Distributed File System

Abstract- *The advent of cloud storage and the increase in importance of highly available, reliable, storage systems are based on distributed file systems. The popularity of consumer devices demands applications that can store files in a distributed environment for a user. The depreciation of monolithic system architecture and central storage systems have led to various attempts at developing distributed file systems. We have implemented a distributed file system in Java, deployed in AWS that provides a reliable and scalable solution.*

I. GOALS AND MOTIVATION

The goal is to develop a distributed file system to present to the user an efficient and coherent solution to store and access files in a distributed environment.
In a centralized system, the data source may be compromised and this may lead to loss of sensitive information. A distributed file system ensures that users can access and store files across multiple environments providing high availability and security. A distributed file system stores the files at multiple locations to provide high availability. Files can be accessed by multiple users across a network irrespective of their location. The invasion of multiple devices demands a secure and reliable file storage system that can help users access files across multiple devices.

II. DISTRIBUTED SYSTEM CHALLENGES

Our Distributed file system demonstrates the following features:

Security -

a. AWS Application load balancer-

We hide the individual server instances by deploying them in AWS under a private subnet.User can only access the public subnet AWS application load balancer URL. The load balancer helps to distribute the load across the different coordinator EC2 instances.

b. User authorization-

The coordinator does not allow unauthorized users to access the application.
A user can upload and perform operations on the file only if he has a valid session in the application.

a. signup- Coordinator creates a record for the user in the application
b. login- Coordinator creates a valid session for an existing user.
c. logout- Coordinator terminates the session of the user.

Resilience - The co-ordinator implements a retry mechanism to ensure that it sends requests to the metadata and document server even in the case their servers are unavailable. This helps the application to be available to the user even if the downstream servers are temporarily unavailable.

Atomicity - The Coordinator ensures all transactions are atomic.In case of upload,if the document server fails to upload the data, the co-ordinator calls the metadata server to delete the file metadata.

Persistent Session - If the Co-ordinator server is unavailable, an already logged in user will be able to access the application once the server is available again.
The unavailability of the server does not invalidate an existing session.

III. Related Work

A popular distributed file system is Transarc's Andrew File System (AFS), the precursor of the Open Software Foundation's (OSF) Distributed File Service (DFS), part of OSF's Distributed Computing Environment.
Examples of successful distributed file systems in the world of personal computers are AppleShare for Macintosh computers and LAN Manager and Novell Netware for IBM-compatible PCs. In the Unix world, a popular reference is Sun Microsystem’s Network File System (NFS). Also, Apollo Computer’s Domain OS file system is quite popular.Other Unix distributed file systems include AT&T’ s RFS, Sprite, and experimental systems such as Amoeba. Several other distributed file storage systems have been developed like Ceph,GFS, Lustre, Hadoop.
NFS was the first modern network file system. It began as an experimental file system developed in-house at Sun Microsystems in the early 1980s. As a standard, NFS grew quickly because of its ability to interoperate with other clients and servers.NFS follows the client-server model of computing. The server implements the shared file system and storage to which clients attach. The clients implement the user interface to the shared file system, mounted within the client's local file space.

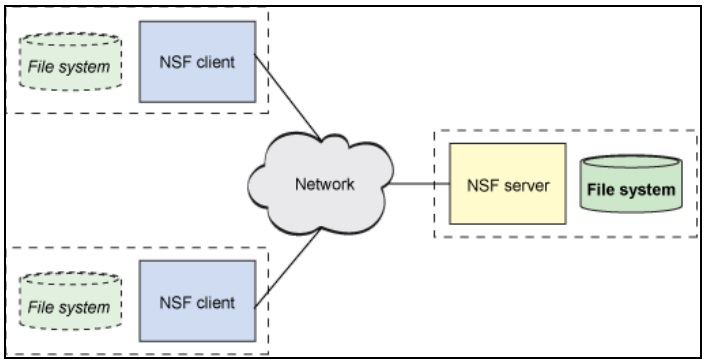

Figure 1. The client-server architecture of NFS

Apollo Domain File System-

Domain provides support for the distribution of typed files via an Object Storage System.. A system-wide Single Level Store (SLS) that provides a mapped virtual-memory interface to objects is built on top of the OSS. The domain distributed file system is layered on the SLS and presents a Unix-like file interface to application programs. A facility called the Open Systems Toolkit uses the file typing mechanism of the OSS to create an extensible I/O system. Users can write non-kernel code to interpret I/O operations. When a file is opened its type is determined and the code implementing I/O operations on that type of object is dynamically loaded by the system.

Andrew File System-

Andrew is a distributed workstation environment that has been under development at Carnegie Mellon University since 1983. It combines the rich user interface characteristic of personal computing with the data sharing simplicity of timesharing. The primary data-sharing mechanism is a distributed file system spanning all the workstations. Using a set of trusted servers, collectively called Vice, the Andrew file system presents a homogeneous, location transparent file name space to workstations. Clients and servers both run the 4.3 BSD version of Unix. It is a relatively heavyweight operation to configure a machine as an Andrew server. This is in contrast to systems such as Sun NFS, where it is trivial for any machine to export a subset of its local file system.

Hadoop-

Hadoop provides a distributed file system using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many hosts, and executing application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers. Many organizations worldwide report using Hadoop.

QFS -

QFS is a dedicated distributed file system designed for the large and medium-sized websites (or intranet), just like GFS.It provides the application-level rather than system-level distributed file storage service. Its purpose is to use cheap commercial equipment to compose mass data storage systems to provide reliable and available data storage especially for small files such as pictures for large and medium-sized websites, enterprises. In a traditional GFS-like system, the system becomes very complicated because it considers the problem of large file storage, file block storage. Nowadays, large and medium-sized websites' huge amounts of small files ask for a more concise and efficient distributed storage system.

OpenStack Swift is another file system in the field of cloud storage. HDFS,GFS are designed for parallel processing of large files and streaming access-based data is not suitable for persistent storage of massive small files.

IV. Project Design

a. Design Goals

1. Heterogeneity

Our distributed file system is designed as a master/slave architecture with a metadata server and multiple document servers. The system uses Spring Boot framework and RESTful APIs, allowing clients to access the system from different platforms on different hardwares.

2. Openness

Our system is designed to be open and extensible, supporting different file formats and storage devices, while maintaining compatibility with existing components.

3. Failure Handling

To handle failures, the metadata server does periodic health checks on all document servers. If a document server becomes unavailable, the system removes that document server and retrieves chunks from other document servers that store replicas.

4. Scalability

To ensure scalability, the system uses load balancing strategies and resource allocation algorithms.

The algorithm examines the available storage space on each chunk server in the network and assigns new chunks to the chunk servers with the largest amount of available space. This ensures that each chunk server can accommodate the new data without becoming overloaded.

5. Quality of Service
Our distributed file system project is designed with QoS in mind to provide a consistent level of service to all clients, regardless of their location or network conditions. One way we achieve QoS is by using chunking, which allows for concurrent saving and retrieving of chunks, thereby increasing speed.

6. Transparency
Our system provides a transparent interface to users, hiding the complexity of distributed systems behind a simple and intuitive file system interface. Users are not required to be aware of the underlying architecture or the location of the data they are accessing.
Overall, our distributed file system project is designed with key technical features to handle the core challenges of distributed systems. By using a master/slave architecture, Spring Boot framework, RESTful APIs, chunking, redundant storage, locking and caching mechanisms, load balancing, and recursive replication/deletion, our system is well-equipped to handle the challenges of heterogeneity, openness, failure handling, quality of service, scalability, and transparency.

b. Components

Metadata Server: The metadata server works as the master node. It manages the metadata associated with files in the system. It keeps track of which document servers hold which chunks of data. The metadata server also performs periodic health checks on all document servers to ensure that they are available and functioning correctly.

