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

INDEX

Project Overview:

This project implements a decentralized peer-to-peer (P2P) messaging system that allows multiple peers to communicate efficiently through topics. Each peer can create topics, subscribe to existing topics, publish messages to those topics, and manage the topics dynamically. The system is designed to be fault-tolerant and scalable, utilizing a Distributed Hash Table (DHT) for efficient data distribution and retrieval.

Tools Used:

Maven, Gradle, XYChart, SpringBoot, JAVA

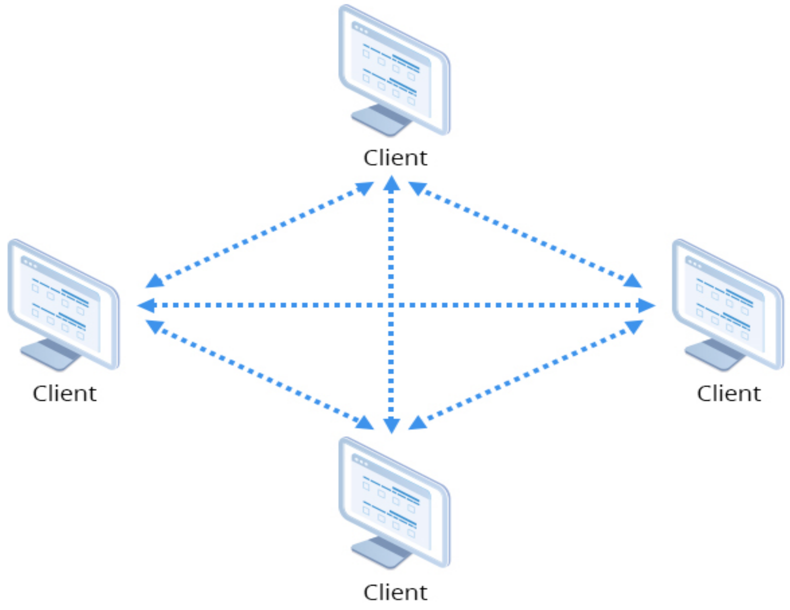
What exactly is a Peer-to-Peer System?

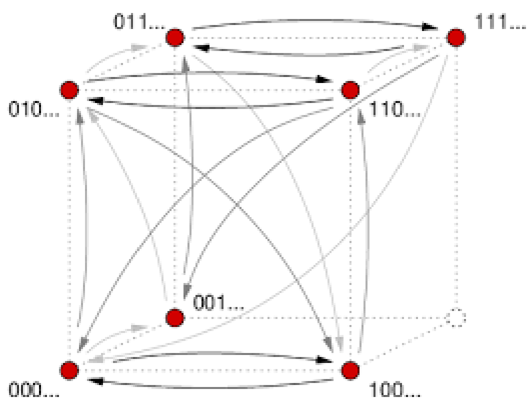
A Peer-to-Peer (P2P) system is a decentralized network architecture in which participants, known as peers, interact and share resources directly with each other without relying on a central server or authority. Each peer in the network can act as both a client and a server, meaning they can request services from other peers and also provide services to them.

Key Characteristics of P2P Systems:

1. *Decentralization:*
   - Unlike traditional client-server models, there is no central server managing the system. All peers are equally privileged and can communicate directly with each other.
2. *Distributed Resources:*
   - Resources (such as files, data, or computing power) are distributed among all peers. This means no single point of control or failure, making the system more resilient.
3. *Scalability:*
   - Since the network relies on peers to contribute resources, it can scale easily as more peers join the system.
4. *Fault Tolerance:*
   - In the event that one or more peers fail, the system can continue to operate, as resources are distributed across multiple peers.

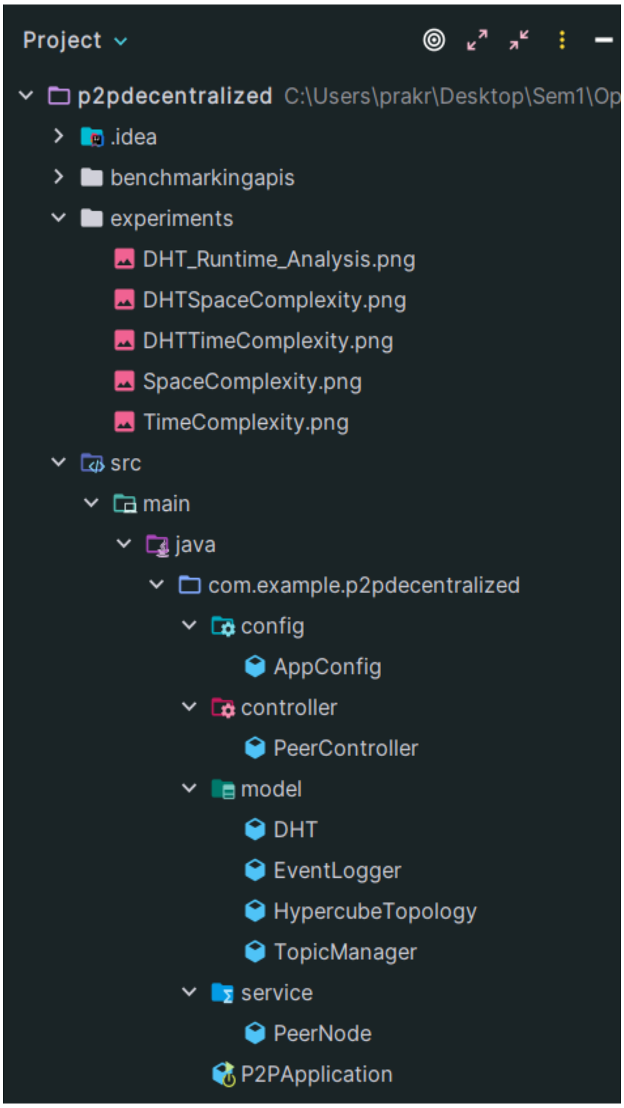
DIAGRAM

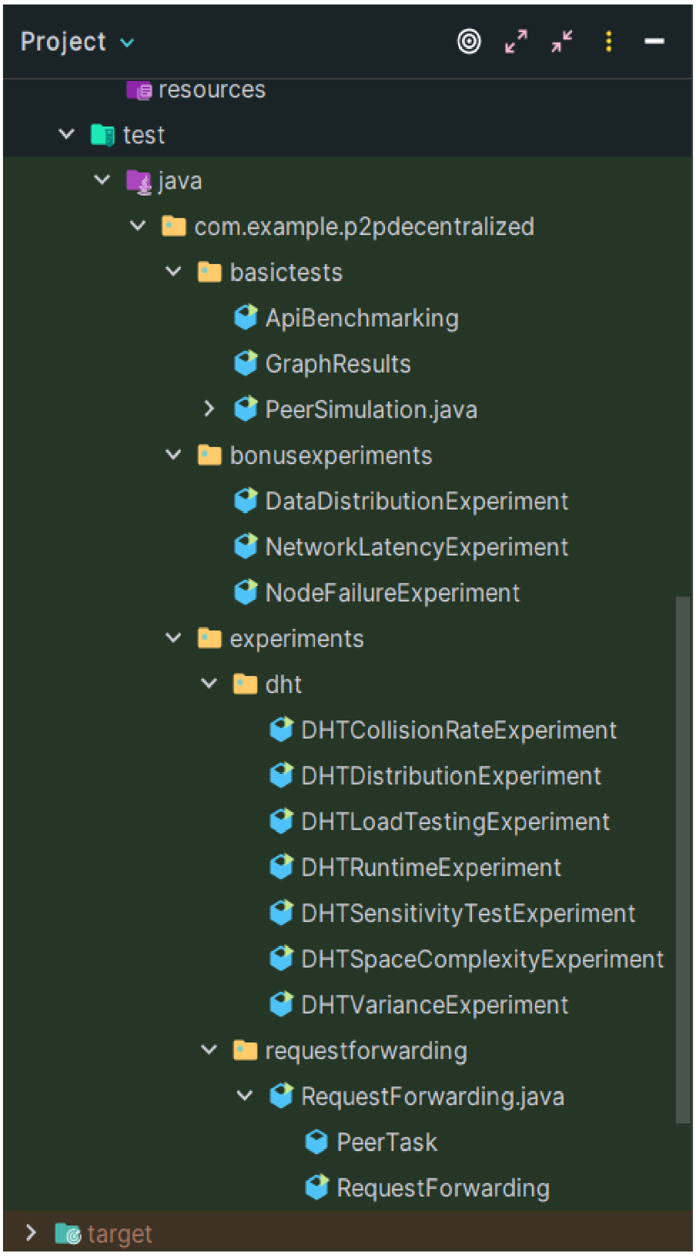




Project Structure

The project is organized into a clear structure, separating concerns into different packages for better maintainability. Below is an overview of the project structure along with descriptions of the key components:


Project
p2pdecentralized C:\Users\prakr\Desktop\Sem1\Op
.idea
benchmarkingapis
experiments
DHT_Runtime_Analysis.png
DHTSpaceComplexity.png
DHTTimeComplexity.png
SpaceComplexity.png
TimeComplexity.png
src
main
java
com.example.p2pdecentralized
config
AppConfig
controller
PeerController
model
DHT
EventLogger
HypercubeTopology
TopicManager
service
PeerNode
P2PApplication


Project
resources
test
java
com.example.p2pdecentralized
basictests
ApiBenchmarking
GraphResults
PeerSimulation.java
bonusexperiments
DataDistributionExperiment
NetworkLatencyExperiment
NodeFailureExperiment
experiments
dht
DHTCollisionRateExperiment
DHTDistributionExperiment
DHTLoadTestingExperiment
DHTRuntimeExperiment
DHTSensitivityTestExperiment
DHTSpaceComplexityExperiment
DHTVarianceExperiment
requestforwarding
RequestForwarding.java
PeerTask
RequestForwarding
target

Main Application Code

(src/main/java/com/example/p2pdecentralized)

- *config*
  - AppConfig.java: This class is responsible for initializing the application context, including the configuration of peer nodes, network parameters, and other essential settings that dictate how the application operates.
- *controller*
  - PeerController.java: This REST controller handles incoming HTTP requests related to peer operations. It provides endpoints for creating topics, subscribing to them, publishing messages, and pulling messages. The controller serves as the primary interface for users to interact with the P2P system.
- *model*
  - DHT.java: This class implements the Distributed Hash Table, which is crucial for mapping topics to peer nodes. It handles the hashing of topic names to distribute them evenly across the available peers and ensures efficient retrieval of messages.
  - EventLogger.java: Responsible for logging significant events and actions within the P2P network, such as when topics are created, messages are published, or nodes join and leave the network. This logging mechanism is vital for debugging and monitoring system behavior.
  - HypercubeTopology.java: Defines the hypercube topology used for connecting nodes in the P2P network. This topology enhances the efficiency of peer communication by allowing for a logarithmic number of hops between nodes.
  - TopicManager.java: Manages the lifecycle of topics in the system, including creation, deletion, subscription, and message publication. This class ensures that topic-related operations are handled appropriately and efficiently.
- *service*
  - PeerNode.java: Represents an individual peer in the network. Each peer maintains its own state, including its subscribed topics and published messages. This class includes methods for subscribing to topics, publishing messages, and interacting with the DHT.

  - P2PApplication.java: The main entry point of the application. This class initializes the application, sets up the server, and starts the peer-to-peer network. It ensures that the necessary components are wired together and ready for operation.

