new resource, they ping the middleware, which in turns activates the pooled resource and yields the RVM's IP address to the allocating process.

B. UML Diagrams

Figure 5 represents the class diagram for the Python user library to perform various file operations.

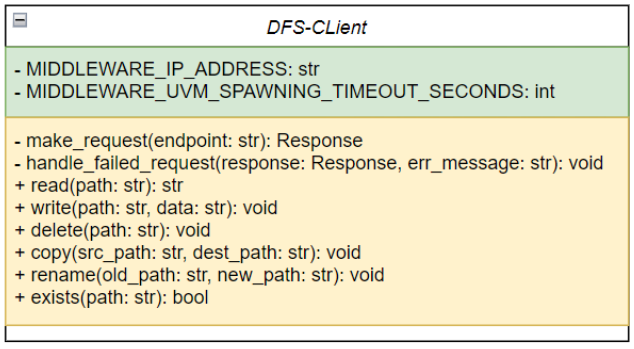

Fig. 5. Class Diagram: Python Library

In the case of the distributed file system at being full capacity (e.g. every single UVM is storing the maximum files allocated) and a write or copy operation is requested (write and copy are both file system operations that generate new file objects), more storage space must be allocated to accommodate these operations.

In figures 6 and 7, an example of the described scenario is shown.

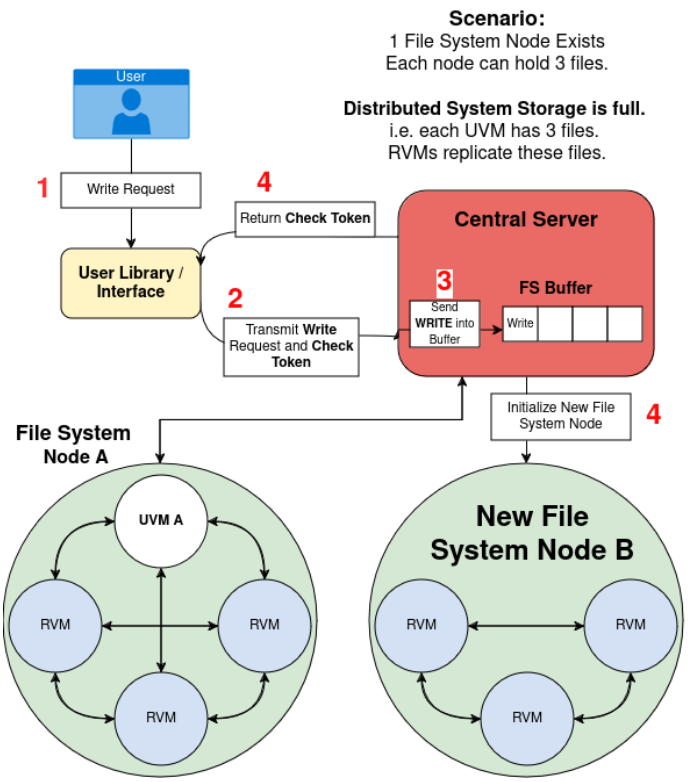

Fig. 6. Communication for Write Operation at System Resource Cap

Figure 6 denotes a user that is requesting to do a write operation on a system that is at full storage capacity.

Initially, only **File System Node A** exists and it stores three files. In this described system, three files were set to be the maximum capacity of each node. As a result, **File System Node A** is set to be at its maximum capacity. In other words, operations that do not generate new file objects are allowed on that node without extra accommodation. This would mean that operations such as read, delete, rename, and exists, would all be valid file system operations.

In our scenario, this is not the case. Our user wants to impose the write operation. The steps in the process of handling this request is denoted in red in Figure 6.

Firstly, our user's request is understood by the user library. Secondly, the request transmitted by the user library to the central server. Additionally, a **Check Token** protocol is initiated: this mechanism allows for the user library to check when there exists a File System Node to fulfill the user's request.

Thirdly, the system checks if the write request is a valid file system operation. In this case, the central server knows that all existing nodes (File System Node A) are at full capacity. Afterwards, the write request is sent into a buffer that is stored in the central server. This buffer is used to preserve the file operations that need to be done in the future.

Fourth, the central server returns the **Check Token** back to the user library. This indicates that the file system operation they have requested for has not be done yet, and that there does not yet exist a file system node that is capable of doing

so. At the same time, a new file system node (File System Node B) is being asynchronously initialized. In this operation, the central server allocates three machines from the machine resource pool to form the three RVMs for the new file system node. With these three RVMs set up, these RVMs would run a leadership election protocol and would then determine a leader. Then, that leader would request another machine from the central server to act as the UVM for that file system node. A full initialization of the file system node is seem in Figure 7.

