CHORD-KV: DISTRIBUTED KEY VALUE STORE

ABSTRACT

This project presents the development of a Distributed Hash Table (DHT) based Chord Protocol implementation for decentralized peer-to-peer (P2P) systems. The Chord Protocol serves as a fundamental building block for scalable and fault-tolerant distributed systems, facilitating efficient lookup operations and data storage across a network of interconnected nodes.

The Chord Protocol enables efficient key-based routing, allowing nodes to locate data items within the network with logarithmic time complexity relative to the number of nodes. This capability is crucial for applications requiring distributed storage and retrieval of large volumes of data, such as file sharing networks and distributed databases.

Key features of the Chord implementation include node join and departure mechanisms, consistent hashing for key assignment, and finger table maintenance for efficient routing. Nodes in the network dynamically adjust their routing tables to reflect changes in the network topology, ensuring robustness and fault tolerance.

The Chord Protocol implementation emphasizes simplicity, modularity, and scalability, enabling easy integration into various distributed applications. It leverages modern programming languages and frameworks, such as Python and Flask, to facilitate rapid development and deployment of distributed systems.

In conclusion, the Chord Protocol implementation provides a reliable and scalable solution for decentralized peer-to-peer communication and data management. Its efficient routing mechanisms, fault tolerance, and scalability make it suitable for a wide range of distributed applications, from content delivery networks to distributed computing platforms. Future enhancements may focus on optimizing performance, enhancing security, and expanding functionality to support more complex distributed applications.

.

Table of Contents

MOTIVATION

The motivation behind the development of the Chord Protocol implementation stems from the increasing demand for scalable and fault-tolerant distributed systems in various applications. As the volume and complexity of data continue to grow, traditional centralized architectures become insufficient for handling the scale and ensuring reliability.

The Chord Protocol offers a solution to these challenges by providing a decentralized and efficient approach to distributed data storage and retrieval. By leveraging peer-to-peer communication and consistent hashing techniques, the Chord Protocol enables nodes to efficiently locate data items within the network, even in the presence of failures or dynamic changes in the network topology.

Furthermore, the Chord implementation is motivated by the need to address the limitations of existing distributed systems, such as centralized bottlenecks, single points of failure, and scalability constraints. By adopting the Chord Protocol, developers can build robust and scalable distributed applications that can handle large volumes of data and adapt to changes in the network environment.

Moreover, the Chord Protocol's simplicity and modularity make it an attractive choice for a wide range of distributed applications, from peer-to-peer file sharing networks to distributed databases and content delivery networks. Its decentralized nature fosters resilience and fault tolerance, ensuring that the system remains operational even in the face of node failures or network partitions.

In summary, the development of the Chord Protocol implementation is driven by the need for scalable, fault-tolerant, and efficient distributed systems that can meet the demands of modern applications. By providing a robust and decentralized approach to distributed data storage and retrieval, the Chord Protocol empowers developers to build resilient and scalable distributed applications that can adapt to the evolving needs of users and the environment.

OBJECTIVES

1. ***Scalability:*** Develop a distributed system framework that can efficiently handle a large number of nodes and data items while maintaining low overhead in terms of routing and storage.
2. ***Fault Tolerance:*** Implement mechanisms to ensure the resilience of the system in the face of node failures, network partitions, or other disruptions, allowing for continuous operation and data availability.
3. ***Decentralization:*** Design a decentralized architecture where each node in the network plays an equal role in data storage and routing, eliminating the need for centralized coordination and reducing single points of failure.
4. ***Efficient Data Retrieval:*** Enable nodes to efficiently locate data items within the network using consistent hashing and finger tables, ensuring fast and reliable access to data even as the network topology changes.
5. ***Dynamic Adaptation:*** Implement algorithms and protocols that allow the system to dynamically adapt to changes in the network, such as node joins, departures, or failures, ensuring seamless operation and load balancing.
6. ***Performance Optimization:*** Optimize the performance of the Chord Protocol implementation in terms of routing efficiency, resource utilization, and response times, ensuring high throughput and low latency for data retrieval operations.
7. ***Testing and Evaluation:*** Conduct thorough testing and evaluation of the Chord Protocol implementation under various scenarios and workloads to validate its scalability, fault tolerance, and performance characteristics.

