DISTRIBUTED HASH TABLE BASED FILE SHARING SYSTEM USING CHORD ALGORITHM

Shaunak Chaudhary, Rahim Firoz Chunara, Iniyan Chandran Ramachandran
Department of Computer Science and Engineering
Santa Clara University
{schaudhary2, rchunara, iniyan}@scu.edu
W1635919, W1649825, W1651510

***Abstract*— This report covers the design and implementation of a peer-to-peer file sharing system that doesn't rely on central servers. We used the Chord protocol, which is based on distributed hash tables (DHTs), to enable efficient lookup of files across a decentralized network of nodes. Each node in the network maintains a finger table with routing information to locate the successor node responsible for a given file. This allows files to be efficiently located and retrieved from other nodes in the network. We implemented file replication strategies to ensure data availability, and concurrency control mechanisms to allow multiple users to access files simultaneously. Key features of the system include the ability for nodes to dynamically join or leave the network, upload or download files, and maintain fault tolerance through monitoring and stabilization routines. The decentralized nature of the system eliminates single points of failure and provides a scalable solution for file sharing without relying on central servers. Experimental results demonstrate that the system can effectively distribute file storage and retrieval across the participating peers in the network. Overall, this report presents a robust and decentralized approach to peer-to-peer file sharing.**

***Index Terms*— Chord, DHT, Barnes-Hut, Parallelization**

I. INTRODUCTION

Centralized client-server architectures for file sharing systems possess inherent limitations in terms of scalability, reliability, and potential single points of failure. This has motivated research into decentralized peer-to-peer (P2P) approaches that distribute the responsibilities of data storage and lookup across multiple nodes in the network. Distributed Hash Tables (DHTs) have emerged as a powerful technique to enable efficient resource discovery and routing in structured P2P networks without relying on central coordination.

This paper presents the design and implementation of a DHT-based file sharing system built using the Chord protocol. Chord employs consistent hashing to map data identifiers onto nodes, forming a logical ring topology. Each node maintains a finger table containing routing information about other nodes, which allows queries for a given file to be directed towards the responsible successor node via O(log N) hops.

The proposed system incorporates several key mechanisms to facilitate decentralized file sharing. Replication strategies replicate files across multiple successors to enhance data availability. Concurrency control through techniques like file locking or leases enables parallel access to shared files. Dynamic node join and leave operations preserve routing consistency as peers enter or depart the network. Periodic stabilization routines detect failed nodes and update successor pointers to maintain fault tolerance.

Through simulations and experimental evaluations, the efficacy of the DHT-driven architecture is demonstrated in terms of distributing storage load, efficient lookup performance, resilience to node failures, and the ability to scale as nodes join or leave the system. The decentralized nature eliminates centralized bottlenecks and single points of failure, making it a promising approach for building robust and scalable file sharing platforms.

II. LITERATURE SURVEY

A. Previous Work

Distributed hash tables (DHTs) have been instrumental in enabling decentralized peer-to-peer file sharing systems that operate without relying on central servers. Popular platforms like BitTorrent and IPFS have successfully leveraged DHTs to facilitate the exchange of files across a decentralized network of nodes.

However, the applications of DHTs extend beyond these well-known examples. Various projects have explored customized implementations of DHTs to cater to specific use cases and objectives. For instance, Freenet, Tribler, Hyperspace, and Ares Galaxy have employed DHTs to enable decentralized file sharing while prioritizing goals such as censorship resistance, privacy, optimized routing algorithms, and improved usability, respectively. Even the pioneering Napster system, which predates many of these projects, incorporated a DHT-driven architecture for decentralized file downloading.

The diverse range of applications and implementations highlights the adaptability and versatility of DHTs in the realm of decentralized file storage and retrieval. Ongoing research efforts aim to further enhance critical aspects of DHT-based systems, such as scalability, efficiency, and security, to meet the evolving demands of various applications.

Additionally, the decentralized nature of DHTs aligns well with the principles of peer-to-peer networks, promoting resilience, load balancing, and fault tolerance. As research in this field continues, we can expect to see innovative approaches that leverage DHTs to facilitate decentralized file sharing while addressing challenges related to performance, privacy, and security.

It is worth noting that while DHTs have proven to be a powerful tool for decentralized file sharing, alternative approaches, such as blockchain-based systems and decentralized storage networks, are also being explored. The interplay between these technologies and their potential synergies could lead to new advancements in the field of decentralized data storage and distribution.

B. Literature Survey

The groundbreaking paper by Stoica et al. [6] on the Chord protocol marks a pivotal moment in distributed systems research. Chord simplifies the design of distributed hash tables by using a consistent hashing technique to distribute keys evenly across the nodes in the network. This design ensures that even as nodes join or leave the system, the average number of keys for which any node is responsible changes minimally. The protocol's elegance lies in its ability to provide a fast (O(log N) complexity) and reliable method for node lookup, crucial for scalable peer-to-peer applications. This foundational work has set the stage for numerous advancements in DHTs, emphasizing the importance of efficient routing, fault tolerance, and load balancing in distributed systems.

Puttaswamy and Zhao [1] extend the conversation around DHTs by advocating for unstructured approaches to better meet the demands of large-scale and diverse application needs. Their argument is based on the observation that structured DHTs, while efficient in lookup operations, may not adequately support the dynamic nature of real-world applications where node churn, heterogeneous capacities, and varied latency requirements are common. By proposing unstructured DHTs, they highlight the potential for improved routing efficiency and resource discovery, particularly in environments where flexibility and adaptability are paramount. This work encourages a reevaluation of DHT architectures to better align with the evolving landscape of distributed computing and peer-to-peer networks.

Xu et al. [2] tackle the challenges of file sharing in DHT-based peer-to-peer (P2P) systems, focusing on scalability and the efficient distribution of resources. They propose strategies that mitigate common issues in P2P systems, such as hotspots and imbalanced resource distribution. By optimizing file sharing, their work contributes to making DHT networks more robust and efficient, reducing the overhead associated with file lookup and transfer. This research is significant for its practical implications, offering solutions that enhance the performance of file-sharing applications and, by extension, the user experience in P2P systems.

The integration of DHTs with social networking, as presented by Shen et al. [3], represents an innovative application of distributed systems technology. Social-P2P leverages the decentralized nature of DHTs to create a social networking framework that is both scalable and privacy-preserving. This approach addresses some of the key concerns in online social networks, such as data ownership and user privacy, by distributing social media content across a P2P network. The work by Shen et al. showcases the versatility of DHTs and opens up new avenues for their application, emphasizing the potential for decentralized technologies to reshape social interactions on the internet.