It is also pertinent to note that these operations are hidden away from the user. With the use of the check token, the user library is able to repeatedly check for file system node availability. With this, processing logic is done to ensure seamless file system operations until the central server's asynchronous operation has completed.

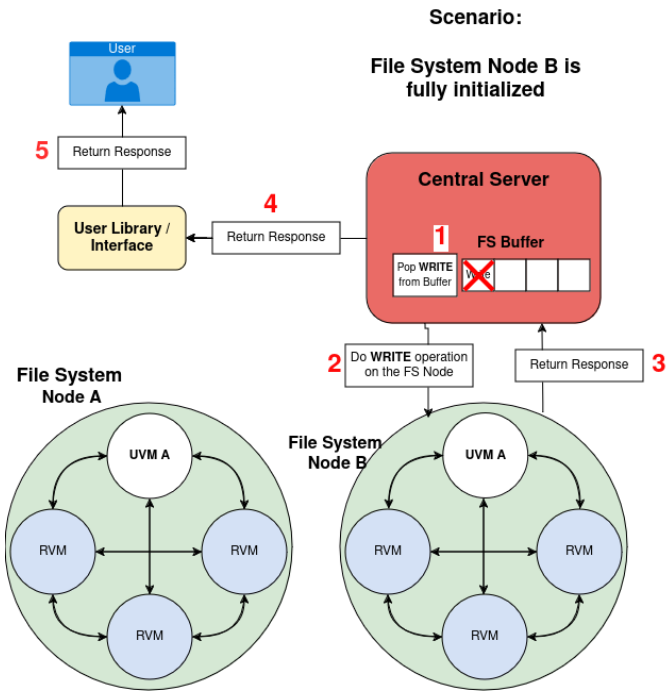

Fig. 7. Communication for Initialized File System Node

As described in the previous section, Figure 7 denotes a full initialization of the file system node. From this point on, the protocol continues to allow for the system to continue normal operation.

Firstly, the write operation that was placed on the buffer in the central server is popped. In translation, this would mean that we no longer need to store the file system operation.

Secondly, we do the write operation (the popped file system operation) on the **new file system node** (File System Node B). Once this operation completes, a response is sent from the file system node back to the central server, indicating the result of the operation.

Fourthly, the central server transmits the response back the the user library.

Lastly, the response is returned to the user.

C. Software Components

All communications take place as HTTP GET requests: including client requests to the middleware, the middleware's pinging of storage devices, and the storage devices' internal communications between their UVM and RVMs. This leads us to one of our future works: using Remote Procedure Calls (RPCs) for UVM/RVM and intra-RVM communications: such accomplishes all of the needs of the HTTP request without the overhead of TCP. However, for the sake of being able to deliver the project in a timely manner, our team settled on delegating all communication to TCP for simplicity.

With regards to data sharing, all data sent between machines is embedded in the route's URL: for example, to write contents "hello!" to file 'greeting.txt', the route would be: '/write/greeting.txt/hello!'. Synchronization of storage device subnetwork IP address files between RVMs and the UVM is done by the RVM leader, again by using HTTP GET requests.

In terms of how our distributed file system infrastructure itself is implemented, we execute everything on EC2 AWS Amazon Linux instances, with open TCP connections from any source. This allows us to quickly spin up new EC2s in development, something especially helpful given that our DFS' EC2 instance count scales quickly with the number of logical UVM/RVM storage devices (the demo alone took 16 machines!).

VII. System Demonstration

A. Success Scenario

Our system can be understood as having two classes of successful scenarios: successful file operations executed by the client library, as well as successful storage device allocation and integration by the middleware. For the sake of this example, we will use "write" in our screenshots as a proxy for the success of all six of our file operations (read, write, delete, copy, rename, and exists), though our recorded demo in the submitted presentation recording does show all six working.

With respect to the correctness of our file operations, figure 8 shows that an RVM (the topmost green terminal window) received a GET HTTP request from its UVM to execute the forwarded client write request (the topmost black terminal window). Figure 9 demonstrates that the file "file.txt" from Figure 8 was actually written to the system, as forwarded to a preallocated resource that stood in as a later RVM that became a UVM (the topmost blue terminal window).

In terms of storage allocation success, in Figure 10 that the client has yet to send a write operation for file "f", and that the bottom four blue terminal windows for pooled resources are still stalled, because they have not been allocated by the middleware yet. However, in Figure 11, we can see that the client write operation to file "f" has succeeded, and that the four blue terminal windows are active showing that they have been allocated as a new storage unit.

