Distributed File Sharing System

Samyuktha Venkatesan
svenkatesan2@scu.edu
SCU ID: 07700006461

Srivarsha Gowdara Chandrashekarappa
sgowdarachandrasheka@scu.edu
SCU ID: 07700005839

Vibha Nayak
vnayak3@scu.edu
SCU ID: 07700006648

***Abstract*— The paper aims to develop a Distributed File Sharing System to manage file storage, retrieval, and sharing across a distributed network. This system will enable users to store files and retrieve them from anywhere in the network, and share them with others. Key components include efficient file storage mechanisms, distributed search algorithms for file retrieval, replication and failure detection for file sharing. The system is designed for scalability, reliability and availability, ensuring that it can handle a large number of users and files while remaining robust and efficient.**

***Keywords*— Chord-DHT, Flask API, Multi-threading, PingAck Protocol, Ring Election Protocol**

I. INTRODUCTION

In today's data-rich environment, managing the vast volume of information generated daily poses a significant challenge. Traditional centralized systems often struggle to handle this influx efficiently. Distributed File Sharing System (DFSS) addresses the challenge of managing and storing data efficiently. We plan to work on implementing such a system to tackle this issue effectively. This system would allow for more effective storage, retrieval, and sharing of files across distributed networks. In addition to its fundamental role in addressing data management challenges, Distributed File Sharing Systems (DFSS) serve as foundational tools across various domains, facilitating data-intensive computations irrespective of data size. These systems empower users to share and modify large datasets seamlessly while maintaining data consistency. As an essential component of distributed systems, DFSS showcases the significance and versatility of distributed architectures, highlighting their ability to streamline collaborative work processes and unlock the potential of vast datasets.

II. PROJECT GOALS

A. High Availability:

Objective: To ensure that files are always accessible, even in the event of node failures.

Implementation: We have implemented redundancy and replication mechanisms on multiple nodes to maintain availability at all times even in case of node failures. Further, we have utilized Chord's ring structure to maintain access paths even if some nodes are down, ensuring that clients can always retrieve their data.

B. Scalability:

Objective: To design the system to efficiently handle an increasing number of nodes and files without performance degradation.

Implementation: We have leveraged Chord's logarithmic scaling properties to ensure that lookups and data retrieval remain efficient as the network grows. Additionally, in also provides dynamic addition and removal of nodes to support seamless scaling.

C. Fault Tolerance:

Objective: To ensure that the system can continue functioning smoothly despite individual node failures or network issues.

Implementation: We have executed redundancy and failover mechanisms. The use of Ping Ack protocol to identify node failures and the usage of Chord's stabilization protocols in the system helps repair the ring and maintain data integrity.

D. Load Balancing:

Objective: To distribute files and requests evenly across nodes to avoid overloading any single node.

Implementation: The system uses consistent hashing to distribute files evenly and implements dynamic load balancing techniques to reassign responsibilities as nodes are added or removed, ensuring even distribution of storage and request handling.

E. Ease of Use:

Objective: To ensure the system is user-friendly for both end-users.

Implementation: We have developed intuitive interfaces for uploading, downloading, and managing files. The system ensures straightforward deployment procedures and user-friendly configuration options. The use of API helps clients interact with the system with ease.

F. Efficient Search and Retrieval:

Objective: To minimize the time it takes to locate and retrieve files.

Implementation: We have optimized Chord's O(log N) lookup protocol for fast file location. And have also implemented efficient routing algorithms to minimize the number of hops required for search queries, enhancing user experience with faster access times.

G. Decentralization:

Objective: To eliminate the need for a centralized architecture to avoid single points of failure and increase system resilience.

Implementation: By using Chord's distributed architecture we have guaranteed that no single node has control over the entire system. By implementing distributed algorithms for leader election, data storage, retrieval, and management, we have allowed the system to function effectively without a central authority. By ensuring that all nodes participate equally in the network's operations, we have enhanced fault tolerance and scalability.

III. MOTIVATION

The motivation for this paper extends beyond addressing the limitations of centralized storage. Distributed File Sharing Systems play a critical role in various domains. They enable efficient processing of large datasets, seamless collaboration on massive files with data consistency, and serve as a foundational tool for distributed systems, highlighting the power of these architectures to streamline collaborative work and unlock the potential of vast datasets. By developing a DFSS, we contribute to the advancement of data management solutions, not only addressing a current challenge but also paving the way for further exploration of distributed systems and their applications in diverse fields.

IV. DISTRIBUTED SYSTEM CHALLENGES ADDRESSED

Developing a robust Distributed File Sharing System (DFSS) necessitates addressing several inherent challenges associated with distributed systems. Our system incorporates several key design choices to mitigate these challenges and ensure efficient, reliable, and scalable file management.

Heterogeneity in distributed systems often encompasses a diverse mix of hardware, software, and operating systems. Our approach tackles this by focusing on platform independence by utilizing standard protocols like HTTP and TCP guarantees compatibility across different platforms, enabling seamless interaction between various devices and software environments. This allows users with different systems to participate in the network.

Node failures and network disruptions can pose significant threats to data availability and system stability. Our design choices address this challenge by replicating files across the network, typically on successor nodes, ensuring data remains accessible even if a node fails. This redundancy minimizes data loss and maintains system availability. We are also implementing automated failure detection mechanisms that enable the system to swiftly identify issues and initiate recovery procedures. This minimizes downtime and ensures system stability in the face of failures. Eliminating single points of failure by distributing critical components across the network enhances overall system availability. If one node fails, others can continue servicing requests, minimizing disruption. As the number of users and data volume grows, a distributed system's ability to handle increased load becomes crucial. Our design choices address this scalability issue by efficient resource management using load-balancing techniques that distribute workloads evenly across the network, preventing the overloading of individual nodes. This optimizes resource utilization and maintains consistent performance as the system scales.

Distributed systems can introduce complexities in terms of resource location, failure handling, and data consistency. Our design choices promote transparency by implementing location/access transparency for users interacting with the system without needing to know the physical location of stored data. This simplifies user experience and promotes user-friendliness. By implementing failure transparency that hides the complexities of failure handling from the user. Even during node or network failures, users experience seamless operation, maintaining a sense of reliability and trust in the system. By implementing replication transparency mechanisms for data replication and synchronization are handled automatically by the system. This ensures users always have access to consistent data views across the network, regardless of underlying replication processes.

Concurrent access to files for upload and download operations is essential for efficient file sharing. Our design enables users to perform these actions simultaneously, allowing for faster file transfers and improved overall system responsiveness.

By incorporating these design choices, our Distributed File Sharing System effectively addresses several key challenges associated with distributed systems. This ensures a robust, scalable, and user-friendly platform for file storage, retrieval, and sharing across a distributed network.

V. PREVIOUS WORK

The ever-increasing volume of data generated daily necessitates efficient solutions for storage, retrieval, and sharing across networks. Distributed File Systems (DFS) have emerged as a powerful response to this challenge. This section explores existing DFS solutions, highlighting their strengths and potential limitations to provide context for our proposed system. Google File System (GFS) introduced the concept of chunking large files into fixed-size blocks, enabling efficient

storage and retrieval. Its focus on high throughput and fault tolerance influenced subsequent DFS designs [2].

HDFS (Hadoop Distributed File System) is ideal for big data analytics, HDFS excels at storing and managing massive datasets efficiently. Its distributed architecture allows for parallel processing and fault tolerance, making it well-suited for large-scale data analysis tasks [2]. Ceph is an object-based storage system that offers high scalability and flexibility. Ceph stores data in objects, enabling efficient management of diverse file types and sizes. Additionally, its distributed design allows for horizontal scaling to accommodate growing storage demands [6].

GlusterFS is an open-source, POSIX-compliant DFS solution, GlusterFS prioritizes performance and versatility. It offers good performance for various workloads and integrates seamlessly with existing Linux environments [1].

