Final Project: Design Doc

I. Introduction

This document describes the designs of a project which extends a P2P system with advanced features to support replication, fault tolerance, and dynamic topology configuration. These enhancements aim to improve system reliability, performance, and adaptability in dynamic environments. The implementation must consider trade-offs between performance, consistency, and overhead while meeting the additional requirements of concurrency and scalability.

The system basically achieves these key features: Distributed Peer-to-Peer Network, Data Topic Management/Replicating, Fault Tolerance, Concurrency and Communication, and Dynamic Topology Support.

This design document provides an overview of the Key Features Implemented, Core Implementation/Design Mechanisms, Trade-offs in Design, Testing and Benchmarking, Possible Improvements, and Conclusion.

II. Key Features Implemented

1. Distributed Peer-to-Peer Network

- **Node Representation and Network Structure**:

  a. Each node is identified by a binary peer_id.

  b. Neighbors are computed dynamically using the compute_neighbors method, ensuring that nodes differ by only one bit in their IDs, adhering to hypercube topology.

  c. Dynamic Topology Support: The system allows nodes to be added or removed dynamically at runtime, with the status updated via detect_node_status and ping_node.

- **Dynamic Routing**:

  a. Routing is implemented with the find_next_hop method, which determines the next hop node to move closer to the target node.

  b. The route_request method forwards requests based on calculated target nodes and routing information.

2. Data Topic Management

- **Topic Creation**:

  a. The hash_function computes the node responsible for storing a topic by hashing the topic name.

  b. The create_topic method creates topics either locally or forwards the request to the responsible node.

c. Each node maintains two storage structures:

d. self.topics: Stores topics and their messages managed by the node.

e. self.replicas: Stores replicas of topics from other nodes for fault tolerance.

- **Topic Replication**:

a. When creating a topic, the create_replicas method generates replicas on neighboring nodes for fault tolerance.

b. The replicate_topic method handles storing replica data.

- **Message Publishing and Subscription**:

a. publish_message publishes messages to a topic. If the node is not responsible, the request is routed to the appropriate node.

b. subscribe_to_topic allows a node to subscribe to a topic, with requests routed if necessary.

c. Messages are propagated to all subscribers using propagate_message.

- **Topic Deletion**:

a. Topics can be deleted via the delete_topic method, which also updates replicas to reflect the change.

3. Fault Tolerance

- **Node Failure Detection**:

a. Neighbors' statuses are periodically checked using ping_node and updated in self.active_nodes.

b. Offline nodes are excluded from active routes, and rejoining nodes are handled by handle_node_rejoin.

- **Replica-based Fault Recovery**:

a. If a node responsible for a topic goes offline, its replicas ensure continued access to the topic.

b. When a node rejoins, the replicate_topics method restores its topics from replicas.

4. Concurrency and Communication

- **Concurrency Management**:

a. Shared data is accessed with asyncio.Lock to ensure thread safety, such as with self.peer_lock and self.topic_lock.

b. The system uses the asyncio framework for asynchronous operations, enabling efficient network communication and concurrent handling of requests.

- **Network Communication**:

a. Messages are sent asynchronously to other nodes using the send_message method, which manages TCP connections.

b. The handle_client method processes incoming requests, delegating commands to handle_command.

5. Dynamic Topology Support

- **Adding New Nodes**:

a. New nodes are registered in the network via announce_new_node, triggering topic reallocation as needed.

b. Neighbor lists are updated dynamically using the compute_neighbors method.

- **Offline and Rejoining Nodes**:

a. Offline nodes are marked as inactive and removed from neighbor lists.

b. Rejoining nodes synchronize their topics using handle_node_rejoin.

III. Core Implementation/Design Mechanisms

1. Hypercube-based Topology Routing:

The hypercube topology ensures efficient routing, with nodes differing by only one bit in their IDs. Routing uses bit flipping to progressively approach the target node.

2. Replica-based Fault Tolerance:

- Topics are replicated on neighboring nodes to maintain access in case of failure.
- When nodes recover, they synchronize their topics from replicas.

3. Asynchronous Operations and Concurrency Support:

- asyncio enables concurrent handling of multiple requests and efficient communication.
- Locks prevent race conditions when accessing shared data.

4. Dynamic Node Status Detection and Updates:

- Periodic heartbeats (via ping_node) check the status of neighbor nodes, dynamically updating active node lists and neighbor relationships.

5. Publish/Subscribe APIs

The core publish/subscribe APIs include:

- Create Topic: Allows a peer to create a topic if it is responsible for it.
- Delete Topic: Allows a peer to delete a topic if it is responsible for it.
- Publish Message: Allows a peer to publish a message to a topic.
- Subscribe to Topic: Allows a peer to subscribe to a topic and receive messages from it.

IV. Trade-offs in Design

Design Aspect	Advantages	Trade-offs
Fault Tolerance	High availability, robust against failures	Increased latency and synchronization overhead
Hypercube Topology	Efficient routing $O(\log N)$	Complexity in neighbor computation and routing logic
Dynamic Topology Support	Flexibility to add/remove nodes	Increases synchronization challenges and potential for inconsistency
Asynchronous Communication	High concurrency and responsiveness	Increased complexity in error handling and debugging
Consistency vs. Availability	Ensures data integrity (fault recovery)	Performance overhead and potential stale data with eventual consistency
Fault Detection Accuracy	Accurate status tracking	Network overhead or detection latency depending on interval
Scalability	Supports large networks	Higher resource usage for replicas and state management

System Complexity	Rich feature set (dynamic, fault-tolerant, scalable)	Increased difficulty in debugging, testing, and maintaining the system

V. Testing and Benchmarking

1. API Latency Benchmark

Each API benchmark measures the start and end time of the operation and computes the latency as the difference.

2. Throughput Benchmark

Test how many requests can be handled concurrently by the system within a given time frame.

VI. Possible Improvements

1. Dynamic Test Configuration

- Issue: The number of peers, requests, and topic names are hardcoded.
- Improvement: Allow dynamic configuration via input arguments or configuration files.

2. Improved Fault Tolerance

- Issue: The current detect_node_status method periodically checks neighbors but does not handle edge cases like fluctuating node states or slow nodes.
- Improvement: Add a retry mechanism for pinging neighbors and maintain a retry count for each node before marking it as failed.

3. Efficient Topic Replication

- Issue: The create_replicas method does not ensure load balancing when creating replicas.
- Improvement: Use a round-robin or hashing mechanism to distribute replicas evenly across neighbors.

4. Graceful Node Recover

- Issue: Replicating topics during recovery might block other tasks.
- Improvement: Perform replication asynchronously to avoid blocking.

5. Consistent State Synchronization

- Issue: The synchronization mechanism (replicate_topics) lacks a consistent versioning system.
- Improvement: Use a version vector or timestamp to ensure replicas are up-to-date.

VII. Conclusion

This project builds upon a P2P system by incorporating advanced functionalities such as replication, fault tolerance, and dynamic topology management. These features are designed to enhance the system's reliability, performance, and flexibility in ever-changing environments. The implementation necessitates a careful balance between performance, consistency, and resource overhead, while also addressing the demands for concurrency and scalability.