Optimizing Password Cracking Efficiency through Chord Protocol Implementations

Ashley Varghese
varghes8@purdue.edu
Purdue University
USA

Diksha Jaiswal
djaiswa@purdue.edu
Purdue University
USA

Jack Lowrie
jlowrie@purdue.edu
Purdue University
USA

Paretosha Singh
singh632@purdue.edu
Purdue University
USA

Rakshit Munjadas Kadam
rkadam@purdue.edu
Purdue University
USA

Shruti Ugalmugale
sugalmug@purdue.edu
Purdue University
USA

ABSTRACT

The Chord protocol is a pioneering peer-to-peer (P2P) distributed hash table (DHT) system designed for scalable and efficient resource lookup in decentralized networks. This project implements the Chord protocol using Python, simulating its operation on the Mininet network emulator. The implementation includes essential functionalities such as consistent hashing, finger tables for efficient routing, and fault tolerance mechanisms [7].

To evaluate the protocol, we replicated the simulations described in the original Chord research paper, analyzing performance metrics such as lookup latency and scalability with increasing network size. Additionally, we demonstrated the protocol's practical applicability by implementing a distributed password-cracking application on top of the Chord network, showcasing its potential for real-world distributed computing tasks.

This report outlines the development process, experimental setup, and results obtained. The findings validate the protocol's scalability and robustness, providing insights into its efficiency and suitability for large-scale distributed systems.

CCS CONCEPTS

• **Networks** → **Network protocols**; *Peer-to-peer networks*; • **Security and privacy** → *Cryptography*; *Distributed systems security*; *Authentication*; *Intrusion/anomaly detection and malware mitigation*; • **Theory of computation** → *Design and analysis of algorithms*; • **Computer systems organization** → *Distributed architectures.*

KEYWORDS

Chord protocol, password cracking, distributed systems, security

ACM Reference Format:
Ashley Varghese, Diksha Jaiswal, Jack Lowrie, Paretosha Singh, Rakshit Munjadas Kadam, and Shruti Ugalmugale. 2025. Optimizing Password Cracking Efficiency through Chord Protocol Implementations. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rapid proliferation of distributed systems and peer-to-peer (P2P) networks transformed the way data is stored, shared, and accessed in decentralized environments. These systems empower nodes, often referred to as peers, to communicate directly without relying on centralized servers, thereby improving scalability, fault tolerance, and resource utilization. Recently P2P networks have been explored in relation to IoT[3] and federated learning[1]. While these benefits have fueled the adoption of P2P networks, they have also introduced new challenges. Among these, the efficient discovery and retrieval of resources in a dynamic network of peers stand out as critical issues. Traditional centralized approaches to resource discovery are infeasible in P2P networks due to the absence of central coordinators and the need for scalable solutions.

The Chord protocol, introduced by Stoica et al. [7], is a pioneering solution to these challenges, offering a robust, scalable framework for resource lookup in P2P systems. Chord operates on the principles of a distributed hash table (DHT), where each node in the network is responsible for a specific portion of the key space. By utilizing consistent hashing, Chord ensures that keys are distributed uniformly among nodes, minimizing the disruption caused by nodes joining or leaving the network. This property is particularly important for maintaining the stability and efficiency of large-scale, dynamic networks.

The core innovation of Chord lies in its logical ring structure and its efficient routing mechanism. Each node in the network maintains a small set of routing information, known as the finger table, which allows it to locate any resource within $O(\log n)$ hops in a network of n nodes. This logarithmic lookup efficiency, combined with Chord's self-healing capabilities, makes it an ideal solution for scalable and fault-tolerant P2P applications.

In the research paper *Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications* [7], the authors demonstrated the versatility of Chord through various applications, including a large-scale distributed combinatorial search. Inspired by this work, our project aims to implement the Chord protocol and explore its practical utility in modern distributed systems. By replicating key aspects of the original research and validating its findings, we aim to deepen our understanding of Chord's strengths and limitations.

The primary objectives of this project are threefold:

(1) **Protocol Implementation**: To develop a robust implementation of the Chord protocol in Python, focusing on its essential components such as ring formation, resource lookup, consistent hashing, and fault tolerance mechanisms.
(2) **Performance Validation**: To simulate the protocol in a controlled environment using Mininet, a network emulation tool, and conduct experiments to evaluate its performance under varying network conditions and configurations.