The development of a distributed cache for the Hadoop Distributed File System (HDFS) by Zhang et al. [4] illustrates the utility of DHTs in cloud computing environments. By improving real-time file access through a distributed cache that leverages DHTs, this work addresses a critical bottleneck in cloud storage systems: latency. The significance of this research lies in its demonstration of how DHTs can enhance the scalability and performance of cloud services, making them more suitable for real-time applications. This contribution is particularly relevant as the demand for cloud computing continues to grow, highlighting the need for innovative solutions to improve data access times and system responsiveness.

Riley and Scheideler [5] explore the use of DHTs within computational grids, focusing on the efficient management of distributed computing resources. Their research underscores the adaptability of DHTs to a wide range of computational and storage tasks, from job scheduling to data storage. By applying DHTs to grid computing, they illustrate how these structures can facilitate resource management in distributed environments, improving the efficiency and scalability of computational grids. This work contributes to a broader understanding of the potential for DHTs to optimize distributed computing beyond traditional peer-to-peer file sharing or data storage applications.

III. Distributed System Challenges Addressed

A. Scalability

Dynamic Node Management: Our implementation supports dynamic node entry and exit with minimal disruption. The joinNode method allows new nodes to integrate seamlessly into the network, re-calibrating the network's topology without significant overhead. This flexibility is crucial for scalable systems, as it allows the network to expand or contract based on demand.

Efficient Finger Tables: The creation and maintenance of finger tables in our code significantly reduce the number of steps required to locate a node responsible for a particular key. Since the size of the finger table is logarithmic in relation to the number of nodes, the increase in the network size does not linearly increase the complexity of lookup operations, thus preserving the system's scalability.

B. Fault Tolerance

Our system demonstrates fault tolerance in several ways. It continuously monitors the immediate successor of each node through periodic "ping" operations. This allows the system to detect and recover from node failures quickly. When a failure is detected, the system initiates a stabilization process to update the predecessor and successor links, ensuring the network remains connected and operational. The use of a finger table also indirectly contributes to fault tolerance by providing multiple pathways for finding a key, thus allowing the system to handle node failures gracefully.

C. Efficiency in Data Lookup

The efficiency of data lookup in our system is achieved through the implementation of the finger table. Each node maintains a finger table with entries pointing to other nodes in the system, allowing for fast routing of queries. The lookup process uses these entries to reduce the distance to the target key by half with each step, on average, leading to an efficient O(log N) lookup time, where N is the number of nodes in the system. This design ensures that even as the network grows, the efficiency of data lookup is maintained.

D. Load Balancing

Uniform Key Distribution: The use of a consistent hashing function for node and key mapping ensures a uniform distribution of responsibilities across the network. This distribution helps in preventing any single node from becoming overloaded with too much data or too many requests.

Replication and Redundancy: While our code primarily focuses on establishing a robust Chord protocol foundation, the principles of replication and redundancy are implied through file transfer mechanisms (transferFile). Implementing data replication across multiple nodes can further enhance load balancing and data availability, a direction we might consider for extending our project.

IV. KEY DESIGN GOALS

A. Design Goals

1) Heterogeneity:

a) Using Python for Cross-Platform Support: We chose Python as our development language, fully capitalizing on its robust cross-platform support. This decision was strategic, ensuring our system could be deployed across various operating systems—be it Windows, Linux, or macOS—without requiring any modifications to our code base. This universal compatibility is crucial for a distributed system where nodes could span across different technological ecosystems. Our reliance on Python's standard libraries, including socket for networking, threading for managing concurrent operations, and hashlib for implementing consistent hashing, was instrumental. These libraries are meticulously designed to be agnostic to the underlying platform, thereby reinforcing our system's heterogeneity.

b) Universal Networking with TCP/IP: The use of the TCP/IP protocols, implemented through Python's socket library, was a crucial design decision for our project. This industry-standard approach ensured that our nodes could communicate effectively across different network infrastructures, ranging from the vast expanse of the global internet to localized network environments within organizations or homes. By adhering to these widely adopted protocols, we eliminated the need for specialized configurations or proprietary communication mechanisms. Our system seamlessly integrates with existing network setups, allowing for a plug-and-play experience that abstracts away the complexities of network communication. This choice not only enables our system to function reliably across various network topologies but also future-proofs our solution, ensuring compatibility with evolving network technologies and standards. By leveraging the ubiquitous nature of TCP/IP, we have created a system that can be readily deployed and scaled without encountering barriers or compatibility issues. Furthermore, the socket library in Python provided us with a straightforward and powerful interface to harness the capabilities of these protocols. Its ease of use and extensive documentation allowed us to focus on the core functionality of our system, rather than getting bogged down in the intricate details of network programming. In essence, our decision to integrate TCP/IP protocols through Python's socket library laid a solid foundation for a robust, flexible, and scalable system that can adapt to diverse network environments while maintaining seamless communication between nodes.

2) Openness:

a) Cross-Platform Networking: This choice ensures that our system can communicate over networks irrespective of the underlying operating system of each node. For example, the socket.socket(socket.AF_INET, socket.SOCK_STREAM) instantiation in our code is a clear indicator of utilizing TCP/IP standards for reliable, stream-oriented communication, enabling our nodes to join and interact within the distributed network seamlessly.

b) Universally Accepted Protocols: By relying on TCP/IP for our network communications, we ensure our system's ability to operate on a universal set of protocols that are widely accepted and used. This not only guarantees interoperability with other systems and technologies but also facilitates easy integration, as these protocols form the backbone of most modern network communications.

3) Security:

a) Foundational Security Measures: Our current system utilizes SHA-1 hashing provided by Python's hashlib for node and data identification. This is a foundational security measure, ensuring a level of obfuscation and integrity in our distributed network.

Integrity: The use of hashing for data verification, along with planned digital signatures, will maintain the integrity of the data as it is stored and transferred within the network.

4) Failure Handling:

a) Periodic Health Checks: Periodic Health Checks: Our system features a proactive approach to failure detection with the implementation of periodic health checks of successor nodes. The pingSucc method encapsulates this functionality within our code, where each node sends a "ping" message to its immediate successor at regular intervals. These health checks help in early detection of node failures, which is vital for maintaining the stability of the network.

Dynamic Topology Updates: Upon detection of a node failure, our system swiftly reconfigures the network topology. This is achieved by updating the predecessor and successor references, ensuring that the network remains interconnected and operational. The reconfiguration process is embedded within our failure detection mechanisms, triggering a seamless transition when a node becomes unresponsive or leaves the network.

b) Stabilization Protocol: An integral part of our failure handling strategy is the stabilization protocol that runs periodically. This protocol is responsible for updating the finger tables and verifying the immediate successor and predecessor of each node, ensuring that the system self-corrects and adapts to changes due to node failures.

Resilience through Redundancy: While not explicitly implemented in the current version of our code, our design allows for future inclusion of data redundancy mechanisms. By replicating data across multiple nodes, we can ensure that the system remains resilient to data loss, even in the event of multiple concurrent node failures.

c) System Recovery: In the face of node failures, our system is designed to ensure data persistence and accessibility. Recovery mechanisms within our code are primed to redistribute the responsibilities of a failed node to other active nodes, thereby preserving the integrity and availability of the data managed by the system.