Document Servers: Document servers store the actual data of files in the system. They are responsible for storing and retrieving chunks of data, as well as performing recursive replication and deletion to ensure data availability and integrity. They work in a master-slave architecture, where the leader takes the file and performs chunking or merging as needed. It then ensures task completion by calling worker APIs to perform operations. The worker nodes operate at the chunk level and provide APIs for uploading, retrieving, or deleting chunks.

Coordinator (Client Server): Client Server serves as the application interface that can be accessed by the user. The co-ordinator acts as an orchestrator between the metadata server and the document server.The coordinator performs the following actions:
Signup - Create a record for a new user in the application.
Login - Co-ordinator creates a valid session for an existing user.
Logout - Co-ordinator terminates the session of the user.
Upload - Users can upload a file in DFS.
Download - Users can download a file from DFS.
Delete - User can delete an existing file from the application.

c. Algorithms

1) Chunking
Files are broken into chunks between 64KB and 10MB to allow for efficient transfer of data and parallelism in saving and retrieving chunks.

2) Replica Management
Each chunk is replicated across 3 document servers to ensure data availability in the event of a failure. The metadata server assigns chunks to document servers with the largest amount of available space to balance the load across the system.

3) Recursive Replication/Deletion
Document servers automatically replicate chunks to other document servers to ensure redundancy and data availability. They also perform recursive deletion of chunks when files are deleted to free up storage space and maintain data integrity.

4) Resource Allocation Algorithm
To distribute data across multiple storage nodes in the network, our system uses a resource allocation algorithm. When a new file is uploaded, it is divided into smaller "chunks" of data. The algorithm then examines the available storage space on each document server in the network and assigns chunks to the document servers with the largest amount of available space. This ensures that each document server can accommodate the new data without becoming overloaded. Additionally, the algorithm balances the load across the network to prevent any single document server from becoming a bottleneck.

5) Resource Discovery
Our system also includes a resource discovery mechanism to keep track of available resources (i.e., storage nodes) in the network. When a new document server comes online, it informs the metadata server about its presence and the amount of available storage space. The metadata server updates its records and assigns chunks to the new document server as appropriate. This allows the system to dynamically adjust to changes in the network and maintain high availability and reliability.

6) Health Checks

The metadata server performs periodic health checks on all document servers to ensure that they are available and functioning correctly. If a document server is found to be unavailable, the metadata server updates its records and reassigns its chunks to other available document servers.

d. Architecture

Our distributed file system uses a master-slave architecture with a metadata server and multiple document servers. Servers communicate with each other using RESTful APIs.

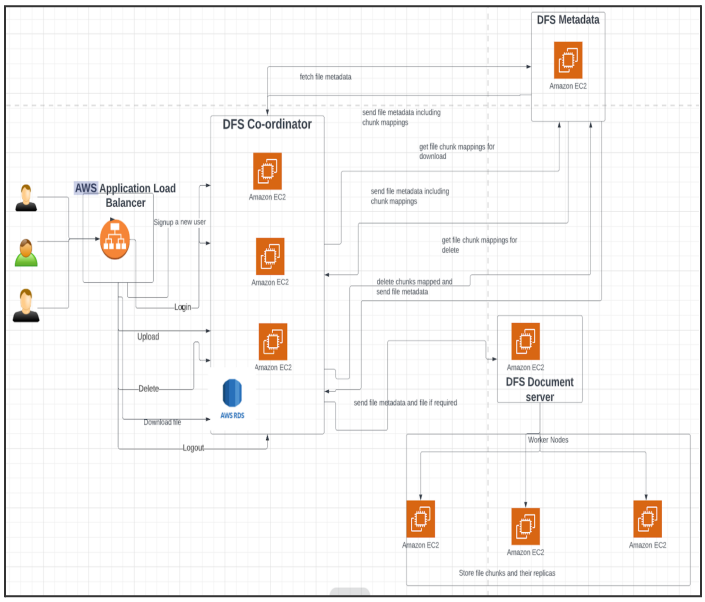

Figure 2. Application architecture

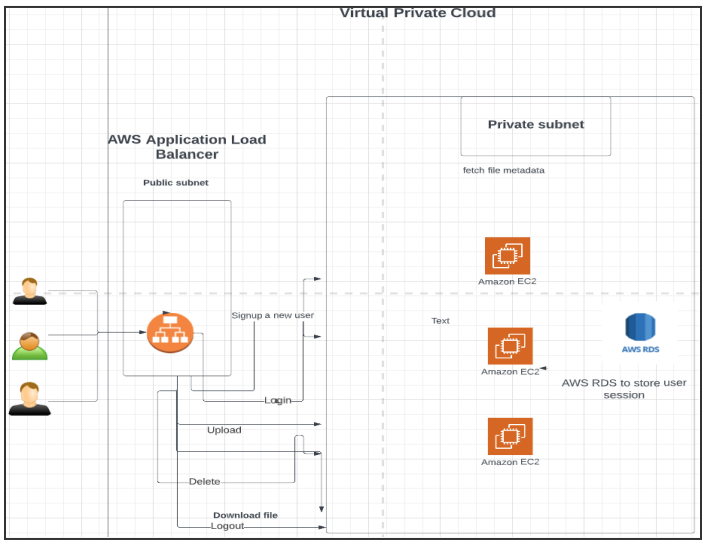

Figure 3. Coordinator architecture

The following is a workflow introduction of the main functions

When a client sends a request to upload a file, the metadata server receives the file name and size and generates a unique file ID, chunk size, number of chunks, and chunk IDs. The metadata server then selects three worker nodes with the most available storage space for each chunk, using the mapping between chunk ID and worker node list. The metadata server then replies with the file ID, chunk size, and mapping.

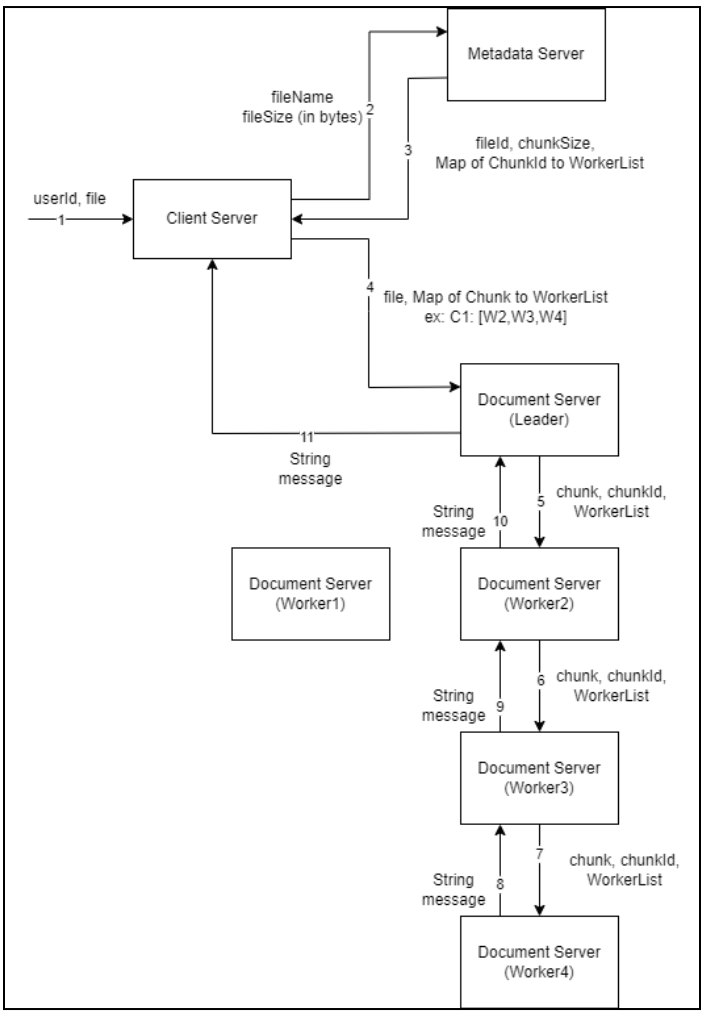

Figure 4. Upload use case diagram

Once the metadata has been generated, the client server sends the file to the document server. The leader document server takes the file, performs the chunking or merging as needed, and then ensures task completion by calling worker APIs to perform operations at the chunk level. The worker nodes operating at the chunk level are responsible for storing and retrieving chunks of data. They perform

recursive replication and deletion to ensure data availability and integrity. This means that each worker node replicates data and calls the next worker in line until the recursion is complete and each worker has completed successfully.

