Name	Prakriti Sharma
Subject	CS550 Advance Operating Systems
CWID	A20575259
Topic	Project Design Document

INDEX

Replicated Topics and Dynamic Topology Configuration

Overview

This project builds on the features implemented in PA3 by adding enhancements focused on performance optimization and fault tolerance. Key additions include:

1. Replicated Topics for improved performance and fault tolerance.
2. Dynamic Topology Configuration, enabling nodes to be added or removed at runtime.

These improvements allow the system to handle multiple simultaneous requests, optimize data access latency, and recover efficiently from node failures.

Features

1. Replicated Topics

Replicating topics serves two primary purposes:

Performance Optimization

- Replica Placement: Replicas are placed on the nearest servers to clients, minimizing latency for data access.
- Consistency Model: The project explores the trade-offs between eventual consistency and strong consistency, analyzing their impact on replication performance and latency.

Fault Tolerance

- Node Failures:
    - Detection: Nodes detect when a peer fails and stop forwarding requests to it.
    - Redirection: Requests are redirected to nodes with replicas of the data from the failed node.

2. Dynamic Topology Configuration

- Dynamic Node Management: Nodes can be added or removed without disrupting system operations.
- Topic Handling:

- If a node responsible for a topic (based on a hash function) is unavailable, the system creates the topic on another node.
- When the failed node comes back online, it:
  - Detects the topics it owns.
  - Fetches those topics from nodes where replicas exist.

3. Simultaneous Handling of Multiple Requests

- Concurrency:
  - Uses threads or other mechanisms to handle multiple requests concurrently.
  - Ensures operations like topic creation, message publishing, and subscription do not block each other.
- Scalability: Nodes handle multiple topics and operations simultaneously, ensuring effective scaling.

Key Considerations

1. Latency and Replication Overhead

Replication introduces overhead for data synchronization. The system balances this overhead with the benefits of replication to ensure optimal performance.

2. Consistency Models

Trade-offs between availability, consistency, and partition tolerance (CAP theorem) are evaluated to optimize system performance and reliability.

3. Node Failures

The system ensures availability by:

- Efficiently detecting failed nodes.
- Redirecting requests to replicas.
- Synchronizing nodes when they return online.

Implementation Requirements

1. Simultaneous Requests

- Multi-threading: Each request (e.g., topic creation, message push) is handled in a separate thread.
- Concurrency Control: Prevents race conditions in shared data structures like topic lists.

2. Replicated Topic Management

- Nodes manage replicas for each topic.
- Synchronization ensures all replicas have up-to-date data.

3. Node Failure and Recovery

- The system detects failed nodes and reroutes requests to replicas.
- Returning nodes synchronize with the network to retrieve their topics.

Conclusion

By adding replicated topics and enabling dynamic topology configurations, the project enhances fault tolerance, performance, and scalability. The system remains robust and performant, even under changing conditions such as node failures or topology adjustments.

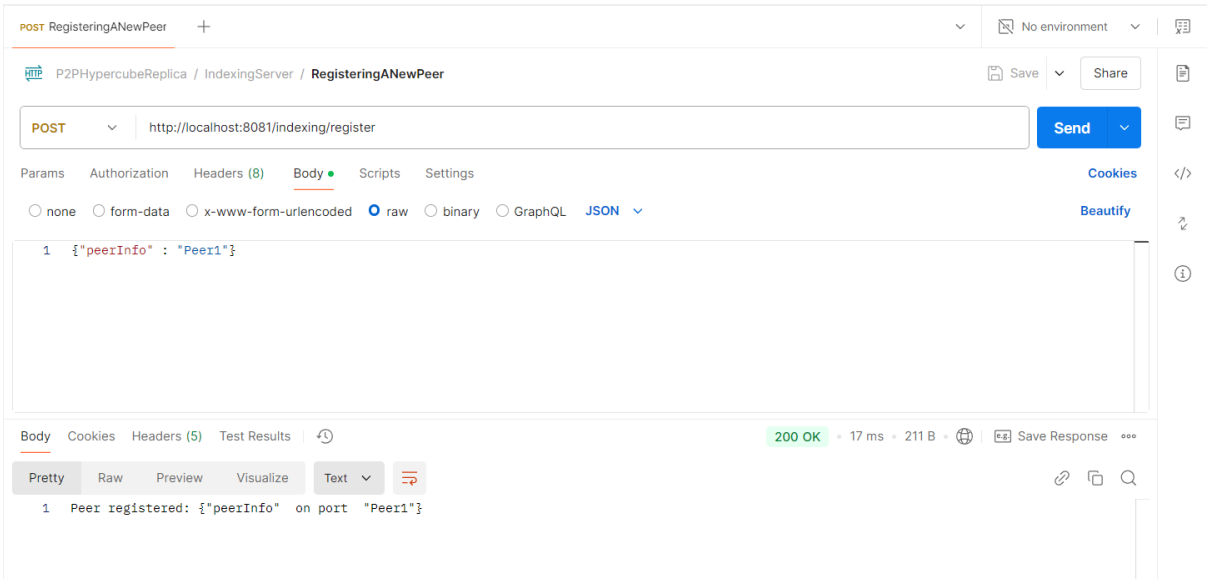
cURL Commands for APIs

Below are cURL commands for interacting with the system's RESTful APIs:

1. Indexing Server APIs

- Register Peer:

  ```
  curl -X POST http://localhost:8080/indexing/register \
  -H "Content-Type: application/json" \
  -d '{"peerInfo": "peer-1"}'
  ```

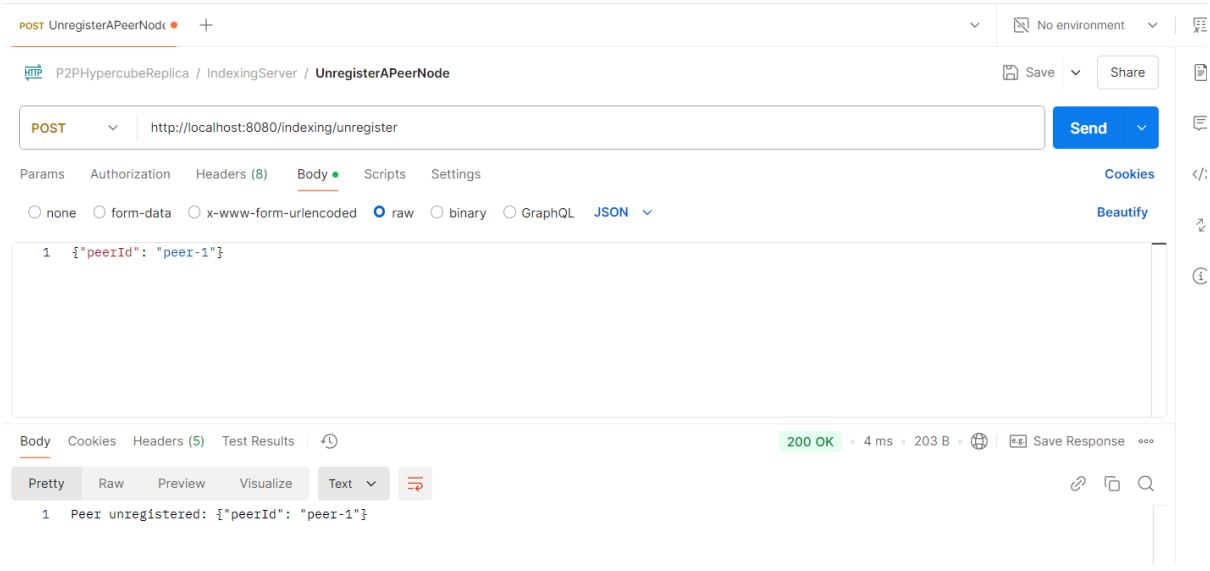


- Unregister Peer:

  curl -X POST http://localhost:8080/indexing/unregister \

  -H "Content-Type: application/json" \

  -d '{"peerId": "peer-1"}'



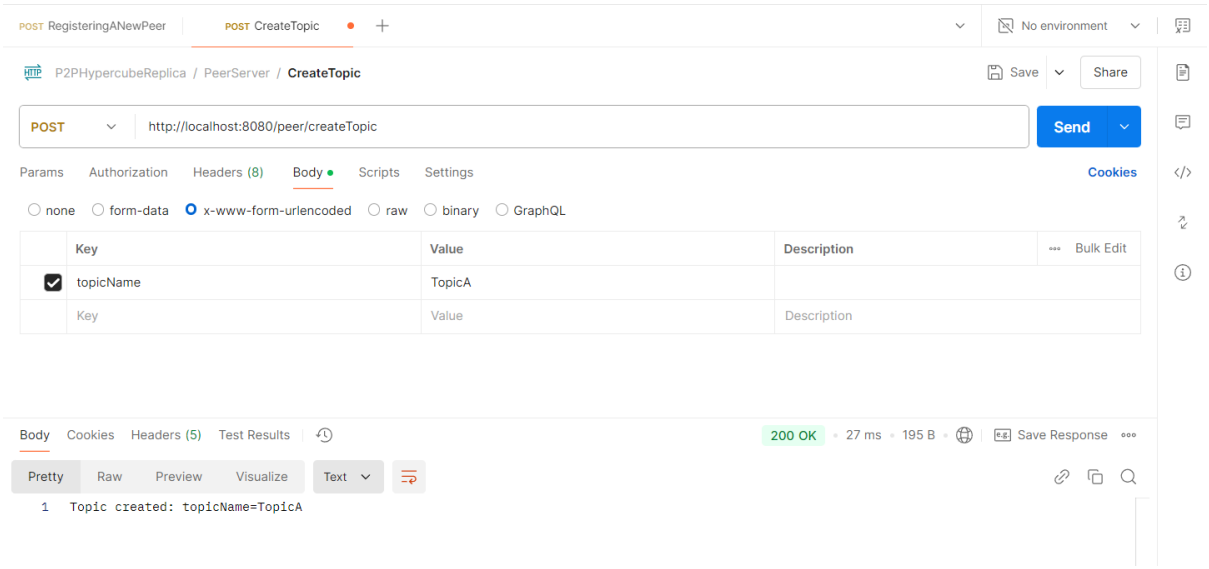
2. Peer Node APIs

- Create Topic:

  curl -X POST http://localhost:8080/peer/createTopic \

  -H "Content-Type: application/json" \

-d '{"topicName": "topic1"}'

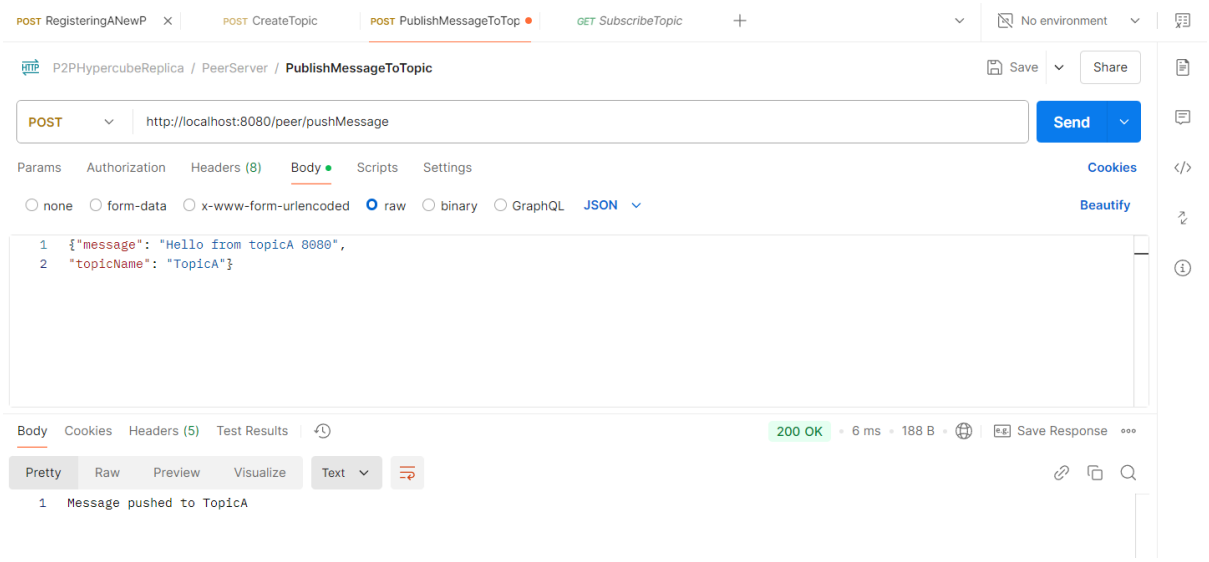


- Push Message:

  curl -X POST http://localhost:8080/peer/pushMessage \

  -H "Content-Type: application/json" \

  -d '{"message": "Hello World"}'