Test Code

(src/test/java/com/example/p2pdecentralized)

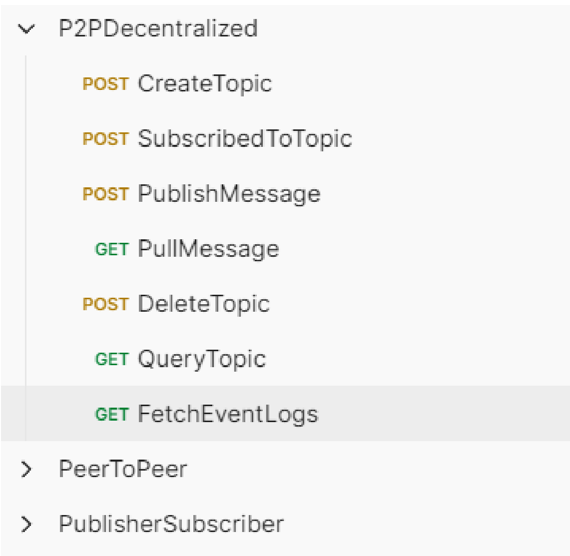
- *basictests*
  - ApiBenchmarking.java: Contains tests for measuring the performance of various API endpoints, focusing on their latency and throughput. It helps ensure that the system can handle expected loads efficiently.
  - GraphResults.java: This class generates graphical representations of benchmarking results, allowing for easy visualization of API performance metrics using libraries like XChart.
  - PeerSimulation.java: Simulates peer interactions and tests how peers communicate with each other under various conditions, such as high load or network latency.
- *bonusexperiments*
  - DataDistributionExperiment.java: Tests the distribution of data across peers to analyze load balancing and data redundancy.
  - NetworkLatencyExperiment.java: Measures network latency between peers, assessing the time it takes for messages to be sent and received.
  - NodeFailureExperiment.java: Tests how the network responds to node failures, including automatic recovery mechanisms and data integrity.
- *experiments*
  - dht
    - DHTCollisionRateExperiment.java: Measures the collision rates in the DHT when multiple topics are hashed, analyzing the effectiveness of the hash function.
    - DHTDistributionExperiment.java: Tests the distribution of topics across the DHT, ensuring that they are evenly spread among the nodes.
    - DHTLoadTestingExperiment.java: Evaluates the performance of the DHT under high load conditions, assessing response times and resource utilization.

  - DHTRuntimeExperiment.java: Measures the runtime efficiency of DHT operations such as insertions, deletions, and lookups.
  - DHTSensitivityTestExperiment.java: Analyzes the sensitivity of DHT performance to various parameters such as topic names and node counts.
  - DHTSpaceComplexityExperiment.java: Assesses the space complexity of the DHT, determining how efficiently it uses memory for storing topics and messages.
  - DHTVarianceExperiment.java: Tests variance in data distribution and access times in the DHT, providing insights into the consistency of performance.

- *requestforwarding*
  - RequestForwarding.java: Implements the logic for forwarding requests across peers in the network. This class ensures that messages can be routed efficiently to their intended destinations.
  - PeerTask.java: A helper class that supports the request forwarding mechanism, managing the execution of tasks related to message delivery and peer communication.

API Endpoints

The system exposes several RESTful API endpoints that allow peers to interact with the messaging system. Below is a detailed description of each API along with example cURL commands for testing.



1. Create Topic

- Description: This API creates a new topic in the system and assigns it to a peer node. Each topic can have multiple subscribers.
- HTTP Method: POST
- Endpoint: /api/topic
- Request Body: JSON object containing the topic name.
- Example cURL Command:

```
curl -X POST "http://localhost:8080/api/topic" -H "Content-Type: application/json" -d
'{"topic": "YourTopicName"}'
```

2. Subscribe to Topic

- Description: Allows a peer to subscribe to an existing topic, enabling it to receive messages published to that topic.
- HTTP Method: POST
- Endpoint: /api/subscribe
- Request Body: JSON object containing the topic name.
- Example cURL Command:

```
curl -X POST "http://localhost:8080/api/subscribe" -H "Content-Type: application/json" -d
'{"topic": "YourTopicName"}'
```

3. Publish Message

- Description: Publishes a message to a specified topic. All subscribers to that topic will receive the message.
- HTTP Method: POST
- Endpoint: /api/publish
- Request Body: JSON object containing the topic name and message.
- Example cURL Command:

```
curl -X POST "http://localhost:8080/api/publish" -H "Content-Type: application/json" -d
'{"topic": "YourTopicName", "message": "YourMessage"}'
```

4. Pull Messages

- Description: Retrieves messages published on a specified topic. Only the messages that the requesting peer is authorized to access will be returned.
- HTTP Method: GET
- Endpoint: /api/pull
- Query Parameters: topic - the name of the topic to pull messages from.
- Example cURL Command:

```
curl -X GET http://localhost:8080/api/pull?topic=YourTopicName
```

5. Query Topic

- Description: Fetches information about a specified topic, including its location in the DHT and associated metadata.
- HTTP Method: GET
- Endpoint: /api/query
- Query Parameters: topic - the name of the topic to query.
- Example cURL Command:

```
curl -X GET "http://localhost:8080/api/query?topic=YourTopicName"
```

6. Delete Topic

- Description: Deletes an existing topic from the system. This operation removes the topic and all associated messages from the DHT.
- HTTP Method: DELETE
- Endpoint: /api/delete
- Request Body: JSON object containing the topic name.
- Example cURL Command:

```
curl -X DELETE "http://localhost:8080/api/delete" -H "Content-Type: application/json" -d
'{"topic": "YourTopicName"}'
```

7. Fetch Event Logs

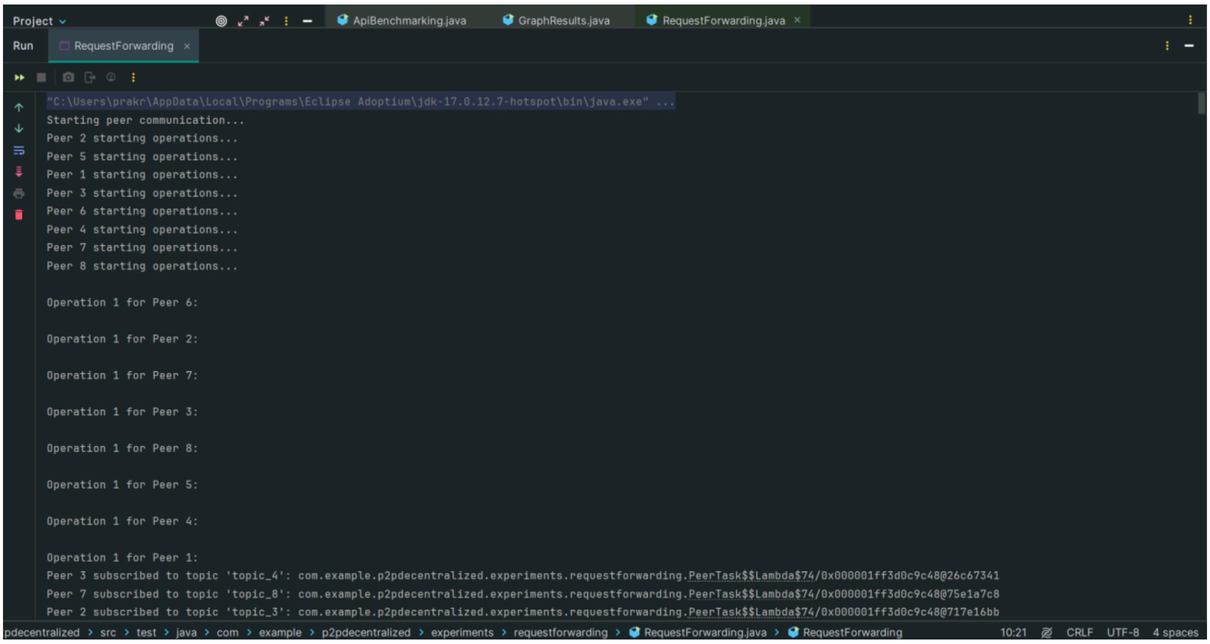
- Description: Retrieves the event logs that detail actions taken on topics and messages within the system. This can help in monitoring and debugging.
- HTTP Method: GET
- Endpoint: /api/logs
- Example cURL Command:

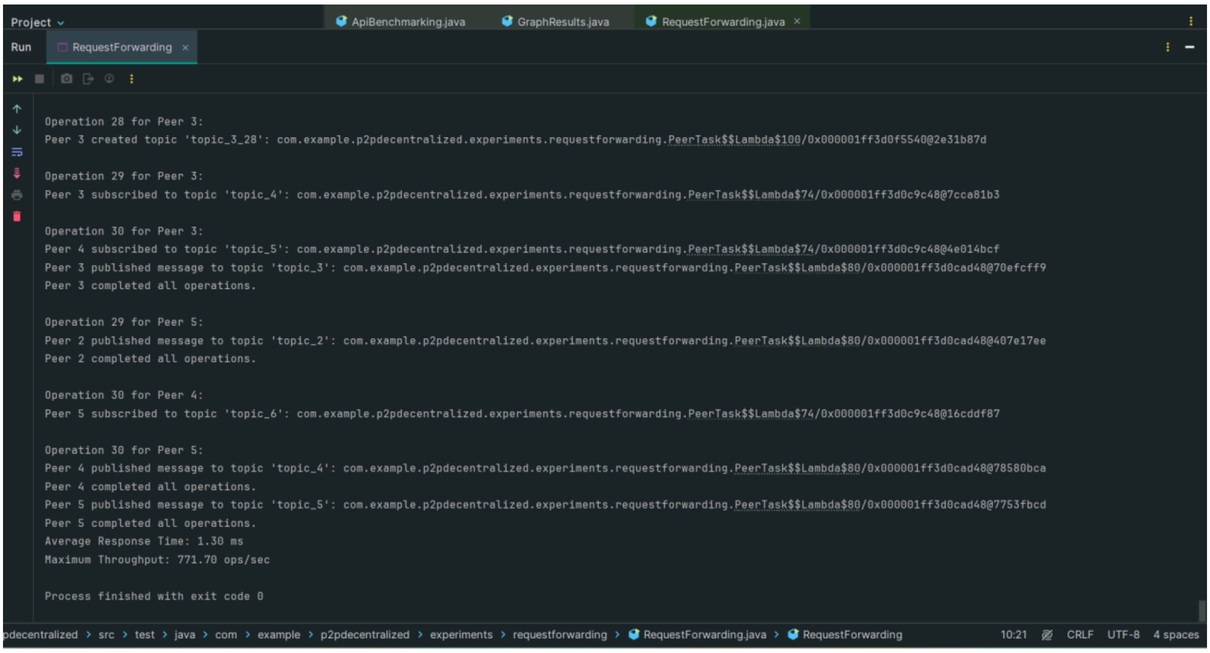
```
curl -X GET http://localhost:8080/api/logs
```

EVALUATION:

1. *Deploying 8 peers. They can be set up on the same machine or different machines.*
    a. *Ensure all APIs are working properly.*
    b. *Ensure multiple peer nodes can simultaneously publish and subscribe to a topic.*

Performing 30 operations amongst 8 peers.
Not pasting the whole output pictures just, the end and the beginning.