In distributed file systems, algorithms play critical roles in addressing various challenges. One such algorithm is introduced in [5] by Aliva Bakshi et al. This work introduces the Write Preference Heap Tree algorithm, aimed at achieving lock-free file consistency. By mitigating delays associated with acquiring read-write locks, this approach offers a practical solution to improving system efficiency in distributed file systems.

A detailed overview of the key components and the role of distributed file systems is provided by [6] by Sabitha R et al. It highlights the challenges faced by distributed file sharing systems in cloud storage, such as scalability, consistency, data integrity, and fault tolerance. The paper also outlines the evolution of these systems, briefly mentioning the advantages of the latest cloud storage solutions like Azure Blob and Amazon S3.

These existing DFS solutions offer valuable functionalities. However, they may have limitations that our proposed system aims to address. Our proposed system focuses on a balance between scalability, performance, and ease of use. It leverages Distributed Hash Tables (DHTs) like Chord for efficient data lookup and storage, aiming to provide a robust platform for general-purpose file sharing across a distributed network. This exploration of existing DFS solutions highlights the evolving landscape of data management technologies. Our project builds upon these advancements, aiming to contribute a user-friendly and scalable solution for distributed file storage and sharing.

VI.PROJECT DESIGN

A. Key Design Goals

1) Heterogeneity: Our system prioritizes platform independence by leveraging standard protocols like HTTP and TCP. This allows seamless interaction between diverse hardware, software, and operating systems, promoting user participation regardless of their specific platform.

2) Openness: While this project focuses on a closed system for controlled development, the core design principles can be extended to incorporate open standards and APIs in future iterations. This openness would allow for integration with third-party applications and services, fostering a more flexible and adaptable system.

3) Security: Our design emphasizes secure data storage and access control mechanisms. User authentication and authorization protocols will be implemented to restrict unauthorized access and ensure data privacy. Additionally, secure communication protocols like HTTPS will be employed to safeguard data during transmission across the network.

4) Failure Handling: Redundancy through data replication across the network minimizes data loss in case of node failures. Automated fault detection and recovery mechanisms will be implemented to swiftly identify and address issues, minimizing downtime and maintaining system availability. Notably, the distributed architecture eliminates single points of failure, ensuring the system remains operational even if individual nodes encounter problems.

5) Concurrency: The system will be designed to enable concurrent file uploads and downloads. This allows multiple users to access and modify files simultaneously, improving overall system responsiveness and efficiency. Mechanisms for conflict resolution and data consistency will be crucial to ensure data integrity during concurrent operations.

6) Quality of Service: Performance metrics like latency and availability will be closely monitored to ensure the system meets desired QoS levels. By employing load balancing techniques, workloads will be evenly distributed across the network, preventing bottlenecks and maintaining consistent performance even under heavy load.

7) Scalability: Elastic scalability will be a key feature, allowing the system to gracefully adapt to fluctuations in user base, data volume, and request rates. The ability to add or remove nodes dynamically ensures efficient handling of increased load and promotes the system's ability to grow alongside user needs.

8) Transparency: Users will be able to interact with the system without needing to be aware of the physical location of stored data (location transparency). Additionally, the system will handle complexities like failure recovery and data replication transparently, providing users with a seamless experience even during potential disruptions. This transparency fosters user confidence and simplifies file management within the system.

B. Key Components

*1) Client:*The client interface allows users to upload files to the distributed system and download them from it. This interaction is typically managed through a web interface provided by Flask, which serves as the front-end framework.

The client handles errors gracefully, providing users with meaningful messages and notifications regarding the status of their actions (e.g., File uploaded, File download).

2) DNS Server: The DNS server resolves domain names to IP addresses. In the context of a distributed file-sharing system, it resolves requests to the IP addresses of the load balancers. By employing a round-robin or other load balancing algorithms, the DNS server ensures that requests are evenly distributed among multiple load balancers. This helps avoid overloading any single load balancer and maintains system stability.

3) Load Balancer: Load balancers receive incoming client requests and distribute them across multiple chord nodes. This distribution ensures that no single node is overwhelmed by too many requests. Load balancers monitor the health and performance of chord nodes. If a node becomes unresponsive or overloaded, the load balancer can redirect traffic to other nodes.

4) Chord: Chord is a Distributed Hash Table (DHT) protocol that assigns keys (file identifiers) to nodes. Each node in the Chord network is responsible for a specific range of keys, ensuring distributed storage. The Chord protocol allows for efficient lookup operations with logarithmic time complexity. It uses a finger table at each node, which points to other nodes in the network, enabling quick file retrieval.

Chords can handle a dynamic number of nodes. When nodes join or leave the network, the protocol efficiently reassigns key responsibilities and updates finger tables to maintain balance. Chord is designed to be robust against node failures. If a node fails, its key responsibilities are redistributed among remaining nodes. Additionally, Chord can replicate data across multiple nodes to further enhance fault tolerance.

When a new node joins, it integrates into the existing network by taking over a portion of the key space from another node. When a node leaves, its responsibilities are seamlessly transferred to neighboring nodes. To enhance data availability and reliability, copies of a file are stored at the successor node, ensuring that data is not lost if a node fails.

B) Algorithms

1) Resource Discovery: The primary goal of the Resource Discovery Algorithm in Chord is to locate the node responsible for storing a specific resource identified by a key. This process must be efficient, minimizing the number of hops (transfers from node to node) required to find the desired resource.

Chord organizes nodes and resources using a DHT. Each resource (file) is assigned a unique key, typically derived using a hash function. Each node in the Chord network is also assigned a unique identifier from the same hash space. Resources are mapped to nodes using consistent hashing. A resource with a given key is stored on the node whose identifier is equal to or follows the key in the circular identifier space. Each node maintains a finger table, a routing table that contains references to other nodes in the network. This table allows a node to efficiently locate the node responsible for any given key. The finger table entries at nodes point to nodes at specific intervals around the identifier circle, helping to quickly narrow down the search for a key.

To find a resource(file), a node initiates a lookup for the resource's key. The lookup protocol involves querying nodes along the path towards the node responsible for the key. The querying node uses its finger table to forward the request to the node that is closer to the target key. This process continues iteratively until the node responsible for the key is reached.

The client (or a node) initiates a lookup for a resource by computing the hash of the resource's identifier to get the key. The node uses its finger table to identify the closest preceding node (in terms of the key's hash) and forwards the lookup request to that node. The request propagates through the network, with each intermediate node using its finger table to forward the request closer to the target node responsible for the key. Once the node responsible for the key is reached, it returns the resource or the location information to the requesting node.

By using this algorithm, the logarithmic lookup time (O(log N), where N is the number of nodes) ensures that resource discovery is quick even in large networks. Chord can efficiently handle a growing number of nodes and resources, making it well-suited for large-scale distributed systems. The decentralized nature of Chord ensures that even if some nodes fail, the system can still locate resources by rerouting requests through alternative nodes.

*2) Replication:*The primary goal of replication protocols in Chord is to ensure that copies of a resource (file) are stored on multiple nodes. This redundancy helps in maintaining data availability even if some nodes fail or leave the network.

A predetermined number of copies of each resource are maintained in the network. This number, known as the replication factor, determines how many nodes will store the same resource. Each node in Chord maintains a successor list, which contains a list of the next few nodes (successors) in the identifier circle. These successors are potential candidates for storing replicas.

When a resource is stored on a node, additional copies are also stored on its successors, in our case the system will be storing additional copies in one successor node.

When a resource (file) is inserted into the system, the node responsible for the key stores the resource and then replicates it to its successors as per the replication factor. Each successor node stores a copy of the resource and maintains metadata to indicate it is a replica. When a resource is requested, the lookup process initially seeks the primary copy. If the primary node is unavailable, the request is directed to one of the successor nodes holding a replica. This ensures that the resource can still be accessed even if the primary node fails.

When a new node joins the network, it takes over a portion

of the key space from its predecessor and may receive replicas of resources to maintain the replication factor. When a node leaves the network, it transfers its resources and replicas to its successor to ensure continuous availability. Replication ensures that resources are available even if some nodes fail, significantly improving the system's fault tolerance.