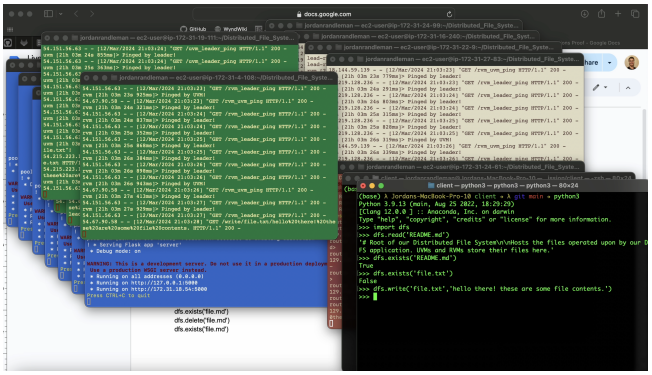

Fig. 8. Write Operation Routed to an RVM

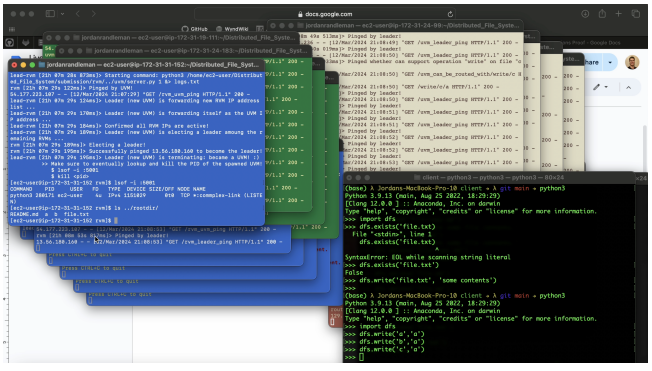

Fig. 9. Write Operation Result on an RVM

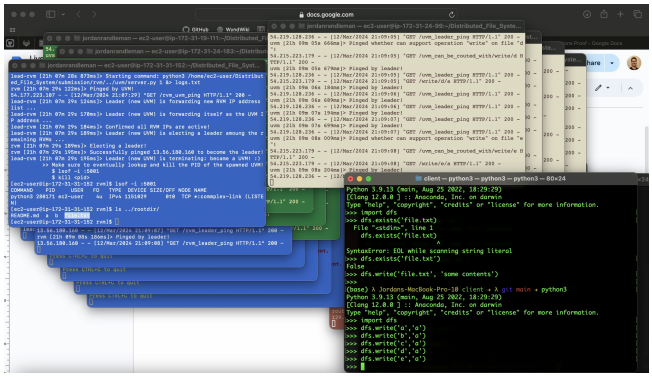

Fig. 10. UVM Prior Allocation

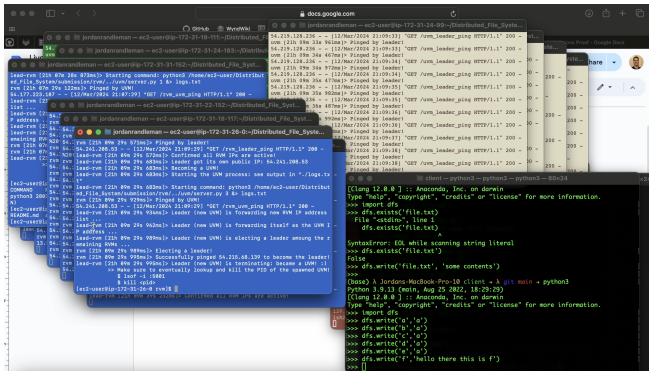

Fig. 11. UVM After Allocation

B. Failure Scenarios

Our system has three classes of failure scenarios, each of which corresponds to different types of downed resources. UVMs, RVMs, and lead RVMs can all fail at any given point, and the system has to be able to navigate allocating and integrating in an alternative machine, in order to return to a stable state. Here, we show that the system adjusts just fine to any of the given scenarios.

Figure 12 shows an RVM (top green window) prior failure, along with a pooled blue window resource on stand by in the top left. Figure 13 shows that RVM having been killed, with that blue pooled resource successfully having been allocated and integrated by that RVM's leader as a replacement RVM.

Figure 14 shows an RVM leader (top green window) prior failure, along with the blue window regular RVM in the top left. Figure 15 shows that lead RVM having been killed, with that blue RVM having successfully become the new lead RVM, with another pooled resource having been allocated and integrated (the second blue window down) to replace that RVM leader in the RVM subsystem.

Figure 16 shows a UVM (top green window) prior failure, along with the blue window lead RVM in the top left. Figure 17 shows that UVM having been killed, with that blue lead RVM having successfully become the new UVM, with another pooled resource having been allocated and integrated (the third blue window down) to replace that ascended RVM leader in the RVM subsystem.