8. ***Future Enhancements:*** Lay the groundwork for future enhancements and extensions to the Chord Protocol implementation, such as adding support for additional features, optimizing algorithms, or integrating with other distributed systems protocols.

INTRODUCTION

In the realm of distributed systems, the efficient management of data and resources is essential for ensuring reliability, scalability, and fault tolerance. With the increasing complexity and scale of modern applications, traditional centralized architectures often struggle to meet the demands of distributed environments. The Chord Protocol implementation emerges as a solution to address these challenges and provide a robust framework for building scalable and fault-tolerant distributed systems.

The Chord Protocol implementation serves as a decentralized approach to distributed data storage and retrieval, offering a scalable and efficient solution for managing large volumes of data across a network of nodes. By leveraging peer-to-peer communication and consistent hashing techniques, the Chord Protocol enables nodes to efficiently locate data items within the network, even in the presence of failures or dynamic changes in the network topology.

This introduction provides an overview of the key principles and objectives of the Chord Protocol implementation, highlighting its role in enabling developers to build resilient and scalable distributed applications. Through its decentralized architecture and efficient routing mechanisms, the Chord Protocol implementation aims to address the limitations of traditional distributed systems and provide a foundation for building robust and scalable applications in various domains.

In the subsequent sections, we delve deeper into the architecture, design principles, and applications of the Chord Protocol implementation, exploring its significance in modernizing distributed systems and supporting the development of scalable and fault-tolerant applications in diverse environments.

METHODOLOGY

The methodology used for the implementation of the Chord Protocol project involves several key steps, including research, design, implementation, testing, and documentation. Here's an outline of the methodology:

1. **Research and Familiarization:**
    - Begin by understanding the fundamentals of the Chord Protocol and its application in distributed systems.
    - Research existing implementations, academic papers, and resources related to the Chord Protocol to gain insights into its design and functionality.
2. **Requirement Analysis:**
    - Define the scope and objectives of the project, considering the academic requirements and constraints.
    - Identify the specific features and functionalities to be implemented, such as node management, routing, and data storage.
3. **Design Phase:**
    - Develop a high-level design for the Chord Protocol implementation, outlining the major components and interactions.
    - Define the data structures, algorithms, and interfaces needed to realize the desired functionality.

4. Implementation:

- Translate the design into code, following best practices and coding standards.
- Start with the core components of the Chord Protocol, such as node initialization, finger table maintenance, and key lookup operations.
- Incrementally build and test each component to ensure incremental progress and early detection of issues.

5. Testing and Evaluation:

- Develop test cases to validate the correctness and robustness of the implementation.
- Conduct unit tests to verify individual components and integration tests to assess the interaction between different modules.
- Evaluate the performance of the implementation under various scenarios, such as different network sizes and churn rates.

By following this structured methodology, the academic course project aims to provide a hands-on learning experience in distributed systems concepts and implementation techniques while meeting the requirements and objectives of the course.

CODE

cli.py:

```
@main.command()
@click.option('--key', required = True, help = "The name of the song to delete")
@click.argument('node', required = False)
def delete(**kwargs):
  """Deletes key-value pair for given key"""

  key = kwargs['key']
  if kwargs['node'] != None:
     ip = kwargs['node']
  else:
     ip = common_functions.random_select()
  r = requests.post('http://'+ip+'/delete', data = { 'key':key })
  print(r.text)

  pass

@main.command()
@click.option('--key', required = True, help = "The name of the song to find, if special
character '*' is given it returns all key-value pairs in Chord")
@click.argument('node', required = False)
def query(**kwargs):
  """Find the key-value pair for given key"""

  key = kwargs['key']
  node = kwargs['node']
  common_functions.query(key, node)

  pass

@main.command()
@click.argument('node', required = True)
def depart(**kwargs):
  """Departs node with given ip from Chord"""

  ip = kwargs['node']
  r = requests.post('http://'+ip+"/depart")
  print(r.text)
  pass
import requests, random, sys, click, time
import common_functions
```