3) Leader Election: The primary goal of a Leader Election algorithm in Chord is to designate a single node as the leader (or coordinator) to perform specific tasks that require centralized control or coordination among nodes. The leader monitors node health and manages recovery and rebalancing when nodes fail or leave. The leader can optimize the distribution of resources and load balancing within the network. The election process is triggered when the current leader fails, leaves the network, or when a new leader is needed for coordination tasks. Our system uses the Ring Election algorithm in Chord's distributed environment.

The election process is initiated by a node when it detects that the current leader is unavailable or when a new leader is needed. This can happen through timeouts, heartbeats, or explicit messages indicating the leader's failure. Nodes are arranged in a logical ring. Each node passes an election message around the ring, and the node with the highest identifier is elected as the leader. Once a node is elected as the leader, it announces its leadership to all other nodes in the network. Other nodes update their state to recognize the new leader and start communicating with it for coordinated tasks. The leader detects node failures and coordinates the redistribution of resources and responsibilities to maintain system integrity. The leader monitors the load on different nodes and can reassign resources to balance the load evenly across the network.

4) Failure Detection Algorithm: Ping Acknowledgement (Ping Ack) algorithm is crucial for maintaining the health and connectivity of nodes within the network. This algorithm ensures that each node can reliably communicate with others, detect failures promptly, and maintain an updated view of the network topology. In our system the Ping Ack algorithm operates by sending periodic "ping" messages from one node to another to check its availability and responsiveness. When a node, say Node A, wants to verify the status of another node, Node B, it sends a ping message to Node B. Upon receiving this ping, Node B is expected to respond with an acknowledgment message, commonly referred to as an "ack." This acknowledgment serves as a confirmation that Node B is active and capable of communication.

The process begins with Node A initiating a ping message, which includes a unique identifier and a timestamp to track the request. This message is sent to Node B using the network protocol in place (TCP/IP). Node B, upon receiving the ping, processes the message and immediately generates an acknowledgment, embedding the unique identifier and the timestamp received from Node A. This ack message is then sent back to Node A.

Node A, after sending the ping, starts a timer and waits for the acknowledgment within a predefined timeout period. If Node A receives the ack from Node B within this period, it concludes that Node B is functioning correctly and marks it as active in its internal records. This successful ping-ack exchange indicates a healthy network connection between Node A and Node B.

However, if Node A does not receive the acknowledgment within the timeout period, it infers that Node B might be unavailable or experiencing issues. Node A will typically retry the ping a few more times to account for transient network issues or temporary delays. If multiple attempts fail, Node A marks Node B as inactive or failed. This information is then propagated to other nodes in the network, ensuring that the network topology is updated, and necessary steps are taken to reroute requests or replicate data to maintain system reliability.

C. Architecture

The distributed system architecture depicted in the diagram (Fig. 2) of a distributed file-sharing system leverages the Chord protocol for efficient file storage and retrieval within a cloud environment. It incorporates DNS for load balancing and redundancy, ensuring high availability and reliability. The system allows for seamless file uploads and downloads by distributing the requests through load balancers to a Chord-based cloud infrastructure.

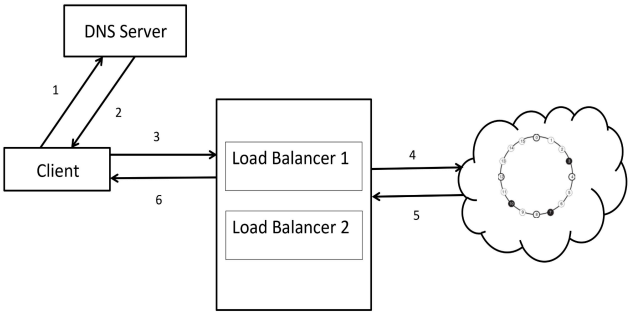


Fig.1 System Architecture

The DNS server plays a crucial role in resolving the IP and port of an available load balancer, acting as the initial point of contact for the client. The process begins when the client requests the DNS server for the IP and port of a load balancer, to which the DNS server responds with the necessary details. The client interacts with the load balancer for uploading or downloading files. For file uploads, the client sends a request to the DNS server, receives the IP and port of a load balancer, and then sends the file to be uploaded to the load balancer. The load balancer forwards the file to the Chord-enabled cloud, receives an acknowledgment from the cloud, and subsequently sends an acknowledgment back to the client. For file downloads, the client requests the DNS server for the IP and port of a working load balancer, receives the details, and sends a request to the load balancer. The load

balancer requests the file from the cloud, receives the requested file, and forwards it to the client.

Load balancers, such as Load Balancer 1 and Load Balancer 2, manage incoming requests from clients and distribute them to the cloud storage. They ensure balanced load distribution and redundancy. For file uploads, they receive the file from the client and forward it to the cloud. For file downloads, they retrieve the file from the cloud and send it to the client. Multiple load balancers are used to enhance fault tolerance and ensure continuous availability.

The Cloud storage leverages the Chord protocol to efficiently manage file storage and retrieval. Chord is a distributed hash table (DHT) protocol that provides a scalable and efficient means to locate files in a distributed system. The cloud receives files from the load balancer and stores them in the Chord ring. For file retrieval, the cloud locates the file within the Chord ring and sends it to the load balancer. Chord organizes nodes in a ring structure, where each node maintains information about its successor nodes. Files are stored at nodes based on consistent hashing, ensuring even distribution of data. Chord's lookup mechanism efficiently locates the node responsible for a given file.

The architecture offers several benefits, including scalability, fault tolerance, efficient file lookup, and load distribution. The use of Chord ensures that the system can scale horizontally, allowing new nodes to be added without significant reconfiguration. The system's redundancy, with multiple load balancers and the distributed nature of Chord, ensures high availability and fault tolerance. Chord's logarithmic lookup time ensures quick file retrieval, enhancing the system's overall performance. Load balancers distribute client requests evenly, preventing any single point of failure and optimizing resource utilization.

1) Activity Diagram

The file upload process similarly starts with the client, who initiates a file upload request. The client first contacts the DNS server to get the IP and port of an available load balancer. The DNS server responds with the details of a working load balancer. With this information, the client sends the file to be uploaded to the load balancer.

The load balancer, upon receiving the file, needs to determine the appropriate target node where the file should be stored. It sends a request to the leader node to get the details of the target node. The leader node, which manages the storage locations of files within the system, responds by sending the details of the target node suitable for storing the new file.

Having obtained the details of the target node, the load balancer proceeds to upload the file to the designated target node. The target node then stores the file, completing the upload process. This process ensures that the file is efficiently distributed within the system, leveraging the load balancer to manage traffic and the leader node to maintain an organized record of file locations

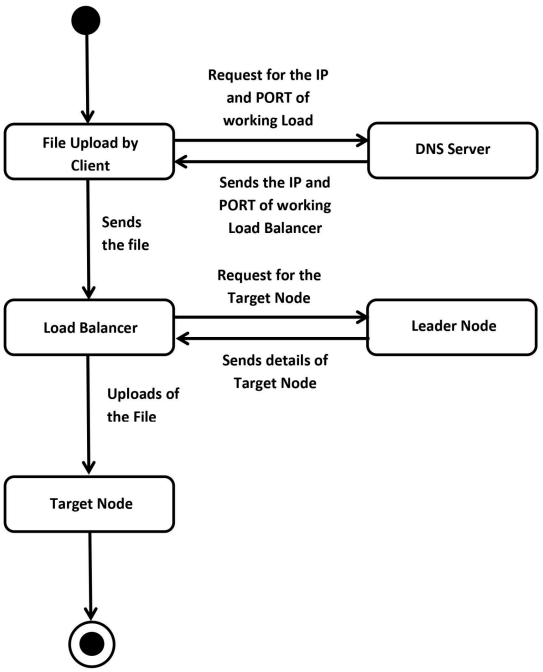


Fig.2 Activity diagram(File Upload)

The file download process in the distributed file-sharing system begins with the client initiating a file download request. The client first sends a request to the DNS server to obtain the IP and port of a working load balancer. The DNS server responds by providing the necessary details of an available load balancer. With this information, the client sends the file name to the load balancer, indicating the file they wish to download.