5) Concurrency:

a) Threading for Parallel Operations: Our system utilizes Python's threading capabilities to manage concurrency, allowing multiple operations to occur simultaneously without interference. Each node in our network can handle multiple incoming and outgoing connections in parallel, which is crucial for maintaining a responsive and efficient network. Within our code, this is implemented through the threading.Thread target invocation, allowing separate threads to manage client requests and background tasks like periodic pinging and stabilization.

Thread-Safe Communication: To ensure thread safety during concurrent operations, our code is structured to handle shared resources carefully. When threads modify shared data, such as updating the finger table or altering the predecessor and successor nodes, we manage concurrent access to prevent data corruption. Python's threading library provides the necessary constructs, such as locks, to synchronize these operations, which our system utilizes when performing updates that need atomicity.

b) Scalable Connection Handling: The design of our system's concurrency model is inherently scalable. By employing threads, our system can scale to handle an increasing number of concurrent operations as the network grows. The listenThread function in our code serves as a listening socket, accepting connections and spawning new threads for each peer connection. This design allows our system to maintain high performance and low latency even as the number of nodes and user requests increases.

Resource Management: Our concurrency approach also focuses on efficient resource management. Threads are lightweight and consume fewer resources than spawning new processes, making them ideal for a distributed system where resource conservation is important. Our implementation ensures that each node uses its computational resources efficiently, managing multiple connections and operations concurrently without overwhelming the node's processing capabilities.

c) Asynchronous Operations: The concurrency in our system also allows for asynchronous operations, which are essential for distributed systems where operations can be time-consuming, such as network I/O or file transfers. Our code ensures that these operations do not block the execution of other critical tasks, maintaining the system's overall responsiveness. For instance, the transferFile method allows for asynchronous file uploads and downloads, ensuring that user requests are handled promptly while the system continues to perform other necessary functions in the background.

6) Scalability:

a) Efficient Resource Utilization: Our system's architecture is designed to efficiently manage resources, ensuring that it can handle a growing number of nodes without a proportional increase in overhead. The implementation of the Chord protocol within our code, particularly the getSuccessor and updateFTable methods, reflects this by utilizing a scalable hashing technique and maintaining a finger table that grows logarithmically with the number of nodes. This ensures that the addition of nodes incurs a minimal increase in routing complexity, allowing the system to scale gracefully.

Adaptive Load Distribution: The consistent hashing mechanism allows our system to distribute data and workload evenly across the network. As nodes join or leave, the system dynamically redistributes the keys, maintaining balanced load distribution among the available nodes. This is a pivotal feature for scalability, as it prevents any single node from becoming a bottleneck, thus enhancing the system's ability to expand.

b) Automated Node Integration: Our code facilitates automated integration of new nodes into the network. Through the joinNode method, a node can seamlessly enter the system, with its presence automatically acknowledged by existing nodes and the necessary updates made to their finger tables. This automation is crucial for scalability as it allows the system to evolve without manual intervention.

Robust Network Topology: The Chord protocol's ring structure, coupled with our implementation of finger tables, creates a robust topology that can withstand significant changes in the network size. This robustness is essential for a scalable system, as it ensures that the network remains stable and efficient despite continuous growth or contraction.

7) Transparency:

a) Simplified User Interface: In our system, we abstract the complexities of the underlying distributed operations from the end-user. Functions for joining and leaving the network, as well as for file management, are all presented through simple interfaces. This design philosophy is captured in our code through methods like printMenu, which provides users with clear and concise options for interaction, hiding the intricate processes running behind the scenes.

Abstraction of Network Mechanics: We ensure that the users are not burdened with the details of the Chord protocol mechanics or the management of the distributed network. The system's operations, such as key placement or data retrieval, are all handled transparently by our code. For example, the lookupID method encapsulates the process of locating a key within the network, providing the user with the result without exposing the underlying steps.

b) Consistent User Experience: Regardless of the network's state—whether it's expanding with new nodes or handling failures—our system maintains a consistent user experience. The complexity of these operations is managed internally within our code, such as through the pingSucc method for failure detection and the updateOtherFTables method for maintaining accurate routing information.

Streamlined Interaction Flow: Our user interaction flow is designed to be intuitive. Even as the network scales and the underlying data structures become more complex, the user interface remains straightforward, ensuring that from a user's perspective, the system's operation remains unchanged.

B. Key Components

1) Node: In the architecture of our Chord-based distributed system, the Node class stands as the cornerstone, encapsulating the essence of network participants. Each instance of this class represents an individual node within the network, meticulously designed to incorporate all necessary attributes and functionalities required for seamless operation within the distributed environment. Central to each node's identity are its IP address and port number, which not only facilitate network communication but also contribute to the generation of a unique identifier (ID) for the node. This ID is derived through the application of the SHA-1 hashing algorithm on the concatenation of the node's IP address and port, ensuring a unique and evenly distributed identifier across the Chord ring.

Upon instantiation, each Node object is equipped with references to its immediate successor and predecessor within the network, establishing a bidirectional linkage that is pivotal for the maintenance of the Chord's ring structure. These references are dynamically updated in response to nodes joining or leaving the network, thereby preserving the integrity and continuity of the ring. Furthermore, the Node class incorporates a finger table, an ordered dictionary that significantly enhances the efficiency of routing and lookup operations within the network. By maintaining shortcuts to strategically selected nodes across the ring, the finger table enables each node to drastically reduce the number of hops required to locate any given key, thereby ensuring an efficient O(log N) lookup time.

To operationalize these functionalities, the Node class is endowed with a suite of methods designed to facilitate network integration, data management, and operational maintenance. The joinNode and sendJoinRequest methods enable nodes to seamlessly integrate into the network, automatically updating successor and predecessor links to reflect the new network topology. Conversely, the leaveNetwork method allows for graceful departure, ensuring that a leaving node's responsibilities are duly transferred and that the network reconfigures itself to maintain stability. Additionally, methods such as transferFile, uploadFile, and downloadFile are integral to the system's file management capabilities, enabling nodes to handle file storage and retrieval operations across the network efficiently.

2) Hash Function: In the design of our Chord-based distributed system, the hash function plays a pivotal role in establishing the system's structure and facilitating its core functionalities. The choice of the SHA-1 hashing algorithm, implemented through Python's hashlib module in our getHash function, is instrumental in achieving a uniform distribution of nodes and data across the network's identifier space. This uniformity is crucial for maintaining balance and ensuring efficient operation of the system.