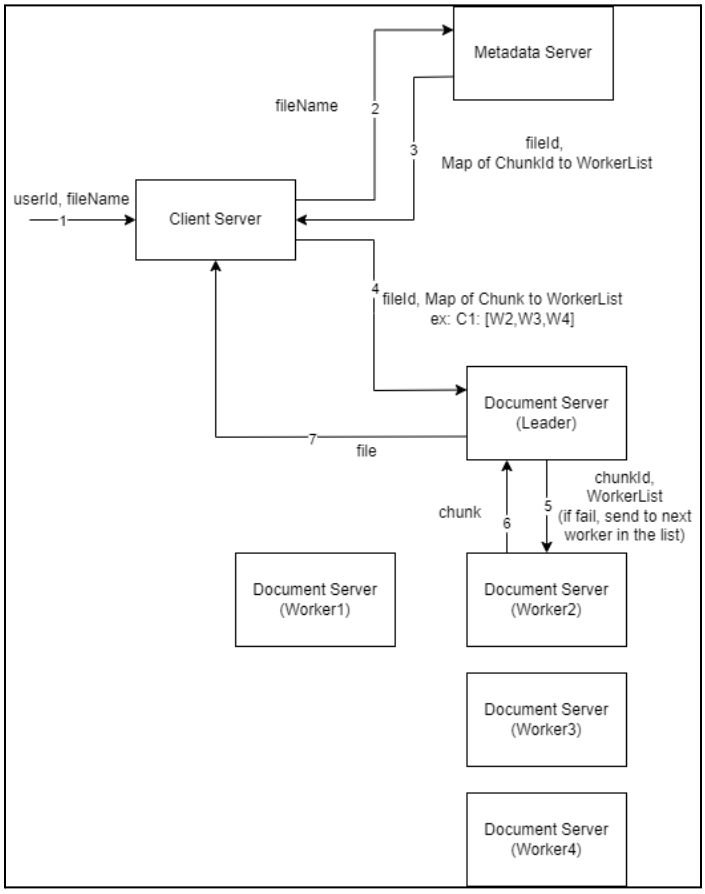

Figure 5. Download use case diagram

When a client requests to download a file, it sends a request to the metadata server containing the file name. The metadata server searches for the file based on the file name and retrieves the file ID and mapping between chunk and worker nodes. It then sends the file ID and mapping back to the client.

Once the client has the file ID and mapping, it sends requests to the document servers to retrieve the chunks of the file from the worker nodes specified in the mapping. The document server acting as the leader retrieves the chunks from the worker nodes specified in the mapping and reassembles the file from the chunks. Once the file has been assembled, it is sent back to the client.

The document servers ensure that the file is correctly downloaded by verifying the integrity of each chunk received from the worker nodes. The worker nodes, on the other hand, are responsible for retrieving the chunks of data requested by the document server and sending them to the client.

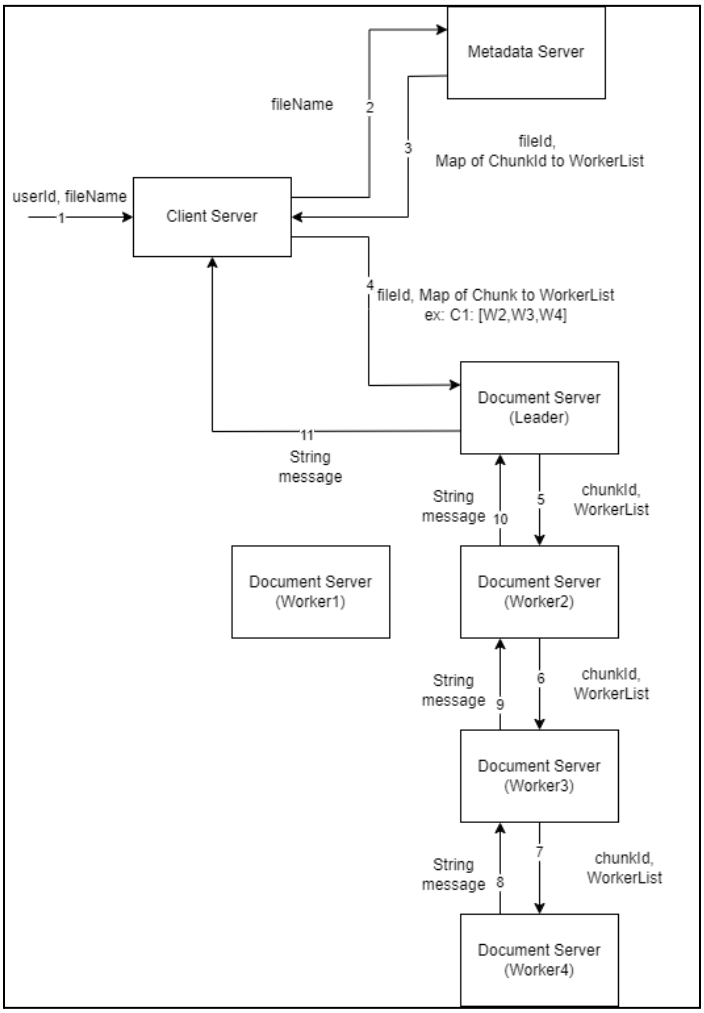

Figure 6. Delete use case diagram

When a client requests to delete a file, it sends a request to the metadata server containing the file name. The metadata server searches for the file based on the file name and retrieves the file ID and mapping between chunks and worker nodes. It then sends the file ID and mapping back to the client.

Once the client has the file ID and mapping, it sends requests to the document servers to delete the chunks of the file from the worker nodes specified in the mapping. The document server acting as the leader sends requests to

the worker nodes to delete the chunks of data. Once all the chunks have been successfully deleted from the worker nodes, the metadata server removes the file info and chunk info from all maps.

The metadata server is responsible for keeping track of the mapping between files, chunks, and worker nodes. It removes the file and its chunks from all maps to ensure that they are no longer accessible to clients. The document servers, on the other hand, are responsible for ensuring that the file is correctly deleted by verifying that all the chunks have been successfully deleted from the worker nodes. The worker nodes, in turn, are responsible for deleting the chunks of data requested by the document server.

V. Evaluation

The selected approach for our distributed file system is a good choice for several reasons. The system uses Spring Boot framework and RESTful APIs, which allow for easy access to the system from different platforms on different hardwares. It is also simple to add more storage nodes.

The algorithms used in our DFS are correct and efficient. The document servers store the actual data of files in the system and perform chunking or merging as needed. The chunking algorithm used in the document servers has a time complexity of O(N), where N is the file size. This algorithm ensures that files are divided into manageable pieces, allowing for faster and more efficient data transfer.