The load balancer, upon receiving the file name, needs to determine the location of the target node where the file is stored. To accomplish this, it sends a request to the leader node, asking for details about the target node. The leader node, which maintains a record of where files are stored across the system, responds by sending the details of the target node that has the requested file.

Once the load balancer has the target node's details, it sends a request to the target node to download the file. The target node, upon receiving the request, retrieves the requested file and sends it back to the load balancer. The load balancer then forwards the requested file to the client, completing the download process. This series of steps ensures efficient file retrieval while distributing the load across multiple nodes and balancing traffic through load balancers.

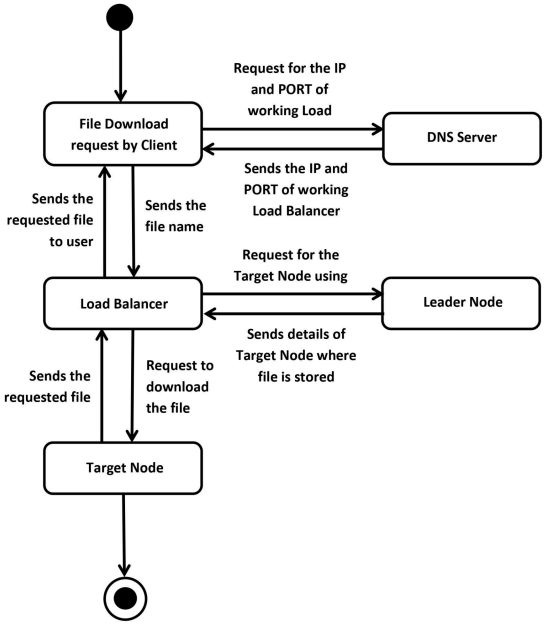


Fig.3 Activity diagram(File Download)

2) Sequence Diagram

The sequence diagram for the file upload process in the distributed file-sharing system illustrates the interaction between the user interface, client, DNS server, load balancer, and Chord nodes.

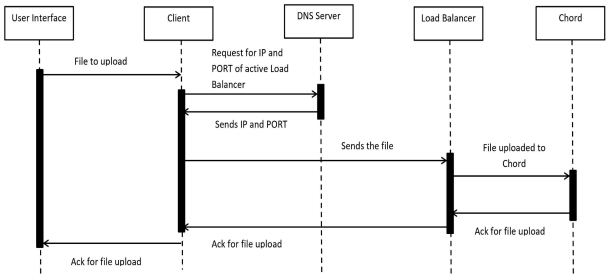


Fig.4 Sequence diagram(File Upload)

The user initiates the upload request through the user interface by selecting the file to be uploaded. This request is forwarded to the client. The client receives the upload request and sends a request to the DNS server to obtain the IP and port of an active load balancer. The DNS server processes the client's request and responds with the IP and port of an available load balancer. The client receives the IP and port details from the DNS server and sends the file to be uploaded to the load balancer. The load balancer receives the file from the client and forwards it to the Chord network for storage. The Chord network stores the uploaded file and sends an acknowledgment back to the load balancer indicating that the file has been successfully stored. The load balancer receives the acknowledgment from the Chord network and forwards it to the client. The client receives the acknowledgment from the load balancer and delivers it to the user interface, confirming the successful upload of the file. This interaction ensures that the client can efficiently upload files to the distributed file-sharing system by leveraging the load balancer and the Chord protocol for file storage.

The below sequence diagram for the file download process in the distributed file-sharing system shows the interaction between the user interface, client, DNS server, load balancer, and Chord nodes.

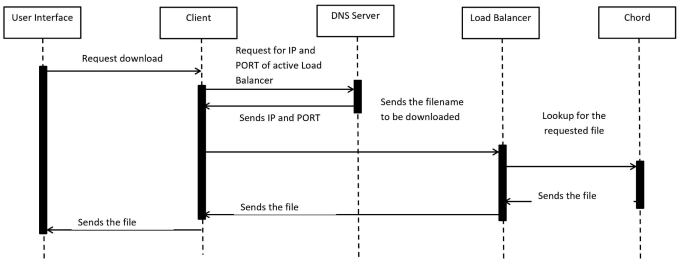


Fig.5 Sequence diagram(File Download)

The user initiates the download request through the user interface. This request is forwarded to the client. The client receives the download request and sends a request to the DNS server to obtain the IP and port of an active load balancer. The DNS server processes the client's request and responds with the IP and port of an available load balancer. The client receives the IP and port details from the DNS server and sends the filename of the file to be downloaded to the load balancer. The load balancer receives the filename and initiates a lookup for the requested file within the Chord network. The Chord network, upon receiving the lookup request, locates the file and sends it back to the load balancer. The load balancer receives the file from the Chord network and sends it back to the client. The client receives the file from the load balancer and delivers it to the user interface, completing the download process.

VII. Evaluation

A. Approach

Chord is a decentralized protocol for distributed hash tables (DHTs) that provides an efficient way to locate resources in a peer-to-peer network. It is commonly used in distributed file sharing systems to manage file storage and retrieval across a network of nodes. Chord is highly scalable and can handle a large number of nodes efficiently. As the number of nodes increases, the lookup time for resources remains logarithmic, ensuring that the system can scale without a significant decrease in performance.

Chord provides fault tolerance by replicating resources across multiple nodes and maintaining successor lists. If a node fails, its responsibilities are transferred to its successors, ensuring continuous availability of resources.Chord's decentralized nature allows for easy addition and removal of nodes without disrupting the entire system. This makes it well-suited for dynamic environments such as cloud deployments where nodes may join or leave frequently.

Chord's lookup algorithm is efficient, with a time complexity of O(log N), where N is the number of nodes. This ensures that resource lookup remains fast even as the system grows. By distributing resources across multiple nodes, Chord helps in load balancing, preventing any single node from becoming a bottleneck.

The correctness of Chord's algorithms is ensured by its design principles and the use of consistent hashing for key assignment. The decentralized nature of the protocol allows for fault tolerance and scalability while maintaining correctness. The complexity of Chord's algorithms is primarily determined by its lookup process, which has a time complexity of O(log N) and a message complexity of O(log N). This ensures that as the network grows, the lookup time for resources remains reasonable.

In conclusion, Chord is a good approach for a Distributed File Sharing System deployed on the cloud due to its scalability, fault tolerance, decentralized nature, efficient lookup, load balancing, and consistency features. Its algorithms are intuitive, correct, and have a manageable complexity, making it suitable for large-scale distributed systems.

B. Issues Addressed

1) Fault Tolerance: Resources (files) are replicated across multiple nodes. The replication factor determines how many copies of each resource are maintained. This ensures that if one node fails, other nodes still have copies of the resource. Each node maintains a list of successor and predecessor If a node fails, its responsibilities are transferred to its immediate successor, ensuring continuous availability. Nodes periodically check the health of their neighbors. If a node detects that its successor has failed, it can update its successor list and transfer the responsibilities to maintain the system's integrity.

2) Performance: Chord provides efficient resource lookup with logarithmic time complexity (O(log N)), where N is the number of nodes. The finger table at each node helps in quickly routing lookup requests to the correct node. Load balancers distribute incoming client requests across multiple chord nodes to ensure even load distribution and prevent any single node from becoming a bottleneck.

3) Scalability: Chord's decentralized and peer-to-peer nature allows it to scale seamlessly. Nodes can join or leave the network without significantly affecting the system's overall performance. The use of consistent hashing ensures that when new nodes join or existing nodes leave, only a minimal amount of data needs to be redistributed, allowing the system to handle a large number of nodes efficiently. Nodes can be added incrementally, and the system will automatically adjust by redistributing keys and updating finger tables, ensuring smooth scalability.

4) Concurrency: Nodes are capable of handling multiple concurrent requests, thanks to their decentralized nature. Each node can independently process requests, reducing bottlenecks.