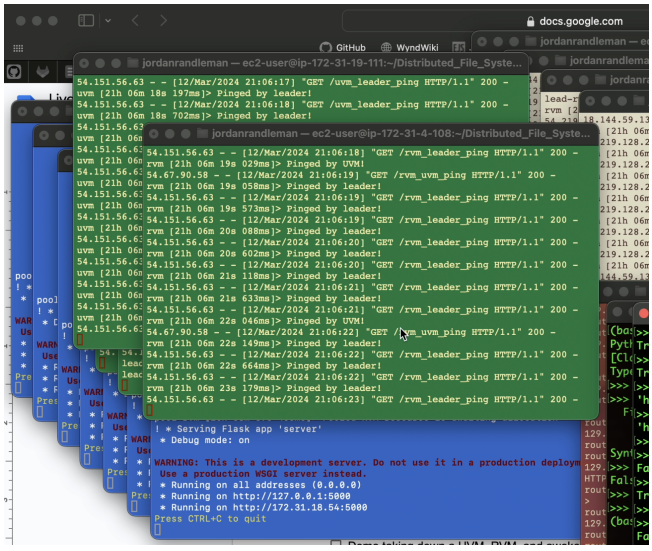

Fig. 12. RVM Prior Failure

VIII. Performance Analysis

The evaluation of performance for distributed file systems is a complex task that can involve a multitude of metrics. These metrics traditionally include throughput, characterized

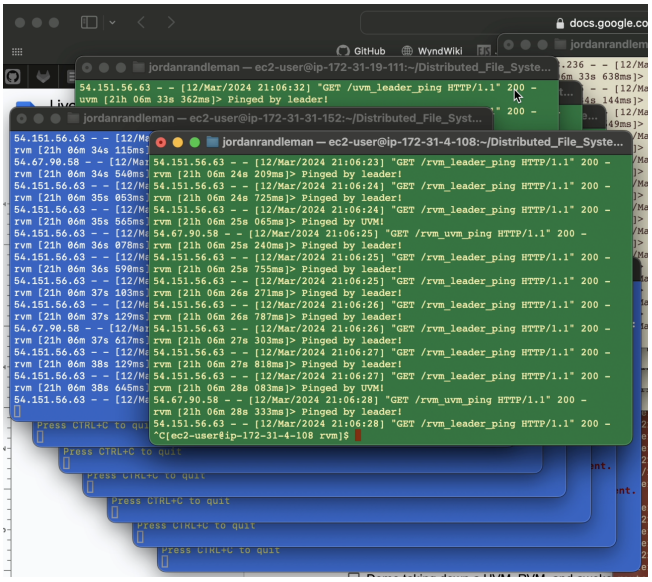

Fig. 13. RVM After Failure (reallocated!)

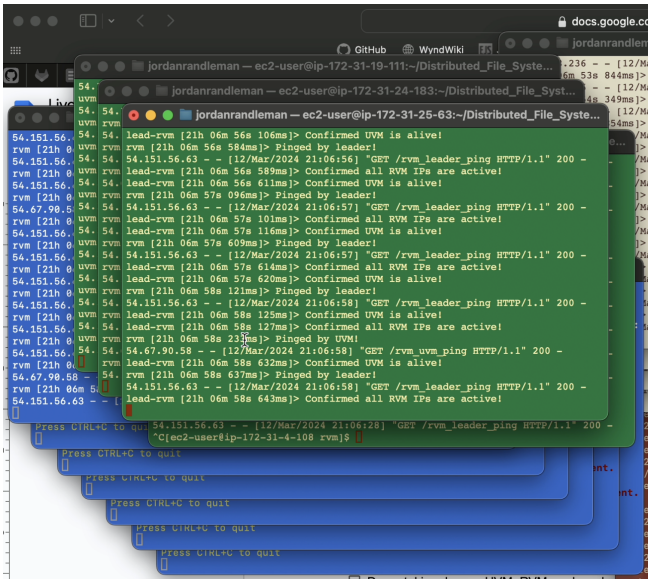

Fig. 14. RVM Leader Prior Failure

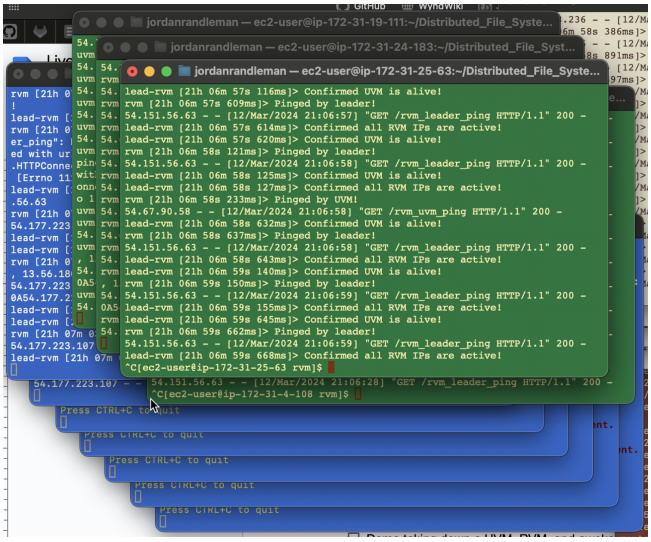

Fig. 15. RVM Leader After Failure (reelected!)

