Indian Institute of Information Technology Allahabad

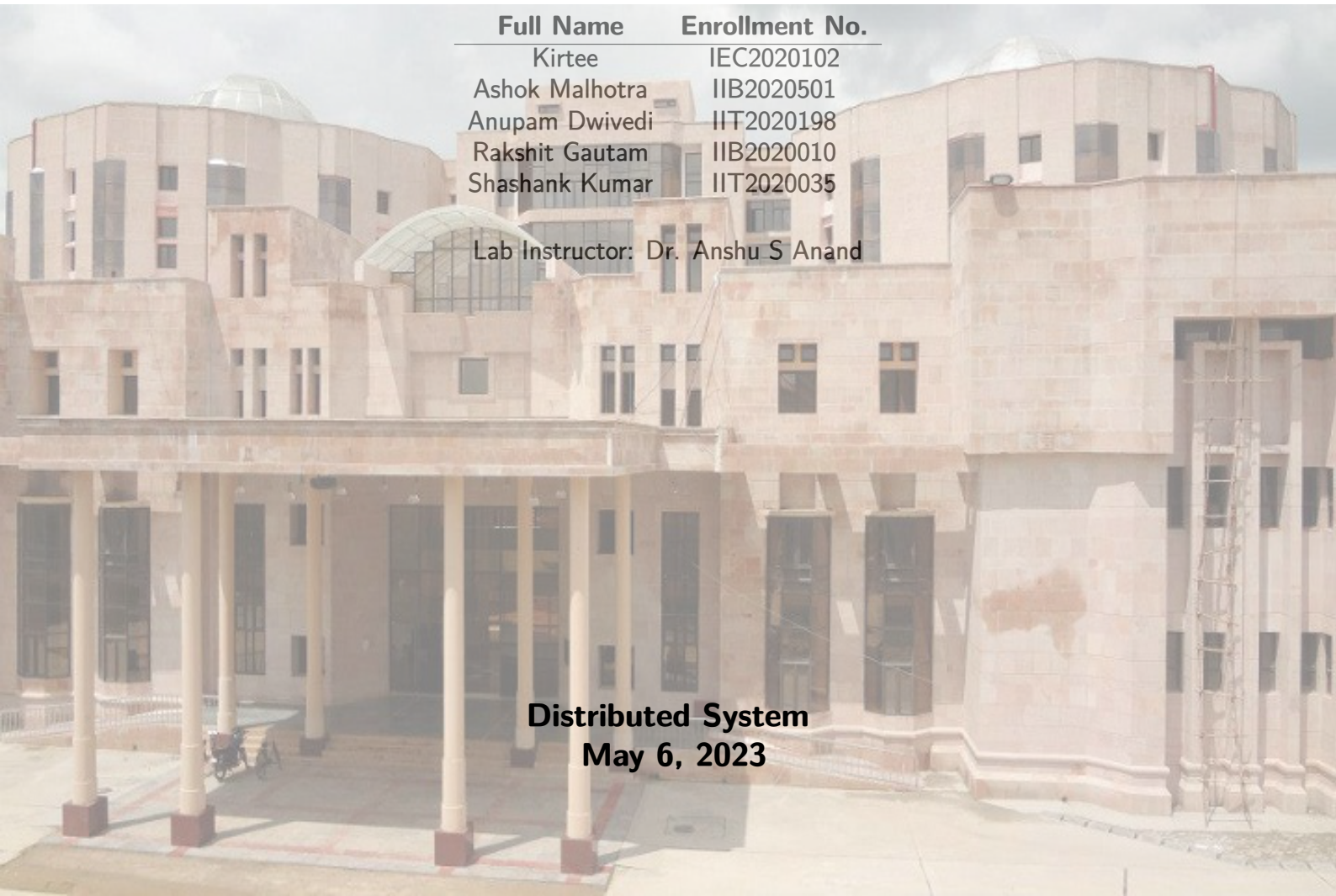
Distributed Hash Table

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Group 29

Full Name	Enrollment No.
Kirtee	IEC2020102
Ashok Malhotra	IIB2020501
Anupam Dwivedi	IIT2020198
Rakshit Gautam	IIB2020010
Shashank Kumar	IIT2020035

Lab Instructor: Dr. Anshu S Anand



Distributed System
May 6, 2023

Contents

1 | Introduction

A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table. key–value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data.One such DHT implementation is Chord, which provides a scalable, decentralized, and fault-tolerant method for organizing and accessing distributed data.