5) Security: Data encryption (both at rest and in transit) protects the data from unauthorized access and tampering. Nodes can use secure communication protocols (e.g., TLS/SSL) for data transmission. The use of secure hash functions in key generation and lookup ensures that the data distribution and retrieval process is secure and tamper-proof.

VIII. Implementation Details

A. Architecture Styles

The architecture style of our distributed file sharing system is a combination of several architectural patterns, with a focus on distributed systems and peer-to-peer networking.

1) Three-Tier Client-Server Architecture: This architecture is observed from the relationship between the clients and the rest of the system components. Clients, serving as end-user applications, interact with the system by making requests to the server components such as the Load Balancer and DNS Server. The server components handle these requests and in turn make requests to the resource management layer and provide responses back to the clients.

2) Peer-to-Peer (P2P) Architecture: The use of the Chord protocol for distributed hash table (DHT) storage and file sharing introduces a peer-to-peer architecture aspect to the system. In a P2P architecture, nodes (Chord nodes, in this case) communicate directly with each other to share resources (files) without relying on centralized servers. This decentralization enables scalability, fault tolerance, and resilience to network failures.

3) Microservices Architecture: The system's design, with components like the Client, DNS Server, Load Balancer, and Chord, depicts a microservices architecture style. Each component serves a specific function or set of functions, encapsulating its logic and communicating with other components through well-defined interfaces.

4) Event-Driven Architecture (EDA): The system can also be thought of as an event-driven architecture. For example,

clients trigger events (such as file uploads or downloads) that are handled asynchronously by the system's components.

5) Scalable Architecture: The use of load balancers and the Chord protocol in our system reflects a scalable architecture that can handle a large number of concurrent users and efficiently distribute workload across multiple nodes. This scalability is crucial for ensuring optimal performance as the system grows in size and usage.

B. UML Diagrams:

1) Class Diagram

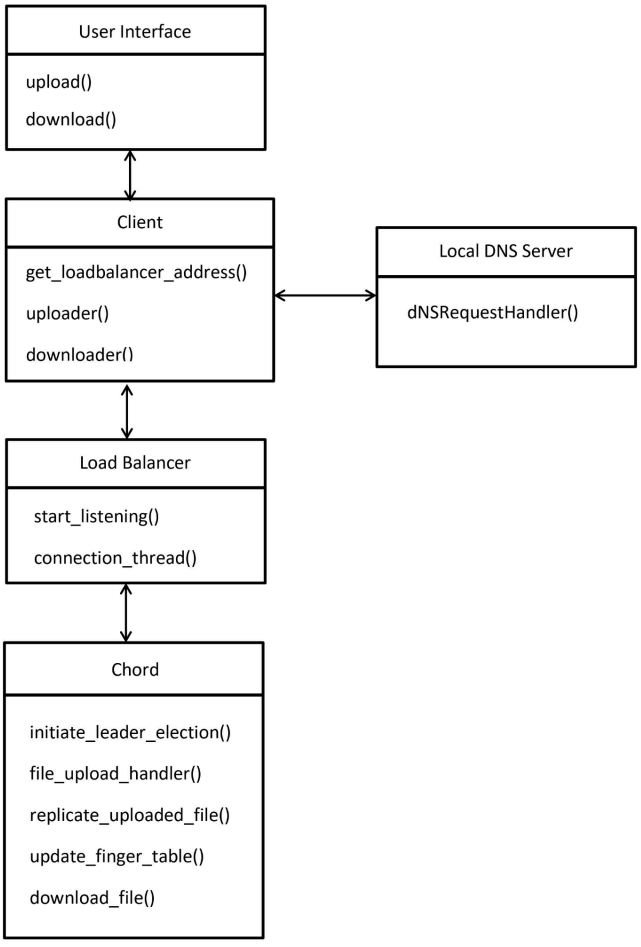


Fig.6 Class diagram

The above class diagram delineates the foundational architecture of our system, encompassing various components facilitating file uploading and downloading operations. The User Interface class initiates these processes through methods like upload() and download(), while the Client class, interacting with it, manages functionalities such as obtaining the load balancer address and executing upload and download operations. A Local DNS Server routes user requests to load balancers, employing round-robin allocation. The Load Balancer class, pivotal in managing traffic, initiates operations and handles individual requests in a multi-threaded setup. Additionally, the Chord class orchestrates distributed file management utilizing a Chord distributed hash table, encompassing functionalities like updating the finger table, leader election, file handling, replication, and download, along with network operations and failure detection mechanisms. Interactions among these classes, depicted by arrows, illustrate the interconnectivity facilitating seamless system functionality, with the User Interface serving as the primary user entry point.

2) Object Diagram:

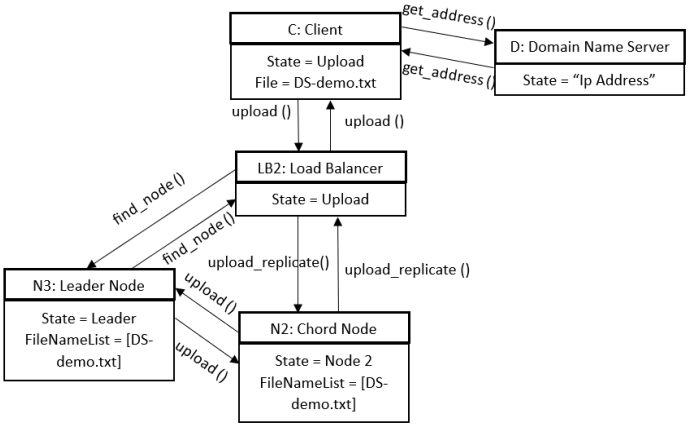


Fig.7 Object Diagram for upload

The object diagram above illustrates the upload process of a file (DS-demo.txt). The client initiates this process by preparing the file for upload and interacting with the load balancer(LB1) to manage the request. The load balancer redirects the upload request to an appropriate node in the network. The leader node (N3) coordinates the upload process and identifies the node where the file should be stored. The appropriate chord node (N2) receives the file from the load balancer, updates its state to include the new file, and sends a replication request to its successor node (N3).

This object diagram illustrates the download process of a file (demo.txt). The client initiates this process by requesting the specific file and interacting with the load balancer (LB2) to manage the request. The load balancer forwards the request to the leader node (N3), which identifies the node containing the file. The leader node then redirects the download request to the appropriate node (N1). Finally, the load balancer retrieves the file from node N1 and delivers it to the client.

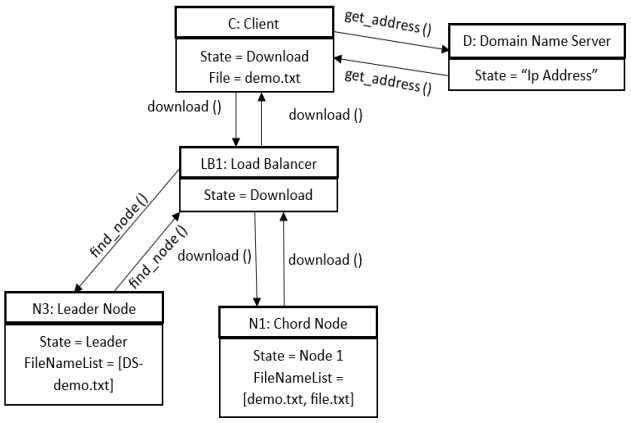


Fig.8 Object Diagram for download

C. Software Components:

The software components and services interact with each other using different protocols and APIs. The user interface is implemented using Flask, which is a Python web framework. Flask allows for the creation of web applications and provides tools and features to make web development easier. It uses HTTP (Web Services) to interact with the client. The Client component interacts with the User Interface through HTTP (Web Services), as mentioned above. It also interacts with the Local DNS Server and Load Balancer using Socket APIs. Socket APIs provide a low-level interface for network communication, allowing the Client to send and receive data over the network.