The metadata server uses a sorting algorithm with an average time complexity of O(NlogN), where N is the number of worker nodes, to select worker nodes with the most available storage space. It ensures that chunks are distributed evenly across the system, improving the system's fault tolerance, performance, and scalability. However, if there are a large number of worker nodes, a more efficient algorithm would be needed.

The following issues are addressed in our project:

i. Fault Tolerance: Our system is fault-tolerant, as it uses recursive replication and deletion to ensure data availability and integrity. In case a worker node becomes unresponsive, it is removed from the node map and chunk map, and its tasks are reassigned to other available worker nodes.

ii. Performance: Our system is designed for high performance, with chunking and merging algorithms that minimize data transfer and reduce latency. The system uses RESTful APIs, which are lightweight and fast, and the metadata server algorithm for selecting worker nodes with the most available storage space ensures that data is stored in the most efficient way possible.

iii. Scalability: Our system is highly scalable, as it can handle a large number of nodes and data storage without compromising performance. The recursive replication and deletion used in our system allows for easy addition and removal of worker nodes.

iv. Consistency: Our system ensures consistency by maintaining accurate metadata about files, chunks, and worker nodes. The metadata server algorithm for selecting worker nodes ensures that chunks are distributed evenly across the system, minimizing the risk of data inconsistencies.

v. Concurrency: Our system handles concurrency by assigning tasks to multiple worker nodes simultaneously, allowing for efficient data transfer and processing.

vi. Security: Our system uses RESTful APIs, which provide secure data transfer between client and server. Additionally, our system uses authentication to ensure that only authorized users have access to files.

VI. Implementation Details

Our implementation of the distributed file system is based on a master/slave architecture, with a metadata server and multiple document servers. The system is implemented using the Java programming language and the Spring Boot framework, which provides a robust and scalable platform for developing and deploying web applications. We have chosen to package our system as a .jar file, which can be easily deployed in different environments.

The system is built using a microservices architecture, where each service is responsible for a specific functionality.

The communication between servers is handled through RESTful APIs, which are lightweight and fast. The system uses JSON format for data exchange, and the communication is secured using HTTPS. The system is also designed to handle concurrency, as multiple client requests can be processed simultaneously.

These class UML diagrams illustrate the main classes and their key methods for the client server, metadata server, and document server of our distributed file system. The diagrams have been simplified for ease of viewing, and only the most important classes are included.

Client Server:

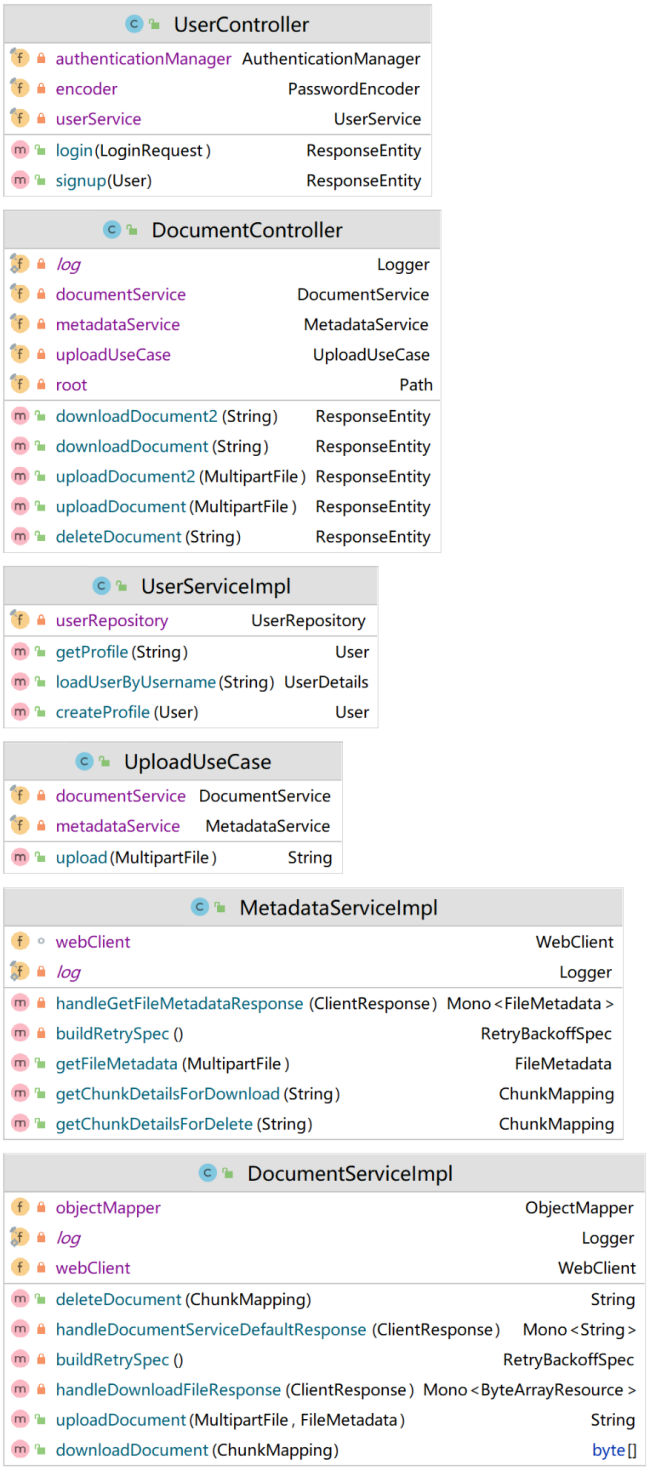

Figure 7. Client server class diagram

Metadata Server:

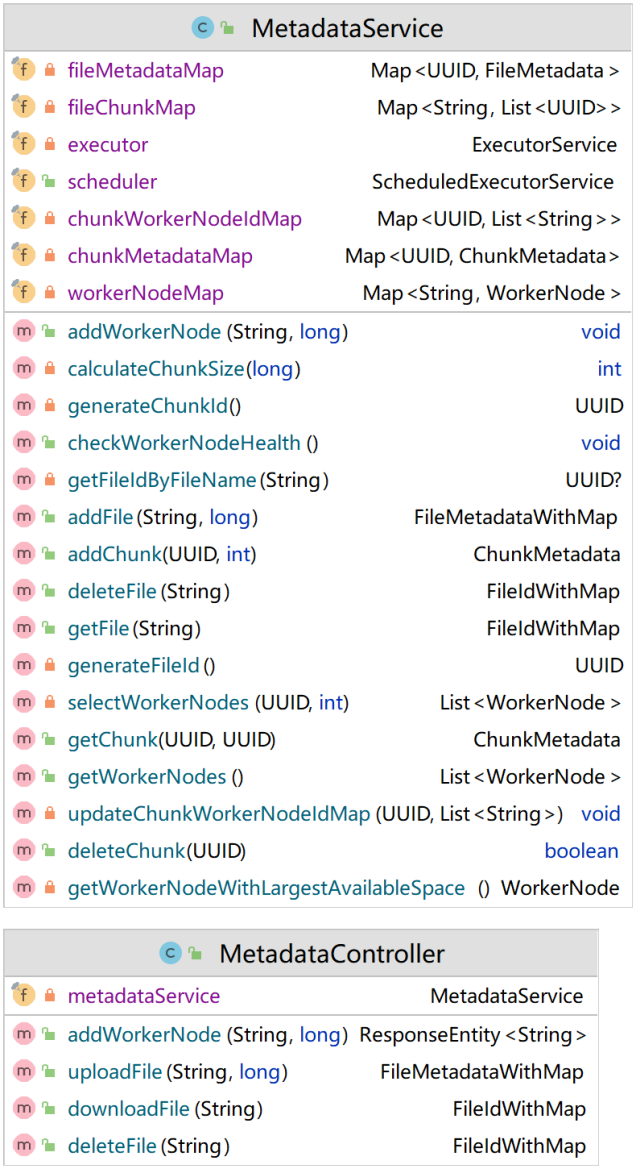

Figure 8. Metadata server class diagram

Document Server:

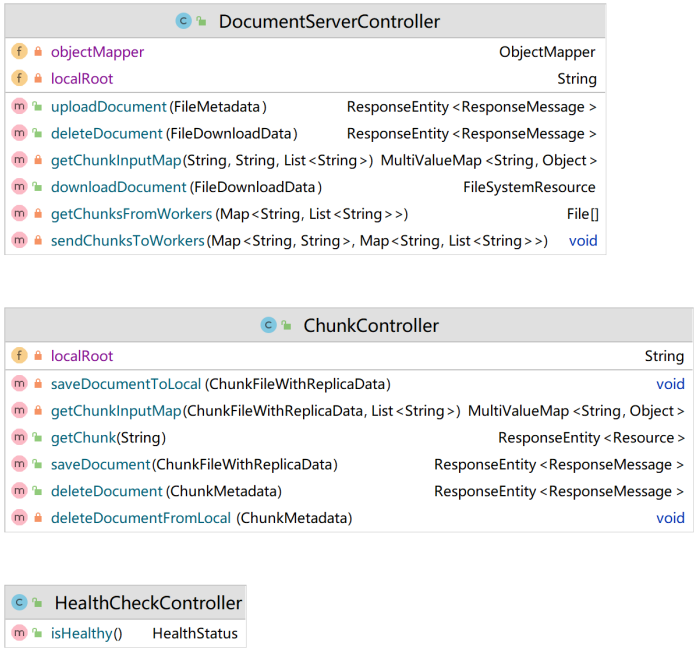

Figure 9. Document server class diagram

VII. Demonstration

In this section, we will demonstrate successful and failure scenarios and how our distributed system behaves in those scenarios.

User Signup- Our application demonstrates security by allowing only authorized users to access the application and perform operations on the files.
A user cannot access

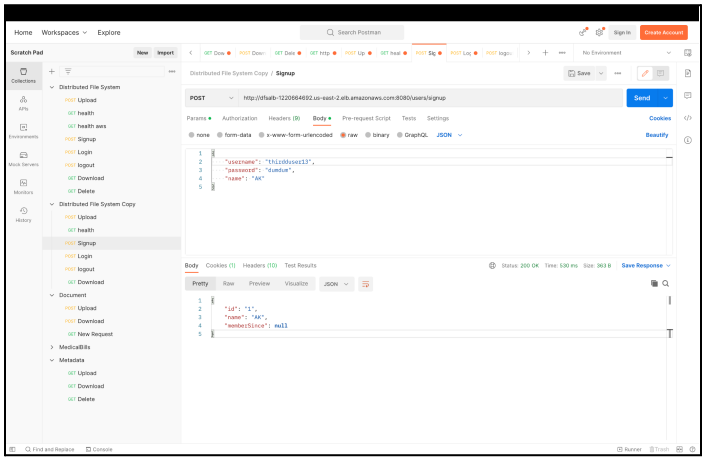

Figure 10. User Signup

User Login -An existing user can create a session by logging in to the application. Login creates a persistent session for the user that is available irrespective of coordinator server failures.

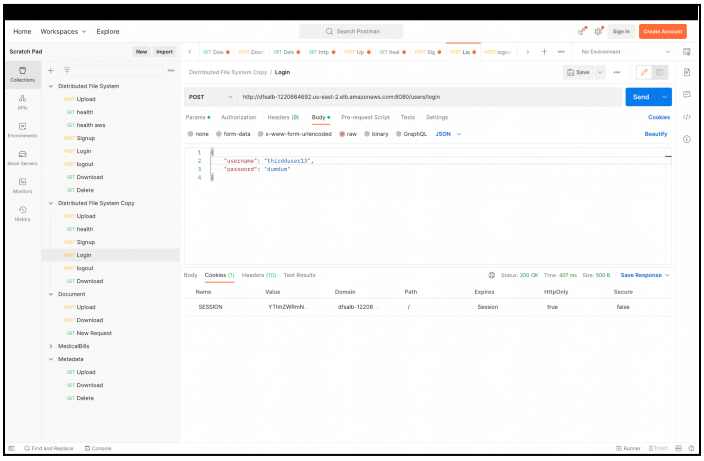

Figure 11: User Login

User Logout- The user can terminate his session using the logout functionality. On logout, the system redirects to a default login link which can be implemented to point to a valid login UI.

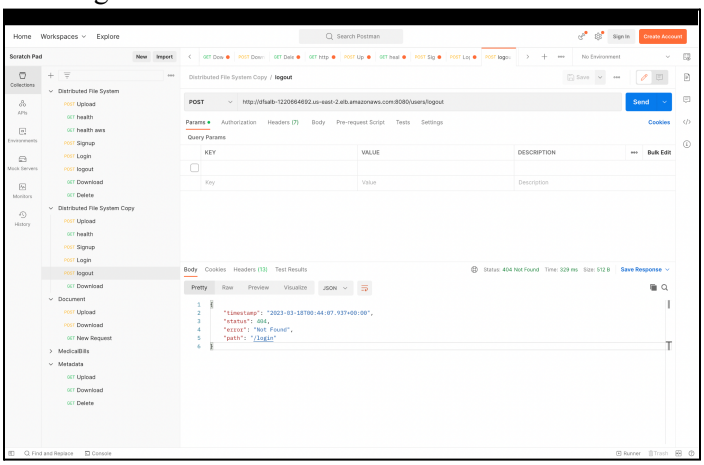

Figure 12: User Logout

Successful Scenario: Upload & Download flow

In this scenario, we will demonstrate the happy case when everything works end to end as expected for a normal flow.
When a user has to upload a file to our Distributed File System, they will first invoke the coordinator which invokes the metadata server asking for the details of chunks and the worker mapping for a given file name and file size. Below is a screenshot of the Metadata server

responding with the map containing a single chunk and multiple worker nodes to which the chunk should be sent in order to perform replication. Since the metadata server looks at the fileSize before creating chunks, in this case due to small file size, it only defines a single chunk for this upload scenario.

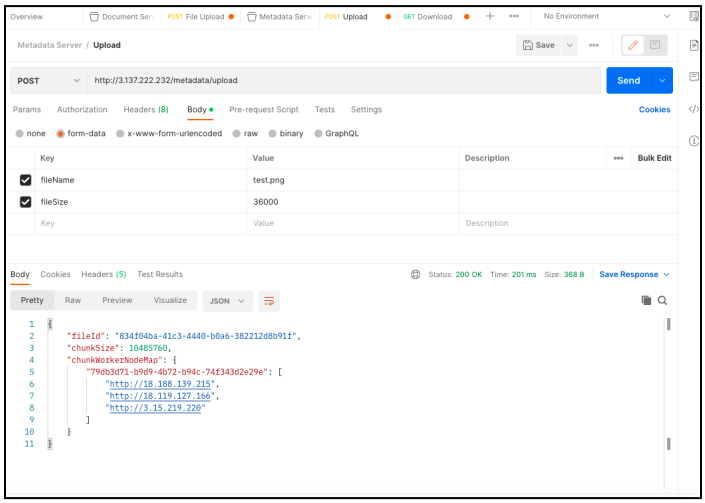

Figure 13: Metadata Server Upload Response

Now, we shall invoke the Leader of the Document Server and ask it to upload the given file along with the map of chunks and worker nodes each chunk needs to be replicated to.