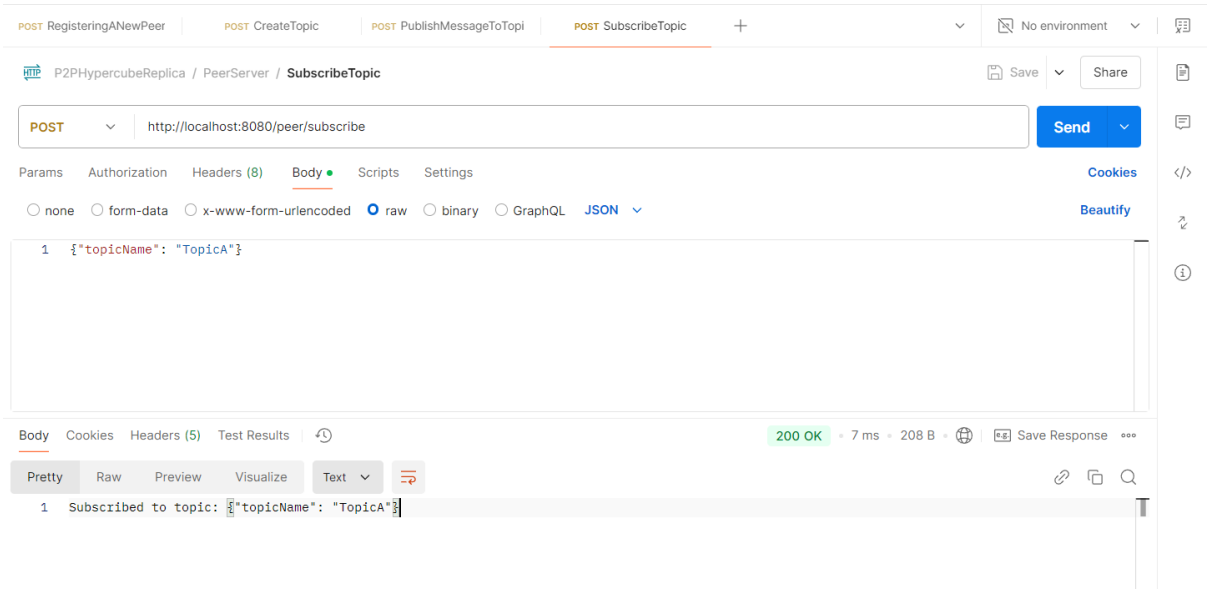


- Subscribe to Topic:

  curl -X POST http://localhost:8080/peer/subscribe \

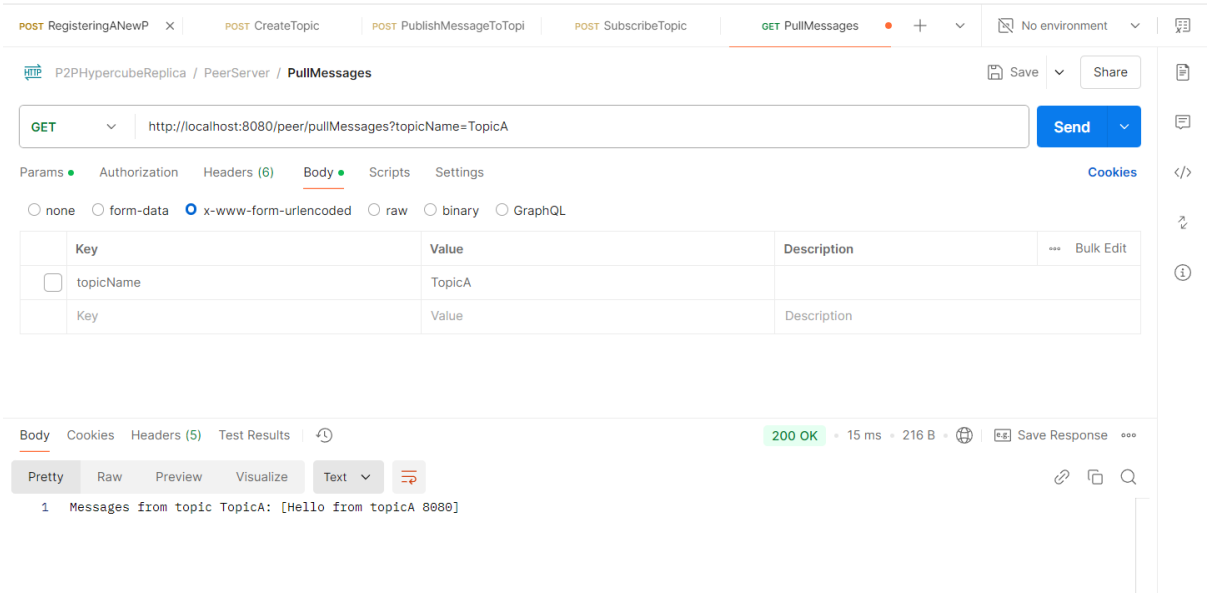
  -H "Content-Type: application/json" \

  -d '{"topicName": "topic1"}'