The hash function's primary responsibility is to convert variable-length input, such as a node's IP address and port number or a file name, into a fixed-length hash value. This value then serves as a unique identifier within the Chord ring or determines the placement of data within the network. By employing SHA-1, we ensure that each input is mapped to a seemingly random point in the identifier space, thereby minimizing the risk of clustering and uneven load distribution among nodes.

One of the inherent challenges in distributed systems is the effective management of resources and the efficient routing of requests. The getHash function directly addresses these challenges by providing a deterministic yet evenly distributed mapping mechanism. For instance, when a new node joins the network, its unique identifier is generated by hashing its IP address and port number. This process ensures that the node is assigned a specific position within the ring, facilitating the correct updating of successor and predecessor references and the rebalancing of data responsibilities.

Similarly, the hash function is applied to data keys when files are added to the system. This determines which node will be responsible for storing the file, based on the closest succeeding node to the hash value of the file's key. Such a mechanism not only simplifies data lookup and retrieval by providing a clear method for locating data but also aids in achieving a balanced distribution of data storage responsibilities across the network.

Moreover, the use of SHA-1 hashing contributes to the scalability of our system. As the network grows, the hash function continues to ensure that new nodes and data are integrated smoothly, without requiring significant reorganization of the network or creating hotspots of activity. This scalability is vital for the long-term viability and performance of the distributed system.

3) Finger Table: Within the architecture of our Chord-based distributed system, as defined by the Python code we've developed, the implementation of the finger table stands out as a key innovation tailored to enhance the network's operational efficiency and scalability. This detailed mechanism, integral to each node's functionality within our system, exemplifies our approach to optimizing lookup processes and ensuring seamless adaptability as the network evolves.

The finger table in each Node instance comprises a set of entries that serve as shortcuts to other nodes across the network, calculated and updated based on the node's unique identifier within the Chord ring. Each entry in this table points to a successor node for different portions of the identifier space, allowing for rapid traversal of the network when locating the successor of a given key. This structure, realized through the updateFTable method in our code, leverages a logarithmic distribution of entries to minimize the path length for any lookup operation, achieving an average complexity of O(log N) hops to locate a specific node or data key.

Our system's code dynamically maintains the finger table's accuracy and relevance through periodic updates. These updates are crucial, especially in a network characterized by frequent node join and leave events. The updateFTable method recalibrates each entry in the table to reflect the current network topology, ensuring that lookup operations remain efficient despite changes in the network's composition. This dynamic maintenance mechanism underscores our system's resilience and capacity to self-stabilize.

Moreover, the finger table's design inherently supports the network's scalability. The addition of nodes necessitates minimal adjustments to the existing nodes' finger tables, primarily affecting only a logarithmic number of entries directly. This feature is demonstrated in our joinNode and leaveNetwork methods, where the integration or departure of nodes triggers targeted updates to the finger tables, preserving the network's efficiency and integrity without imposing significant overhead.

4) Successor and Predecessor Nodes: Within the Chord distributed hash table implemented in our ChordNode class, each node's place in the ring is determined by its successor (self.successor) and predecessor (self.predecessor). These references are essential for the Chord protocol, as they maintain the ring's structure and ensure that the distributed system can perform data lookups and storage operations efficiently. Initially, a ChordNode points to itself as both its successor and predecessor, representing a Chord ring with a single node.

The Chord DHT is dynamic, allowing nodes to join and leave. The join_handler and leave_network methods are crucial for managing these changes. When a new node joins, it must correctly position itself in the ring by finding and setting its correct successor and predecessor, which involves the modify_successor and modify_predecessor methods. These adjustments are critical for ensuring the ring remains intact and that key responsibilities are correctly reassigned.

To handle node failures, we have implemented a stabilization protocol. The ping_successor method ensures that a node's successor is operational. If the successor fails, the node uses its finger table to locate a new successor. This process is fundamental to the resilience of the Chord ring, allowing it to recover gracefully from node failures without disrupting the overall system.

5) Consistency Models: In our distributed system, we understand that data consistency models are not one-size-fits-all, and so we offer three distinct models catered to various application requirements, each represented within the ConsistencyStrategy Enum. The first, Eventual Consistency, implemented in the replicate_file_eventual method, is designed to prioritize availability and fault tolerance. We chose this model for scenarios where it is acceptable for data to be inconsistent temporarily, with the assurance that it will eventually reach a consistent state. This model is especially well-suited for applications where the system must continue to operate despite network partitions or delays.

We recognize that certain applications require a stronger form of consistency, which led us to implement Sequential Consistency through the replicate_file_sequential method. This model guarantees that operations appear in the same sequence on all nodes, providing a more intuitive form of consistency for users. Sequential consistency is particularly useful in collaborative environments where the order of operations—such as edits in a document or updates to a shared database—must be consistent across the system.

For use cases demanding the most stringent consistency, we provide Linearizable Consistency with the replicate_file_linearizable method. This model offers the strongest level of consistency in our system, ensuring that all operations are immediately consistent across all nodes. It's akin to having a single copy of the data, even though it's distributed across multiple nodes. When we deploy this model, clients can be confident that they are always interacting with the latest data, with no delays or temporal inconsistencies. Linearizable consistency is essential for systems where immediate data accuracy is critical, such as financial services or real-time monitoring systems.

6) File Storage and Management: Our approach to file management involves a meticulous process where each file's name is hashed to ascertain its rightful place within the ring. Utilizing the upload_file and download_file methods, we've enabled our nodes to handle file distribution and retrieval operations seamlessly. When a user requests to upload a file, the system calculates the file's hash and directs the file to the appropriate node. This method of direct addressing minimizes the latency often associated with data transfer in distributed networks and simplifies the retrieval process, as the location of each file is predictable and easily determined.

Recognizing the diversity in our users' needs for data consistency and availability, we have incorporated various file replication strategies that complement our file storage mechanism. These strategies are deeply integrated into our node's operations, ensuring that data is not only stored but also replicated across the network following the selected consistency model. For instance, our replicate_file_eventual method ensures data is eventually consistent across the system, which is particularly beneficial in scenarios where network partitioning is a concern. Meanwhile, our replicate_file_linearizable provides instantaneous consistency, a necessity for applications that cannot tolerate stale data.

7) Resource Discovery Protocol: In designing the discover_resources method within our ChordNode class, we aimed to create a streamlined process allowing nodes to determine the ownership of any given resource swiftly. Our use of consistent hashing to assign resources to nodes ensures a balanced distribution, minimizing the potential for hotspots within the network. When a resource is queried, the system calculates the hash of the resource name, leading to the identification of the responsible node with minimal overhead, thus optimizing the lookup process.

Our system thrives on its decentralized nature, which brings the challenge of how to efficiently locate resources without a central directory. To navigate this, we harness the power of the Chord protocol's logical ring structure. Each node only needs knowledge of its immediate successor in the ring and a finger table with references to other nodes to facilitate faster hops around the ring. This setup allows for the rapid discovery of the node responsible for a given key, as the query is passed along the ring in a structured manner, drastically reducing the number of hops compared to a linear search across all nodes.