The Chord protocol is a decentralized peer-to-peer protocol that enables the efficient storage and retrieval of data across a large number of nodes. In Chord, nodes are arranged in a circular ring, with each node responsible for a range of keys in the ring. Each node maintains a routing table that enables it to efficiently route messages to other nodes in the ring.It facilitates one operation: mapping a key to a node. Depending on the application using Chord, that node may be responsible for storing a value linked to the key. Chord uses consistent hashing, a variant of hashing that balances the load since each node receives approximately the same number of keys, to assign keys to Chord nodes. Consistent hashing assumes that nodes are aware of most other nodes in the system, making it difficult to scale to large numbers of nodes. In contrast, each Chord node only requires routing information about a few other nodes, making it more scalable.

In a steady-state of an N-node system, each node maintains information about only O(LogN) other nodes and resolves all lookups via O(LogN) messages to other nodes.

Chord's simplicity, provable correctness, and performance distinguish it from other P2P lookup protocols. Routing a key through a sequence of O(logN) other nodes towards the destination is a simple task for Chord. While a Chord node requires information about O(LogN) other nodes for efficient routing, performance degrades gracefully when that information is out of date. Chord has a simple algorithm for maintaining this information in a dynamic environment, and only one piece of information per node needs to be correct for Chord to guarantee correct (though slow) routing of queries.

2 | System Model

Chord is a distributed hash table that simplifies the design of peer-to-peer systems and applications based on it by addressing difficult problems related to load balance, decentralization, scalability, availability, and flexible naming.

- Load Balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes. This provides a degree of natural load balance, as the cost of a Chord lookup grows as the logarithm of the number of nodes.
- Decentralization: Chord is fully distributed, which means that no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.
- Scalability: The cost of a Chord lookup grows as the logarithm of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
- Flexible Naming: Chord places no constraints on the structure of the keys it looks up, which means that the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

3 | Base Chords Protocol

The Chord protocol is a peer-to-peer lookup service for Internet applications that provides a scalable and decentralized solution for key-value pair storage and retrieval. It is based on a distributed hash table (DHT) that maps keys to nodes in a ring-shaped network topology, enabling efficient routing of queries with a logarithmic complexity in the number of nodes. Chord provides natural load balancing, automatic node joining and leaving, and flexible naming for keys. The protocol is implemented as a library that can be linked to client and server applications, and it provides lookup and notification functions for the applications to interact with the Chord network.

Table 3.1: e 1: Definition of variables for node n, using m-bit identif

Notation	Defination
finger[k].start	(n+2**(k-1) mod 2**m, 1¡=k¡=m
.interval	(finger[k].start, finger[k+1].start)
.node	first node ¿= n.finger[k].start
successor	ssor the next node on the identifier circle; finger[1]
predecessor	the previous node on the identifier circle

(a) Finger 1

(b) Finger 2

Figure 3.1: Chord protocol fingers

4 | Design and implementation

We implemented the Chord protocol using a Client-Server based model and **grpc** in python.

4.1 | gRPC

1. The key advantage of using gRPC over other RPC frameworks is its efficient use of network bandwidth. It uses binary serialization with protocol buffers to encode and decode messages, which is significantly faster and more compact than traditional text-based formats such as JSON or XML. This results in faster communication between clients and servers, which is especially important for high-performance applications.

2. Another feature of gRPC is its support for streaming, which allows clients and servers to send and receive streams of data in real-time. This is useful for building applications that require continuous data transfer, such as real-time analytics or chat applications.

4.2 | Proto Buffers

We design two proto buffers for our implementation- **'dhtpb2'** and **'dhtpb2grpc'**. These are used as the interface definition language (IDL) for describing both the service interface and the structure of the payload messages.