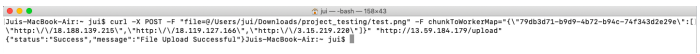

Figure 14: Document Server Upload Response

As you can see in the above screenshot, the file was uploaded successfully. In order to verify that the file was correctly broken into chunks and replicated across workers, we login to each of the Worker Nodes through the EC2 Console (via EC2 Connect) and list the chunks on the server.

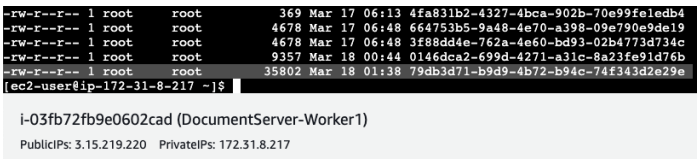

Figure 15: Document Worker 1 Chunk Upload

As shown above, the chunk is stored in the Worker-1.

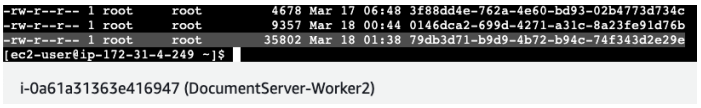

Figure 16: Document Worker 2 Chunk Upload

As shown above, the chunk is stored in the Worker-2 as well.

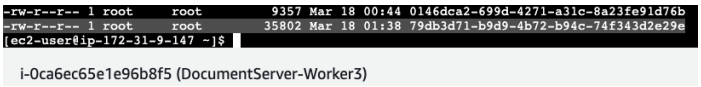

Figure 17: Document Worker 3 Chunk Upload

Finally, as shown above, the chunk is stored in the Worker-3 as well.
Now, we shall demonstrate that after the chunk is uploaded and replication is complete, we will try to download the document by invoking the Leader of the Document server with the same information that we get from the Metadata server using the file name (chunk to worker node mapping).
As you can see below in the screenshot, the file is downloaded successfully.

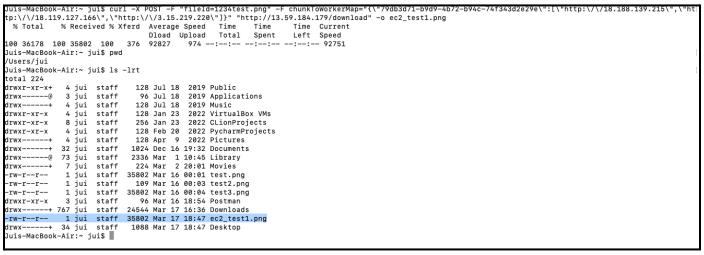

Figure 18: Document Leader File Download

Failure Scenarios

In this section, we shall describe various failure scenarios and how our system behaves in those scenarios. The purpose is to demonstrate that the system can recover from some common failures and still provide reliable availability and consistency.

Failure Scenario 1: Unauthorized user trying to upload the file

User cannot access any upload/download,delete operations if there is no valid session for him. If an unauthorized user tries to upload a file, we get a 403 error message saying that the user is unauthorized to upload the file.

Failure Scenario 2: Download when a Worker node has

crashed

In this scenario, we will demonstrate how our system behaves when a worker node crashes. There could be multiple ways this could happen in a real world scenario: disk space could get exhausted, the instance could get terminated by EC2 or the server running using Spring and Tomcat could crash as well. Here, in this example, we manually kill the server running on a Worker node.

In order to perform this, we shall look at the processes running on the machine. We do this by running the linux command 'ps -auxf' which lists all processes running along with processId. Then in order to kill this process, we execute the kill command - 'sudo kill -9 <process_id>'.

Figure 19: Document Worker Crash

As you can see from the screenshot here, the worker node has crashed. However, when we attempt to download the file from the Document Server - Leader node, it will look at the chunks and the workers and attempt to download the chunk from the workers. When a particular worker crashes, the leader will default to another worker to which the replication was done. As you can see in the screenshot below, the download was successful after crashing one of the worker nodes:

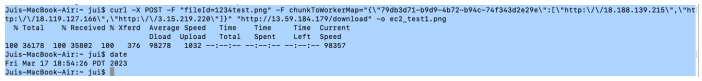

Figure 20: Document Leader Download after Crash

We can continue bringing down more workers and the system will continue to work as expected until we reach the point that we have crashed K-1 worker nodes performing replication if there were K replicas. When all the K replica nodes are crashed, the system would be unable to provide reliable response to the user requests.

Failure Scenario 3: File Deleted From Worker node

In this scenario, we create a failure scenario by deleting the chunk stored locally by a worker node. This kind of scenario is possible when the disk space is exhausted and we would like to empty the disk space manually or if bad chunks are created, we would like to clear them up. It can also happen if a process is cleaning up dead chunks (chunks that are created but there is no association of the chunkId in the metadata server).

So, in this particular scenario, we re-create the situation by manually deleting the file by invoking the 'rm' command in Linux as below:

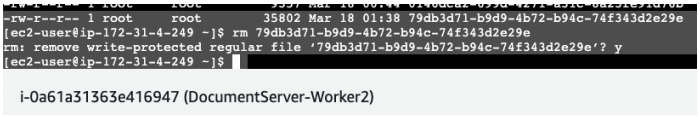

Figure 21: Document Worker Chunk delete

As you can see in the screenshot above, the file was successfully deleted from the Worker node.
Now, we try to download the file from the Leader node and we expect that this failure scenario will be handled.
As demonstrated in the screenshot below, the file was successfully downloaded from the Leader node even though the file was deleted from one of the worker nodes.

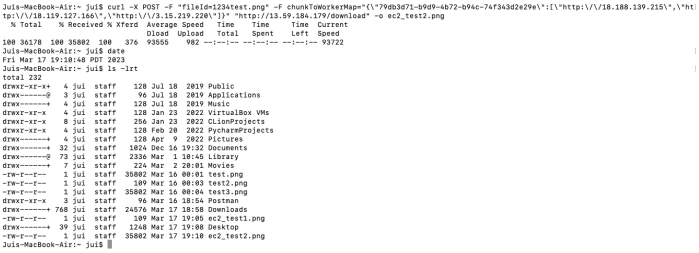

Figure 22: Document Leader Download after Delete

Failure Scenario 4: Communication Failure at a Worker Node

In this failure scenario, we demonstrate that the system works when there is a communication failure that happens to any of the worker nodes.

In order to create this scenario, we took the route of blocking inbound connections to one of the Worker nodes. This is achieved in AWS by modifying the Inbound rules of an EC2 Security Group. A Security Group is a configuration in AWS which is attached to each resource that we create. So, every EC2 instance is attached to a Security group where we define inbound and outbound connection rules. For example, we specify the port we use for SSH connections and the ports on which connections are allowed (TCP/HTTP/HTTPS) and whether a connection can be made from some particular instance or from anywhere in the internet.

For simplicity, we opted to keep the rules open for Inbound connections but we can restrict it as well if needed.
Here is the screenshot of inbound connection rules:

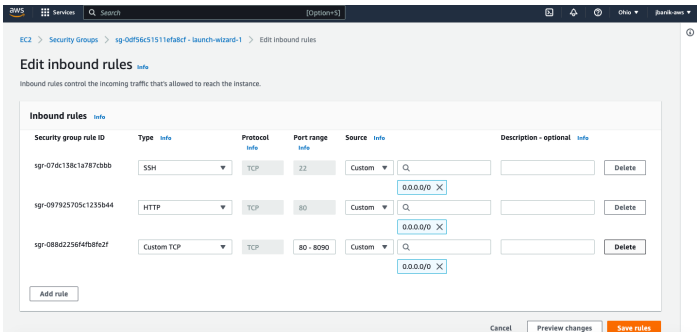

Figure 23: Ec2 Security Group Rules before Communication Failure

Now, in order to block the inbound connection from the Leader node to this particular worker, we will remove all TCP/HTTP connections to this node by deleting the inbound rules. As noted in the screenshot below, the rules have been removed.