(3) **Application Development**: To demonstrate Chord's real-world applicability by building a distributed password-cracking application that leverages the protocol's efficient lookup mechanism to distribute and manage computational tasks across multiple nodes.

This report provides a comprehensive overview of the project, from the theoretical foundation of the Chord protocol to its implementation and validation. By building upon the foundational work presented in the Chord research paper, we aim to highlight the protocol's practicality and its potential for addressing modern distributed computing challenges. The subsequent sections delve into the technical details of the implementation, experimental methodology, and analysis of results, concluding with insights and recommendations based on our findings.

2 RELATED WORK

Distributed systems require methods for ensuring efficient resource location and data retrieval. Consistent hashing is a technique commonly used to distribute keys uniformly across a large number of nodes. Chord builds on the idea of consistent hashing and introduces a structured approach to manage distributed nodes and their data responsibilities.[7]

Many other DHTs, such as Kademlia [5], Pastry [6], and Tapestry [9], have been proposed to solve similar problems in large-scale systems. While these protocols share the concept of a decentralized key-value store, they differ in terms of routing efficiency, fault tolerance, and the structure of their identifier spaces.

Kademlia, for example, uses a XOR distance metric and maintains a more complex routing table than Chord's finger table [5]. Pastry and Tapestry also implement DHTs but are optimized for different aspects such as high-performance routing and latency [6, 9].

Chord stands out for its simplicity and efficient use of a ring-based architecture, which allows for easy node addition and removal without disrupting the system significantly [7].

3 BACKGROUND

Distributed systems and peer-to-peer (P2P) networks have revolutionized the way resources are shared and accessed, enabling decentralized architectures that eliminate the need for central coordination. These systems distribute workload and data storage across multiple nodes, improving scalability, fault tolerance, and resource utilization. However, such decentralization introduces challenges in efficiently locating and retrieving resources, particularly in dynamic environments where nodes frequently join or leave the network.

The Chord protocol, introduced in the seminal research paper *Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications* [7], addresses the resource lookup challenge with a distributed hash table (DHT) approach. Chord provides a scalable and robust framework for mapping a given key to its corresponding value in a network of peers. Unlike traditional systems that require global knowledge of the network, Chord ensures efficient lookups while requiring each node to maintain information about only a small subset of other nodes.

3.1 Core Concepts of Chord

Consistent Hashing: Chord relies on consistent hashing to partition the key space among the participating nodes. Each key is mapped to a node in the network based on a hash function, ensuring uniform key distribution and minimizing data movement when nodes join or leave. This makes the system highly resilient to churn, a common phenomenon in P2P networks where nodes frequently enter or exit the network.[7]

Logical Ring Structure: The Chord network organizes nodes into a logical ring, where each node is assigned an identifier (ID) from the same hash space as the keys. The ring structure facilitates efficient routing, as each node is responsible for a segment of the key space. A key is stored at the first node whose ID equals or succeeds the key's hash value. [7]

Finger Table: To improve lookup efficiency, each node maintains a finger table, which is a routing table containing pointers to other nodes in the ring. This reduces the number of hops required to locate a resource to $O(\log n)$, where n is the total number of nodes in the network. The finger table allows Chord to scale efficiently even in large networks. [8]

Self-Healing and Fault Tolerance: Chord is designed to handle dynamic changes in the network topology. When nodes join or leave, the protocol adjusts the finger tables and key mappings to ensure consistency. These self-healing properties make Chord robust against failures and highly reliable for distributed applications.[7]

3.2 Applications of Chord

The flexibility and efficiency of the Chord protocol enable it to support a variety of applications. In their research, Stoica et al. [7] demonstrated Chord's potential in large-scale distributed combinatorial search, where computational tasks are distributed across multiple nodes in the network. For example, password cracking or combinatorial optimization problems can leverage Chord's efficient resource lookup mechanism to coordinate the computation and retrieval of partial results.

Our project builds upon the foundational concepts of Chord to implement the protocol and validate its performance. By running Chord with Mininet, a network emulation tool, we recreate the network dynamics described in the research paper. Additionally, we demonstrate its utility by implementing a distributed password-cracking application, demonstrating the practical implications of Chord in real-world scenarios.