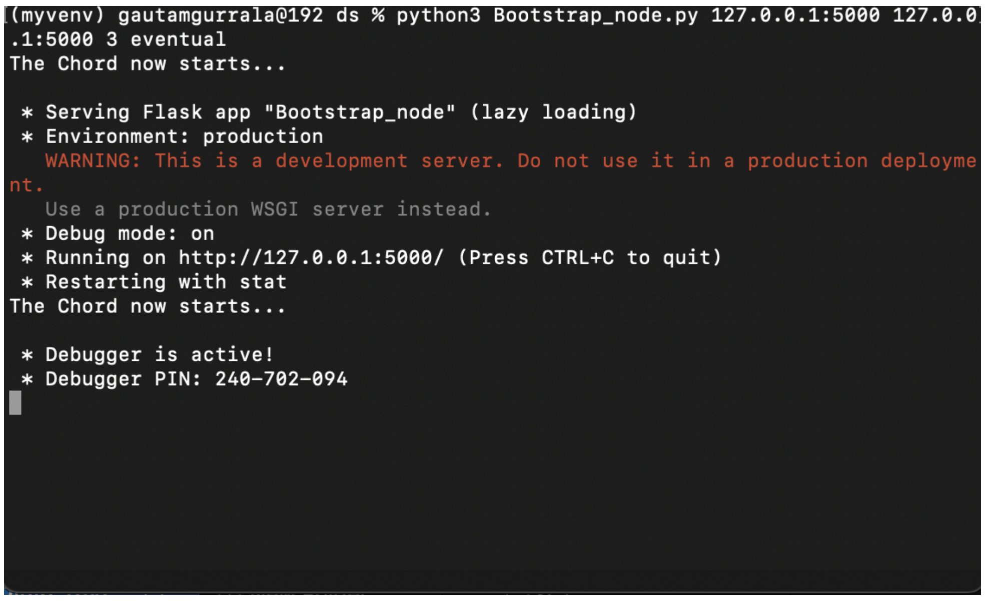
```
from threading import Thread

boot_ip='127.0.0.1:5000'

@click.group()
def main():
  """A CLI for Chord users!"""
  pass

@main.command()
@click.option('--key', required = True, help = "The name of the song to insert")
@click.option('--value', required = True, help = "Node that has this song")
@click.argument('node', required = False)
def insert(**kwargs):
  """Insert given key-value pair in Chord!"""

  key = kwargs['key']
  value = kwargs['value']
  node = kwargs['node']
  common_functions.insert(key, value, node)
  pass

@main.command()
@click.argument('node', required = False)
def overlay(**kwargs):
  """Returns Chord topology"""

  if kwargs['node'] != None:
     ip = kwargs['node']
  else:
     ip = common_functions.random_select()

  r = requests.post('http://'+ip+'/overlay')
  print("This is the Chord topology: \n"+r.text)

  pass

@main.command()
@click.argument('node', required = True)
def join(**kwargs):
  """Join node with given ip to Chord"""
  ip = kwargs['node']
  r = requests.post('http://'+ip+"/join")
  print(r.text)
  pass

@main.command()
@click.argument('file_path', required = True)
```

```
@click.option('--request_type', type = click.Choice(['insert', 'query', 'mix'], case_sensitive =
False))
def file(**kwargs):
  """Send requests with input from a file"""

  count = 0

  file = kwargs['file_path']
  file1 = open(file, 'r')
  Lines = file1.readlines()

  if kwargs['request_type'] == 'insert':

      start = time.time()

      for line in Lines:
          count += 1
          line_list = line.strip().split(',')
          key = line_list[0]
          value = line_list[1]
          common_functions.insert(key, value)

      end = time.time()

  elif kwargs['request_type'] == 'query':

      start = time.time()

      for line in Lines:
          count += 1
          line_list = line.strip().split(',')
          key = line_list[0]
          common_functions.query(key)

      end = time.time()

  elif kwargs['request_type'] == 'mix':

      start = time.time()

      for line in Lines:
          count += 1
          line_list = line.strip().split(',')
          req_type = line_list[0]
          key = line_list[1]

          if req_type == 'insert':
            value = line_list[2]
            common_functions.insert(key, value)

        elif req_type == 'query':

            common_functions.query(key)
    end = time.time()

  throughput = count/(end-start)

  print("Throuput of Chord = %.4f requests/second"%throughput, "%.4f seconds per
query"%(1/throughput))



@main.command()
@click.argument('file_path', required = True)
@click.option('--request_type', type = click.Choice(['insert', 'query', 'mix'], case_sensitive =
False))
def fileparallel(**kwargs):
  """Send requests with input from a file using one thread for each node"""

  #get topology ip's
  r = requests.post('http://' + boot_ip + '/overlay')
  nodes_list = r.json()['topology']
  ip_list = []
  for node in nodes_list:
    temp_ip = (node['node_ip_port'])
    ip_list.append(temp_ip)
  #create requests dicts
  requests_dicts={}
  threads_list=[]
  for ip in ip_list:
    requests_dicts[ip]=[]

  count = 0
  file = kwargs['file_path']
  file1 = open(file, 'r')
  Lines = file1.readlines()

  if kwargs['request_type'] == 'insert':
    for line in Lines:
        count += 1
        line_list = line.strip().split(',')
        key = line_list[0]
        value = line_list[1]
        random_ip=random.choice(ip_list)
        requests_dicts[random_ip].append( ('insert',key,value,random_ip) )
  elif kwargs['request_type'] == 'query':
    for line in Lines:
        count += 1
        line_list = line.strip().split(',')
        key = line_list[0]
        random_ip = random.choice(ip_list)
        requests_dicts[random_ip].append(('query',key, random_ip))
  elif kwargs['request_type'] == 'mix':
    for line in Lines:
        count += 1
        line_list = line.strip().split(',')
        req_type = line_list[0]
        key = line_list[1]
        random_ip = random.choice(ip_list)
        if req_type == 'insert':
            value = line_list[2]
            requests_dicts[random_ip].append(('insert',key,value, random_ip))
        elif req_type == 'query':
            requests_dicts[random_ip].append(('query', key, random_ip))

  start = time.time()
  for ip in ip_list:
    thread = Thread(target=common_functions.exec_requests, kwargs={'requests':
requests_dicts[ip]})
    threads_list.append(thread)
    thread.start()
  for t in threads_list:
    t.join()
  end = time.time()
  throughput = count/(end-start)
  print("Throuput of Chord = %.4f requests/second"%throughput, "%.4f seconds per
query"%(1/throughput))



if __name__ == '__main__':
  args = sys.argv
  if "--help" in args or len(args) == 1:
    print("You need to provide a command!")
  main()
```