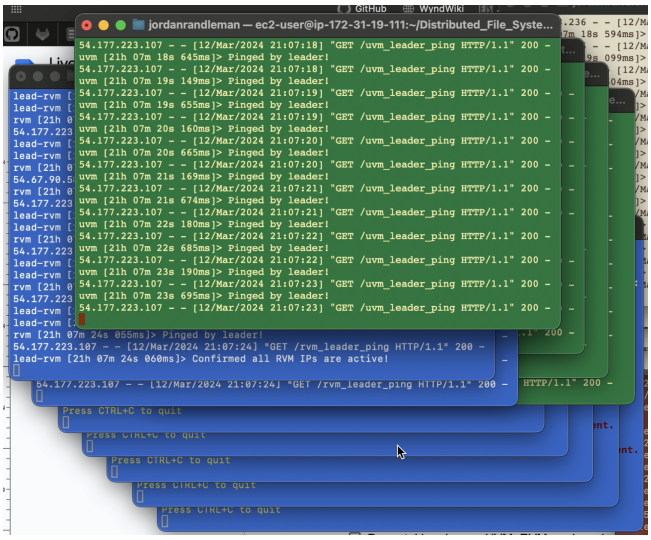

Fig. 16. UVM Prior Failure

by the quantity of data processed within a given time interval; latency, the temporal cost incurred to complete a single file system operation; IOPS, denoting the number of read/write operations achievable per second; concurrency, the system's ability for managing simultaneous operations. However, not all metrics are equally relevant or applicable to our distributed file system. So, in the context of our project, we opted to focus on a few specific metrics to measure performance.

For our distributed file system, we chose to measure performance using the average round-trip-time (RTT) for each supported file operation, as requested and received by the client library.

Round Trip Time encapsulates the following:

1) Network Latency: The delay caused by the transmission of data over the network.
2) Processing Time: The time the system takes to process the request and execute the operation.
3) Data Transfer Time: The time taken to actually transfer data between the client and the file system if the operation involves data movement.

This metric was selected because it provides a meaningful and user-centric insight into our system's responsiveness and performance, as well as because RTT is a straightforward measure to accumulate.

By measuring the average RTT for each file operation, including read, write, delete, copy, rename, and exists, we can evaluate the performance of our system by varying the number concurrent of clients interfacing with our DFS to simulate an increased system load. This approach is targeted to measure

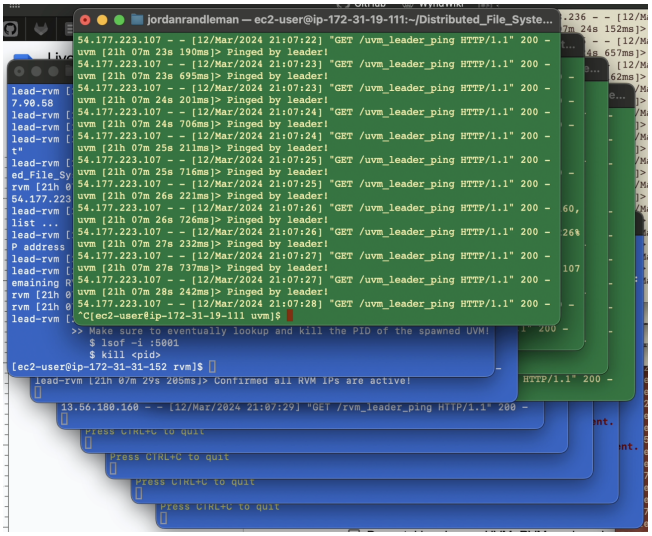

Fig. 17. UVM After Failure (replaced by leader!)

system scalability and concurrency, in effect seeing how well our system scales to its number of clients.

Fault tolerance is another crucial metric that we measure in order to evaluate the performance and reliability of our proposed distributed file system. Fault tolerance is a critical aspect of distributed systems because said systems are designed to operate in environments where individual components may experience failures. Our distributed file system employs a combination of UVMs and RVM pools to achieve fault tolerance by ensuring continuous operation in the event of failures.

To assess the fault tolerance capabilities of our system, we conducted a series of experiments focused on timing storage device allocation, UVM failure recovery, RVM failure recovery, and RVM leader election. These four scenarios represent conditions under which new virtual machines need to be allocated and integrated into the network, thereby making them performance bottlenecks that need to be accounted for.