4 PROJECT SETUP

In this research, we implemented and simulated the Chord protocol for the purpose of cracking passwords in a distributed environment. To simulate the Chord network and evaluate its performance in a realistic peer-to-peer (P2P) environment, we used *MiniNet*, a popular network emulator that allows for the creation of virtual networks with multiple hosts and switches. MiniNet is ideal for simulating large-scale distributed systems and networking protocols, as it enables the creation of virtual topologies with the flexibility of real-world networks, making it a suitable tool for our study.

4.1 MiniNet Simulation Environment

MiniNet is an open-source tool that provides a lightweight platform for creating virtual networks with hosts, switches, and links, all running on a single physical machine. It allows the creation of virtual environments for simulating network protocols, routing, and distributed systems without requiring physical hardware. For our study, MiniNet enabled the simulation of a Chord-based P2P network, where each node in the Chord network represented a virtual host in the MiniNet topology.

The key features of the MiniNet setup used in this experiment are as follows:

- **Nodes (Hosts):** Each node in the Chord network was represented by a host in MiniNet. We configured multiple virtual machines (VMs) to simulate the nodes in the Chord ring. These nodes communicated with each other through the network topology defined within MiniNet.
- **Switches:** MiniNet allowed us to define switches that interconnect the virtual hosts. The switches are responsible for forwarding network packets between hosts in the simulation.
- **Topology:** A custom topology was created to simulate the Chord protocol's ring-like structure. Each node in the system was connected in a way that mirrored the ring topology, ensuring that the routing mechanisms (such as finger tables) were implemented and tested within a realistic network environment.

4.2 Environment Configuration

The setup included:

- **Operating System:** Ubuntu, tested on AWS EC2 instances.
- **Python Environment:** Python 3.11, with dependencies managed via a virtual environment.
- **Mininet:** A network emulator used to simulate Chord's peer-to-peer architecture.

5 IMPLEMENTATION

The implementation of the Chord protocol focused on reproducing the core features outlined in the research paper *Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications* [7]. This involved developing a Python-based simulation of Chord capable of running in a Mininet environment. The implementation also extended to running experiments described in the paper and building a distributed application atop the protocol.

5.1 Codebase Structure

The implementation is organized into a modular structure, as outlined in the project repository. The key components include:

- **Core Chord Protocol**
  - **Node Module (`chord/node.py`)**: Implements the primary Chord node functionalities, including maintaining finger tables, successor/predecessor pointers, and performing lookups.
  - **Address Module (`chord/address.py`)**: Manages addressing and ID space calculations, enabling efficient key lookups.
  - **Network Module (`chord/net.py`)**: Handles socket-based communication between nodes, enabling message passing and routing.
- **Experiments and Testing**
  - **Experiments (`experiments/`)**: Scripts to deploy and evaluate the Chord protocol on Mininet topologies. Includes support for simulating dynamic network changes such as node joins and failures.
  - **Test Suite (`tests/`)**: Automated tests ensuring correctness of the Chord implementation, such as consistency of finger tables and routing accuracy.
  - **Setup Verification (`setup/`)**: Tools like `chk_config.py` and `chk_mininet.py` to validate the installation and network environment.
- **Applications** An additional application, distributed password cracking, was implemented to demonstrate Chord's capabilities. This application uses the distributed keyspace provided by Chord to allocate password-cracking tasks across multiple nodes.

5.2 Key Features Implemented

The implementation of the Chord protocol incorporated its fundamental algorithms: `join`, `stabilize`, `notify`, and `fix_fingers`. These operations ensure the correctness and robustness of the Chord network. Below, we describe each feature in detail.

5.2.1 Join. The `join` method enables a node to integrate into an existing Chord ring through a known node's IP and port.

```
def join(self, known_ip, known_port):
    self.predecessor = None
    known_node_address = Address(known_ip,
        known_port)
    response = self._net.send_request(
        known_node_address, 'FIND_SUCCESSOR'
        , self.address.key)
    if response:
        self.successor = self._parse_address
            (response)
        self.start()
        self.fix_fingers()
```

Process:

- The joining node sends a `FIND_SUCCESSOR` request to a known node in the ring.
- The node updates its successor upon receiving a response.