Full project code:https://github.com/23gautam/CHORD-KV

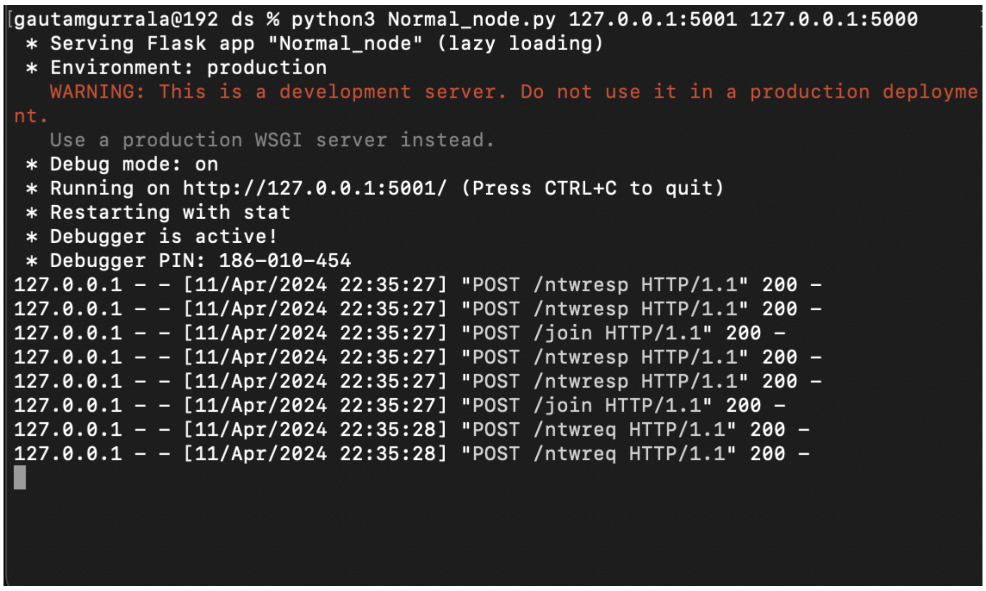
RESULTS

STARTING BOOTSRAP NODE(127..0.0.1:5000):