IX. Testing Results

A. File Operations

For our file operation measurements, the Python time module was used to precisely record the milliseconds taken for each file operation. To ensure the accuracy and reliability of the results, each metric was calculated over an average of 10 complete runs. This means that each file operation was executed 10 times per client, and the time taken for each execution was recorded. The recorded times were then averaged to obtain a single metric for each file operation. So, if we have ten clients testing the read operation, each client will measure its read RTT 10 times, causing 100 reads to occur.

The results of our measurements are seen in Figures 18 and 19.

From a high level point of view, these results show that the distributed file system performs well in most of the operations over several clients, indicating that our DFS could be an

Metrics: File Operations by Concurrent Client Count

- *Source: metrics.py*

File Operation	1 Client	5 Clients	10 Clients
read:	26.167ms	30.296ms	63.979ms
write:	43.131ms	50.776ms	83.777ms
delete:	41.901ms	46.6ms	84.565ms
copy:	42.146ms	46.168ms	91.063ms
rename:	43.987ms	47.281ms	77.709ms
exists:	28.763ms	30.63ms	61.922ms

Fig. 18. File Operation Metrics

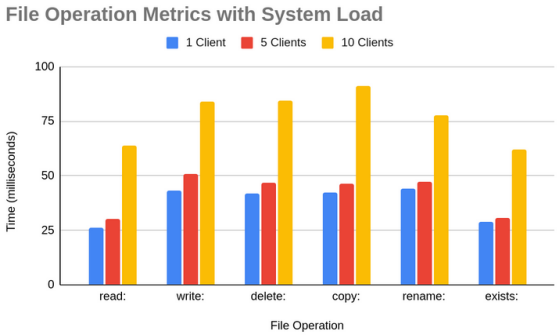

Fig. 19. File Operations Visualized

increasingly practical implementation with tightened intra-VM communication.

In addition, our system scales very well according to the system load in comparison to a traditional file system that does not support concurrency. In a traditional synchronous file system, we expect to see a linear increase in file system operations.

All of the file system operations in our system exhibit relatively good scaling performance. As a baseline, the latencies of read and write operations are either around 30ms or 40ms respectively when tested with a single client. As we increase the number of clients performing concurrent operations, we expect to see some increase in average response times due to coordination overhead. However, the results show that this increase is sub-linear and quite modest.

Most significantly seen when comparing the performance of one client to five clients, each additional client adds only a small amount of latency towards the overall operation time. Although we do see larger differences at ten clients compared to one client, the increase is still sub-linear and lower than a traditional synchronous file system. This demonstrates that our system can effectively utilize the parallelism offered by multiple storage nodes to share the load and maintain good performance scalability.

One possible reason for the greater latency at 10 clients could be that the middleware is receiving too many requests at a single time, creating a centralized bottleneck, however,

the sheer amount of disk I/O being done when testing 10 clients (e.g. 100 back-to-back writes) likely also accounts for the delays faced! In order to mitigate these stalled concurrent file operations, our future work could include improving how the middleware routes client file operations to our UVMs: currently, the middleware just pings the first UVM that can support the file operation. If we instead used a Distributed Hash Table scheme to store the files, we could improve our DFS' load balancing to parallelized more file operations (eliminating pressure on each storage device's disk).

However, it is also important to note that for all file system operations, inherent system latency overhead exists for aspects like data distribution, replication, and consistency management. While these latencies can be optimized by conducting more fine-tuned testing on precisely how small VM communication timeouts can be safely set (e.g. how long each machine allows for delays from other nodes in the network), these delays are fundamentally baked into our algorithm's distributed nature.

B. Resource Allocations

The results of timing resource allocations (as done by new UVM allocations and storage device failures) are summarized below:

1) UVM Allocation and Recovery: This measurement represents the time required for the middleware component of our distributed file system to allocate and provision a new UVM instance, along with its associated pool of RVMs, from the pool of preallocated VMs. This process includes spawning and awakening a pool of RVMs, waiting for them to elect an RVM leader, having that RVM leader notice that the UVM is missing, then having that RVM leader finally escalate its own privileges to become the subnetwork's new stand-in UVM.

UVM allocation by middleware: 8.285 seconds

Efficient UVM allocation is essential for ensuring timely recovery from exceeded file system capacities, as well as to maintain system availability. Thankfully, this operation is relatively rare (how often is the entire file system's capacity going to be exceeded?), and hence does not impact the vast majority of user experiences with our DFS.

2) RVM Failover and Recovery: In the event of an RVM failure, its leader will detect that the machine has gone down, and then automatically launch a new RVM instance to replace the failed device.

Time to launch a new RVM upon failure: 0.523 seconds

The measured time of 0.523 seconds indicates the responsiveness of our system in detecting and recovering from RVM failures, minimizing potential downtime. Note, however, that this is largely because we support a 0.5 second timeout for RVM leaders to detect that one of their RVMs have gone down. Hence, the actual process of integrating freshly allocated RVMs is very quick!