1. **Dhtpb2**: This code file includes the compiled definition of a protocol buffer schema in the file "dht.proto".
   [a] The DESCRIPTOR variable is a descriptor object that represents the schema defined in "dht.proto". It contains metadata about the protocol buffer types and fields, such as their names and data types.
   [b] The builder.BuildMessageAndEnumDescriptors function is called to generate the message and enum descriptors for the schema defined in DESCRIPTOR. These descriptors are Python classes that provide a way to construct, manipulate, and serialize instances of protocol buffer messages defined in the schema.
   [c] The builder.BuildTopDescriptorsAndMessages function is called to create top-level messages defined in the schema and their associated Python classes, which can be used to create and manipulate messages of the corresponding types.
2. **Dhtpb2grpc**
   [a] The code defines a Python class called DHTStub that serves as a client-side stub for a gRPC service defined in the dht.proto file. The class initializes with a grpc.Channel object, which it uses to create gRPC stub functions for each of the RPC methods defined in the DHT service.
   [b] It defines a class DHTServicer which implements methods such as join, leave, store, retrieve, findSuccessor, etc., which correspond to the different functionalities offered by the DHT service. Each method raises a NotImplementedError exception, indicating that they are not yet implemented, and sets the gRPC status code to UNIMPLEMENTED and a message to Method not implemented!.
   [c] The function **addDHTServicertoserver** takes an instance of DHTServicer and a gRPC server, and registers the servicer's methods with the server, specifying the request and response serializers for each method. This function maps the servicer's methods to corresponding gRPC handlers that will be called by the server when a client makes a request to the service.
   [d] The methods in this class are join(), leave(), store(), retrieve(), findSuccessor(), notify(), getPredecessor(), getSuccessor(), insertData(), retrieveData(), and replicate().Each of these methods is a gRPC unary-unary RPC call that takes a request object and some additional parameters and returns a response object. The target parameter specifies the destination address of the RPC call.The protobuf types used as input and output messages for these RPCs are imported from a file named **dhtpb2.py** in the dht package.
3. **Dht.proto**
   [a] The syntax = "proto3"; statement indicates that this file uses the protocol buffer version 3 syntax.
   [b] The package dht; statement specifies that all the messages and service in this file belong to the dht package.
   [c] The message keyword is used to define new message types. The NodeInfo message has two fields, id and address, both of type string. The KeyValue message has two fields, key and value, also of type string. The SuccList message has a single repeated field, vt, which is a list of NodeInfo messages.
   [d] The Empty message is an empty message type that is used as a placeholder in the service definition.
   [e] The service keyword is used to define an RPC service. The DHT service has 16 RPC methods, each defined using the rpc keyword. Each method specifies a request message and a response message. For example, the join method takes a NodeInfo message as input and returns a NodeInfo message as output. The getSuccessorList method takes an empty message as input and returns a SuccList message as output.

4.3 | Server

The server contains a class DHT which inherits from rpc.DHTServicer, which suggests that it is meant to be used as a gRPC service.

1. The class has an init method, which is the constructor of the class. It takes an address parameter and initializes various instance variables of the DHT object, such as self.nodeid, self.data, self.successor, self.predecessor, self.fingerTable, and self.successorList. It then calls several methods, such as self.stabilize(), self.fixFinger(), self.updateSuccessorList(), self.checkPredecessor(), and self.checkSuccessor(), which perform periodic tasks to keep the DHT nodes updated and consistent. Finally, the constructor starts a gRPC server, registers the DHT object as a gRPC service, and starts the server to listen on the specified address.

2. The DHT class also has other methods, such as self.stopServer(), which stops the gRPC server; self.hash(self, key), which hashes a given key using SHA-1 algorithm and returns an integer; self.isbetween(self, key, start, end), which checks whether a given key is between start and end in a circular space; self.join(self, request, context=None), which allows a node to join the DHT by contacting an existing node and updating its predecessor and successor; and self.leave(self, request,context), which allows a node to leave the DHT by updating its successor and predecessor.