Outcome: The new node is integrated into the Chord ring, preserving the ring's structure.

5.2.2 Stabilize. The `stabilize` method ensures correctness in the ring topology by verifying and updating the successor.

```
def stabilize(self):
    x_response = self._net.send_request(self
        .successor, 'GET_PREDECESSOR')
    x = self._parse_address(x_response) if
        x_response else None
```

```
    if x and self._is_between(self.address.
        key, self.successor.key, x.key):
        self.successor = x
    self.notify(self.successor)
```

Process:

- A node queries its successor for its predecessor.
- If the predecessor is a better candidate, the node updates its successor.

Outcome: Stabilization ensures that node relationships are correct under dynamic conditions.

5.2.3 Notify. The `notify` method informs a potential successor about a node's presence.

```
def notify(self, potential_successor):
    response = self._net.send_request(
        potential_successor,
        'NOTIFY',
        f"{self.address.key}:{self.address.
            ip}:{self.address.port}"
    )
    return response == "OK" or response == "
        IGNORED"
```

Process:

- A node sends a `NOTIFY` message to its successor.
- If the successor accepts the notification, it updates its predecessor.

Outcome: Predecessor-successor relationships are dynamically adjusted.

5.2.4 Fix Fingers. The `fix_fingers` method incrementally updates the entries in a node's finger table.

```
def fix_fingers(self):
    start = (self.address.key + (1 << self.
        _next)) % (1 << Address._M)
    responsible_node = self.find_successor(
        start)
    self.finger_table[self._next] =
        responsible_node
    self._next = (self._next + 1) % Address.
        _M
```

Process:

- The method calculates the start key for the next finger and finds the responsible node.
- The finger table entry is updated accordingly.

Outcome: Efficient routing is maintained through an up-to-date finger table.

5.3 Challenges and Solutions

- **Maintaining Correctness Under Churn**
  Dynamic network changes posed challenges in maintaining consistent routing tables. This was mitigated by implementing periodic stabilization and finger table updates.
- **Integration with Mininet**
  Adapting the Python implementation to work seamlessly with Mininet required custom scripts and validation tools, such as `chk_config.py`.
- **Performance at Scale**
  Scaling simulations to a large number of nodes highlighted bottlenecks in stabilization routines, which were optimized by batching updates.

The implemented code and associated scripts are hosted in the Chord GitHub Repository.

APPLICATIONS OF CHORD

6 PASSWORD CRACKING

The objective of this project was to implement a distributed system for cracking passwords using a Chord network[7]. Each node in the network will attempt to crack a portion of the password space, coordinating with other nodes to avoid duplicate work and to increase the efficiency of the search.

6.1 Key Components

6.1.1 Node Initialization: Each node initializes by joining the Chord ring. It finds its position in the ring based on its hash and establishes connections to its predecessor and successor in the network.

6.1.2 Password Hash and Search Space Division: The password hash to be cracked (e.g., an MD5 hash) is input into the system. The search space (possible password combinations) is divided among the nodes. This division is based on the Chord key space allocation, where each node is responsible for a segment of the hash space.

6.1.3 Cracking Algorithm:

- Each node generates potential passwords (using a combination of characters and numbers up to a certain length).
- These passwords are hashed using the specified cryptographic hash function.
- Each generated hash is compared against the target hash.
- If a match is found, the node announces the success to the network.[2]

6.1.4 Distributed Coordination:

- Joining the Network: Nodes use the Chord join function to enter the network and find their position in the ring.
- Lookup Operations: Nodes use Chord's find_successor and closest_preceding_node functions to route requests through the network.
- Stabilization: Regularly running stabilization protocols ensure that the ring maintains correct pointers even as nodes join and leave.

6.2 Challenges and Considerations

6.2.1 Scalability: The system needs to efficiently scale with the number of nodes. The efficiency of the Chord lookup operations, which are O(log N), where N is the number of nodes, plays a critical role here.

6.2.2 Load Balancing: Even distribution of the password search space is crucial for maximizing the system's overall cracking speed.