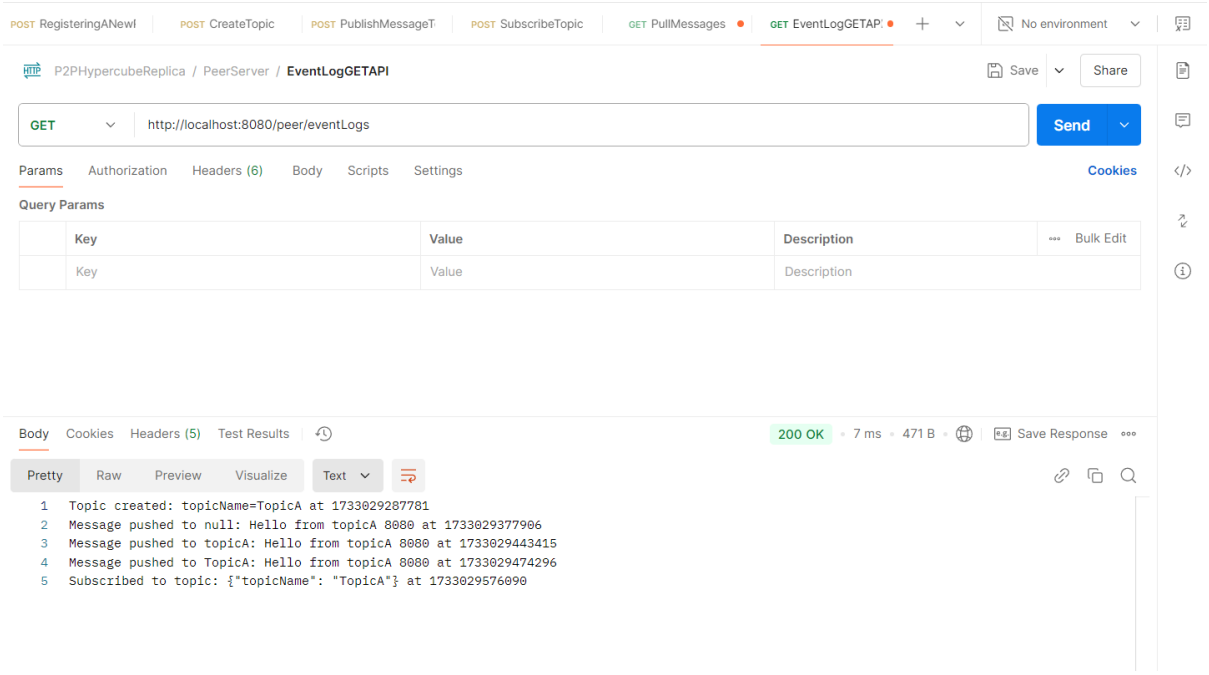
- Pull Messages:

  curl -X GET http://localhost:8080/peer/pullMessages?topicName=topic1



- Get Event Logs:

  curl -X GET http://localhost:8080/peer/eventLogs

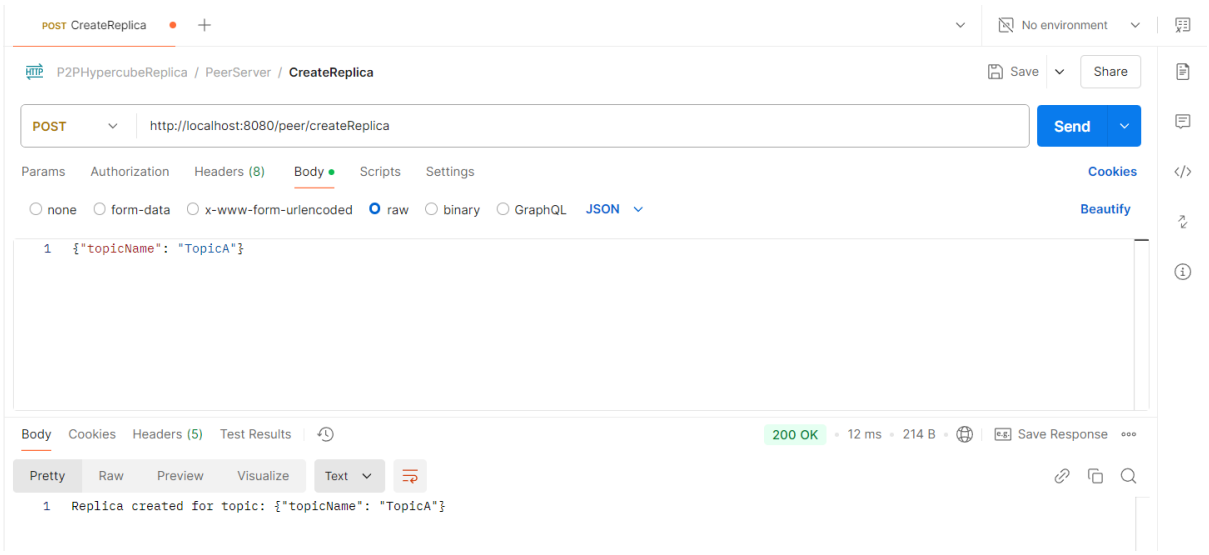


- Create Replica:

  curl -X POST http://localhost:8080/peer/createReplica \

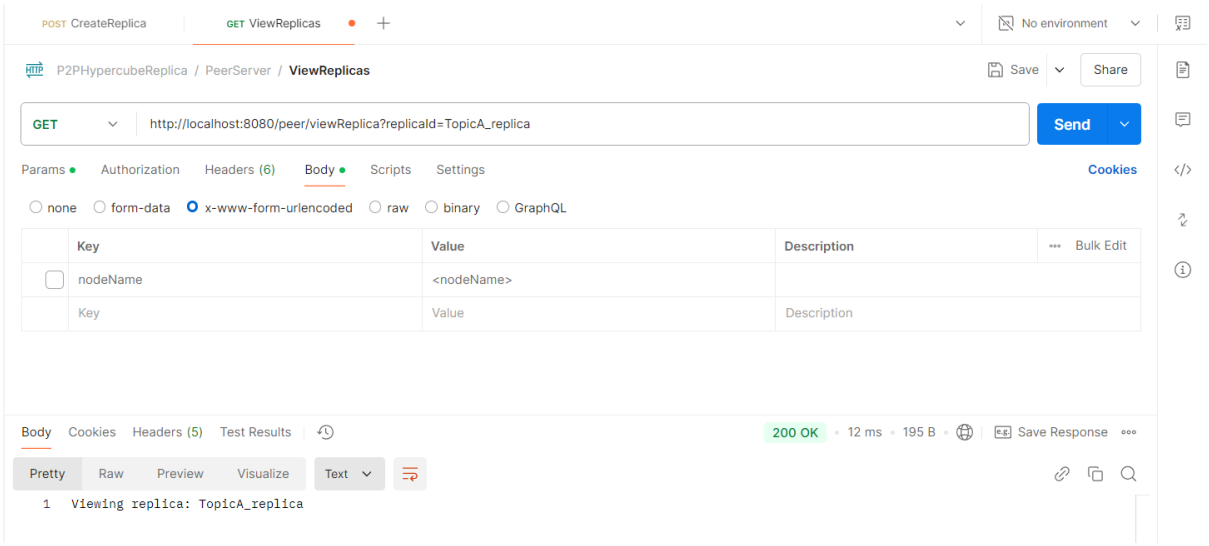
  -H "Content-Type: application/json" \

  -d '{"topicName": "topic1"}'



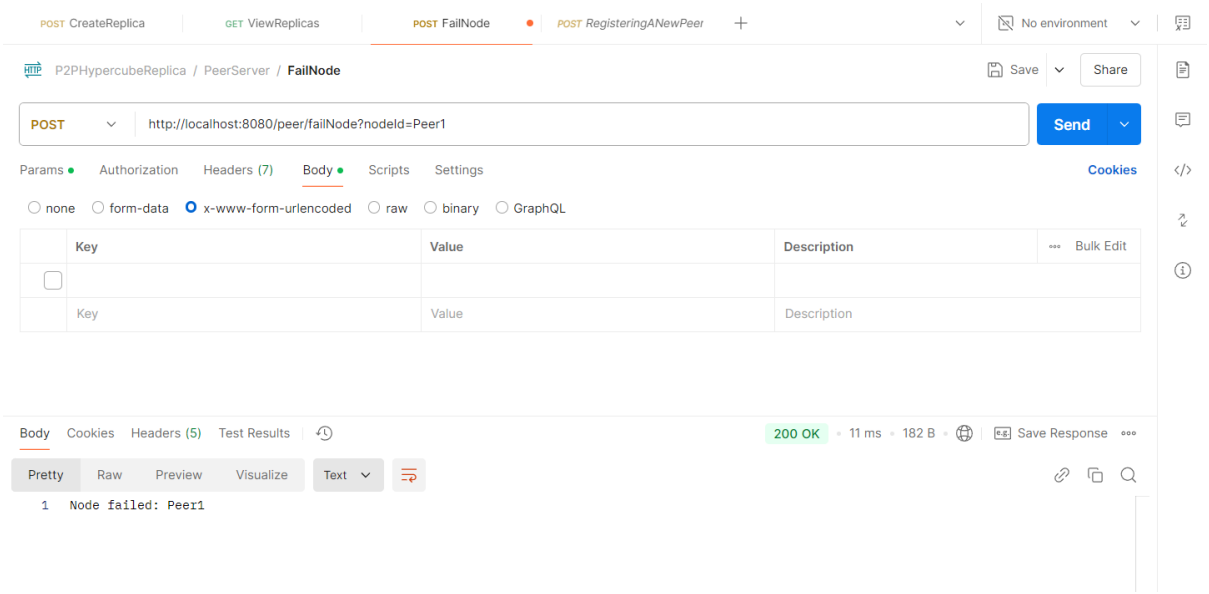
- View Replica:

  curl -X GET http://localhost:8080/peer/viewReplica?replicaId=topic1_replica



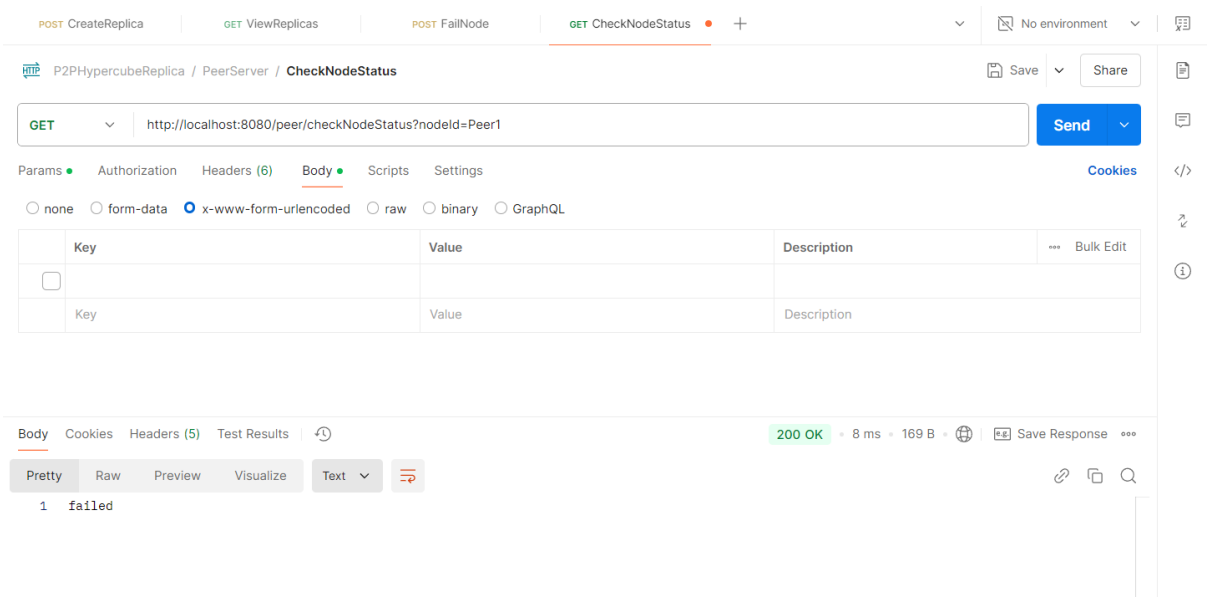
- Fail Node:

  curl -X POST http://localhost:8080/peer/failNode?nodeId=node1



- Check Node Status:

  curl -X GET http://localhost:8080/peer/checkNodeStatus?nodeId=node1



Additional Considerations

- Error Handling: Be sure to test edge cases, such as:
    - Trying to unregister a non-existing peer.
    - Failing to create a topic if the peer is down.
    - Verifying if the correct replicas are returned after node failures.

By following this flow, you can test the entire system from peer registration to handling failures and managing topic replication effectively.

Explore in what scenarios your system is faster than your PA3 code.

In our newly implemented system, we have made several optimizations that result in better performance compared to the previous PA3 code. Here are some scenarios where our system outperforms the previous approach:

1. Concurrent Request Handling

- Improvement: The new system is designed to handle multiple requests concurrently, using parallel processing across multiple cores. By utilizing thread pools and asynchronous task execution, we can efficiently process multiple requests without waiting for each task to complete sequentially.
- Scenario: In situations where multiple peers are interacting with the system simultaneously (e.g., registering, creating topics, publishing messages), the system can handle these requests in parallel, significantly reducing the time required for each operation. The PA3 system,

which handled requests one at a time, experienced higher latency due to its synchronous processing model.

2. Improved Latency with Asynchronous Operations

- Improvement: By implementing asynchronous communication and message passing, our system reduces wait times when handling requests such as message publishing and topic subscription. The PA3 code likely handled these operations synchronously, which resulted in increased latency when multiple peers were involved.
- Scenario: When multiple peers publish messages or subscribe to topics, our system ensures that the messages are sent or received without waiting for each previous operation to complete. This reduces response time and improves overall throughput, whereas the PA3 code's synchronous nature meant that each peer had to wait for the previous peer's request to complete, leading to higher latency.

3. Efficient Replication Handling

- Improvement: Our system's replica creation and update processes are optimized to minimize the time spent synchronizing replicas across peer nodes. We leverage optimized data structures and communication protocols that ensure replicas are created and updated in a fraction of the time compared to the PA3 system.
- Scenario: When a new replica is created or updated, the system quickly synchronizes data across nodes by utilizing background tasks that don't block the main thread. In contrast, PA3 likely used blocking operations that caused delays when synchronizing data, especially under high load.

4. Load Distribution Across Multiple Cores

- Improvement: The new system takes full advantage of the multi-core architecture, distributing the workload across all 8 cores of the machine. This parallelization ensures that requests are handled in a highly efficient manner.
- Scenario: For example, when multiple peers are interacting with the system, the workload is divided across the cores, ensuring that no single core is overwhelmed. In contrast, PA3 may have had a single-threaded execution model or poorly optimized load balancing, leading to slower performance during high demand.

5. Optimized Throughput Calculation

- Improvement: Our system includes advanced techniques for throughput optimization, including batch processing and pre-processing of messages before they are sent to replicas or stored. This reduces overhead and increases the overall throughput of the system.
- Scenario: When a large number of messages are pushed to the system, our batch processing approach ensures that the messages are grouped and processed in chunks, minimizing unnecessary overhead. PA3 likely processed messages individually, which slowed down the system when dealing with high volumes of data.