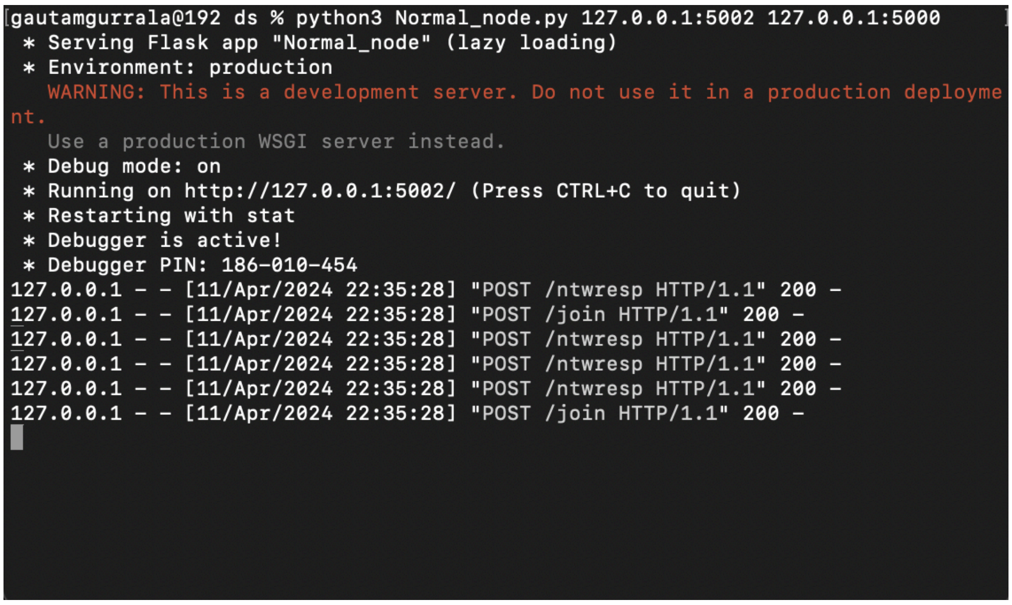
```
(myvenv) gautamgurrala@192 ds % python3 Bootstrap_node.py 127.0.0.1:5000 127.0.0
.1:5000 3 eventual
The Chord now starts...

 * Serving Flask app "Bootstrap_node" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
The Chord now starts...

 * Debugger is active!
 * Debugger PIN: 240-702-094
```

STARTING NODE(127.0.0.1:5001):



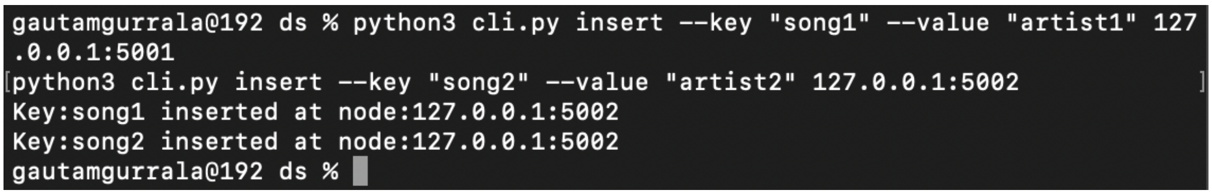
```
gautamgurrala@192 ds % python3 Normal_node.py 127.0.0.1:5001 127.0.0.1:5000
 * Serving Flask app "Normal_node" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 186-010-454
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /join HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:27] "POST /join HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwreq HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwreq HTTP/1.1" 200 -
```

STARTING NODE(127.0.0.1:5002):