6.3 Conclusion

Using Chord for distributed password cracking provides a robust, scalable, and efficient framework for parallel processing tasks across a decentralized network. The project highlights the adaptability of distributed hash table protocols like Chord for diverse applications beyond simple key-value stores, including security-related tasks such as password cracking.

7 INTERNET OF THINGS (IOT)

Distributed hash tables play a critical role in the scalability and efficiency of IoT platforms, facilitating seamless device communication and data management across heterogeneous devices in smart environments. In the context of IoT, DHTs help manage a large number of connected devices, allowing for efficient discovery and interoperability among devices without relying on a central coordinator. For instance, Javed et al. (2020) [3] discuss a scalable IoT platform that leverages hashing to enhance discovery processes and maintain data consistency across a wide network of distributed sensors and actuators, ensuring robust and scalable interaction within smart city infrastructures.

The Hajime botnet represents a unique application of distributed technologies, specifically leveraging peer-to-peer networks to create a decentralized botnet. Unlike traditional botnets which rely on a few central servers for command and control, Hajime uses a DHT-like structure to distribute its control mechanisms, making it more resilient against takedown attempts and enabling it to scale dynamically. This decentralized approach not only makes it difficult to disrupt but also shows the potential of DHTs for resilient network applications beyond typical uses.s

8 FEDERATED LEARNING

Federated learning is an emerging field where DHTs can significantly enhance scalability and privacy. By decentralizing the data storage and computation across multiple nodes, federated learning systems can perform machine learning tasks without needing to centralize sensitive data. This approach is particularly beneficial in environments where data privacy is paramount, such as in healthcare or financial services. The work by Bellet et al. (2022) [1] illustrates how topology in decentralized federated learning can compensate for data heterogeneity, leveraging DHTs to optimize data routing and aggregation across diverse and geographically dispersed nodes . Similar thesis (2022) [4] explores scalable federated learning through DHT-based overlays, addressing key challenges such as efficient resource allocation and robust data sharing among participants in a distributed learning network .

These applications highlight the versatility and potential of DHTs and related distributed technologies to address complex and large-scale challenges across various domains, illustrating their critical role in the development of next-generation distributed systems

9 PERFORMANCE COMPARISON

9.1 Performance

The Chord protocol demonstrates impressive efficiency in query resolution due to its structured use of logarithmic scaling through the finger table. This structure allows each node to make significant jumps toward the target key, drastically improving performance over simpler methods like sequential node traversal.

In practice, experimental data validates Chord's theoretical efficiency. For example, with a network size consistent with n=8, the mean number of hops per key request remains very low, averaging around 1.07, with a maximum of 3 hops. This efficiency is further validated in larger networks such as when n=16, where the mean increases slightly to 1.98, still maintaining low latency with the maximum hops capped at 3. These results confirm that Chord efficiently reduces path lengths even as network size increases.

However, it is essential to consider the impact of network dynamics on Chord's performance. In environments with high churn—where nodes frequently join and leave—the overhead associated with updating finger tables and stabilizing the network introduces additional latency. This can diminish the benefits of Chord's logarithmic efficiency, especially in larger and more volatile networks. Therefore, while Chord is highly effective under stable conditions, its performance in real-world applications may vary depending on network stability and node churn rate.

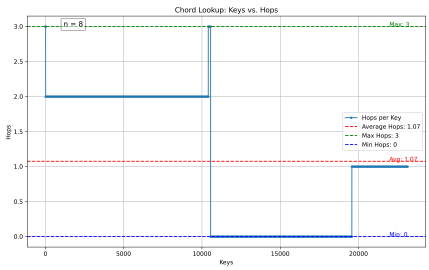

Figure 1: Chord performance with 8 nodes: Keys vs Hops

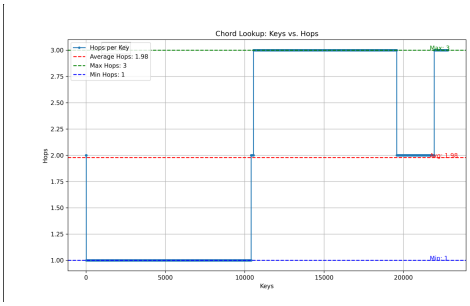

Figure 2: Chord performance with 16 nodes: Keys vs Hops

9.2 Scalability

One of Chord's core strengths lies in its scalability. As the number of nodes in the network grows, the protocol ensures that each node maintains only $O(\log N)$ routing information in its finger table. This small, fixed overhead allows Chord to handle thousands of nodes without significant degradation in lookup performance.