We designed our resource discovery mechanism to be inherently scalable. The discover_resources function is not merely a tool for locating resources but a testament to the scalability of our system. As new nodes join and the network grows, our method dynamically adjusts, with each node's finger table being updated to maintain the efficiency of resource lookups. This means that as our system expands, the resource discovery process remains robust and time-efficient, crucial for maintaining performance in a large-scale distributed system. Through this method, we can assure users that the system's capacity to swiftly locate and retrieve resources will persist, regardless of the network's size.

8) Concurrency Control: We implemented two concurrency control strategies, embodied in the Concurrency-Control Enum: Optimistic and Pessimistic. The optimistic approach, encapsulated in the acq_lock_opt method, is guided by the assumption that collisions are rare and hence allows operations to proceed with minimal locking. This strategy is ideal for environments where processes are largely independent and do not frequently interfere with each other.

For scenarios where data conflicts are more probable, we incorporated the pessimistic concurrency control method, illustrated in the acq_lock_pess function. This method proactively prevents access to a resource by other operations until the current one completes. It's a strategy that prioritizes data consistency and serializability, particularly beneficial in systems with high contention for resources. By providing these two concurrency control strategies, our system can be tuned to match the specific requirements of an application, ensuring that the balance between system efficiency and transaction reliability is maintained.

The flexibility of choosing between these concurrency strategies allows our system to adapt to the diverse demands of various applications. We have engineered our ChordNode class to select the appropriate locking mechanism during runtime based on the system's current state and workload. In doing so, we enable our system to handle a wide array of operations – from read-heavy to write-heavy workloads – while ensuring that the data remains consistent and the performance, optimal. As we move forward, our commitment as a team is to continually refine these strategies, embracing the complexities of distributed systems to deliver a service that is both robust and versatile in the face of concurrency.

V. Working and Structure

1) Full Working:

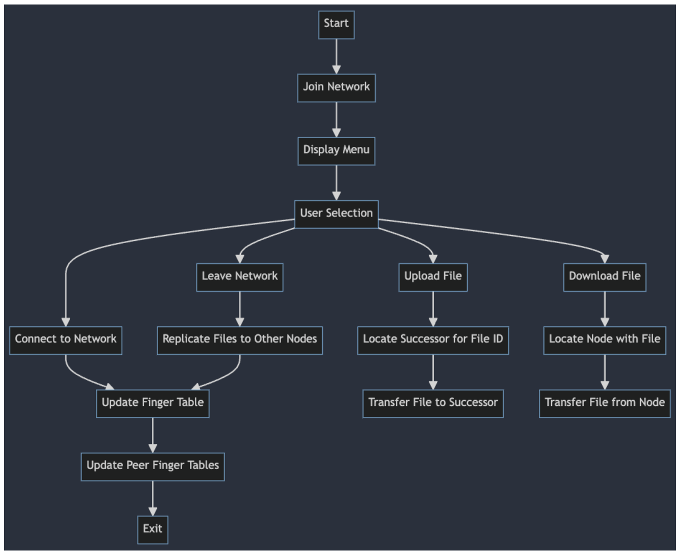


Image1: Workflow of P2P File Sharing

This flowchart provides a high-level overview of the main operations and flow of control in the Chord Network system, including joining and leaving the network, uploading and downloading files, and the related file replication and node discovery processes.

- The program starts.
- A ChordNode instance joins the Chord Network.
- The main menu is displayed to the user.
- The user selects an action from the menu.
- Based on the user's selection, one of the following actions is performed:
  - Connect to the Network: The node connects to the Chord Network by updating its predecessor, successor, and finger table.
  - Leave the Network: The node leaves the Chord Network by replicating its files to other nodes, updating its predecessor, successor, and clearing its finger table.
  - Upload File: The node locates the successor node responsible for the file ID, transfers the file to the successor, and potentially replicates the file based on the chosen consistency strategy.
  - Download File: The node locates the node containing the requested file and transfers the file from that node.
- After performing the selected action, the node updates its finger table and the finger tables of its peers.
- The program exits.

2) Interaction between user and system functions:

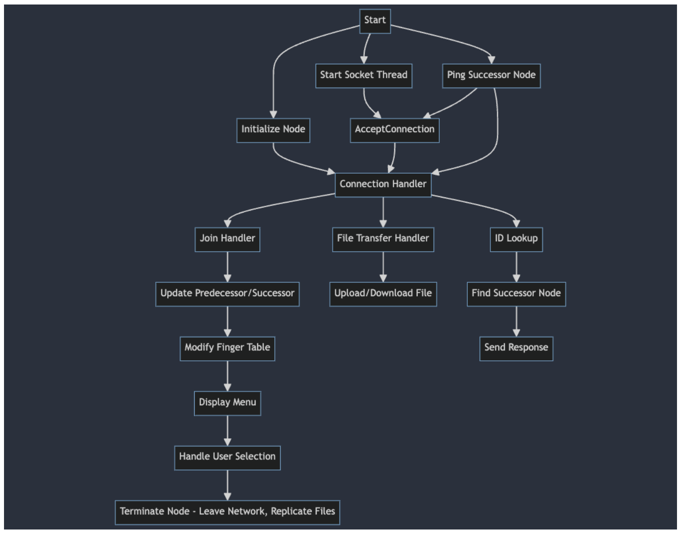


Image2: Use Case Diagram - Interactions

This Use Case Diagram provides an overview of the interactions between the users (nodes and clients) and the various functionalities of the Chord node system, such as joining and leaving the network, uploading and downloading files, managing finger tables, and replicating files across the network nodes.

- Actors
  - Node: Represents a node in the Chord network that participates in the distributed system.
  - Client: Represents a client application or user that interacts with the Chord network, typically to upload or download files.

- Use Cases
  - Join Network: Allows a node to join the Chord network by connecting to an existing node and updating its predecessor, successor, and finger table information.
  - Leave Network: Allows a node to leave the Chord network by notifying its predecessor and succes-

sor nodes, replicating its files, and updating the network accordingly.
  - Upload File: Allows a client or node to upload a file to the Chord network, which triggers file replication based on the chosen consistency model.
  - Download File: Allows a client or node to download a file from the Chord network by locating the appropriate nodes responsible for storing the file.
  - Manage Finger Table: Represents the system's functionality to manage and update the finger tables of nodes, ensuring efficient routing and data lookups within the Chord network.
  - Replicate File: Represents the system's functionality to replicate files across multiple nodes based on the chosen consistency model (eventual, sequential, or linearizable).