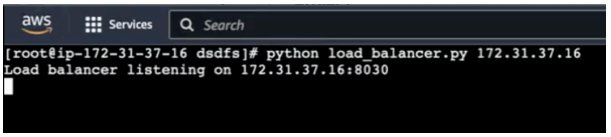
The DNS server component is implemented using dnslib, a Python library for working with DNS. The dnslib library provides tools for generating custom DNS resolvers and encoding/decoding DNS packets. It interacts with other components using socket APIs. The Load Balancer interacts with the Client and the Chord system using Socket APIs. The Chord components interact with each other and with the Load Balancers using Socket APIs. It implements a distributed hash table, providing services for storing and retrieving data in a distributed system.

The software we used for implementing our system includes Python, Flask Web Framework, and Socket APIs. Python is a versatile programming language that can be used for a wide range of applications, including web development, distributed systems, and more. Its simplicity and readability make it a popular choice. There are abundant resources, libraries, and frameworks available for various purposes. This made development in Python more efficient and allowed easier integration with other tools and technologies. Flask Web Framework provides a simple and intuitive way to develop web applications. It offers features like routing, template rendering, and request handling, making it easier to build web-based interfaces. It uses HTTP (Web Services) to interact with the client. Socket APIs, such as those provided by Python, enable real-time communication between clients (e.g., web browsers) and servers. This is crucial for building interactive and dynamic applications. Socket APIs communicate directly with the operating system's TCP/IP protocol stack, allowing them to work independently from any application running on the host. This provides flexibility and efficiency in handling network communication between components. It supports various forms of communication like request response and peer-to-peer.

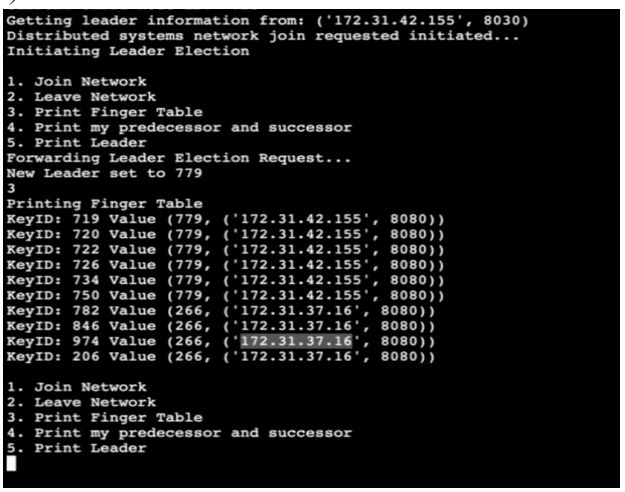
VIII. DEMONSTRATION

A. Component Startup:

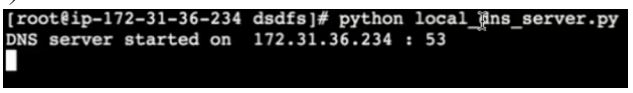
1) LoadBalancer:



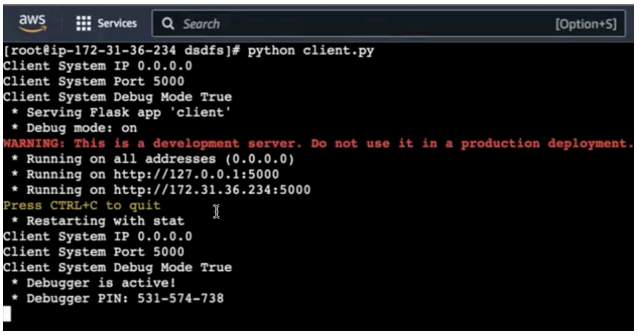
2) Node:

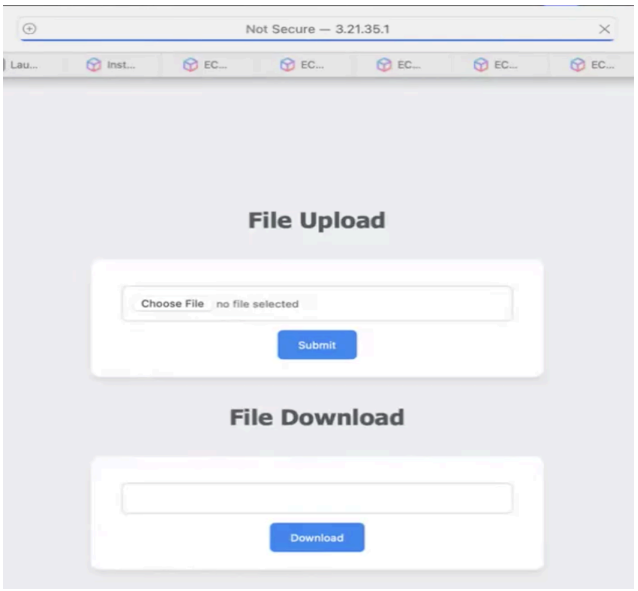


3) DNS Server:



4) Client:



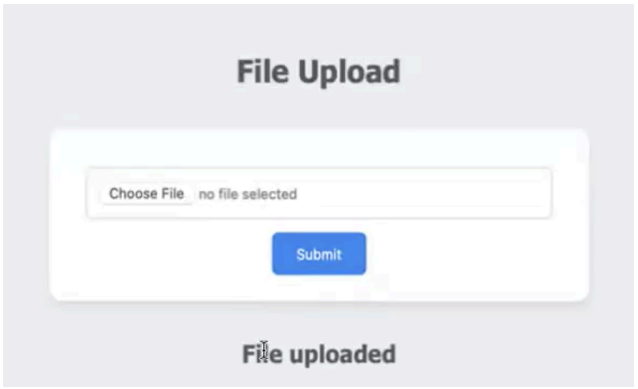


All Nodes are up and running successfully.

B. Successful Scenario:

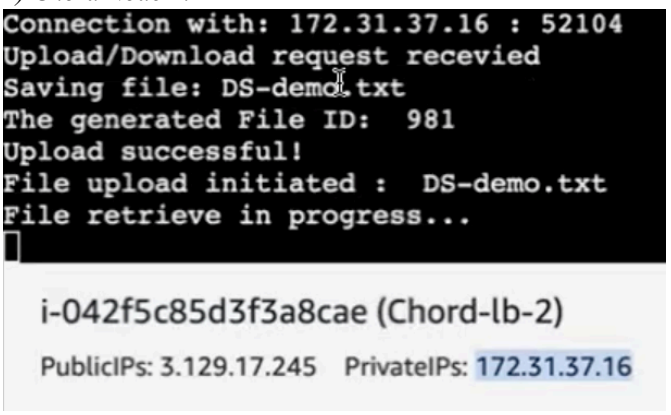
The below figures depict the happy flow of uploading of files from client to chord nodes.

1) Client:



The client chooses the file to upload and chooses upload to make a upload request to the server.

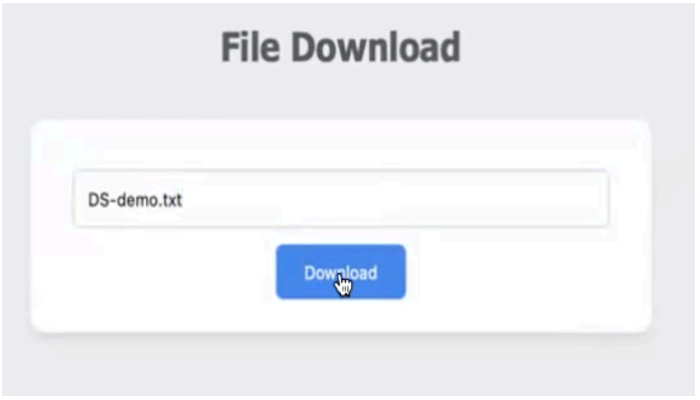
2) Chord Node 2:



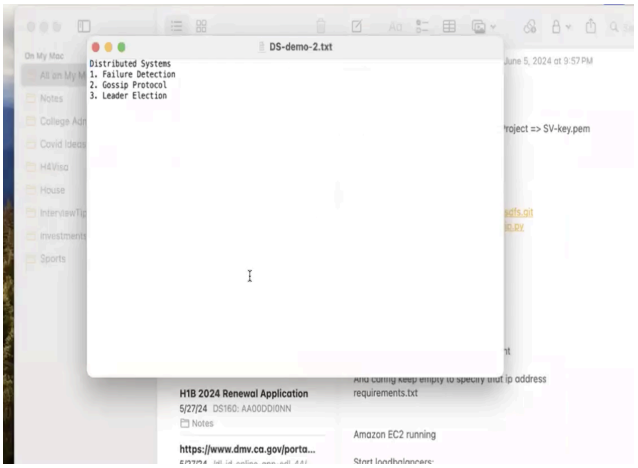
3) Chord Node 3:

The below figures depict the happy flow of downloading of files from client to chord nodes.