2. *Similar to PA2, you need to benchmark the latency and throughput of each API.*
    a. *Deploy 8 peers. Benchmark each API on each node using randomly generated workload.*
    b. *Graph your results*

Benchmarking Procedures

Setup

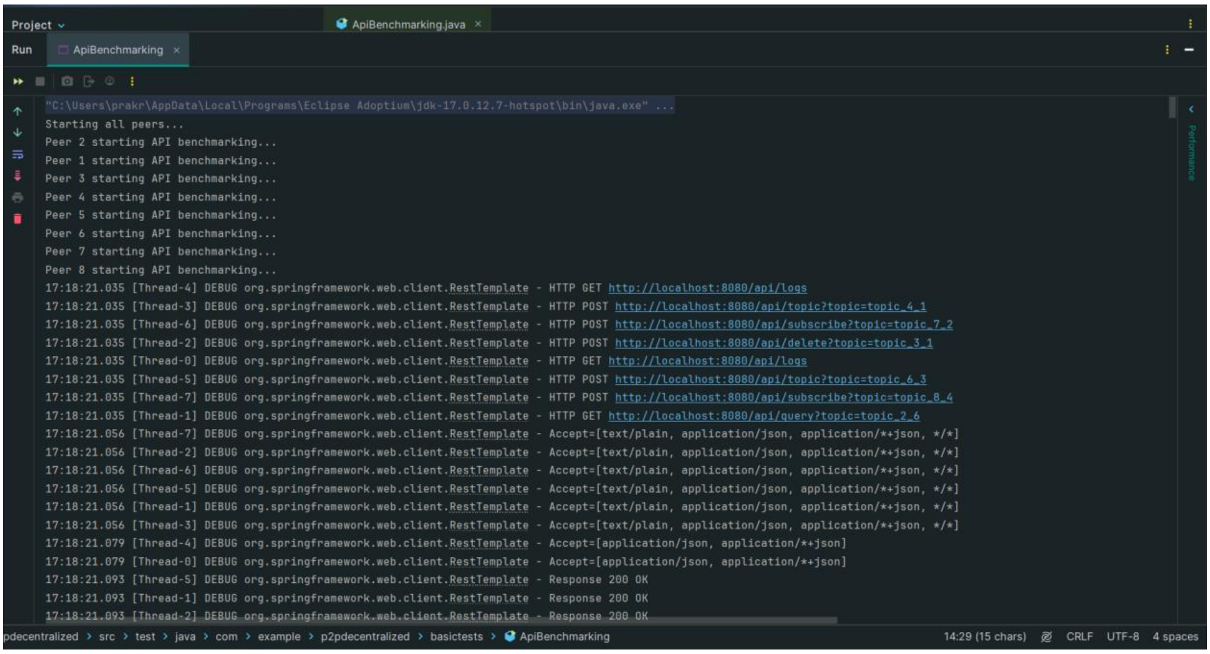
- Environment: Tests were conducted in a local environment with 8 peer nodes running concurrently. Each peer was tasked with simulating various operations over a set of topics.

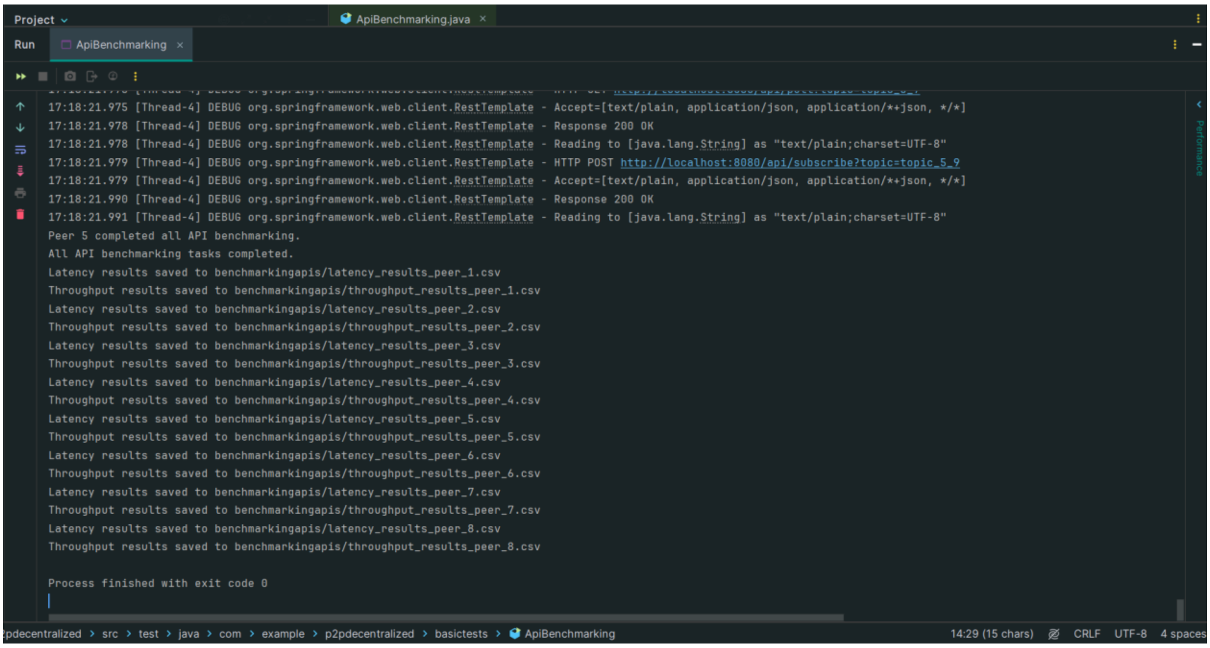
Methodology

1. Simulated Workloads: Randomized topics and messages were generated for testing, ensuring a diverse set of operations.

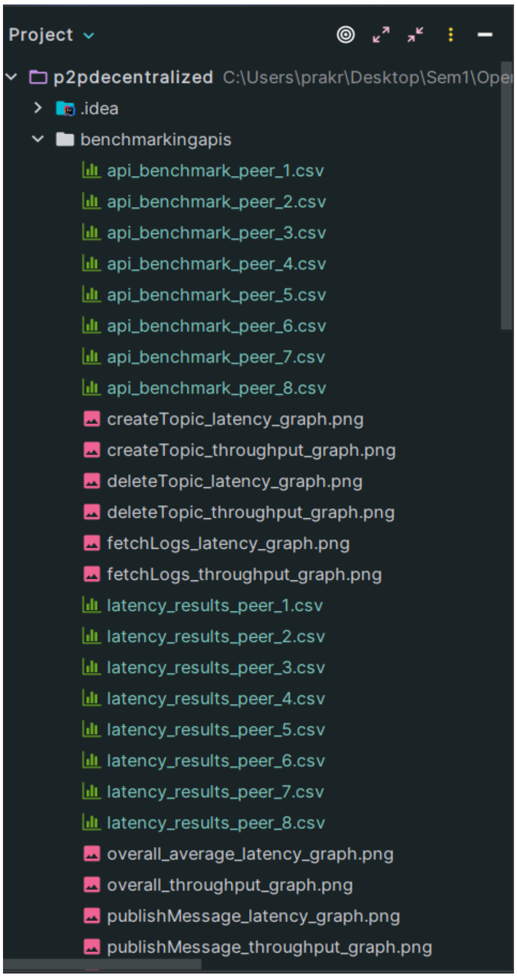
2. Monitoring Tools: Tools such as JVisualVM were used to monitor JVM performance, CPU, and memory usage during benchmarking.

3. Data Collection: Latency and throughput data were collected during each test run, stored in CSV files for analysis.

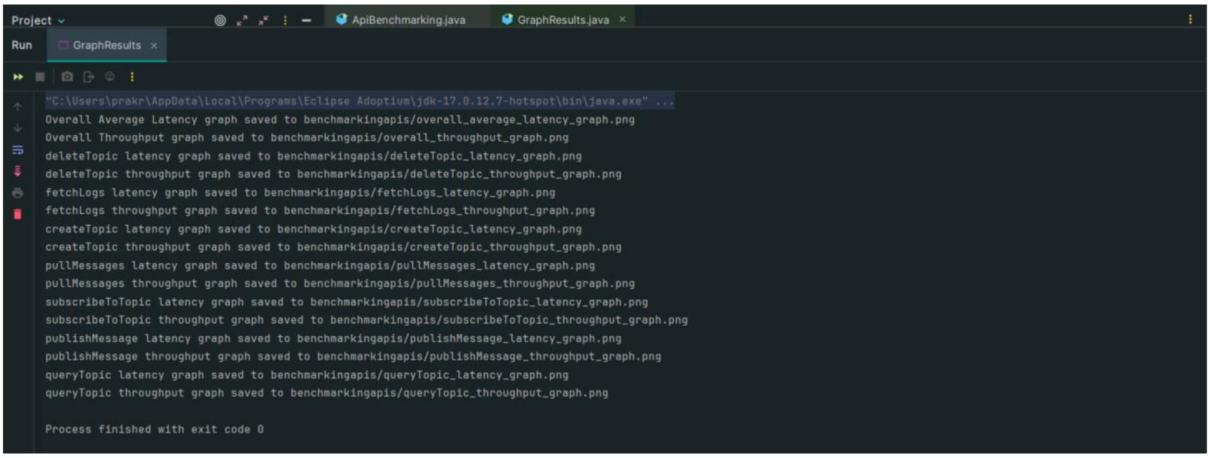




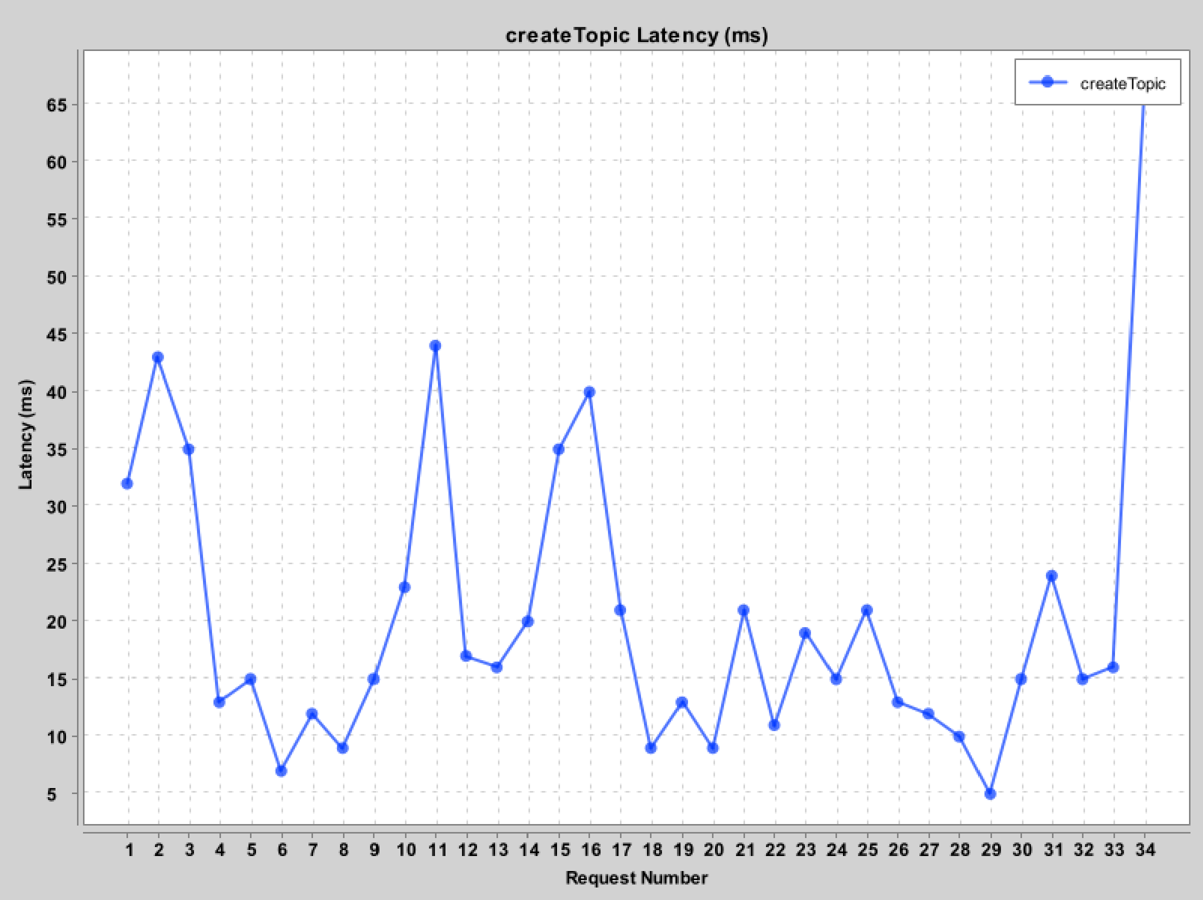
Here the files are saved in the /benchmarking folder.