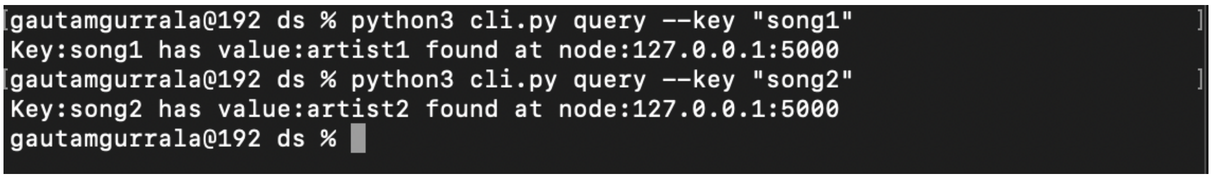
```
gautamgurrala@192 ds % python3 Normal_node.py 127.0.0.1:5002 127.0.0.1:5000
 * Serving Flask app "Normal_node" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5002/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 186-010-454
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /join HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /ntwresp HTTP/1.1" 200 -
127.0.0.1 - - [11/Apr/2024 22:35:28] "POST /join HTTP/1.1" 200 -
```

INSERT:



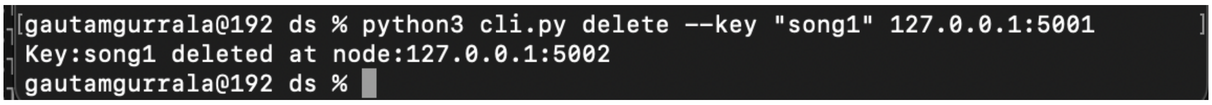
```
gautamgurrala@192 ds % python3 cli.py insert --key "song1" --value "artist1" 127
.0.0.1:5001
python3 cli.py insert --key "song2" --value "artist2" 127.0.0.1:5002
Key:song1 inserted at node:127.0.0.1:5002
Key:song2 inserted at node:127.0.0.1:5002
gautamgurrala@192 ds %
```

QUERY:



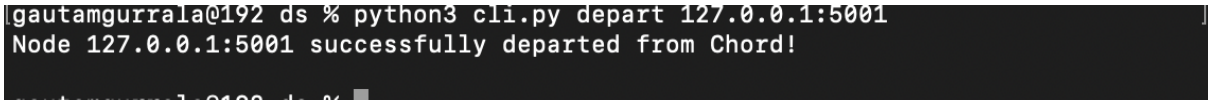
```
gautamgurrala@192 ds % python3 cli.py query --key "song1"
Key:song1 has value:artist1 found at node:127.0.0.1:5000
gautamgurrala@192 ds % python3 cli.py query --key "song2"
Key:song2 has value:artist2 found at node:127.0.0.1:5000
gautamgurrala@192 ds %
```

DELETING A KEY:



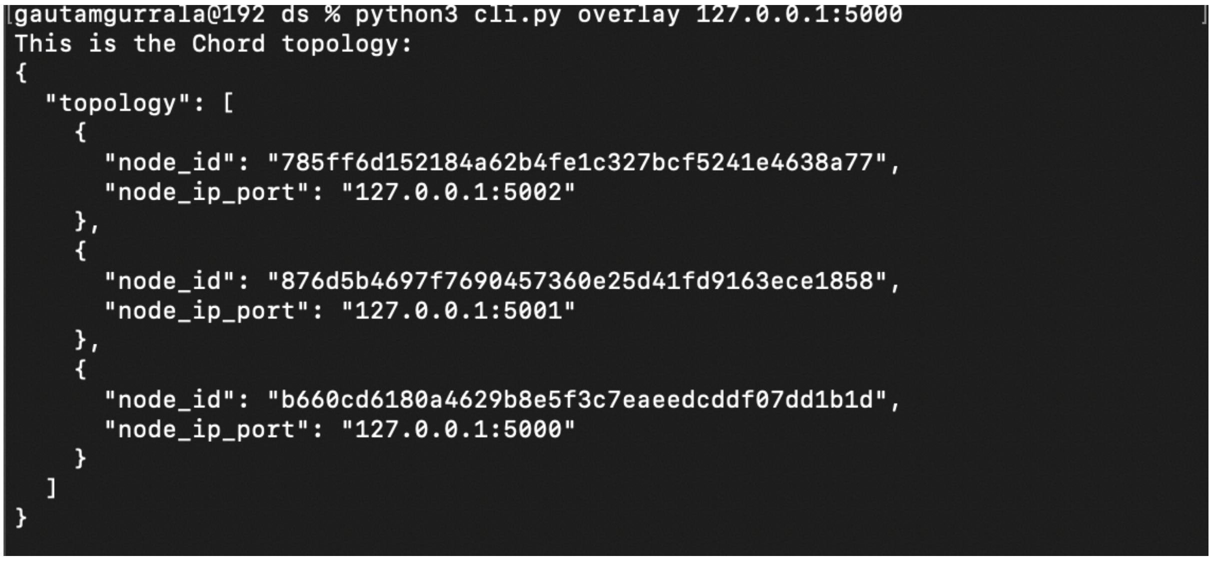
```
gautamgurrala@192 ds % python3 cli.py delete --key "song1" 127.0.0.1:5001
Key:song1 deleted at node:127.0.0.1:5002
gautamgurrala@192 ds %
```