1) Client:



The client enters the file name and selects download to make a download request to the server.



Results: Clients can upload and download files successfully.

B. Failure Scenarios:

The below snapshots depict how the system responds when one of the nodes crashes.

1) Node Failure:

```
KeyID: 282 Value (779, ('172.31.42.155', 8080))
KeyID: 298 Value (779, ('172.31.42.155', 8080))
KeyID: 330 Value (779, ('172.31.42.155', 8080))
KeyID: 394 Value (779, ('172.31.42.155', 8080))
KeyID: 522 Value (779, ('172.31.42.155', 8080))
KeyID: 778 Value (779, ('172.31.42.155', 8080))

1. Join Network
2. Leave Network
3. Print Finger Table
4. Print my predecessor and successor
5. Print Leader
Forwarding Leader Election Request...
New Leader set to 779
Connection with: 172.31.37.16 : 52104
Upload/Download request recevied
Saving file: DS-demo.txt
The generated File ID:  981
Upload successful!
File upload initiated :  DS-demo.txt
File retrieve in progress...
Connection with: 172.31.42.155 : 33900
Upload/Download request recevied
Download request for file: DS-demo.txt
File retrieve in progress...
^CTraceback (most recent call last):
  File "/home/ec2-user/Distributed-System/dsdfs/chord_node.py", line 611, in <module>
    chord_node.spin_up()
  File "/home/ec2-user/Distributed-System/dsdfs/chord_node.py", line 596, in spin_up
    self.config_thread()
  File "/home/ec2-user/Distributed-System/dsdfs/chord_node.py", line 571, in config_thread
    action = input()
KeyboardInterrupt
^CException ignored in: <module 'threading' from '/usr/lib64/python3.9/threading.py'>
Traceback (most recent call last):
  File "/usr/lib64/python3.9/threading.py", line 1468, in _shutdown
    lock.acquire()
KeyboardInterrupt:

[root@ip-172-31-37-16 dsdfs]#
```

The Node is crashed manually by abruptly stopping the node.

2) Node Detecting Failure:

```
Leader Election complete

1. Join Network
2. Leave Network
3. Print Finger Table
4. Print my predecessor and successor
5. Print Leader

Succesor node is offline!
Self Stabilization in progress...
Failed successor: ('172.31.37.16', 8080)

1. Join Network
2. Leave Network
3. Print Finger Table
4. Print my predecessor and successor
5. Print Leader
```

The failure of the node gets detected by predecessor nodes using the Ping-Ack Protocol. The node subsequently initiates the Stabilization protocol.

Result: The failed node is detected and finger tables are stabilized. Additionally, client upload/download still works seamlessly.

The second failure scenario depicts the case when a node leaves the network.

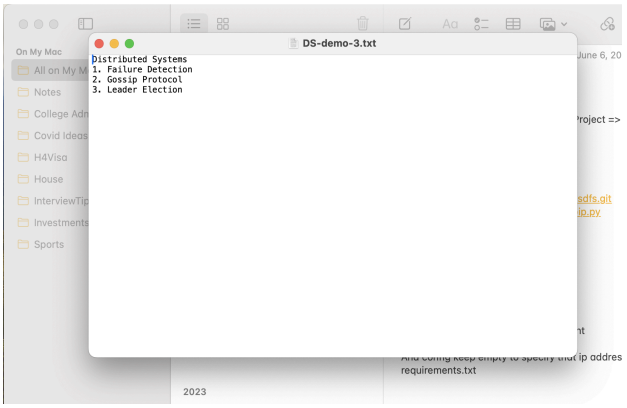
2) Node Leaving Network:

```
1. Join Network
2. Leave Network
3. Print Finger Table
4. Print my predecessor and successor
5. Print Leader
Connection with: 172.31.37.16 : 55988
Upload/Download request recevied
Saving file: DS-demo.txt
The generated File ID:  981
Upload successful!
2
Replicating the files to neighboring nodes before leaving
File list for replication: ['DS-demo.txt']
File retrieve in progress...
File replicated
Leaving the network...
Node: 718
 Address: ('172.31.38.210', 8080)

1. Join Network
2. Leave Network
3. Print Finger Table
4. Print my predecessor and successor
5. Print Leader
```

Result: The node leaving the network replicates all of its files to other nodes, and initiates leader election if the node was the leader.

The third failure scenario depicts when one of the nodes fails and its successor leaves, as shown above, the file gets replicated to other nodes.



Result: The client is still provided with a seamless downloading of file.

IX. PERFORMANCE ANALYSIS

A. Overall Result and Reflection on Design Choice

The Distributed File Storage System implemented using our key design features listed leverages several components that influence its overall performance.

The Distributed Hash Table (DHT) lookup and storage operations show a logarithmic growth rate. Socket communication adds network overhead, but the overall impact should be minimal for typical file operations. Concurrent requests for DHT operations were handled efficiently by leveraging threads. Each thread manages a separate lookup or storage request, improving overall throughput. However, this came with challenges in contention for network resources and potentially slow down communication which were handled by implementing mechanisms like thread pools and non-blocking I/O to help mitigate this by managing concurrent socket operations efficiently.

Leader election algorithm's choice remains crucial. We implemented the ring election protocol to achieve this ensuring efficient leader selection even with a growing number of nodes. Leader election typically occurs periodically or upon leader failure. But our design made sure that the users were not directly impacted by the leader 's election. This was implemented to ensure leader election doesn't significantly delay file operations.

The performance impact of replication and resource discovery remains constant. However, socket communication adds overhead during data replication across nodes which is comparatively minimum considering the fact that any form of messaging will require bandwidth for communication.

By carefully selecting algorithms and implementing efficient thread management with techniques like thread pools and non-blocking I/O, our DFSS achieves good scalability. Optimizing thread usage and network communication strategies still remains crucial for maintaining good performance under increasing user load and concurrent requests. Network bandwidth limitations can significantly impact performance, especially for large file transfers. Also, the processing power and memory capacity of nodes will influence overall system responsiveness. The type and frequency of file operations (uploads/downloads/replication) affects the performance as we will see in the results section.

B. Measurements Metrics to Analyze

Here's a breakdown of key metrics we considered to analyze our system's functionality:

Performance: Latency and Bandwidth Utilization

Average Latency: To measure the average time it takes for a file operation (lookup, upload, download) to complete.

Bandwidth Utilization: To track the network bandwidth consumed by the system during file transfers. This metric helps identify potential bottlenecks. By collecting and analyzing these metrics, we gained a comprehensive understanding of DFSS's performance and identified areas for improvement.

Scalability: We measured operation time with an increase in the number of requests coming into the distributed network to track the average time for critical operations (upload, download). This helps measure the system's ability to handle concurrent file requests as the number of users accessing the system grows.

Availability and Reliability: We measured average time to detect failure of nodes through the Ping ACK implementation. This gives a view into how fast crash or unavailability of nodes is identified by the distributed nodes. This also addressed the Mean Time to Repair by triggering replication of files so that access to files that were uploaded is still valid.

C. Simulation of Workloads and Performance Measurement

The table showcases a stress testing approach to simulating workloads. Stress testing involves placing a high load on the system to assess its performance.