3. The join method is used to join a node to the DHT network by contacting an existing node and updating its predecessor and successor. The method creates a NodeInfo object with the node's id and address and assigns it to the node's successor and predecessor if they are None. If the node already has a successor, the method forwards the join request to its successor node.

4. The leave method is used to remove a node from the DHT network. The method first checks if the node to be removed is the current node. If it is, it updates the predecessor's successor and successor's predecessor and sets the current node's successor and predecessor to None. Then it stops the server. If the node to be removed is not the current node, the method forwards the leave request to the current node's successor.

5. The store method is used to store a key-value pair in the DHT network. The method first hashes the key to an integer value and checks whether the current node should store the key-value pair or forward it to its successor. If it should store the pair, it stores it in the data dictionary. If it should forward the pair, it sends an insertData request to its successor.

6. The retrieve method is used to retrieve a key-value pair from the DHT network. The method first hashes the key to an integer value and checks whether the current node should retrieve the pair or forward the request to its successor. If it should retrieve the pair, it looks up the key in its data dictionary. If it should forward the request, it sends a retrieve request to its successor.

7. closestPrecedingNode(self, request) - This method is used to find the closest preceding node to a given request node in the finger table of the current node. It starts from the highest index in the finger table and checks if the node at that index is not None and if it is between the current node and the request node. If the node at that index satisfies these conditions, it is returned as the closest preceding node.

8. stabilize(self) - This method is used to maintain the stability of the DHT by updating the successor node and notifying the successor about the current node's existence. It checks if the successor node is not None, creates a gRPC channel to the successor node, and retrieves its predecessor node. If the predecessor node of the successor node is not None and is between the current node and its successor node, it updates the current node's successor to that node. It also notifies the successor node about its existence.

9. notify(self, request, context) - This method is used by a successor node to notify its predecessor node about its existence. If the predecessor node of the current node is None or if the successor node is between the predecessor and the current node, the current node's predecessor is updated to the successor node.

10. fixFinger(self) - This method is used to update the finger table of the current node. It selects a random index in the finger table and finds the successor node for the key that corresponds to that index. If a valid successor node is found, it updates the finger table with that node.

11. getPredecessor(self, request, context) - This method is used to retrieve the predecessor node of the current node.

12. getSuccessor(self, request, context) - This method is used to retrieve the successor node of the current node.

13. insertData(self, request, context) - This method is used to insert a key-value pair into the DHT. It computes the hash of the key and stores the key-value pair in the corresponding node. If there are immediate successor nodes to the current node, it replicates the key-value pair to them as well.
14. retrieveData(self, request, context) - This method is used to retrieve the value associated with a given key in the DHT. It computes the hash of the key and checks if the key exists in the data stored at the current node.
15. replicate(self, request, context) - This method is used to replicate a key-value pair to the current node.
16. updatePredecessor(self, request, context) - This method is used to update the predecessor node of the current node.
17. updateSuccessor(self,request,context) - This method is used to update the successor node of the current node.
18. getSuccessorList(self,request,context) - This method is used to retrieve the successor list of the current node. It returns a SuccList object that contains the first immediate successor of the current node, if it exists.

4.4 | Client

We defined a Client which interacts with the server. The Client class has several methods for interacting with the DHT:

1. init(self, address): Initializes a new Client object with the specified network address. It creates a NodeInfo object for the client node with a unique ID generated by hashing the address using SHA-1 and storing it as a string. It also creates a gRPC channel to connect to the DHT at the specified address and creates a stub for making RPCs to the DHT.
2. join(self, node): Joins the client node to the DHT by calling the join RPC on the stub with the provided NodeInfo object for the existing node in the DHT that the client should connect to.
3. store(self, data): Stores the provided data in the DHT by calling the store RPC on the stub with the provided data.
4. retrieve(self, data): Retrieves the value associated with the provided data key from the DHT by calling the retrieve RPC on the stub with the provided data.
5. stabilize(self): Performs the stabilize RPC on the stub to maintain the consistency of the DHT and ensure that each node has the correct predecessor and successor nodes.
6. notify(self, node): Notifies the specified node of a change in the predecessor node by calling the notify RPC on the stub with the provided NodeInfo object for the new predecessor node.
7. fixFingers(self): Fixes the finger table of the client node by calling the fixFingers RPC on the stub.
8. getPredecessor(self): Gets the predecessor node of the client node by calling the getPredecessor RPC on the stub. It also logs the predecessor node ID and starts a timer to call this method again after 10 seconds.
9. getSuccessor(self): Gets the successor node of the client node by calling the getSuccessor RPC on the stub. It also logs the successor node ID and starts a timer to call this method again after 10 seconds.
10. leave(self, node = None): Leaves the DHT by calling the leave RPC on the stub with the provided NodeInfo object for the client node or the specified node. If no node is specified, the client node is used

