Project Proposal Document

Team 17:

- Keval Jain (2021111030)
- Sriteja Reddy Pashya (2021111019)
- Romica Raisinghani (2021101053)

Project Assigned - **Decentralised DHT — CAN**

1. Problem Statement

This project aims to develop a decentralized Distributed Hash Table (DHT) using Content Addressable Network (CAN) principles. The system will enable efficient key-value storage and retrieval in a distributed environment without central coordination. Our implementation will address the following challenges:

- Efficient partitioning of a multi-dimensional coordinate space among peers
- Reliable routing of requests with minimal network hops
- Maintaining system availability despite node failures
- Ensuring data security and integrity
- Optimizing performance through strategic caching and replication

2. Deliverables

2.1 In-Scope Features

1. **Virtual Coordinate Space:**
    - Implementation of a d-dimensional coordinate space.
    - Mechanisms for mapping keys to coordinates using consistent hashing.
2. **Basic Node Operations**:
    - Node join protocol with zone splitting
    - Basic neighbour maintenance
    - Simple node departure handling
3. **Routing Protocol**:
    - Greedy routing using Euclidean distance metric
    - Maintenance of neighbour sets
4. **Key-Value Operations**:
    - PUT: Store a key-value pair
    - GET: Retrieve a value by key
    - DELETE: Remove a key-value pair
5. **Basic Fault Tolerance**:
    - Simple replication to immediate neighbours
    - Basic failure detection via heartbeats

Enhanced Fault Tolerance: Our initial replication was limited to immediate neighbors, but to handle large-scale failures, we'll extend it to multi-hop neighbors. We'll introduce a parameter r to ensure each node has at least r replicas. If immediate neighbors fail, data will replicate to nodes within a few hops, identified via the coordinate space and updated with heartbeats. We may adjust the hop distance or replication logic based on testing. ****

6. **Security:**
   - AES-GCM to encrypt key-value pairs before storing them - For this we will generate a symmetric encryption key and encrypt values before storing (PUT operation), decrypt values during retrieval (GET operation).
   - Optional mTLS for node-to-node communication
     mTLS is a good-to-have feature, not a core deliverable. Our focus is on implementing AES-GCM encryption for key-value pairs and HMAC-SHA256 for integrity. After completing these and other essentials (like routing and replication), we'll add mTLS for secure node-to-node communication using a simple certificate system if time permits.
   - Access control mechanisms
7. **Data Integrity**
   - Hash validation of stored values
   - HMAC for Integrity Verification - Use HMAC-SHA256 to generate a cryptographic hash for each key-value pair. For this Compute HMAC when storing a key-value pair. Store the HMAC alongside the value. On retrieval, recompute the HMAC and compare it with the stored value.
   - Periodic integrity checks
   - Conflict resolution for inconsistent data
8. **Load Balancing**
   - Uniform partioning on node join
   - Dynamic zone adjustment based on load
   - Hot-spot detection and mitigation
   - Request distribution strategies
9. **Caching:**
   - Nodes maintain an LRU cache for recently accessed key-value pairs.
   - On a GET request, check cache first; if cache hit, serve locally; otherwise, forward.
   - Auto-populate cache on request forwarding; expired via TTL mechanism.
10. **Replication:**
    - Nodes track GET frequency and replicate hot data to direct neighbors when overloaded.
    - Replicated nodes serve requests probabilistically (`p`) or forward (`1-p`) to balance load.

    - Replicas expire via TTL or request frequency drop; use gRPC for message passing.

11. **Visualization Tool**:
    - As the project progresses, we will figure out ways to visualize our implementation.
      We'll create a web-based visualization tool showing the CAN coordinate space in 2D, with these MVP features:
      1. *Nodes*: Displayed as labeled points in the space.
      2. *Zones*: Shown as rectangles around each node to indicate partitions.
      3. *Key-Value Pairs*: Marked as colored dots within zones.
      4. *Routing Paths*: Animated lines showing GET/PUT request paths between nodes.

      We'll use Flask (Python) or Gin (Go) for the backend and D3.js for visuals, finalizing the stack after core deliverables are complete. We might add extras like showing failed nodes if time allows, but it's not set yet.

3. Technology Stack

We will use **Go/Python** as our primary programming language

Supporting technologies:

- **gRPC**: For node-to-node communication protocols
- **BoltDB/LevelDB**: For persistent storage of key-value pairs

4. Detailed Features

Key-Value Operations

The implementation will support standard DHT operations:

1. **PUT**: Hash the key to coordinates, route to the responsible node, store the value
2. **GET**: Hash the key, route to the responsible node, retrieve the value
3. **DELETE**: Hash the key, route to the responsible node, remove the value