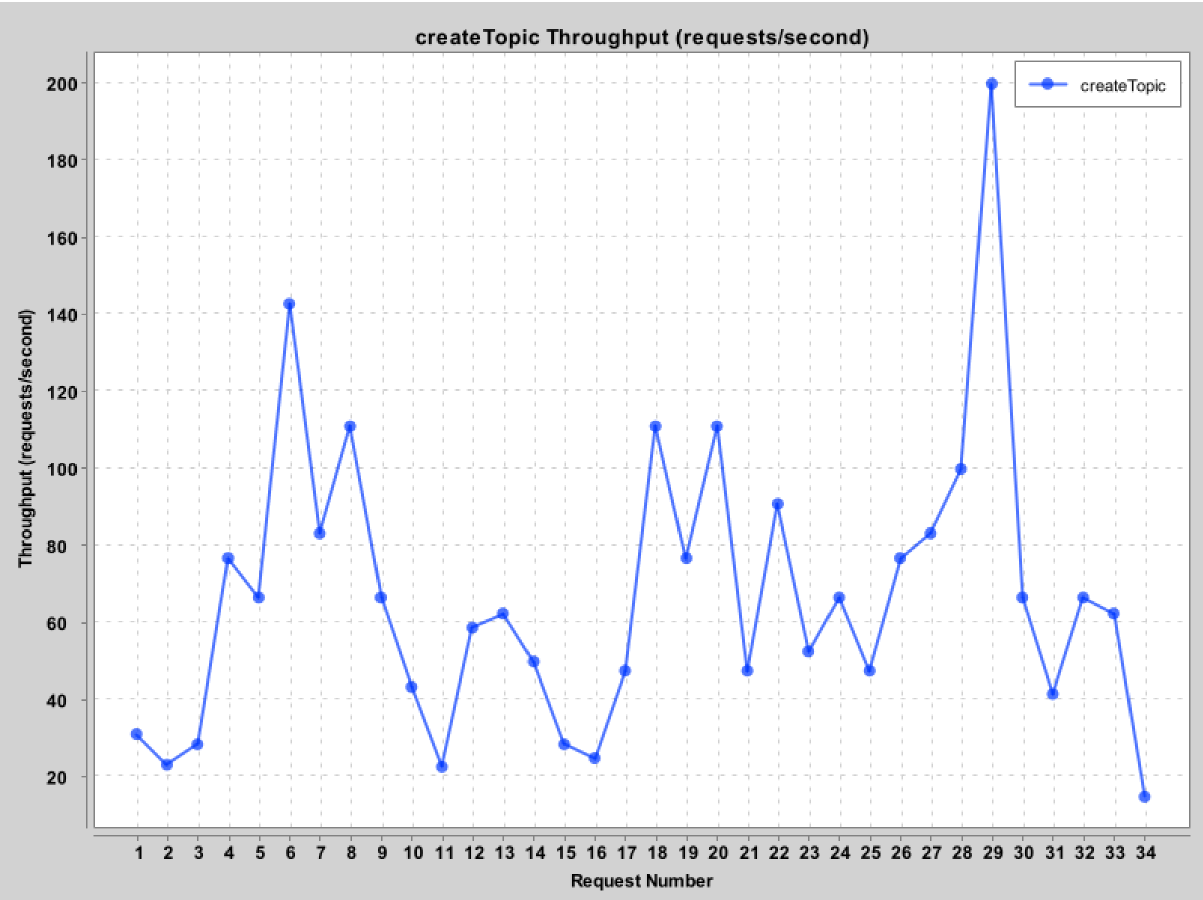


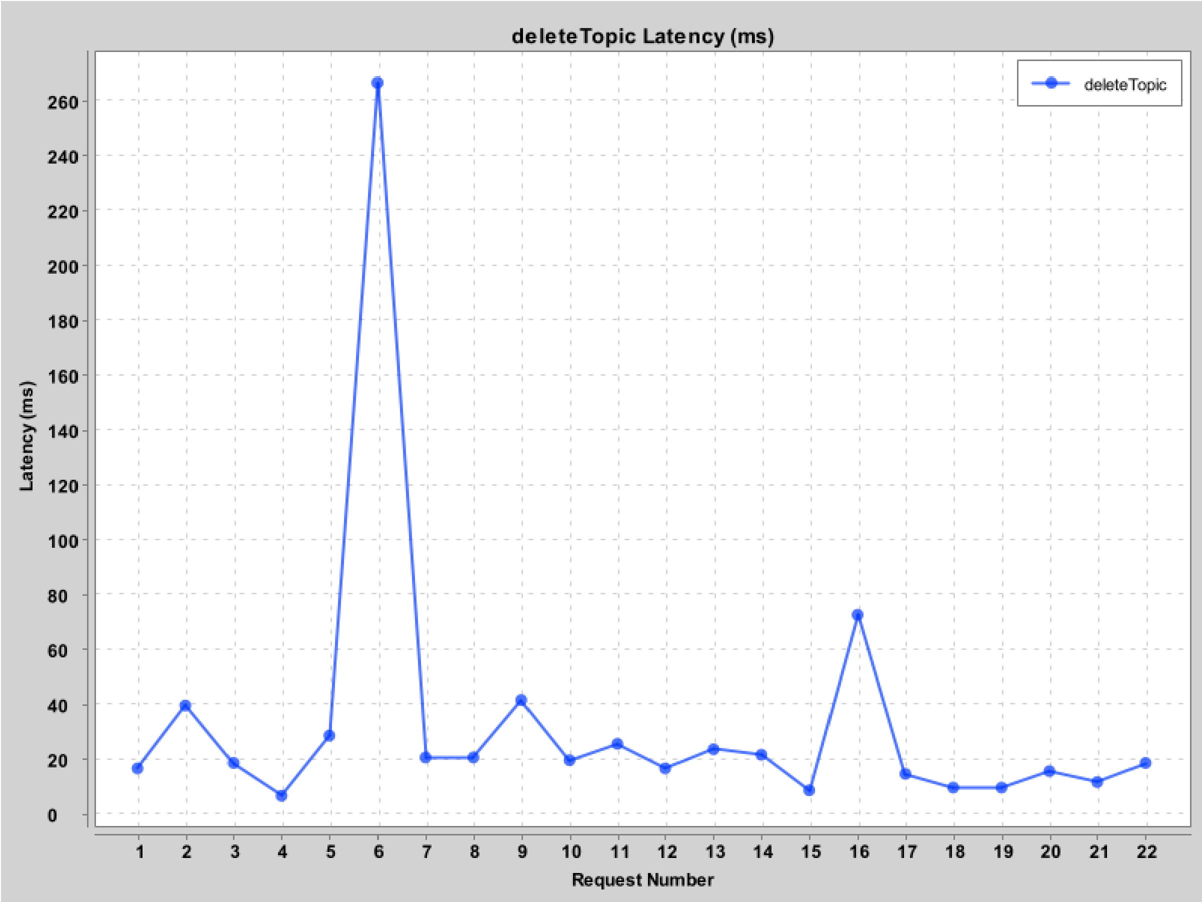
Building these csvs using GraphResults file.

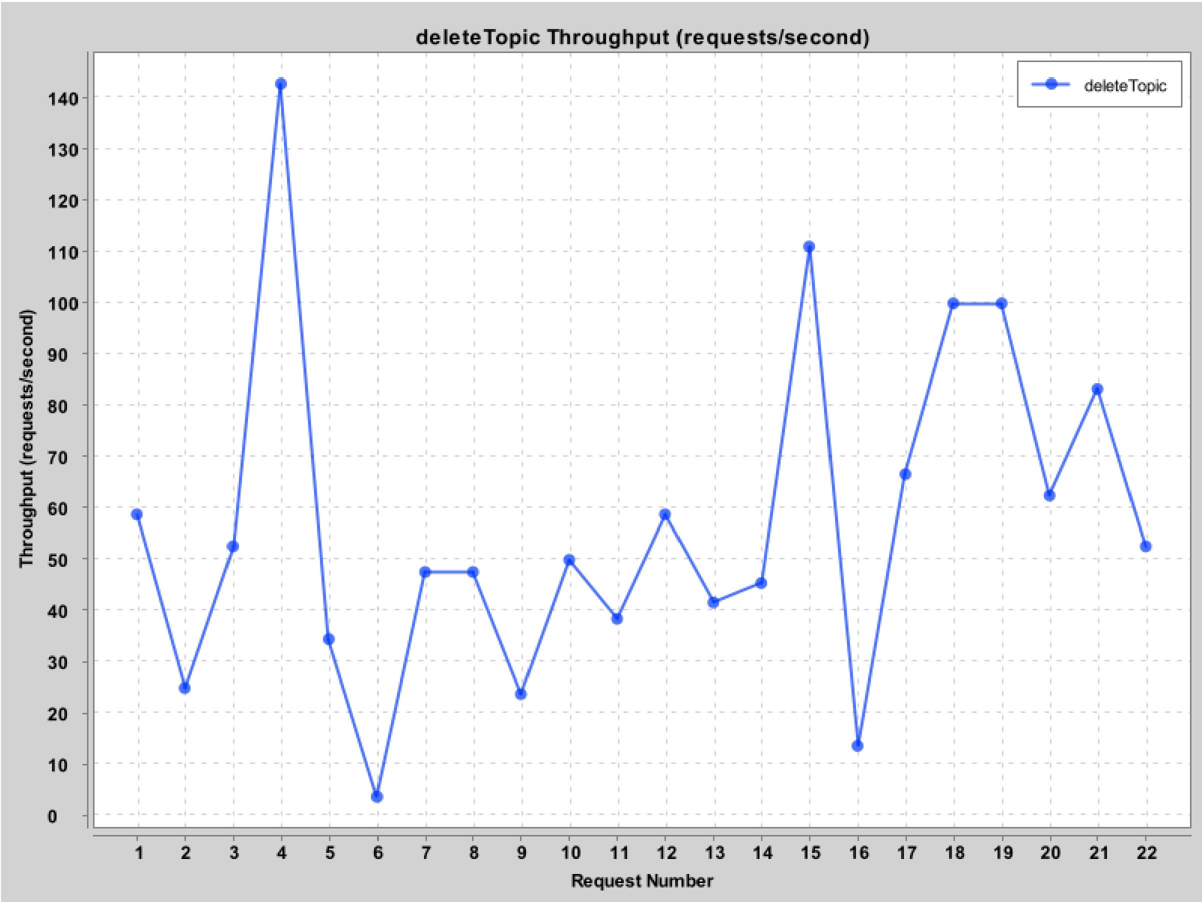


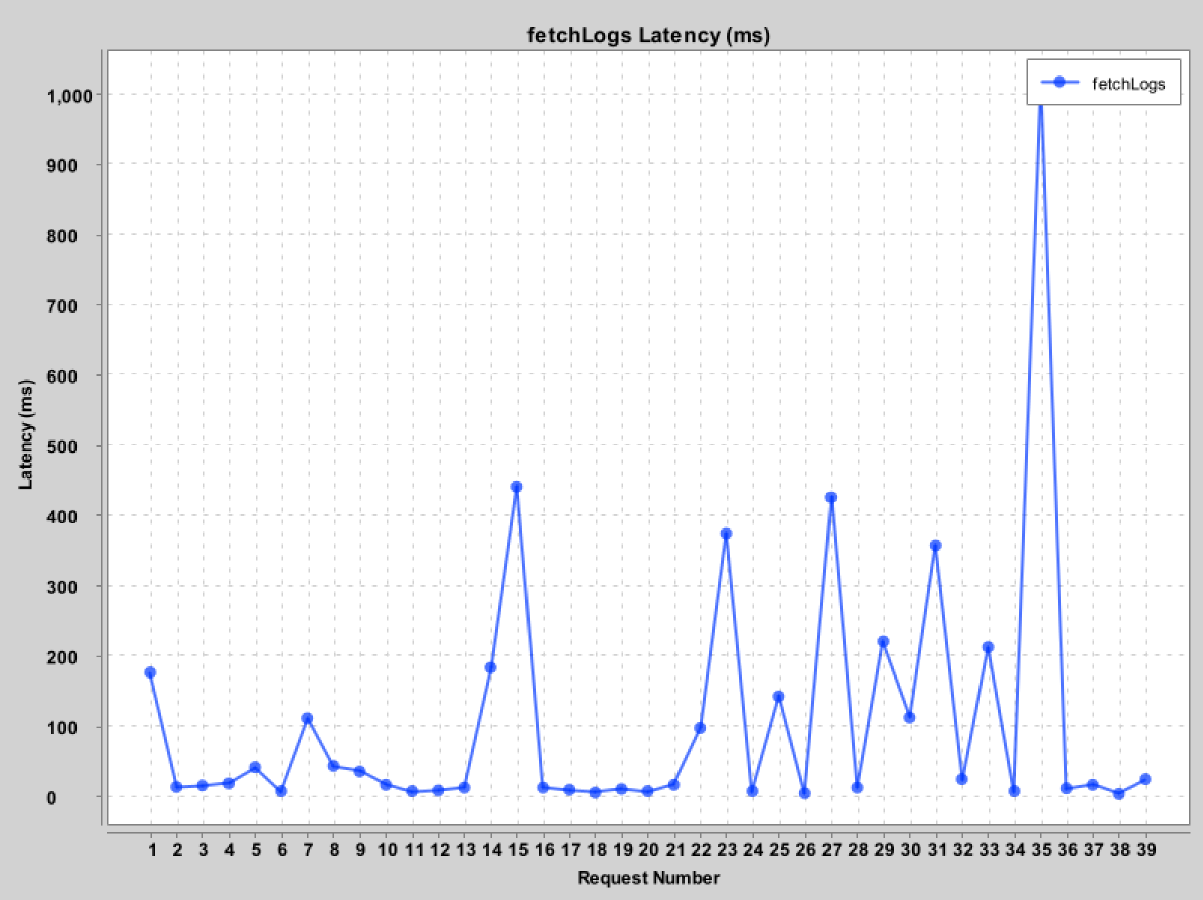
Here are the images of the graphs.