3) *File Uploading, Downloading & Replication:*

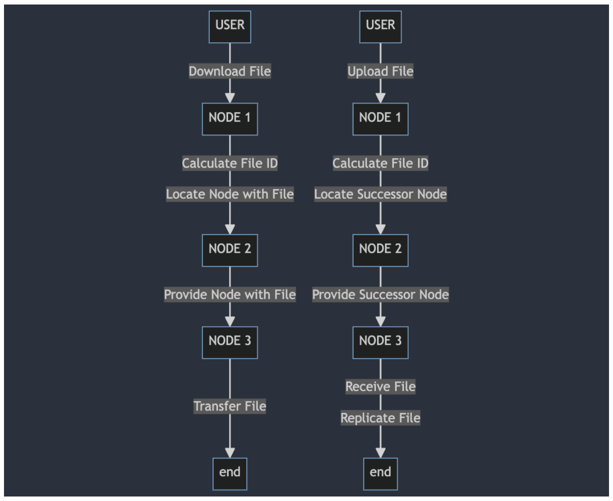


Image3: File Management

This sequence diagram illustrates the interactions between the user, the ChordNode instances, and the system for various operations:

- File Upload
  - The user initiates a file upload.
  - Node 1 receives the file upload request and calculates the file ID.
  - Node 1 locates the successor node responsible for the file ID by contacting Node 2.
  - Node 2 provides the successor node (Node 3) to Node 1.
  - Node 1 transfers the file to Node 3.
  - Node 3 receives the file and replicates it if necessary.

- For file download
  - The user initiates a file download.
  - Node 1 receives the file download request and calculates the file ID.
  - Node 1 locates the node containing the file by contacting Node 2.
  - Node 2 provides the node with the file (Node 3) to Node 1.
  - Node 1 requests the file from Node 3.
  - Node 3 transfers the file to Node 1.

4) *Flow of Interactions with Chord Network:*

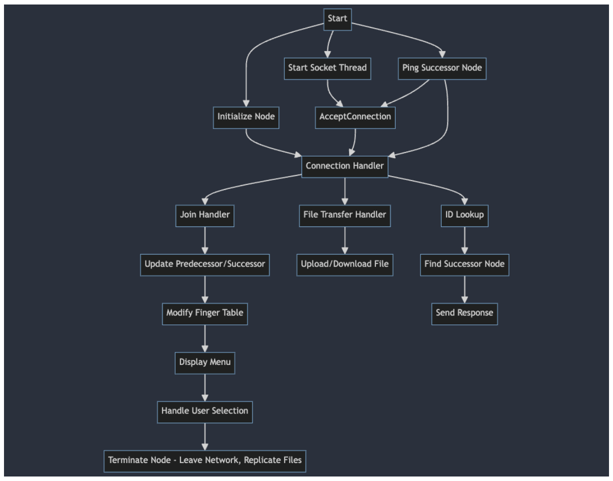


Image4: Process Interaction

The activity diagram also shows the interactions between different processes, such as the Connection Handler interacting with other handlers, and the processes for updating the predecessor, successor, and finger table after network topology changes.

- Initialize Node: This process initializes a ChordNode object with the provided IP address, port number, and other configuration parameters.
- Start Socket Thread: This process starts a separate thread to listen for incoming connections from other nodes in the Chord network.
- Ping Successor Node: This process periodically pings the successor node to detect and handle node failures.
- Accept Connection: This process accepts incoming connections from other nodes.
- Connection Handler: This process dispatches the incoming connection to the appropriate handler based on the connection type (e.g., join, file transfer, ID lookup).
- Join Handler: This process handles a node joining the Chord network by updating the predecessor and successor information.
- File Transfer Handler: This process handles file upload and download requests between nodes.
- ID Lookup: This process finds the successor node responsible for a given key (file ID) in the Chord

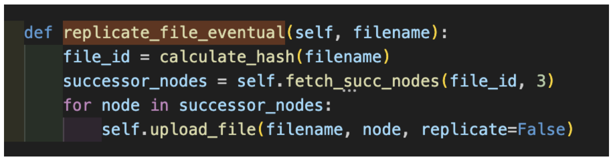
network.

- Update Predecessor/Successor: This process updates the predecessor and successor information for the current node based on the join or leave events.
- Modify Finger Table: This process updates the finger table of the current node based on changes in the network topology.
- Display Menu: This process displays the menu options to the user for interacting with the Chord network.
- Handle User Selection: This process handles the user's selection from the menu, such as joining the network, leaving the network, uploading files, or downloading files.
- Terminate Node: This process handles the termination of the current node, including leaving the network and replicating files to other nodes before exiting.

VI. Evaluation

1) *Effectiveness of Chosen Approaches - Replication:*

a) *Eventual Consistency Approach:*



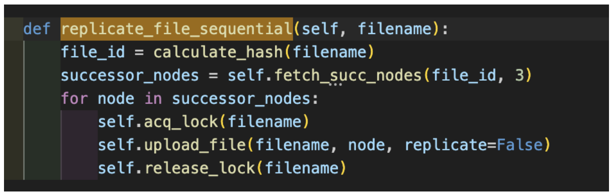
```
def replicate_file_eventual(self, filename):
    file_id = calculate_hash(filename)
    successor_nodes = self.fetch_succ_nodes(file_id, 3)
    for node in successor_nodes:
        self.upload_file(filename, node, replicate=False)
```

Image5: Eventual Consistency

The replicate_file_eventual function embodies our system's eventual consistency strategy. This method is designed to prioritize system availability and tolerance to network partitions by asynchronously replicating files across nodes.

In this approach, after calculating the file's hash to determine its rightful place in the network, the file is replicated across a predefined number of successor nodes without waiting for each replication to be acknowledged before proceeding. This non-blocking behavior enhances the system's ability to remain available and responsive, even in the face of node failures or network issues. The trade-off is that data may not be immediately consistent across all nodes, but it is guaranteed to become consistent over time. This model is computationally less complex, making it highly efficient for large-scale applications where slight delays in data consistency are acceptable.

b) *Sequential Consistency Approach:*



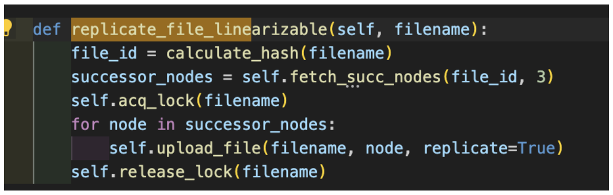
```
def replicate_file_sequential(self, filename):
    file_id = calculate_hash(filename)
    successor_nodes = self.fetch_succ_nodes(file_id, 3)
    for node in successor_nodes:
        self.acq_lock(filename)
        self.upload_file(filename, node, replicate=False)
        self.release_lock(filename)
```

Image6: Sequential Consistency

For applications where the order of operations is critical, our replicate_file_sequential function ensures that all operations are perceived in the same sequence across the network.