In addition to replacing failed RVM instances, our system also ensures the availability of an RVM leader, which is responsible for coordinating and managing the pool of RVM devices.

Time to elect and start a new RVM leader upon failure: 3.259 seconds

The measured time of 3.259 seconds represents the duration required to elect and launch a new RVM leader in the event of a leader failure. Like our "downed RVM replacement" measurement, this time is largely bounded by a 3 second timeout that we set for RVMs to detect that their leader is down (and hence the leader election protocol should be restarted). The actual process for electing the leader upon detection is significantly faster!

When a UVM fails, the RVM leader is responsible for detecting the failure and initiating the UVM's replacement process.

Time for an RVM leader to replace a failed UVM: 0.943 seconds

The measured time of 0.943 seconds indicates the efficiency of our system in identifying and recovering from UVM failures, ensuring minimal disruption to user operations. Our system currently has a 0.5 second timeout set in order for RVM leaders to detect that their UVM has gone down, meaning that the actual time to escalate that RVM's privileges to a UVM (and to have that RVM replaced by a newly allocated device) is about 0.443 seconds.

Overall, the fault tolerance and availability of our system is very reasonable. The system ensures that nodes are replaced quickly and correctly. However, it is also pertinent to note that the most expensive operations are UVM allocation and leadership election, though the former can be managed by boto3, and the later depends on some tunable system timeout hyperparameters.

X. Conclusion

A. Future Work

In the realm of distributed file systems, enhancing performance, functionality, and security is an ongoing process. Our system, while functional, has areas that can be enhanced from improvements. Some key areas that we selected to improve were access control and security, storage management, data placement and routing algorithms, middleware recovery, and remote-procedure-calls for intra-node communication. Many of these improvements (apart from access control and security) would enable faster and more efficient performance of our distributed file system.

A foundational aspect of any distributed file system is ensuring data security and controlling access. Our current model, which lacks fine-grained access control, exposes all files to all users. This approach, while simplistic, fails to meet the security requirements of a production environment. To address these concerns, we propose to implement encryption both in transit and at rest. This ensures that data confidentiality is maintained, protecting against both external breaches and insider threats. Utilizing strong, industry-standard encryption algorithms (e.g., AES-256) will help in safeguarding data against unauthorized access. Additionally, we seek to strengthen user authentication and refine authorization mechanisms. We can adopt secure, robust protocols for authentication to verify

user identities. Furthermore, implementing a more nuanced authorization scheme, such as Access Control Lists (ACLs), user groups, and a file permission matrix, will enable us to define and enforce precise access controls based on user roles and responsibilities, significantly enhancing our system's overall security posture.

Our current system sets capacity limits based on the number of files, which may not be the most efficient approach to managing storage resources. Altering this strategy to focus on percentage storage usage offers several advantages such as flexibility in storage limits and adaptability of the storage devices. By tracking and limiting storage based on usage percentages, we can more effectively manage available space, ensuring a fair distribution of resources among users and preventing any single user from monopolizing disk space. This allows for flexibility in our system. Additionally, this approach allows the system to dynamically adjust and adapt as storage capabilities expand or contract, providing a more scalable and responsive storage management solution.

Optimizing data placement and routing can significantly enhance performance and reliability. The introduction of more sophisticated data placement strategies, such as those that leverage caching mechanisms or incorporate complex logic based on access patterns, can reduce latency and improve system performance. By intelligently placing data closer to where it is frequently accessed, we can minimize bottlenecks and speed up data retrieval. Revamping our routing algorithms to better support the enhanced data placement strategy will further optimize data access. By designing algorithms that consider factors such as network congestion, node availability, and data locality, we can ensure more efficient and reliable data routing within the system.

Addressing the issue of resilience, it is critical to implement comprehensive recovery mechanisms to handle failures at the middleware level as it currently stands as our single point of failure. Some ways to implement this is through using strategies for redundancy, data replication, and failover processes to mitigate the impact of this single point of failure and ensure system availability and data integrity. For example, it could be possible to implement a redundancy mechanism with a structure that is similar to the ring structure to provide greater fault tolerance for our middleware. In the case of a failure, another node in that ring would be elected to be the new middleware (central server).

To further improve system performance, we can adopt the use of remote-procedure-calls (RPCs) over the use of HTTP requests. RPCs, being lighter and more efficient for internal communications, can reduce overhead, lower latencies, and ultimately enhance the overall throughput of the system. Furthermore, these improvements enhance user experience and satisfaction.

Through strategic enhancements in access control and security, storage management, data placement and routing, middleware recovery, and intra-node communication, we can significantly advance the capabilities and performance of our distributed file system. Implementing these improvements will not only bolster system security and efficiency but also provide a more robust, scalable, and user-friendly platform for managing and accessing distributed data.