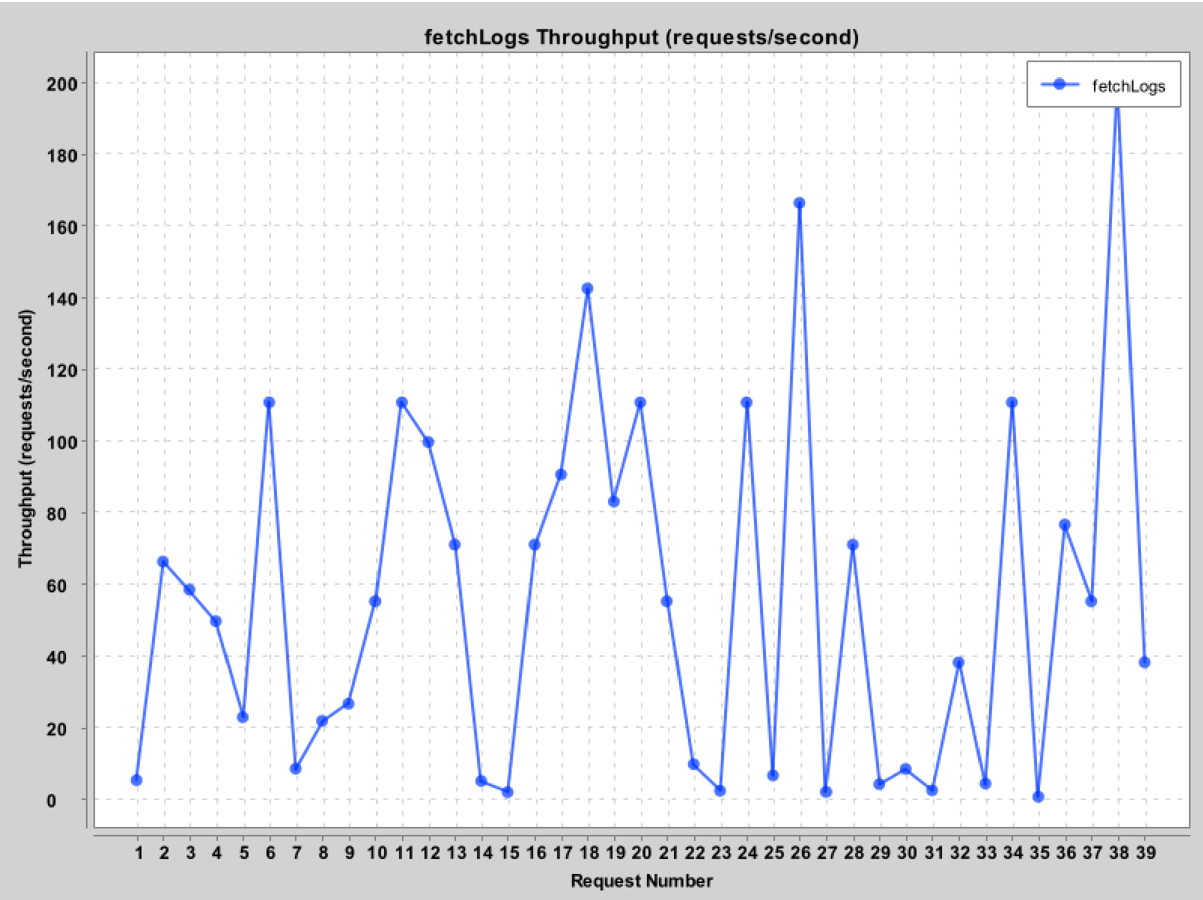

createTopic Latency (ms)
createTopic
Latency (ms)
65
60
55
50
45
40
35
30
25
20
15
10
5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
Request Number

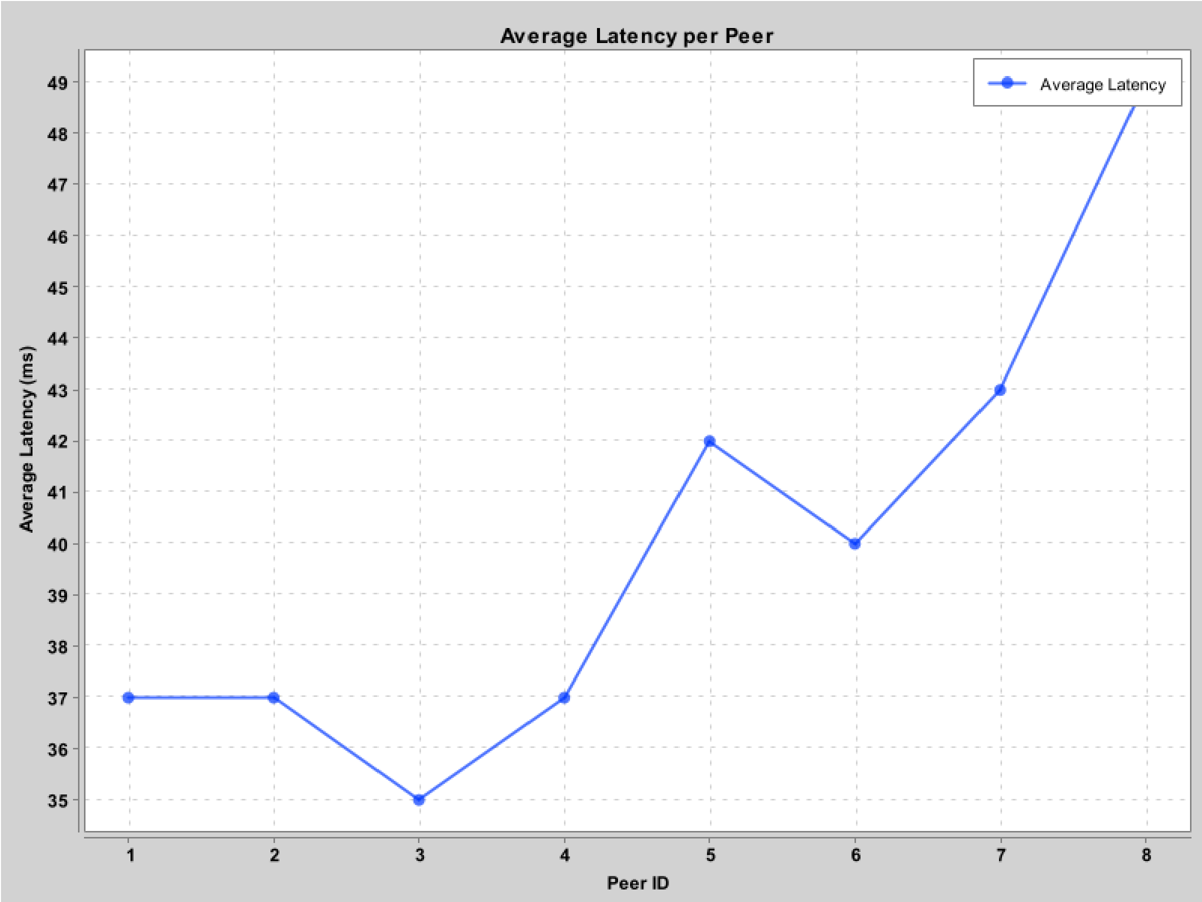

createTopic Throughput (requests/second)
createTopic
Throughput (requests/second)
200
180
160
140
120
100
80
60
40
20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
Request Number

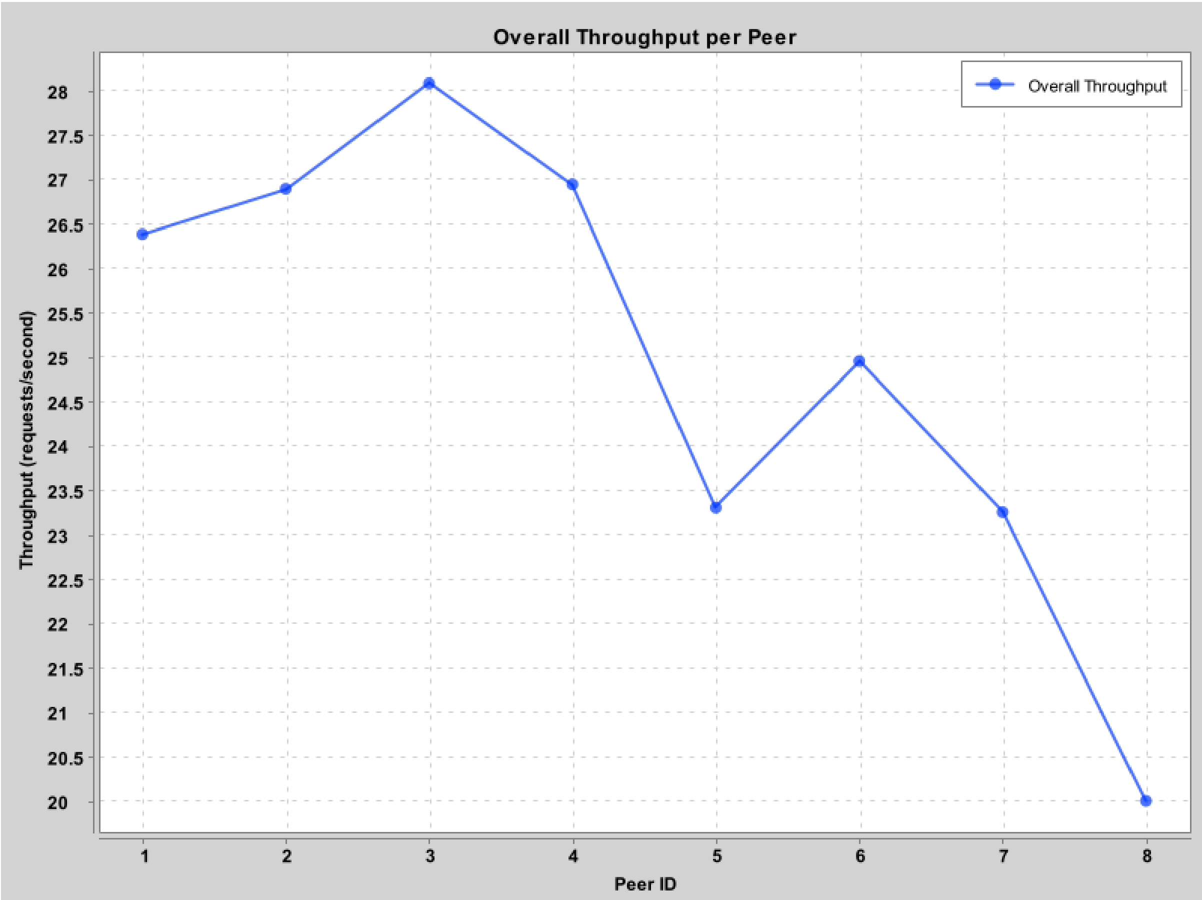

deleteTopic Latency (ms)
deleteTopic
Latency (ms)
0
20
40
60
80
100
120
140
160
180
200
220
240
260
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
Request Number

