Tanmay Pramanick – A20541164

PA3 – Decentralized P2P Publisher-Subscriber System

DESIGN DOCUMENT

1. Introduction

This document outlines the design and functionality of a decentralized P2P (Peer-to-Peer) publisher-subscriber system. Unlike Assignment 2, which used a centralized server, this system relies on a distributed hash table (DHT) where each peer node is responsible for a unique set of topics. The nodes are connected in a hypercube topology, creating a structured yet flexible network that allows efficient topic management without a central coordinator. The overall goal is to improve scalability, reliability, and efficiency in managing topics across multiple nodes.

2. Architectural Overview

Our design transforms the centralized model into a fully distributed system where:

- Each peer node stores a distinct set of topics. No two nodes store the same topic, ensuring that data isn't duplicated.
- Nodes communicate directly with their immediate neighbors using asynchronous I/O, which means each node can handle multiple requests without being blocked by other operations.
- The system uses a hypercube network topology, where each node is connected to only a few others, making routing efficient while keeping the architecture simple.

Key Components

- **Peer Node:**

These nodes store topics, process publish/subscribe requests, and route messages.

- **Client (Publisher/Subscriber):**

Clients interact with peer nodes to publish or retrieve messages.

Publishers create topics and send messages, while subscribers pull messages from their subscribed topics.

3. Roles of Each Component

Peer Node

Each peer node is responsible for:

- **Storing Topics:** Using a hash function, each topic is mapped to a specific peer node, so only that node will host it. This ensures that each topic has a single home within the network.

- **Routing Messages:** The hypercube structure helps each node find the shortest path to any other node by hopping through its neighbors.

- Providing APIs for basic operations like creating topics, publishing messages, subscribing to topics, and pulling messages.

Client (Publisher/Subscriber)

- **Publishers:** Create topics on specific nodes and publish messages to them. They interact directly with the responsible peer node using the DHT to determine where each topic should be stored.

- **Subscribers:** Query the DHT to find topics, subscribe to them, and pull messages directly from the peer nodes that store the topics.

4. Design Decisions and Trade-offs

a) Decentralized DHT for Topic Storage

- **Decision:** Use a distributed hash table, where each node holds a unique portion of topics.
- **Trade-off:** This removes the single point of failure and enables scalability but can lead to uneven load distribution across nodes.

b) Hypercube Topology

- **Decision:** Connect nodes in a hypercube to minimize hops between them.
- **Trade-off:** Efficient routing with a balanced number of connections, but adding or removing nodes disrupts the fixed structure.

c) Hashing for Topic-to-Node Mapping

- **Decision:** Use hashing to map topics to nodes without manual configuration.
- **Trade-off:** Simple and fair distribution but can create hotspots if the hashing isn't perfectly uniform.

d) Asynchronous I/O for Concurrency

- **Decision:** Asynchronous I/O enables concurrent request handling on each node.
- **Trade-off:** Maximizes responsiveness but introduces complexity in managing errors and debugging.

e) Client-Side Routing Logic

- **Decision:** Clients independently locate topic nodes using routing logic.
- **Trade-off:** Reduces node overhead but increases client-side complexity.