5 | Implementation Results

The implementation results for the protocol are summarized using the following images. In Fig 5.1 we get the main screen that is displayed when we run the protocol. In Fig 5.2 we demonstrate how to add a node. In Fig 5.3 we demonstrate how to remove a node. In Fig 5.4 it is demonstrated how a key-value pair is added to the DHT. Similarly in Fig 5.5 retrieval of a value for the corresponding key is demonstrated.

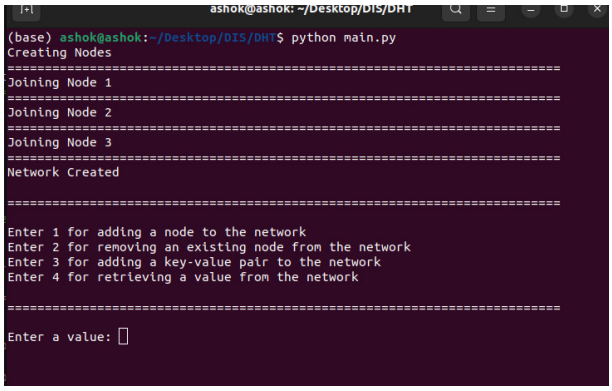


Figure 5.1: DHT 1

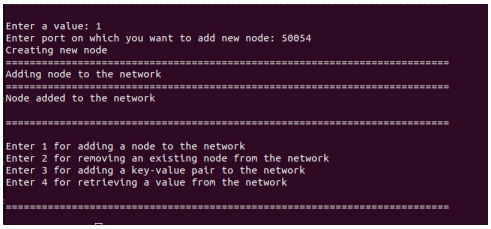


Figure 5.2: DHT 2

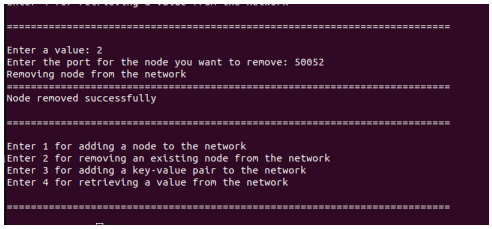


Figure 5.3: DHT 3

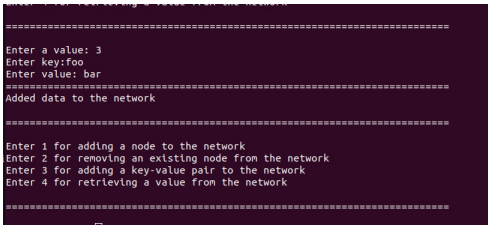


Figure 5.4: DHT 4

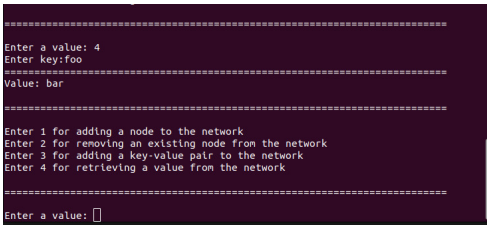


Figure 5.5: DHT 5

6 | References

- Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications by Ion Stoica , Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
- Distributed Hash Tables; Chord by Smruti R. Sarangi Department of Computer Science Indian Institute of Technology New Delhi, India https://www.cse.iitd.ernet.in/~srsarangi/courses/2020/col_819_2020/docs/chord.pdf
- Wikipedia; Chord(peer-to-peer) https://en.wikipedia.org/wiki/Chord_(peer-to-peer)