We gradually increase the number of concurrent file operations (number of files) to simulate a growing number of users uploading and downloading files. This helps evaluate how the system performs under heavy load.

Average latency for upload and download operations with 2 nodes on local system:

No. of Files	Upload Latency	Download Latency
1 file	4.011 sec	2.062 sec
10 files	4.148 sec	2.101 sec
20 files	4.215 sec	2.152 sec
50 files	4.68 sec	2.190 sec

Table1. Avg latency in local system

Average latency for upload and download operations with 3 nodes on AWS EC2 instances with 3 nodes in the network:

No. of Files	Upload latency	Download Latency
1 file	0.488 sec	0.282 sec
10 files	1.021 sec	0.644 sec
20 files	1.652 sec	0.834 sec
50 files	4.423 sec	2.835 sec

Table.2.Avg Latency in AWS EC2 with 3 nodes

Average latency for upload and download operations with 6 nodes on AWS EC2 instances with 5 nodes in the network:

No. of Files	Upload latency	Download Latency
1 file	0.410 sec	0.201 sec
10 files	0.981 sec	0.638 sec
20 files	1.282 sec	0.804 sec
50 files	3.823 sec	2.118 sec

Table.3 Avg Latency for AWS EC2 with 5 nodes

X. TESTING RESULTS

Here are the test cases designed to evaluate the functionality and performance of your Distributed File Sharing System (DFSS), focusing on the key design features you mentioned:

Test Case 1: Basic File Upload and Download to verify basic file upload and download functionality.
Steps:
i. Upload a file to the system.
ii. Download the uploaded file.
iii. Verify the downloaded file is identical to the original file.
Outcome: The file upload and download worked successfully, and the downloaded file contained the same content as the uploaded file.

Test Case 2: Concurrent File Upload and Download to assess the system's ability to handle concurrent file accesses.
Steps:
i. Simulate multiple users uploading and downloading different files concurrently.
ii. Monitor the system's performance metrics (latency)
Outcome: Uploads and downloads worked concurrently. Latency was within acceptable limits but grew with increase in incoming traffic to the distributed system.

Test Case 3: Leader Election to verify the leader election process and leader functionality.
Steps:
i. Simulate a node failure or network disruption that triggers leader election.
ii. Verify a new leader is elected successfully.
iii. Perform file upload/download operations and ensure they are routed to the new leader.
Outcome: A new leader was elected promptly after a failure. File operations continued as expected seamlessly after the leader election.

Test Case 4: Data Replication to verify data replication across the network.
Steps:
i. Upload a file to the system.
ii. Simulate a node failure.
iii. Verify the uploaded file remains accessible and can be downloaded from another node.
Outcome: The replicated data was accessible even after a node failure. Downloads succeeded using data from the predecessor/successor nodes.

Test Case 5: Resource Discovery to assess the efficiency of resource discovery mechanisms.
Steps:
i. Add new nodes to the network.
ii. Verify new nodes are discovered and integrated into the system promptly.
iii. Perform file operations and ensure they are routed to the appropriate nodes.
Outcome: New nodes were discovered as soon as it was added to the network. File operations worked successfully to utilize resources from the newly discovered nodes. Finger tables also got updated at all neighboring nodes.

Test Case 6: This test evaluates the system's performance for handling large file transfers.
Steps:
i. Upload a large file to the system.
ii. Download the large file.
iii. Monitor the time taken for upload and download operations.
Outcome: The system handles large file transfers successfully. The upload and download times increased with the increase in

file size.

Test Case 7: System Scalability to assess how the system performs with increasing load and number of nodes.
Steps:
i. Gradually increase the number of concurrent users and file operations.
ii. Monitor performance metrics (latency) as the load increases.
Outcome: The system maintains acceptable performance levels even with increased load. Latency should increase gradually.

Test Case 8: System Failure Recovery to evaluate the system's ability to recover from node failures.
Steps:
i. Simulate a node failure or network disruption.
ii. Monitor the system's recovery process and downtime.
iii. Verify the system resumes normal operation after recovery.
Outcome: The system recovers from failures promptly as the files get replicated.

XI.CONCLUSIONS

The project has successfully addressed the challenge of efficient and secure content management through the development of a Distributed File Sharing System (DFSS). This system allows users to effortlessly store, retrieve, and share files across a distributed network. Users can access their data from any location with an internet connection, promoting collaboration and remote access. The innovative design prioritizes scalability, performance, and reliability, ensuring smooth operation even with a growing user base and data volume.

A. Lessons Learned

Challenges of Distributed Systems: Distributed systems introduce unique challenges compared to centralized systems. Issues like concurrency control, data consistency across nodes, and leader election require careful design and implementation strategies.

Trade-offs and Optimization: Developing a DFSS often involves making trade-offs between different functionalities. Factors like security, performance, and scalability may need to be balanced to achieve the desired system characteristics.

Testing and Evaluation: Thorough testing and evaluation are essential for identifying weaknesses, bottlenecks, and potential security vulnerabilities. Implementing a comprehensive test suite with diverse test cases is crucial for ensuring system reliability and performance under various load conditions.

Collaboration and Communication: Effective collaboration and communication are essential for successful project development. Clear communication within the development team and with potential users ensures everyone is aligned with project goals and design choices.

B. Possible Improvements

Here's an elaboration on the future enhancements identified for the Distributed File Sharing System (DFSS):

Access Control Mechanisms: Refine access control mechanisms to allow granular control over user permissions. Users can be assigned roles (owner, editor, viewer) to grant specific access levels to different files or folders. This enhances security by ensuring only authorized users can access and modify data.

Caching Mechanisms: Introduce content caching strategies to improve data retrieval speeds. Frequently accessed files can be cached on specific nodes or user devices, reducing the need to retrieve them from remote locations and minimizing latency.

Web Interface Enhancements: Refine the web interface for user-friendliness. Consider features like drag-and-drop functionality for file uploads, intuitive search options, and progress bars for file transfers. These enhancements can significantly improve user experience.

Support for Different File Types: Explore extending support for different file types beyond common document formats. This could involve integrating with specialized storage solutions for multimedia files (audio, video) or scientific data formats.

Automated File Management Features: Investigate implementing features like automatic file versioning or conflict resolution for concurrent edits. This can help users manage file revisions and avoid data loss due to conflicting edits.

To summarize, with its efficient file storage methods and distributed search algorithms, the DFSS surpasses traditional file management solutions. Its ability to solve a wide range of content management problems makes it a valuable tool for personal and professional use. Future enhancements like mobile applications and integration with cloud storage providers hold the potential to further expand its functionality and user base. This DFSS represents a significant step forward in distributed file management, offering a secure, scalable, and user-friendly platform for storing, retrieving, and sharing content in today's increasingly digital world.

XII. CITATIONS

1. IEEE. (2020). “A Scalable - High Performance Lightweight Distributed File System” In 2020 7th NAFOSTED Conference on Information and Computer Science (NICS). https://ieeexplore.ieee.org/document/9335887?signout=success
2. IEEE. (2023). “Survey of Distributed File Systems: Concepts, Implementations, and Challenges” In 2023

14th International Conference on Information and Communication Technology Convergence (ICTC). https://ieeexplore.ieee.org/document/10392447
3. IEEE. (2023). “Decentralized File Sharing 2023” In 2023 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS). https://ieeexplore.ieee.org/document/10062977
4. IEEE. (2024). “Challenges and Solutions for Distributed File System” In 2024 2nd International Conference on Disruptive Technologies (ICDT). https://ieeexplore.ieee.org/document/10489789
5. IEEE. (2019). “A Novel Approach for Maintaining Consistency in Distributed File System” In 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP) https://ieeexplore.ieee.org/document/8882935
6. IEEE. (2023). “Distributed File Systems for Cloud Storage Design and Evolution” In 2023 First International Conference on Advances in Electrical, Electronics and Computational Intelligence (ICAEECI) https://ieeexplore.ieee.org/document/10370956