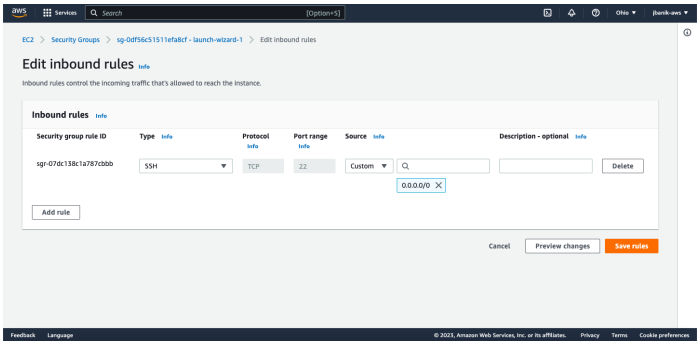

Figure 24: Ec2 Security Group Rules after Communication Failure

Now, we shall try to download the file by calling the Leader node for this file. As shown in the screenshot below, the leader node is still able to download the file and return to the client because it can handle these failure scenarios.

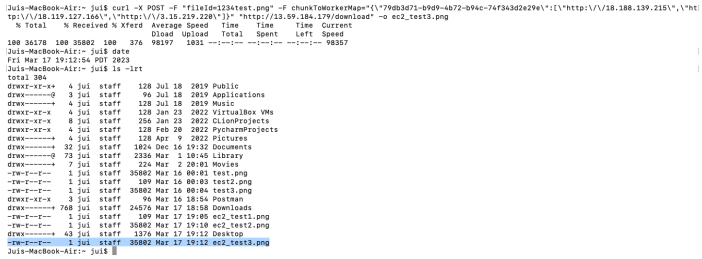

Figure 25: Document Leader File Download after Communication Failure

VIII. Performance Analysis

Overall Result

The overall result of building this distributed file system was effective due to three reasons:
1) We were able to build chunking and merging logic for the files which improved performance of the file handling by reducing the network bandwidth required during file storage
2) We were able to improve availability and reliability of the distributed file system by performing replication of chunks into multiple worker nodes
3) Implementation of resource discovery and health checks ensure that new worker nodes are automatically discovered when they come up and removed when they crash due to some reason.

Measurement Metrics

Latency: The latency of uploading a file to the Distributed File System. Latency is a key performance indicator as it is a metric that is clearly visible to the end users.

Availability: The availability of the system is a measurement of how much time a system is up and running and available to the end users. Availability is a key metric that end users look upon to determine whether the system is reliable. This is also important because to the end users, this provides transparency as internal details of replication and multiple servers are masked.

Workload Simulation and Metric Measurement

Latency: We tested our system with multiple file sizes and

noted the latency of the system against the file size. Here is what we observed:

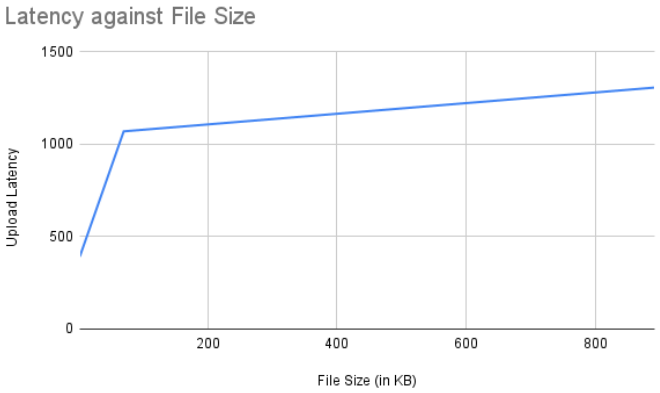

Figure 26: Latency benchmark

The latency grows rapidly as we increase file size from a very small file to medium size (around 100 KB) but soon stabilizes because the growth is linear w.r.t the file size. And the next 10X increase of file size (100 KB to 1 MB), is not creating a steep jump in the latency.

Here is what we observe for availability of the system:

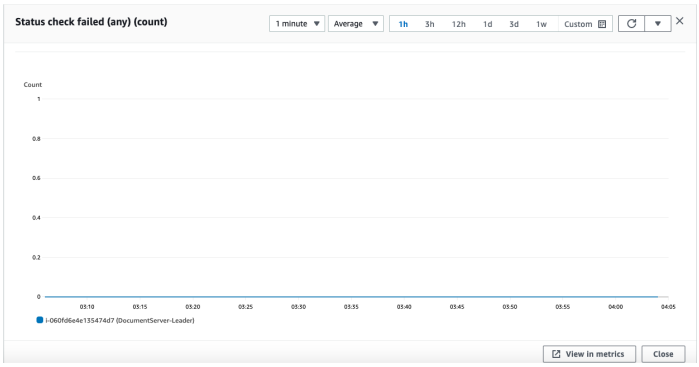

Figure 27: Availability benchmark

This graph shows the metric of how many times a health check on our machines failed. This graph is a representation of Availability of the system. As we can see, due to usage of AWS services that provide high availability (~99.99%), our systems are highly available.

IX. Testing Results

In this section, we shall talk about the result from testing different scenarios on our distributed file system.

Test Case 1: File Upload of Small Size

In this scenario, we shall take a file of relatively small size (1-2 KB) and try to upload this on the server. Typically, this is a happy case scenario but we try to break it into 4 chunks in order to test that the system can perform chunking even for small size documents when needed. This can be especially helpful when the worker nodes are close to maximum storage capacity.

Here is a screenshot of uploading a file of 2KB using Postman:

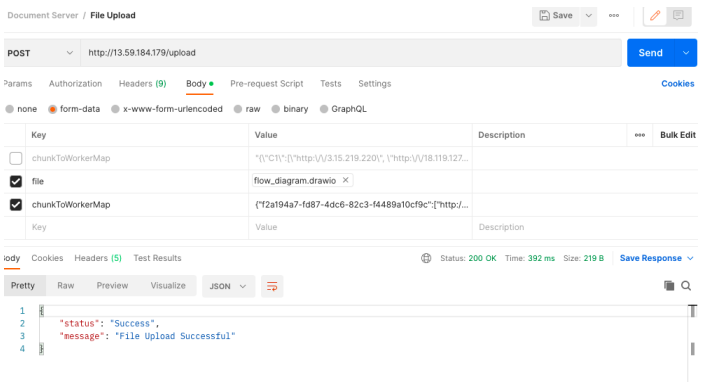

Figure 28: File Upload Small Size

As you can see from the screenshot above, in this test case, the system behaves as expected by creating multiple chunks. Here is also a screenshot from the EC2 instance (Worker Node 1) to show that there are chunks of size 487 bytes each.

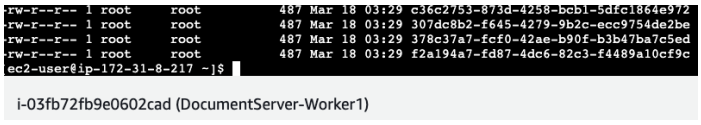

Figure 29: File Upload - Chunk creation

Therefore, the system behaves as expected in this scenario.