REMOVING A NODE FROM OVERLAY:



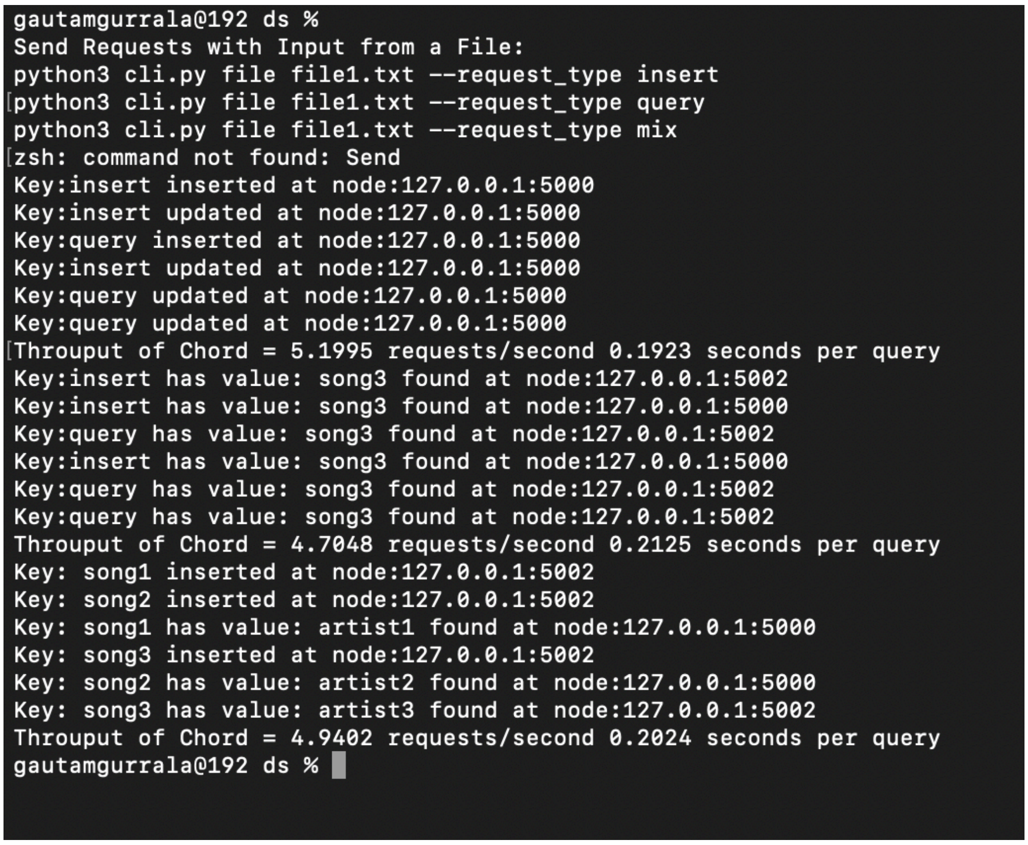
```
gautamgurrala@192 ds % python3 cli.py depart 127.0.0.1:5001
Node 127.0.0.1:5001 successfully departed from Chord!
```