B. Impact

In this paper, we have presented the design and implementation of a robust distributed file system that addresses the key challenges inherent in distributed computing. The primary objectives of our DFS include enabling seamless concurrent file operations within a concurrent client context via idempotent file operations, ensuring scalability to handle growing data volumes, and fortifying the system with fault-tolerance mechanisms to maintain high availability.

To allow users to interact with our system, we have created a simple Python client library that acts as the user interface. This library supports idempotent file operations such as read, write, delete, copy, rename, and exists. The communication between this library and the distributed file system nodes would be facilitated by a central router, which we call our middleware. The middleware has the important job of both routing file operation requests to the corresponding File System Node, and transmitting its response back to the user. Additionally, the middleware would be in charge of handling a machine resource pool, which denotes machines that are waiting to be used in the case of File System Node failure.

Central to our DFS architecture are the File System Nodes. Under the hood, a File System Node consists of User Virtual Machines (UVMs) and Replacement Virtual Machines (RVMs), which work in tandem to provide fault tolerance and replication. In the case of a UVM failure, the leader election protocol dynamically elects a leader among the RVMs, which in turn broadcasts with the other RVMs in the file node subnetwork to monitor the that subnetwork's health. That system's health also includes whether the UVM is active: having thereby been detected as down, the RVM leader will then swap itself in as the system's new UVM, and allocate a replacement RVM to take its own place. Alternatively, on a RVM failure, the leader requests an additional machine from the middleware to replace the failed RVM. This protocol is crucial for maintaining the system's availability, as it allows for the swift recovery and replacement of failed UVMs with healthy RVMs, as well as replacing failed RVMs, minimizing disruptions to file system operations.

The resource allocation strategy leveraged in our DFS is designed to efficiently utilize the capabilities of any machine hardware. By limiting the number of files per UVM, we have instituted a strategic trigger for horizontal scaling actions. When the file capacity threshold is reached, the system automatically provisions additional UVMs to accommodate the growing storage needs, demonstrating its horizontal scalability and adaptability to changing data requirements.

The concurrency control mechanisms implemented in our DFS are crucial for maintaining data consistency and integrity, allowing multiple users to perform idempotent file operations simultaneously without adverse effects on the system's consistency. The synchronization of resource access is achieved

through a robust locking mechanism that ensures atomic operations on shared data structures, preventing race conditions and data corruption.

Through a comprehensive evaluation, we have demonstrated the system's ability to withstand failures, scale effectively, and preserve data consistency. The fault-tolerance capabilities are highlighted by the seamless replacement of failed UVMs with RVMs, ensuring continuous service and data availability. The scalability of the system is evident in its capacity to both support a large number of concurrent users, as well as by dynamically provisioning new storage resources as needed, accommodating the growing file storage requirements of client needs.

In conclusion, the success of our distributed file system represents another lightweight solution towards a scalable, fault-tolerant, and concurrency-aware system for managing files across a distributed network.

REFERENCES

[1] Ceri, Stefano, Barbara Pernici and Gio Wiederhold. "Distributed database design methodologies." Proceedings of the IEEE 75 (1987): 533-546. https://ieeexplore.ieee.org/abstract/document/1458038.

[2] S. Sanghavi, B. Hajek and L. Massoulie, "Gossiping With Multiple Messages," in IEEE Transactions on Information Theory, vol. 53, no. 12, pp. 4640-4654, Dec. 2007, doi: 10.1109/TIT.2007.909171. https://ieeexplore.ieee.org/abstract/document/4385780.

[3] Peter Macko and Jason Hennessey. 2022. Survey of Distributed File System Design Choices. ACM Trans. Storage 18, 1, Article 4 (February 2022), 34 pages. https://doi.org/10.1145/3465405.

[4] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. 2003. Ivy: a read/write peer-to-peer file system. SIGOPS Oper. Syst. Rev. 36, SI (Winter 2002), 31–44. https://doi.org/10.1145/844128.844132.

[5] Sanjay Ghemawat, , Howard Gobioff, and Shun-Tak Leung. "The Google File System." . In Proceedings of the 19th ACM Symposium on Operating Systems Principles (pp. 20–43).2003. https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf.

[6] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. Frangipani: a scalable distributed file system. SIGOPS Oper. Syst. Rev. 31, 5 (Dec. 1997), 224–237. https://doi.org/10.1145/269005.266694.

[7] Distributed File System Course Project Document. https://www.andrew.cmu.edu/course/14-736-s20/applications/labs/proj3/proj3.pdf.

[8] Managing Amazon EC2 instances. https://boto3.amazonaws.com/v1/documentation/api/latest/guide/ec2-example-managing-instances.html.