Test Case 2: File Upload of Medium Size

In this scenario, we will try to upload a file of medium size (<100 KB) and see how the system behaves in this scenario by creating multiple chunks.

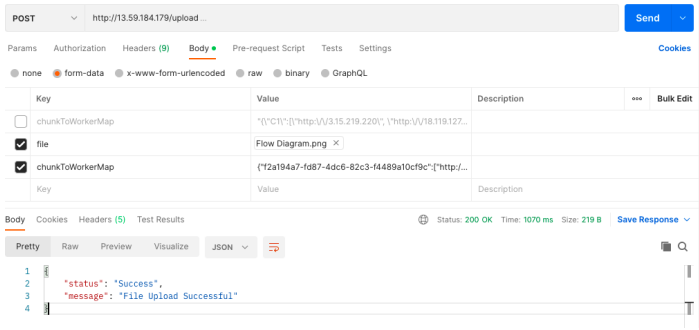

Figure 30: File Upload - Medium Size

As seen in the screenshot above, the file upload is successful for a file of size 70 KB. In this scenario, we observe that the latency increased due to the file size increase and the need to create all the chunks on all the replicas.

Test Case 3: File Upload of Large Size

In this scenario, we shall test the system using a file upload of large size (~1 MB) and see how the system behaves in this scenario and the latency expectation in this case.

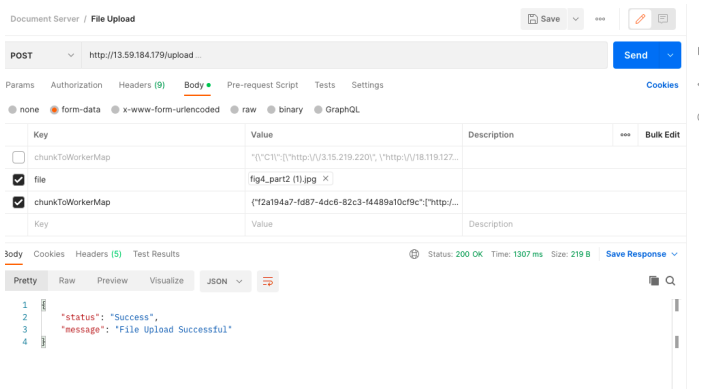

Figure 31: File Upload - Large Size

In this scenario, our observation is that the file upload of this large file took the longest (1300 msec approximately). Here, as we see that the latency is high, there are multiple optimizations possible to the upload strategy by the use of advanced algorithms. For example, if we apply quorum to replication where we only replicate to two out of three replicas and not wait for the third replication to complete before returning a response to the client, we can respond much faster. Another optimization possible is that we upload to one server synchronously and publish an event which shall be consumed by other replica workers and create a copy of the file by calling the primary replica. However, in this design choice, there is a possibility of losing the data because if the primary replica fails in the duration that the file was uploaded and the secondary replicas were able to consume the asynchronous event.

Test Case 4: File Download of Large Size

In this scenario, we demonstrate the download of a large file which was previously uploaded to the distributed file system.

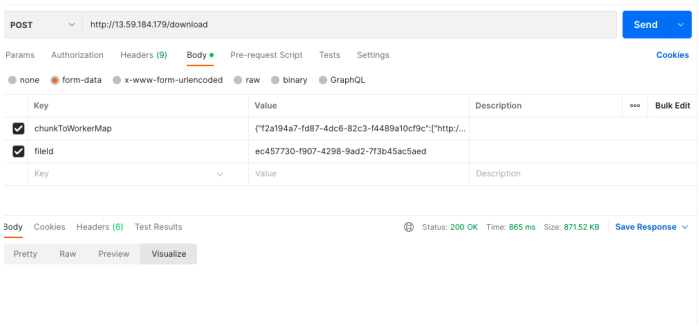

Figure 32: File Download - Large Size

As we can see from the screenshot above, we were able to obtain the file using the Download API. Even though the file is not visible (due to the image format in Postman) , the file was downloaded successfully as we received a 200 OK response from the server.

Test Case 5: File Delete of Large Size

In this scenario, we demonstrate the deletion flow. So, in order to delete the file we need to call the Document server with fileId and chunk data. The server ensures that all the copies of the file are deleted.

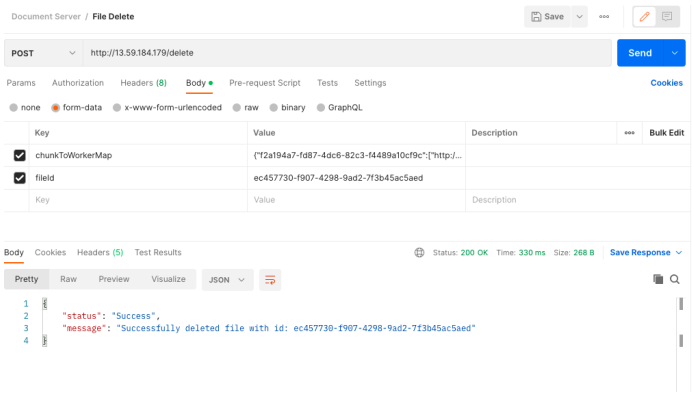

Figure 33: File Delete - Large Size

As we can see from the screenshot above, the file deletion is completed and we receive a 200 OK response from the server. We manually verified that these chunks are no longer on the server.

X. Conclusion

Development of distributed systems have been attempted in the past but they have become popular in recent times due to globalization of activities. We have developed a highly scalable and reliable distributed file system using AWS. Our system addresses all key concerns of a distributed application and helps users store files across geo locations and machines.

Improvements:

1. End to end chunking- Chunking improves the operation of file storage by reducing the amount of data that travels over the network. In our application, the document server chunks the files and stores them across nodes. In order to reduce bandwidth requirements of the application, we can enable the coordinator to deal with all file operations in chunks. This will ensure minimum data traffic and reduce the load on the network.

XI. References

1. R. Tobbicke, "Distributed file systems: focus on Andrew File System/Distributed File Service (AFS/DFS)," Proceedings Thirteenth IEEE Symposium on Mass Storage Systems. Toward Distributed Storage and Data Management Systems, Annecy, France, 1994, pp. 23-26, doi: 10.1109/MASS.1994.373021.
2. Di Liu and Shi-Jie Kuang, "A kind of distributed file system based on massive small files storage," 2012 International Conference on Wavelet Active Media Technology and Information Processing (ICWAMTIP), Chengdu, China, 2012, pp. 394-397, doi: 10.1109/ICWAMTIP.2012.6413521.
3. R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II, Las Vegas, NV, USA, 2005, pp. 205-213 Vol. 2, doi: 10.1109/ITCC.2005.42.
4. Q. Wang, C. Wang, K. Ren, W. Lou and J. Li, "Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing," in IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 5, pp. 847-859, May 2011, doi: 10.1109/TPDS.2010.183.
5. M. T. Rashid, D. Zhang and D. Wang, "EdgeStore: Towards an Edge-Based Distributed Storage System for Emergency Response," 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China, 2019, pp. 2543-2550, doi: 10.1109/HPCC/SmartCity/DSS.2019.00356.
6. A. Khiyaita, H. E. Bakkali, M. Zbakh and D. E. Kettani, "Load balancing cloud computing: State of art," 2012 National Days of Network Security and Systems, Marrakech, Morocco, 2012, pp. 106-109, doi: 10.1109/JNS2.2012.6249253.
7. H. -C. Hsiao, H. -Y. Chung, H. Shen and Y. -C. Chao, "Load Rebalancing for Distributed File Systems in Clouds," in IEEE Transactions on Parallel and Distributed Systems, vol. 24, no. 5, pp. 951-962, May 2013, doi: 10.1109/TPDS.2012.196.