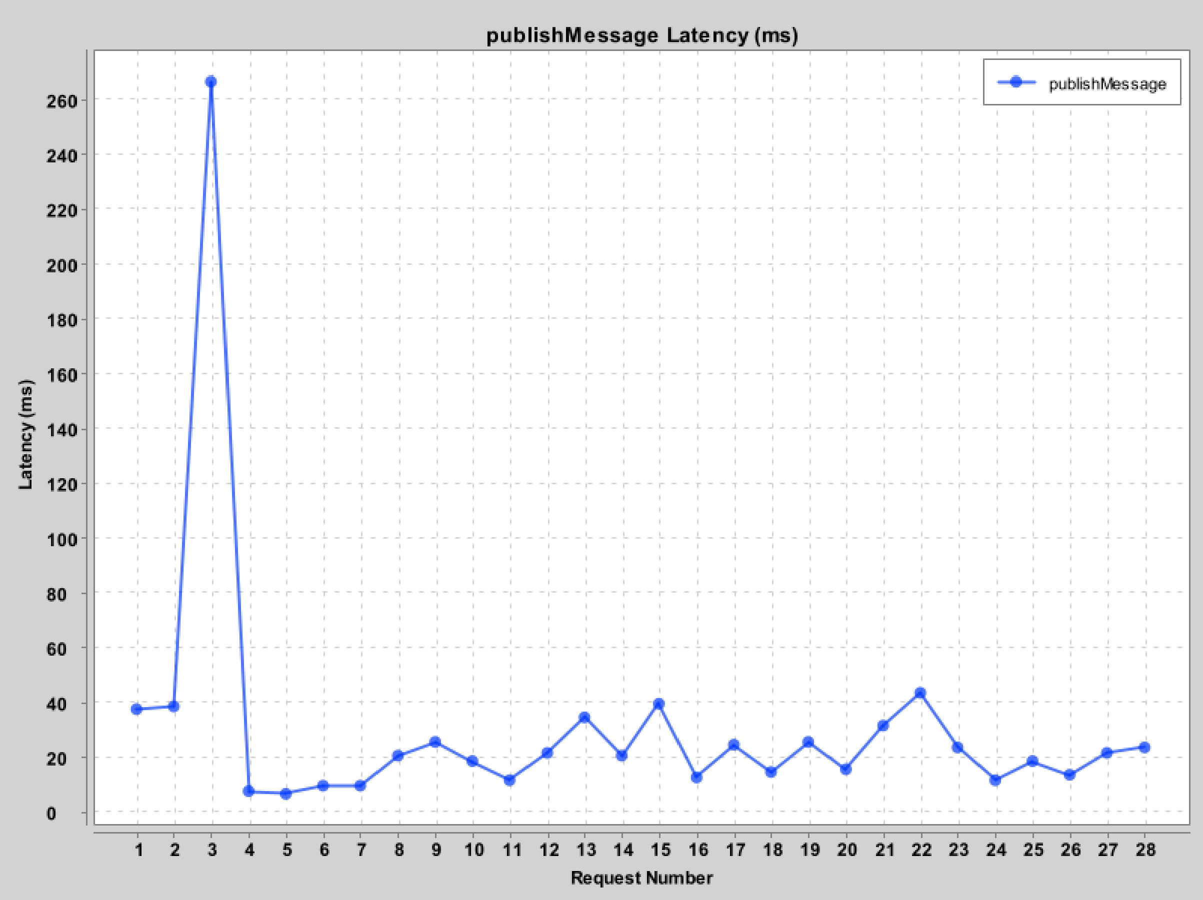

deleteTopic Throughput (requests/second)
deleteTopic
Throughput (requests/second)
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
Request Number

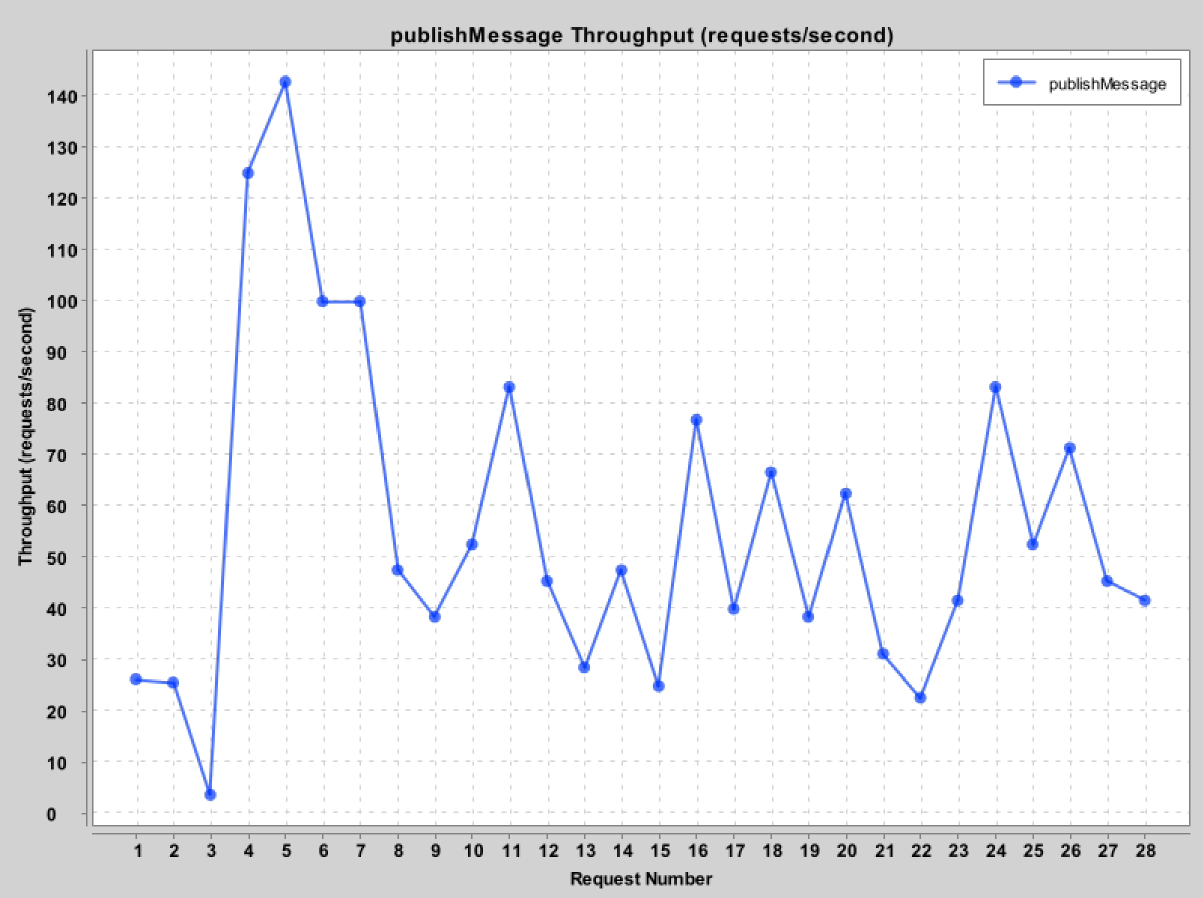

fetchLogs Latency (ms)
fetchLogs
Latency (ms)
1,000
900
800
700
600
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Request Number

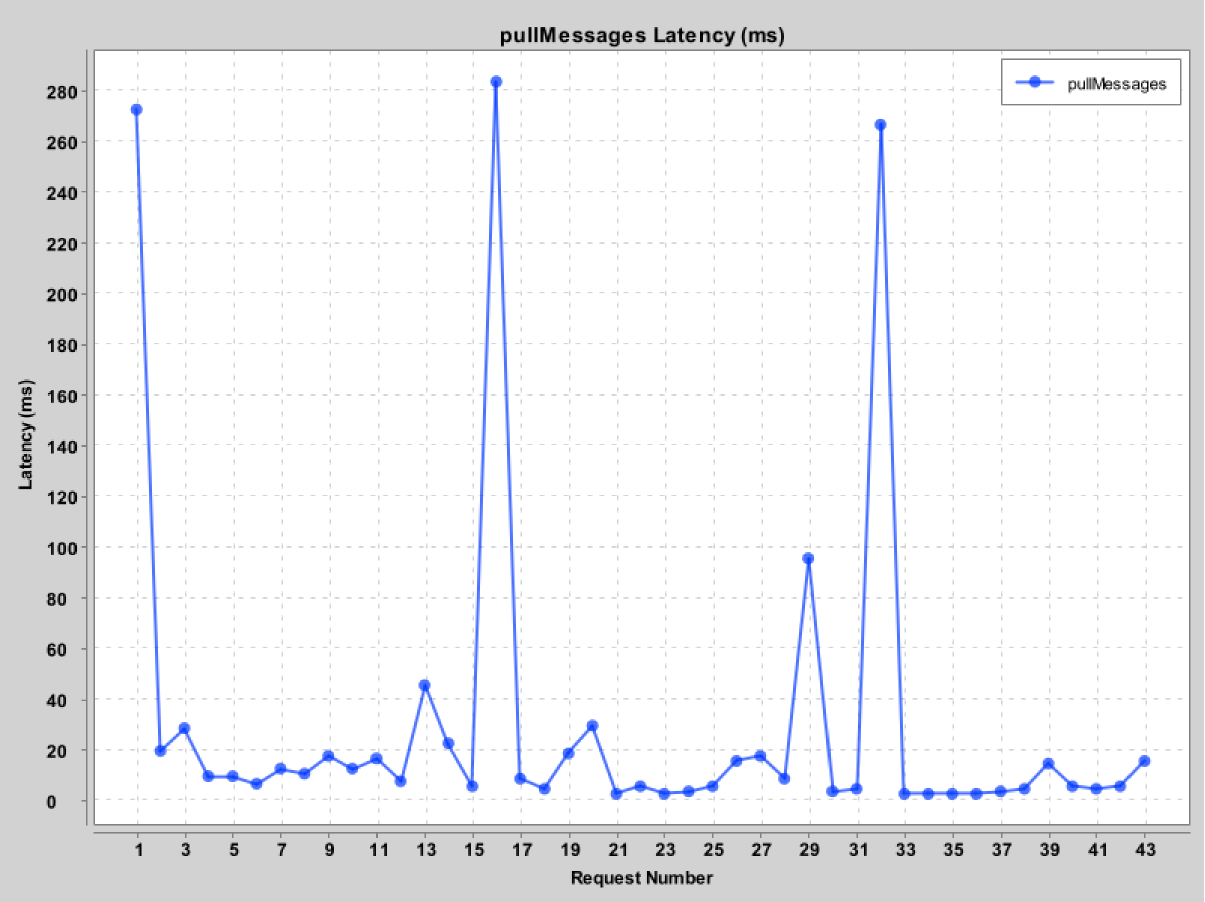

fetchLogs Throughput (requests/second)
fetchLogs
Throughput (requests/second)
200
180
160
140
120
100
80
60
40
20
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Request Number

