Name	Prakriti Sharma
Subject	CS550 Advance Operating Systems
CWID	A20575259
Topic	PA3

Design Document for Decentralized Peer-to-Peer System

Table of Contents

1. Overview

This document describes the design and implementation of a decentralized peer-to-peer (P2P) system, which builds upon the centralized architecture established in Programming Assignment 2 (PA2). The primary goal of this project was to enhance scalability, fault tolerance, and network efficiency by utilizing a decentralized structure where all nodes (peers) are equal participants in the network.

The system implements a Distributed Hash Table (DHT) for storing topics, allowing for efficient storage and retrieval without relying on a central authority. A hypercube topology is used to connect peer nodes, which facilitates efficient communication and routing of requests. This document outlines the design choices made during the implementation, including the functionalities supported, potential trade-offs, and possible improvements.

2. System Architecture

2.1 Peer Nodes

Peer nodes are the fundamental units of the system, where each node is capable of independently hosting topics and processing client requests. The design enables multiple peer nodes to run concurrently, either on the same machine or distributed across different machines. Each peer node must fulfill several responsibilities:

- Hosting Topics: Each peer can create, store, and manage topics, ensuring that topics are non-overlapping across the network.
- Request Handling: Peer nodes handle incoming requests for publishing messages, subscribing to topics, and managing topics.
- Communication: Nodes communicate with each other to forward requests and replies, maintaining network cohesion.

2.2 Distributed Hash Table (DHT)

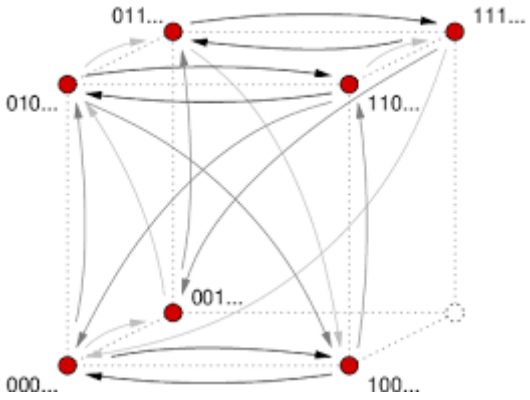
The DHT is a crucial component of the architecture that allows for the distributed management of topics. In this system:

- Storage Mechanism: Each peer node is responsible for a subset of the DHT, defined by a unique range of hashes derived from topics. This ensures that no two nodes manage the same topics, adhering to the design requirement of non-overlapping topics.
- Hash Function: A custom hash function was implemented to map topic names to locations within the DHT. This function is designed to minimize collisions and distribute topics evenly among peers, ensuring efficient access and management.
- Decentralization: The DHT structure allows for operations without a central coordinator, enabling the system to scale efficiently as nodes are added or removed.

2.3 Hypercube Topology

The peer nodes are organized in a hypercube topology, characterized by:

- Unique Identifiers: Each node is assigned a unique binary identifier (e.g., 000, 001, 010), where the number of bits corresponds to the number of nodes. In our case, eight nodes (3-bit identifiers) create a fully connected network.
- Connection Logic: Nodes are directly connected if their binary identifiers differ by exactly one bit. This connection scheme ensures that messages can be routed efficiently, with a maximum of log2(n) hops required to reach any node in the network.
- Routing Requests: The hypercube structure allows for a straightforward request routing mechanism, where a peer can forward requests to its neighbors until the target node is reached.



3. Implementation Details

3.1 Peer Node Implementation

The implementation of peer nodes consists of several key classes:

- PeerNode.java: This class represents individual peer nodes. It manages incoming connections and requests while interfacing with the DHT to store and retrieve topics. It also contains the logic for forwarding requests to other peers based on the routing mechanism.

Key functionalities include:

  - Listening for incoming connections on a specified port.
  - Accepting client requests and interacting with the TopicManager.
  - Connecting to other peers based on the computed hash of the requested topic.
- TopicManager.java: This class is responsible for managing topics within the DHT. It handles operations such as creating topics, subscribing users to topics, and removing topics.