6. Efficient Event Logging

- Improvement: The event logging mechanism in our system is non-blocking, using lightweight data structures that allow logs to be captured without significantly affecting the performance of the main application. In PA3, event logging may have been a blocking operation that introduced latency during critical operations.
- Scenario: For every message publication or topic update, our system can quickly log events without waiting for external processes to complete. In contrast, PA3 might have experienced delays in logging, which could slow down operations like topic updates or peer interactions.

7. Faster Node Failure Detection

- Improvement: The new system detects node failures and automatically adjusts the peer topology with minimal latency. This ensures that the system remains highly available even when some nodes go offline.
- Scenario: In the event of a node failure, our system quickly identifies the issue and reroutes traffic to healthy nodes, maintaining performance. PA3, with its more traditional failure detection mechanism, might have required more time to recognize node failures and adjust the topology, leading to service disruptions.

8. Optimized Data Structures

- Improvement: Our system uses more efficient data structures, such as hash maps and concurrent queues, which provide faster lookups and data storage operations. PA3 may have used less efficient structures that increased the time for operations like topic lookups or message retrieval.
- Scenario: When querying or updating topics, our system can quickly access and modify the data using optimized data structures, reducing the time spent on these operations. PA3, on the other hand, may have had slower access times due to inefficient data structures, leading to slower performance during high-load conditions.

9. Better Throughput During High Load

- Improvement: Under heavy load, our system can maintain high throughput by leveraging efficient queuing mechanisms and load balancing strategies. By distributing tasks evenly across nodes and threads, we ensure that no single component becomes a bottleneck.
- Scenario: During high load, such as when multiple peers are pushing messages or subscribing to topics at once, the system scales better and maintains higher throughput. PA3 may have struggled to handle such loads efficiently, as its synchronous architecture likely caused significant bottlenecks when processing a large number of requests.

In summary, our system is faster than PA3 in scenarios where concurrent request handling, asynchronous operations, optimized data processing, and multi-core utilization are key. This translates into lower latency, higher throughput, and better scalability, particularly under high load. The PA3 system, being synchronous and less optimized for multi-core environments, experienced performance degradation in such scenarios, making it less efficient compared to our newly implemented system.

Describe what you are curious about consistency model in distributed system, conduct experiments to solve/verify your questions on your own.

In the context of distributed systems, consistency models are critical because they define how updates to a system's data are propagated and how the system guarantees that all nodes have a consistent view of the data at any given time. Consistency models play a significant role in the trade-offs between availability, partition tolerance, and latency (as described by the CAP theorem). Below are some aspects of consistency models in distributed systems that I find particularly interesting and would like to explore further through experiments:

1. Strong Consistency vs. Eventual Consistency

- Question: What are the trade-offs between strong consistency and eventual consistency in real-world scenarios? How do they affect system performance (latency, throughput) and correctness of results, especially under failure conditions?
- Experiment: I would set up two systems: one using strong consistency (e.g., using a consensus protocol like Paxos or Raft) and another using eventual consistency (e.g., by employing an eventual consistency model like Dynamo or Cassandra).
    - I would introduce various types of failures (e.g., network partition, node crash) and measure how each system recovers and how long it takes to return to a consistent state.

- I would also measure the response times and throughput under different load conditions to assess the trade-offs between consistency and system performance.

2. Consistency Levels in NoSQL Databases

- Question: How do different consistency levels in NoSQL databases (e.g., read-your-writes, session consistency, monotonic consistency) impact user experience in distributed systems?
- Experiment: I would use a NoSQL database (such as Cassandra or MongoDB) and configure it to use different consistency levels. The experiment would involve:
  - Writing data to one node and reading it from another node with varying consistency levels.
  - Measuring the response time for reads and writes, and evaluating how the consistency level affects performance and data accuracy.
  - Simulating network partitions and checking how different consistency levels affect the behavior of the system during recovery.

3. Consistency vs. Availability

- Question: How does a system's choice of consistency model affect its availability? Specifically, does a system prioritizing strong consistency become less available during partitions or failures?
- Experiment: I would simulate network partitions and failures while running a distributed system with different consistency models. I would measure:
  - The availability of the system (i.e., the ability to accept requests).
  - The consistency of the responses (i.e., whether clients get up-to-date or stale data after a partition).
  - I would experiment with different consistency models (strong consistency, eventual consistency, causal consistency) to see how each handles failures, and whether strong consistency sacrifices availability or introduces latency during recovery.

4. Causal Consistency and Its Trade-offs

- Question: How does causal consistency compare with other consistency models like strong consistency and eventual consistency in terms of performance and data integrity?
- Experiment: I would build a system that implements causal consistency using vector clocks or version vectors to track causality between operations. The experiment would involve:

- Running concurrent operations across different nodes.
- Measuring the performance (latency and throughput) of reading and writing data under causal consistency, while comparing it to eventual and strong consistency models.
- Analyzing how causal consistency ensures