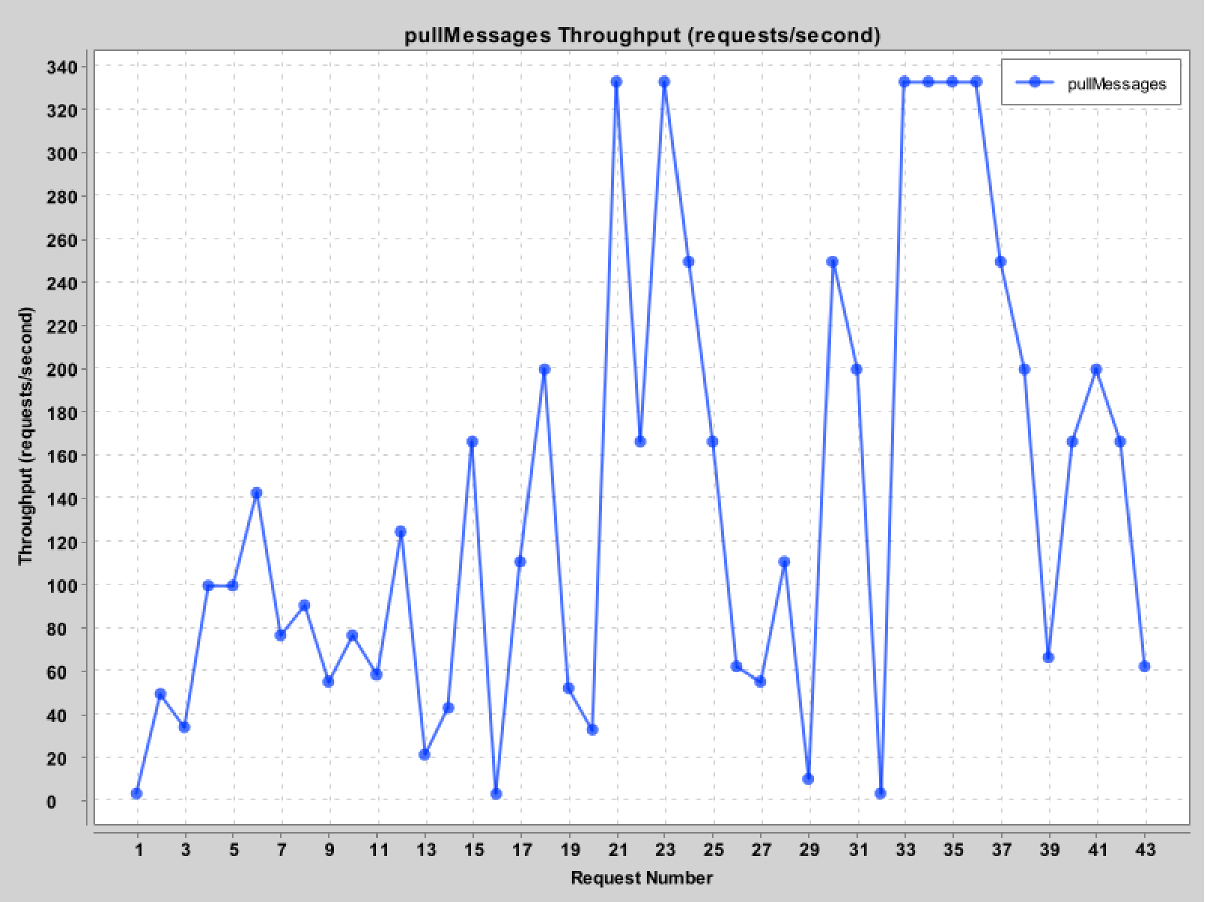

Average Latency per Peer
Average Latency
Average Latency (ms)
49
48
47
46
45
44
43
42
41
40
39
38
37
36
35
1
2
3
4
5
6
7
8
Peer ID

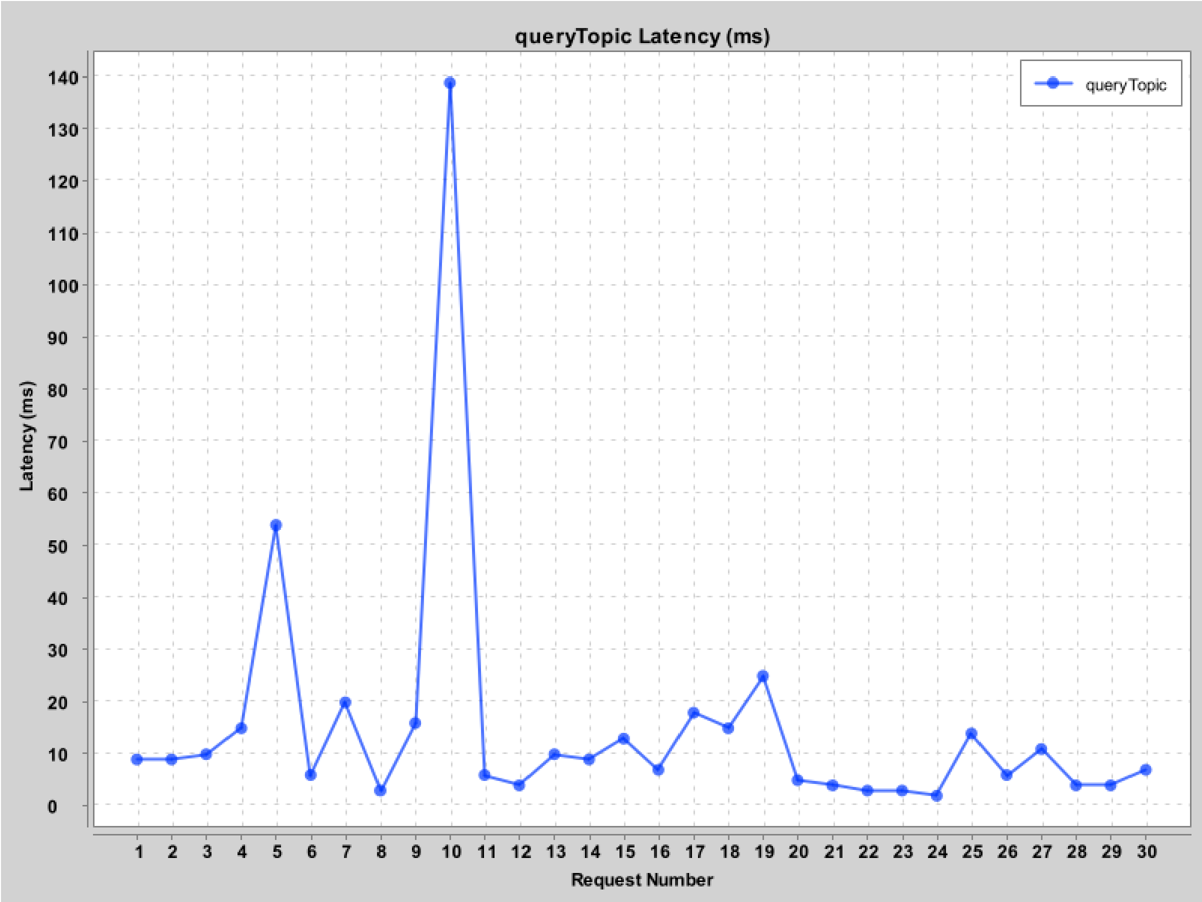

Overall Throughput per Peer
Overall Throughput
Throughput (requests/second)
28
27.5
27
26.5
26
25.5
25
24.5
24
23.5
23
22.5
22
21.5
21
20.5
20
1
2
3
4
5
6
7
8
Peer ID

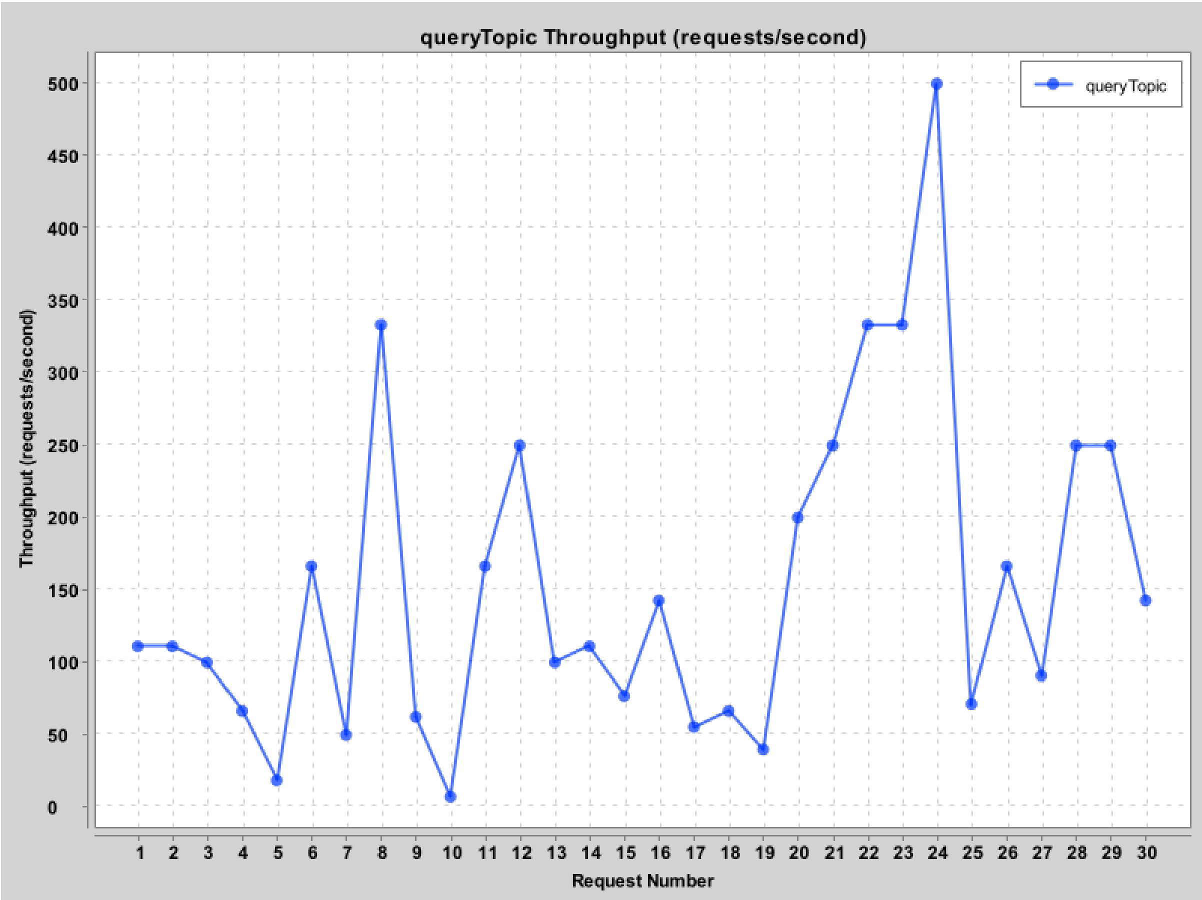

publishMessage Latency (ms)
publishMessage
Latency (ms)
0
20
40
60
80
100
120
140
160
180
200
220
240
260
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Request Number