In this model, operations are synchronized across nodes by acquiring and releasing locks before and after the replication process. This guarantees that all nodes will apply updates in the same order, preserving operation sequence across the system. While this approach ensures a higher level of consistency, it does introduce additional computational overhead due to the lock management and the sequential nature of the replication process, making it more suited to environments where data integrity and order are paramount.

c) *Linearizable Consistency Approach:*



```
def replicate_file_linearizable(self, filename):
    file_id = calculate_hash(filename)
    successor_nodes = self.fetch_succ_nodes(file_id, 3)
    self.acq_lock(filename)
    for node in successor_nodes:
        self.upload_file(filename, node, replicate=True)
    self.release_lock(filename)
```

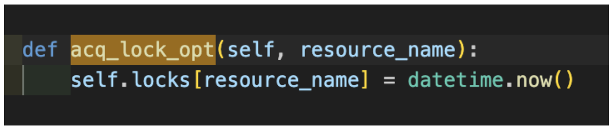
Image7: Linearizable Consistency

Our replicate_file_linearizable function provides the strongest level of consistency, ensuring that all operations are instantly visible to all nodes, which is crucial for systems where immediate data accuracy is required.

In this model, a global lock is acquired before replication starts, and released only after the update has been successfully propagated to all relevant nodes, ensuring that any read operation returns the most recent write. This approach is particularly effective for use cases such as financial transactions or any system that cannot tolerate stale reads. The trade-off for achieving linearizable consistency is a higher computational complexity and the potential for increased latency due to the need for synchronization across nodes.

2) *Effectiveness of Chosen Approaches - Concurrency:*

a) *Optimistic:*



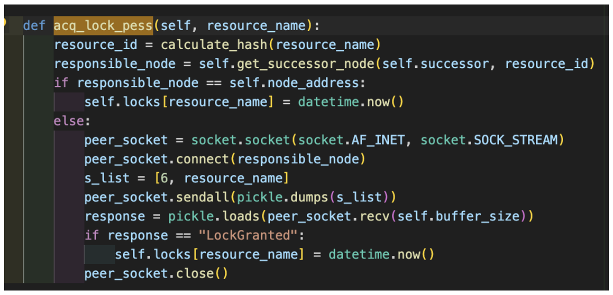
```
def acq_lock_opt(self, resource_name):
    self.locks[resource_name] = datetime.now()
```

Image8: Optimistic Concurrency

Optimistic locking is employed in scenarios where conflicts are expected to be rare. It allows operations to proceed without acquiring locks upfront, under the assumption that the likelihood of simultaneous conflicting operations is low. This strategy is encapsulated in our acq_lock_opt method, which marks the start of a transaction by noting a timestamp or version number, rather than enforcing immediate exclusivity.

The system checks for conflicts at the end of the transaction. If a conflict is detected—meaning another operation has modified the resource since the timestamp was recorded—the transaction is rolled back and possibly retried. This approach minimizes locking overhead, enhancing system throughput and reducing latency. It is especially effective in read-heavy workloads where write conflicts are infrequent, thereby maximizing operational efficiency without significantly compromising data consistency.

b) Pessimistic:



```
def acq_lock_pess(self, resource_name):
    resource_id = calculate_hash(resource_name)
    responsible_node = self.get_successor_node(self.successor, resource_id)
    if responsible_node == self.node_address:
        self.locks[resource_name] = datetime.now()
    else:
        peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        peer_socket.connect(responsible_node)
        s_list = [6, resource_name]
        peer_socket.sendall(pickle.dumps(s_list))
        response = pickle.loads(peer_socket.recv(self.buffer_size))
        if response == "LockGranted":
            self.locks[resource_name] = datetime.now()
        peer_socket.close()
```

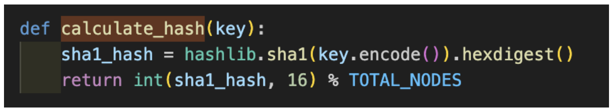
Image9: Pessemistic Concurrency

For environments where operations frequently conflict, we utilize pessimistic locking. This strategy assumes that conflicts are common and proactively prevents concurrent access to resources by acquiring locks before performing any operations. The acq_lock_pess function demonstrates this approach, securing exclusive access to the resource upfront to avoid potential conflicts.

This method ensures that once a transaction begins, it can complete without interference from other operations. While it significantly reduces the risk of data inconsistency, it can lead to increased waiting times for operations, as they must wait for locks to be released. Pessimistic locking is therefore most effective in write-heavy environments or applications where maintaining strict data consistency is critical.

3) Effectiveness of Chosen Approaches - Resource Discovery:

a) Consistent Hashing for Resource Placement:

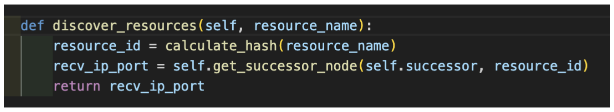


```
def calculate_hash(key):
    sha1_hash = hashlib.sha1(key.encode()).hexdigest()
    return int(sha1_hash, 16) % TOTAL_NODES
```

Image10: Hashing

The foundation of our resource discovery mechanism lies in Chord's consistent hashing algorithm. This algorithm assigns each node and resource a position on a hash ring, ensuring a distributed yet predictable placement of resources. Our implementation uses the calculate_hash function to determine the position of resources: By hashing the resource identifier (such as a filename or key) and calculating its position on the ring, we can efficiently determine which node is responsible for storing that particular resource. This method significantly reduces the complexity of locating resources, as it avoids the need for a central lookup table and ensures that the load is evenly distributed among the nodes in the system.

b) Efficient Lookup with Ordered Node Links:



```
def discover_resources(self, resource_name):
    resource_id = calculate_hash(resource_name)
    recv_ip_port = self.get_successor_node(self.successor, resource_id)
    return recv_ip_port
```

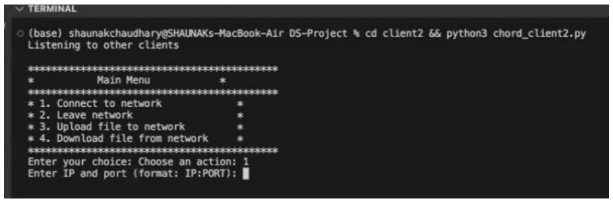
Image10: Lookup

Our system's resource discovery is further enhanced by the ordered nature of nodes within the Chord ring. Each node maintains knowledge of its immediate successor (self.successor), and through a series of queries, a node can find the responsible node for a given resource. The discover_resources method embodies this approach, leveraging the network's structure for efficient lookups

This method effectively utilizes the Chord protocol's principles, ensuring that resource discovery operations are both fast and reliable. The ordered linkage of nodes, combined with the consistent hashing of resources, allows for resource discovery that scales logarithmically with the number of nodes, maintaining high efficiency as the system grows.