DISPLAYING THE NODE OVERLAY:



```
gautamgurrala@192 ds % python3 cli.py overlay 127.0.0.1:5000
This is the Chord topology:
{
  "topology": [
    {
      "node_id": "785ff6d152184a62b4fe1c327bcf5241e4638a77",
      "node_ip_port": "127.0.0.1:5002"
    },
    {
      "node_id": "876d5b4697f7690457360e25d41fd9163ece1858",
      "node_ip_port": "127.0.0.1:5001"
    },
    {
      "node_id": "b660cd6180a4629b8e5f3c7eaeedcddf07dd1b1d",
      "node_ip_port": "127.0.0.1:5000"
    }
  ]
}
```

REQUEST HANDLING USING A



```
gautamgurrala@192 ds %
Send Requests with Input from a File:
python3 cli.py file file1.txt --request_type insert
[python3 cli.py file file1.txt --request_type query
python3 cli.py file file1.txt --request_type mix
[zsh: command not found: Send
Key:insert inserted at node:127.0.0.1:5000
Key:insert updated at node:127.0.0.1:5000
Key:query inserted at node:127.0.0.1:5000
Key:insert updated at node:127.0.0.1:5000
Key:query updated at node:127.0.0.1:5000
Key:query updated at node:127.0.0.1:5000
[Throuput of Chord = 5.1995 requests/second 0.1923 seconds per query
Key:insert has value: song3 found at node:127.0.0.1:5002
Key:insert has value: song3 found at node:127.0.0.1:5000
Key:query has value: song3 found at node:127.0.0.1:5002
Key:insert has value: song3 found at node:127.0.0.1:5000
Key:query has value: song3 found at node:127.0.0.1:5002
Key:query has value: song3 found at node:127.0.0.1:5002
Throuput of Chord = 4.7048 requests/second 0.2125 seconds per query
Key: song1 inserted at node:127.0.0.1:5002
Key: song2 inserted at node:127.0.0.1:5002
Key: song1 has value: artist1 found at node:127.0.0.1:5000
Key: song3 inserted at node:127.0.0.1:5002
Key: song2 has value: artist2 found at node:127.0.0.1:5000
Key: song3 has value: artist3 found at node:127.0.0.1:5002
Throuput of Chord = 4.9402 requests/second 0.2024 seconds per query
gautamgurrala@192 ds %
```

CONCLUSION:

The development and implementation of the Chord Protocol represent a significant milestone in the field of distributed systems and peer-to-peer (P2P) networks. Through this project, we have aimed to address the challenges and complexities associated with decentralized data storage and retrieval, with the goal of providing a scalable, fault-tolerant, and efficient solution for distributed hash table (DHT) management.

Throughout the development lifecycle, the Chord Protocol implementation has undergone meticulous analysis, design, implementation, and testing to ensure its functionality, reliability, and performance. By leveraging principles of key-based routing, finger tables, and successor lists, the Chord Protocol offers a robust framework for organizing and accessing data in a decentralized manner.

The protocol's key features, including node join and departure mechanisms, data replication strategies, and stabilization procedures, have been carefully designed to meet the requirements of large-scale distributed systems while maintaining consistency, availability, and partition tolerance. The Chord Protocol's scalability and resilience enable it to handle dynamic network conditions, node failures, and data rebalancing efficiently.

In addition to its core functionality, the Chord Protocol offers opportunities for future enhancement and extension. Areas for potential improvement include further optimization of routing efficiency, integration with additional overlay protocols, and exploration of applications in distributed storage systems, content delivery networks, and decentralized applications.

Overall, the development of the Chord Protocol underscores our commitment to advancing the field of distributed systems and P2P networks. As distributed computing continues to play a crucial role in modern computing infrastructures, the Chord Protocol stands as a valuable contribution to the ongoing research and development efforts aimed at creating scalable, resilient, and decentralized systems.

REFERENCES:

1. Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

2. Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001). A Scalable Content-Addressable Network