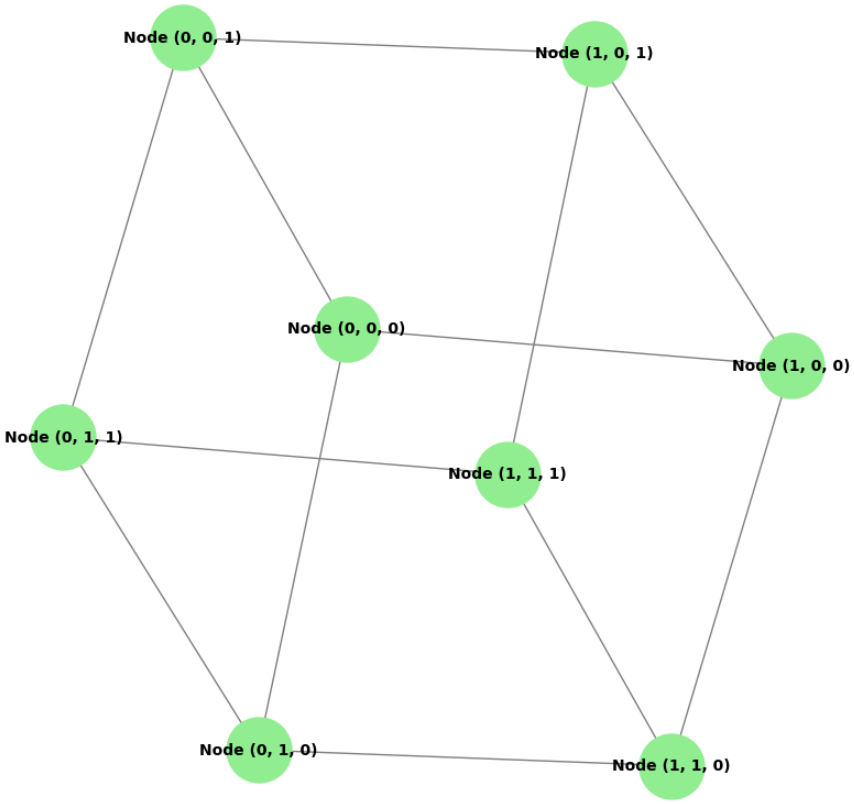
f) No Topic Replication

- **Decision:** Each topic is stored on a single node without redundancy.
- **Trade-off:** Simplifies design and storage needs but limits fault tolerance; topics are lost if a node fails.

g) Fixed Network Size

- **Decision:** Assume a static set of 8 nodes in a hypercube structure.
- **Trade-off:** Simplifies implementation but restricts scalability.

5. System Flow



1. **Node Startup**: Each peer node initializes with a unique identifier and calculates its neighbors in the hypercube structure.
2. **Topic Assignment Using Hashing**: Topics are hashed to assign them to specific nodes. The node identified by the hash function becomes the “owner” of that topic, meaning only that node will store and manage it.
3. **Client Actions**:
    - **Publishers** use hashing to locate the node that “owns” the topic they want to publish to. They then create the topic and send messages directly to that node.

- **Subscribers** use hashing to locate the node responsible for the topic they want to follow. They subscribe to the topic on that node and retrieve messages from it.

6. APIs and Functionality

Peer Node APIs:

- **Create Topic**: Registers a new topic on the designated peer node.
- **Publish Message**: Publishes messages to a topic if it exists on the peer.
- **Subscribe**: Registers a subscription to a topic on the peer node.
- **Pull Messages**: Retrieves messages from a topic on the peer node.
- **Delete Topic**: Deletes a topic from the peer node.

Client APIs:

- **Create Topic**: Sends a request to the appropriate node to create a new topic.
- **Delete Topic**: Deletes a topic from its designated node.
- **Publish Message**: Publishes a message to a specific topic.
- **Subscribe**: Subscribes to a topic to receive updates.
- **Pull Messages**: Pulls messages from the subscribed topic.

Sufficiency of APIs:

The APIs provided for creating topics, publishing, subscribing, and message pulling have proven to be effective in enabling decentralized P2P interactions. They cover essential functionalities and allow each peer node to handle independent client requests without needing centralized coordination. However, in a real-world setting, additional features like topic replication and fault tolerance mechanisms could further strengthen reliability.

7. Bottlenecks and Observations

Identified Bottlenecks

a) **Load Balancing**: Since topic ownership is determined by hashing, certain nodes may become overloaded if a high-traffic topic maps to them. A more advanced load balancing mechanism, such as topic replication or dynamic rehashing, could address this.

b) **Routing Efficiency**: The hypercube topology ensures low latency routing in a static network but is restrictive in handling node changes. Adapting a more flexible topology could allow better scalability but at the cost of routing complexity.

c) **Lack of Fault Tolerance**: Currently, if a node fails, all topics hosted on that node are lost. Adding topic replication across multiple nodes or periodically syncing topic data with neighbor nodes could mitigate this issue, though it would add to the storage and communication overhead.

9. Future Improvements and Extensions

a) **Topic Replication:** Implementing a system where topics are replicated across a few nodes could increase fault tolerance. This would require a modified hashing system that allows redundant topic storage.
b) **Dynamic Node Addition/Removal:** Although this version assumes a fixed network, future versions could support dynamic peer additions or removals. This would necessitate a more flexible topology and a distributed mechanism for rehashing topics when nodes join or leave.
c) **Enhanced Load Balancing:** The current hashing-based topic assignment can lead to uneven load distribution. Adaptive load balancing, where high-traffic topics are spread across multiple nodes, could improve system efficiency under heavy loads.
d) **Improved Error Handling and Debugging:** Asynchronous I/O introduces complexity in error tracking. Adding centralized logging or a monitoring tool could help manage and debug potential communication failures between nodes.

10. Conclusion

This decentralized publisher-subscriber system distributes topic storage across multiple peer nodes in a hypercube topology. The design eliminates any central dependency, allowing for robust scalability and resilience. Each node independently manages its subset of topics and coordinates with others to fulfil client requests. This model offers strong performance and flexibility, making it suitable for systems where real-time data sharing and scalability are critical.