VII. DEMONSTRATION

A. User Interface



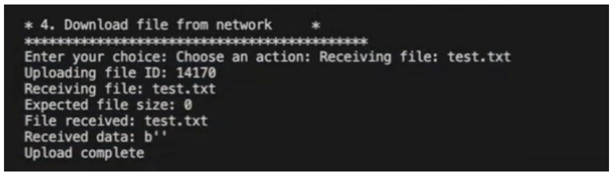
```
(base) shaunakchaudhary@SHAUNAKs-MacBook-Air DS-Project % cd client2 && python3 chord_client2.py
Listening to other clients

******************************************
*           Main Menu            *
******************************************
* 1. Connect to network          *
* 2. Leave network               *
* 3. Upload file to network      *
* 4. Download file from network  *
******************************************
Enter your choice: Choose an action: 1
Enter IP and port (format: IP:PORT): 
```

Image11: Display Menu

In our distributed system, the displayMenu function plays a crucial role in interfacing with the user, offering a straightforward and interactive way to navigate the system's features. This method encapsulates the simplicity and efficiency we aimed for in our design, presenting users with a clear and concise list of operations they can perform, such as connecting to the network, uploading and downloading files, and leaving the network. Through this method, we ensure that users, regardless of their technical expertise, can easily interact with our distributed system, making distributed computing accessible and user-friendly.

B. Upload

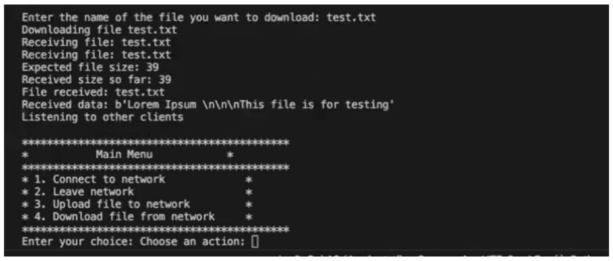


```
* 4. Download file from network     *
********************************************
Enter your choice: Choose an action: Receiving file: test.txt
Uploading file ID: 14170
Receiving file: test.txt
Expected file size: 0
File received: test.txt
Received data: b''
Upload complete
```

Image12: Uploading File

The upload_file user interface (UI) component in our distributed system is meticulously designed to simplify the process of sharing files across the network, ensuring that users can easily contribute their resources. This function represents a critical aspect of our system's user interaction, facilitating the secure and efficient uploading of files to the distributed network.

C. Download

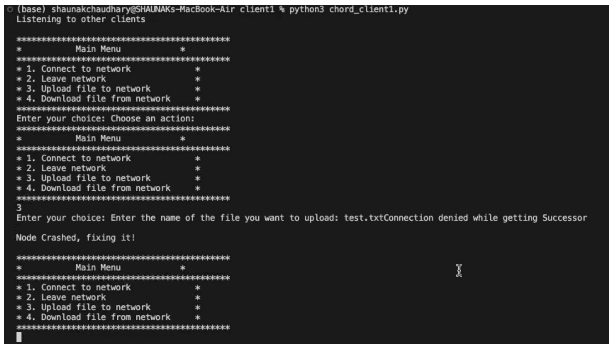


```
Enter the name of the file you want to download: test.txt
Downloading file test.txt
Receiving file: test.txt
Receiving file: test.txt
Expected file size: 39
Received size so far: 39
File received: test.txt
Received data: b'Lorem Ipsum \n\n\nThis file is for testing'
Listening to other clients

********************************************
*           Main Menu          *
********************************************
* 1. Connect to network          *
* 2. Leave network               *
* 3. Upload file to network      *
* 4. Download file from network  *
********************************************
Enter your choice: Choose an action: []
```

Image13: Downloading File

D. Crash



```
(base) shaunakchaudhary@SHAUNAKs-MacBook-Air client1 % python3 chord_client1.py
Listening to other clients

********************************************
*           Main Menu          *
********************************************
* 1. Connect to network          *
* 2. Leave network               *
* 3. Upload file to network      *
* 4. Download file from network  *
********************************************
Enter your choice: Choose an action:
********************************************
*           Main Menu          *
********************************************
* 1. Connect to network          *
* 2. Leave network               *
* 3. Upload file to network      *
* 4. Download file from network  *
********************************************
3
Enter your choice: Enter the name of the file you want to upload: test.txtConnection denied while getting Successor

Node Crashed, fixing it!

********************************************
*           Main Menu          *
********************************************
* 1. Connect to network          *
* 2. Leave network               *
* 3. Upload file to network      *
* 4. Download file from network  *
********************************************
█
```

Image14: Crash Handling

Our UI is designed to keep the user informed throughout their interaction with the system. During a recent incident, a user attempted to upload a file, and our system detected a node crash. Immediately, the UI displayed a clear message: "Connection denied while getting Successor Node Crashed, fixing it!" This feedback is crucial because it acknowledges the issue without overwhelming the user with technical details.

This incident showcases our system's proactive crash detection and automatic recovery procedures. After recognizing the failure, the system initiated a self-healing process. Once the issue was addressed, the Main Menu was presented again, indicating that the user could retry their action. This seamless handling of node crashes ensures that users experience minimal disruption and maintains their trust in the system's resilience.

References

[1] K. P. N. Puttaswamy and B. Y. Zhao, "A Case for Unstructured Distributed Hash Tables," 2007 IEEE Global Internet Symposium, Anchorage, AK, USA, 2007, pp. 7-12, doi: 10.1109/GI.2007.4301423. keywords: Peer to peer computing;Large-scale systems;Routing;Information retrieval;Application software;Network topology;Computer science;File systems;Mechanical factors;Impedance,

[2] Xu, Zhiyong & He, Xubin & Bhuyan, Laxmi. (2005). Efficient file sharing strategy in DHT based P2P systems. 151 - 158. 10.1109/PCCC.2005.1460541.

[3] Haiying Shen, Ze Li, and Kang Chen, "Social-P2P: An Online Social Network Based P2P File Sharing System," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 10, pp. 2874–2889, Oct. 2015.

[4] J. Zhang, G. Wu, X. Hu and X. Wu, "A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services," 2012 ACM/IEEE 13th International Conference on Grid Computing, Beijing, China, 2012, pp. 12-21, doi: 10.1109/Grid.2012.17. keywords: Real-time systems;Cloud computing;Libraries;Servers;Data models;Random access memory;File systems;distributed cache system;cloud storage;HDFS;real-time file acces;in-memory cloud,

[5] C. Riley and C. Scheideler, "A distributed hash table for computational grids," 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., Santa Fe, NM, USA, 2004, pp. 51-, doi: 10.1109/IPDPS.2004.1302971. keywords: Distributed computing;Grid computing;Peer to peer computing;Concurrent computing;Computer networks;Heuristic algorithms;Dynamic programming;Computer network management;Computer applications;Topology,

[6] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 4 (October 2001), 149–160. https://doi.org/10.1145/964723.383071