Theoretical simulations and experimental data show that even when doubling the network size, the mean path length increases

only marginally. This scalability ensures that even in large-scale distributed systems, resources can be located efficiently. By comparison, centralized systems struggle with scalability as they require a single node to maintain a global state, which becomes a bottleneck as the network grows.

Despite its scalability, Chord's reliance on consistent hashing and finger tables introduces challenges when node identifiers or key distributions are uneven. Skewed distributions can lead to load imbalances, where certain nodes become overloaded with keys. Techniques such as virtual nodes or load-balancing algorithms can mitigate this issue, but they introduce additional complexity.

9.3 Fault Tolerance

Fault tolerance is another critical aspect where Chord excels, although its practical effectiveness depends on the implementation of stabilization protocols. Chord is inherently resilient to failures due to its distributed nature and the use of successor lists. These lists provide redundancy, ensuring that even if a node fails, its responsibilities can be quickly transferred to the next available successor.

However, the experiments in this project focused primarily on static networks, where all nodes were assumed to be stable. This limitation means the fault tolerance of the implemented system was not rigorously tested under high-churn conditions. In real-world scenarios, frequent node failures can disrupt routing, especially if successor lists or finger tables are not regularly updated. Implementing robust stabilization routines can mitigate these issues but adds computational and network overhead.

10 CHALLENGES AND THEIR IMPACT

10.1 Challenges in Implementation

Several challenges were encountered during the implementation of the Chord protocol:

- **Routing Logic and Boundary Conditions**: Ensuring correct query routing across the identifier space was a non-trivial task. The circular nature of the identifier space introduced edge cases, such as when keys wrapped around from the maximum identifier to the minimum. Early implementations failed to account for these scenarios, leading to incorrect query resolutions and routing loops.
- **Finger Table Construction**: Building accurate finger tables required precise interval calculations and ensuring each entry pointed to the correct node. Errors in finger table entries caused misrouted queries and inflated path lengths during initial testing.
- **Simulating Large Networks**: Simulating networks with thousands of nodes and hundreds of random queries was computationally intensive. Resource limitations constrained the scope of the experiments, particularly for evaluating dynamic behaviors like node churn or failure recovery.
- **Integration with Mininet**: Adapting the Python implementation to work seamlessly with the Mininet network emulator posed integration challenges. Network delays and inconsistencies in Mininet occasionally affected query resolution times, requiring additional debugging.
- **Load Imbalance**: Consistent hashing assumes uniform key distribution, but in practice, hash collisions or skewed distributions led to load imbalances. This issue became more pronounced in smaller networks where fewer nodes had to handle disproportionately large key assignments.

10.2 Impact of Challenges on Results

These challenges had both direct and indirect impacts on the results:

- **Accuracy of Path Length Metrics**: Initial issues with finger table construction and boundary conditions led to inflated path lengths in early simulations. After resolving these issues, the measured path lengths closely aligned with theoretical expectations, validating the logarithmic efficiency of Chord.
- **Focus on Static Networks**: Due to computational constraints, the experiments primarily evaluated Chord in static networks. While this approach simplified the analysis, it did not provide insights into Chord's fault tolerance or performance under high-churn conditions. Future work should include simulations of dynamic environments to evaluate these aspects.
- **Skewed Results Due to Load Imbalance**: Uneven key distributions occasionally produced outlier path lengths. These outliers slightly increased the mean path length but were mitigated by focusing on percentile-based metrics, such as the 1st and 99th percentiles.
- **Real-World Applicability**: Despite the challenges, the implementation demonstrated Chord's practicality for real-world distributed applications. The efficient lookup mechanism was successfully applied to a distributed password-cracking application, showcasing the protocol's utility beyond theoretical simulations.

11 CONCLUSION

11.1 Dynamic Network Evaluation

This project primarily focused on static networks, where nodes and keys remain unchanged during simulation. Future work should incorporate dynamic behaviors, such as frequent node joins, departures, and failures, to evaluate Chord's fault tolerance mechanisms under high-churn conditions. Implementing and testing stabilization routines to maintain ring consistency and finger table accuracy in such scenarios will provide a deeper understanding of Chord's robustness in real-world environments.