Core functionalities include:

    - Computing the hash of a topic to determine its storage node.
    - Managing subscriptions to ensure clients receive messages from topics of interest.

- DHT.java: This class implements the structure of the distributed hash table, managing the segments of the DHT that each peer is responsible for. It provides methods for storing and retrieving topics based on their hashed values.

3.2 API Support and Functionality

All APIs from PA2 were integrated into the new architecture, ensuring continuity of functionality. The supported APIs include:

- Create Topic: Clients can create new topics, which are distributed to the appropriate peer based on the hash.
- Subscribe to Topic: Clients can subscribe to existing topics, receiving updates as messages are published.
- Publish Message: Clients can publish messages to a topic, which are then routed to the appropriate peer for storage.
- Pull Messages: Clients can request messages from a specific topic.
- Delete Topic: Clients can delete a topic, which involves removing it from the responsible peer's DHT segment.
- Fetch Event Logs: Each peer maintains a log of events (connections, messages sent/received) for monitoring and debugging purposes.

3.3 Asynchronous Communication

Asynchronous I/O is utilized for handling requests to improve system responsiveness. The implementation details include:

- Non-blocking I/O: Java NIO (New Input/Output) is used to create non-blocking server sockets, allowing peer nodes to accept multiple connections without stalling other operations.
- Thread Management: Each incoming request is handled in a separate thread, enabling peers to process multiple requests concurrently. This design allows for high throughput and responsiveness, even under heavy load.

3.4 Logging and Monitoring

Logging is implemented to maintain transparency and facilitate debugging:

- EventLogger.java: This class captures key events, such as connections established, messages sent/received, API calls made, and timestamps.

- Log Management: Each peer node maintains its log, providing insight into its operations and the overall health of the network. These logs can be analyzed to monitor performance and identify issues.

4. Trade-offs in Design Choices

Several trade-offs were made in designing the decentralized P2P system:

- Complexity vs. Performance: Utilizing a hypercube topology introduces complexity in routing logic. However, this design choice enhances performance by allowing efficient direct communication among nodes, reducing the need for longer routing paths that would exist in simpler topologies.
- Asynchronous I/O vs. Simplicity: While asynchronous I/O improves performance by allowing multiple requests to be handled concurrently, it adds complexity in terms of error handling and managing the state of connections. This complexity necessitates careful synchronization mechanisms to avoid race conditions.
- DHT Design: A simple hash function was employed to minimize implementation complexity and ensure that topics are evenly distributed among peers. However, this choice may have limitations regarding load balancing in scenarios with uneven topic distributions.

5. Possible Improvements and Extensions

There are several avenues for enhancing the decentralized P2P system:

- Dynamic Node Management: Currently, the system assumes a fixed number of nodes. Implementing dynamic management would allow nodes to join or leave the network without disrupting ongoing operations. This could involve adjusting the DHT and routing mechanisms dynamically.
- Enhanced Load Balancing: Introducing advanced load-balancing algorithms could improve topic distribution across peers. Techniques such as consistent hashing or a more sophisticated distribution algorithm could help maintain balance as new topics are added.
- Security Features: Adding encryption for messages and secure authentication for peer connections would enhance the system's resilience against attacks. This is particularly important in a decentralized environment where nodes may not trust one another.
- Data Replication: Implementing data replication strategies could increase fault tolerance. By maintaining copies of topics on multiple nodes, the system can ensure availability even in the event of node failures, which is crucial for maintaining service continuity.
- Performance Metrics: Developing a comprehensive performance metrics framework could help evaluate the system's efficiency and identify bottlenecks. Metrics such as average response time, throughput, and latency could be tracked and analyzed.

6. Conclusion

The design and implementation of a decentralized peer-to-peer system effectively transition from a centralized architecture, providing enhanced scalability and resilience. The integration of a distributed hash table (DHT) and hypercube topology facilitates efficient topic management and communication between peers.

This document outlines the system's architecture, key implementation details, and design trade-offs. Future improvements can further enhance the system's robustness and scalability, ensuring that it meets the demands of a real-world application while maintaining efficient operation. The current system serves as a strong foundation for further development and experimentation in decentralized network architectures.