11.2 Optimizing Fault Tolerance Mechanisms

The current implementation relies on static successor lists to handle node failures. Future enhancements could include dynamic successor selection and proactive replication strategies to improve fault recovery time. Additionally, exploring techniques such as predictive node failure detection and periodic network health checks could further enhance the system's resilience.

11.3 Load Balancing and Resource Distribution

While consistent hashing ensures uniform key distribution in theory, practical implementations often encounter skewed distributions, leading to load imbalances. Future work should explore advanced load-balancing techniques, such as virtual nodes or adaptive key reassignment, to mitigate these issues. Incorporating dynamic load monitoring and redistribution mechanisms can further optimize resource utilization across the network.

11.4 Evaluation of Large-Scale Networks

Simulating networks with tens of thousands of nodes would better capture Chord's scalability limits. This requires optimizing the simulation framework to handle large-scale networks efficiently. Leveraging distributed computing platforms or high-performance cluster environments for these simulations could provide insights into the protocol's performance at Internet-scale deployments.

11.5 Integration with Real-World Applications

The distributed password-cracking application implemented in this project demonstrates Chord's potential for real-world use cases. Future extensions could involve integrating Chord with other distributed systems, such as decentralized storage platforms, content delivery networks, or blockchain networks. Evaluating Chord's performance in these diverse application domains would highlight its versatility and practicality.

11.6 Real-Time Monitoring and Visualization

Developing tools for real-time monitoring and visualization of Chord's operations would greatly benefit both research and practical deployments. Such tools could provide insights into network state, key distribution, and routing efficiency, enabling faster debugging and optimization.

12 CONTRIBUTIONS

- **Ashley Varghese** Developed simulations to validate model accuracy and optimize system performance, enhancing predictive analytics capabilities. Worked on core functionalities of chord, particularly in ring creation and node traversal.
- **Diksha Jaiswal** Implemented the stabilize function, was responsible for the creation of the finger_table presentation materials, and drafting the abstract, introduction, and related work sections of the project report.
- **Jack Lowrie** Designed and implemented the basic chord protocol and program architecture, Presented chord protocol motivation/overview slides, Set up environment, including test node scripts, sample node client scripts with background maintenance, recorded chord demos.
- **Paretosha Singh** Assisted in developing simulations to validate model accuracy, drafted the differences in setup, challenges sections and future work of the project report.
- **Rakshit Munjadas Kadam** Implemented the finger_table function, and drafting the abstract, introduction, and related work sections of the project report.
- **Shruti Ugalmugale** : Developed the password cracking algorithm and was responsible for the creation of the corresponding presentation materials, slides, and report, gathered and coordinated submission process.

REFERENCES

[1] Aurélien Bellet, Anne-Marie Kermarrec, and Erick Lavoie. 2022. D-Cliques: Compensating for Data Heterogeneity with Topology in Decentralized Federated Learning. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 1–11. https://doi.org/10.1109/SRDS55811.2022.00011

[2] Piyush Gupta and Prashant Kumar. 2012. Distributed Computing for Password Cracking. *Journal of Network Security* 2012, 4 (2012), 12–16.

[3] Asad Javed, Avleen Malhi, Tuomas Kinnunen, and Kary Främling. 2020. Scalable IoT Platform for Heterogeneous Devices in Smart Environments. *IEEE Access* 8 (2020), 211973–211985. https://doi.org/10.1109/ACCESS.2020.3039368

[4] Taehwan Kim. 2022. *Scaled: Scalable Federated Learning via Distributed Hash Table Based Overlays.* Master's thesis. Virginia Tech.

[5] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.

[6] Antony Rowstron and Peter Druschel. 2001. A Peer-to-peer Information System Based on the XOR Metric. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 329–350.

[7] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review* 31, 4 (2001), 149–160.

[8] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishna. [n. d.]. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf. Accessed: Fall 2024.

[9] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. 2004. Tapestry: A Resilient Global-scale Overlay for Service Deployment. In *IEEE Journal on Selected Areas in Communications*, Vol. 22. IEEE, 41–53.