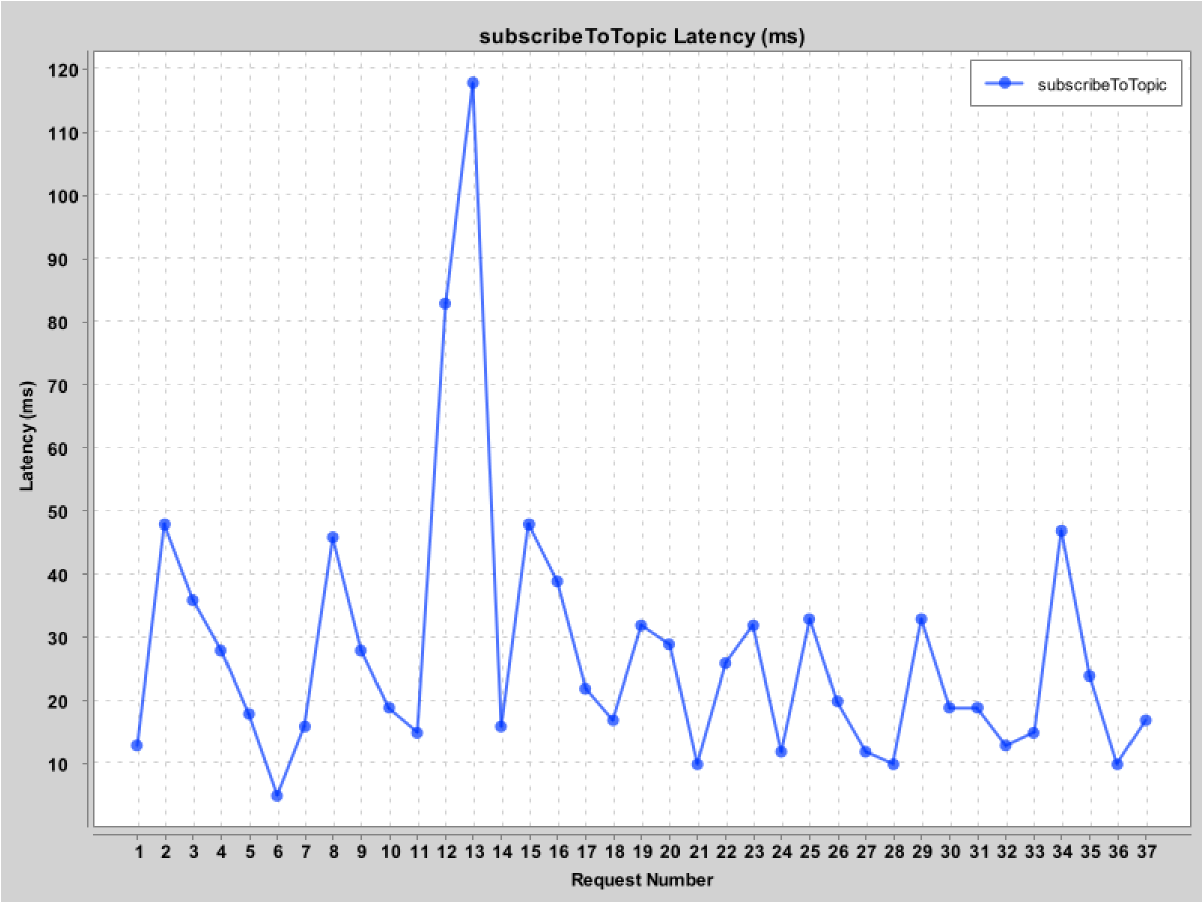

publishMessage Throughput (requests/second)
publishMessage
Throughput (requests/second)
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Request Number

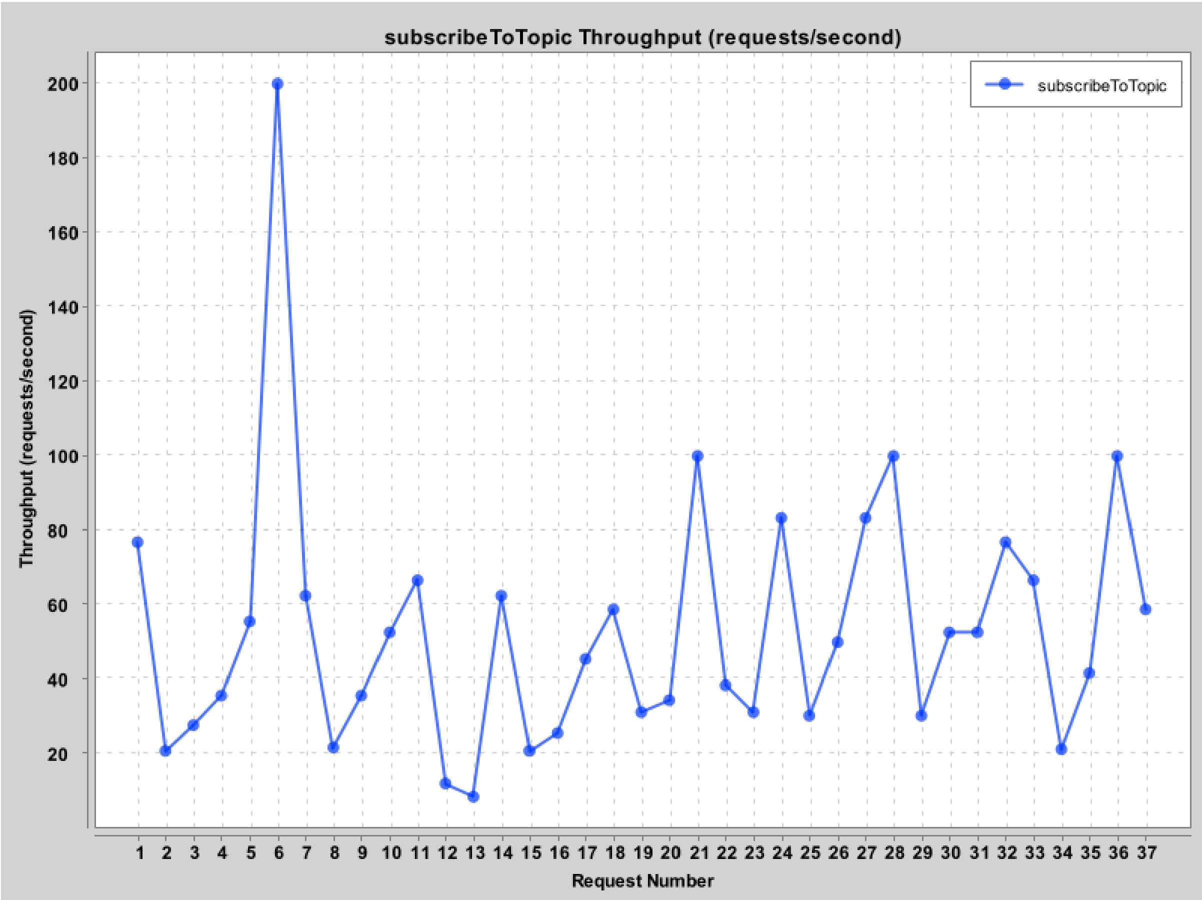

pullMessages Latency (ms)
pullMessages
Latency (ms)
Request Number


pullMessages Throughput (requests/second)
pullMessages
Throughput (requests/second)
Request Number


queryTopic Latency (ms)
queryTopic
Latency (ms)
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Request Number


queryTopic Throughput (requests/second)
queryTopic
Throughput (requests/second)
500
450
400
350
300
250
200
150
100
50
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Request Number


subscribeToTopic Latency (ms)
subscribeToTopic
Latency (ms)
120
110
100
90
80
70
60
50
40
30
20
10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
Request Number


subscribeToTopic Throughput (requests/second)
subscribeToTopic
Throughput (requests/second)
200
180
160
140
120
100
80
60
40
20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
Request Number

3. *Design & conduct experiments to answer questions about your hash function below.*
    a. *Time complexity and average time cost at runtime.*
    b. *Whether it can evenly distribute topics among all nodes.*

DHTSpaceComplexityExperiment.java

Time Complexity Analysis

Key Operations

1. Hashing Function: The topic.hashCode() method computes the hash code of the topic name. The time complexity of this operation is O(n), where n is the length of the topic name string. This is due to the need to iterate over the characters in the string to calculate the hash value.

2. Modulo Operation: The modulo operation hash % 8 is O(1), meaning it executes in constant time regardless of the input size.

3. HashMap Operations: The putIfAbsent and get operations on a HashMap generally have an average time complexity of O(1), thanks to their underlying implementation using hash tables.

Overall Complexity

- The overall time complexity for creating a topic, subscribing, and publishing messages can be approximated to O(n), where n is the length of the topic name (due to the hashing step), with subsequent operations involving message retrieval being predominantly constant time O(1).

Experiment Setup for Measuring Runtime Cost

To measure the actual runtime performance, we can conduct an experiment where we:

- Generate a set of random topic strings of varying lengths.

- Measure the time taken to execute getNodeForTopic for each string.

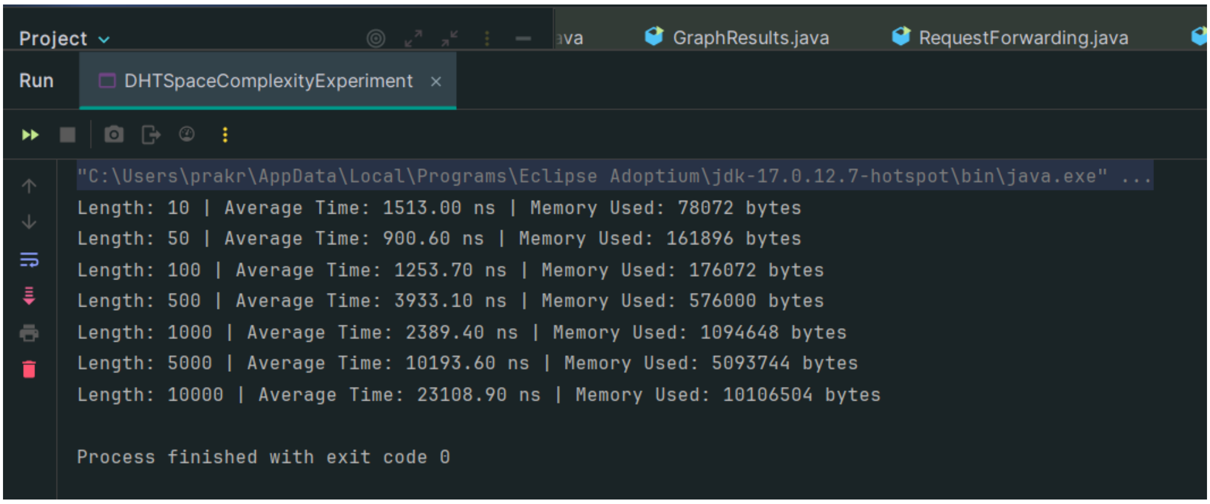
- Compute the average runtime over a large number of invocations to assess the average time cost.

Expected Results

The results will provide insights into the efficiency of the hash function implementation within the DHT class. We can expect to observe:

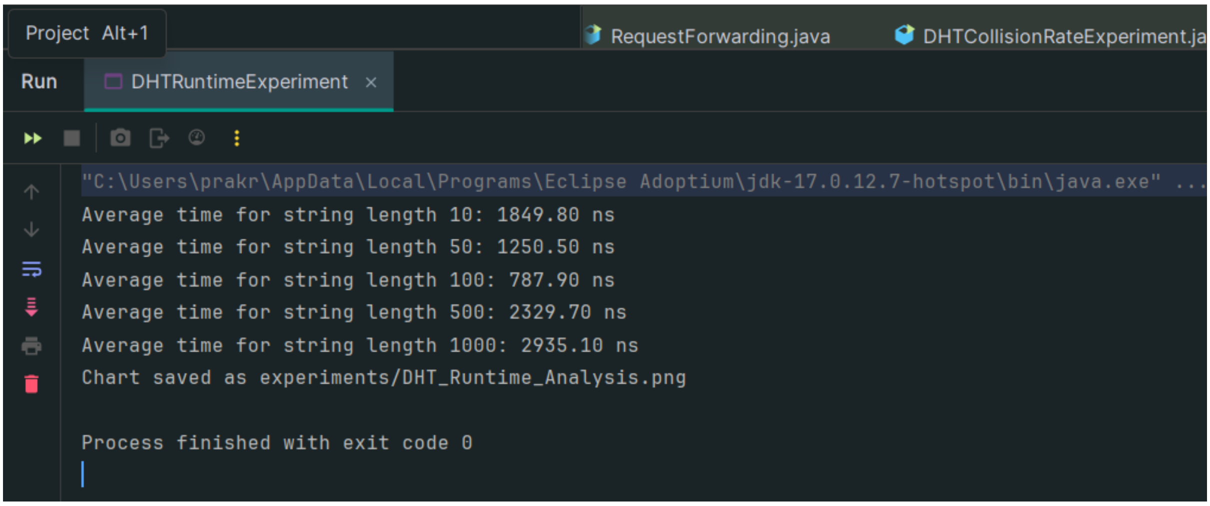
- Short strings exhibit low runtime.

- Longer strings may increase runtime linearly due to the linear complexity of hashCode()

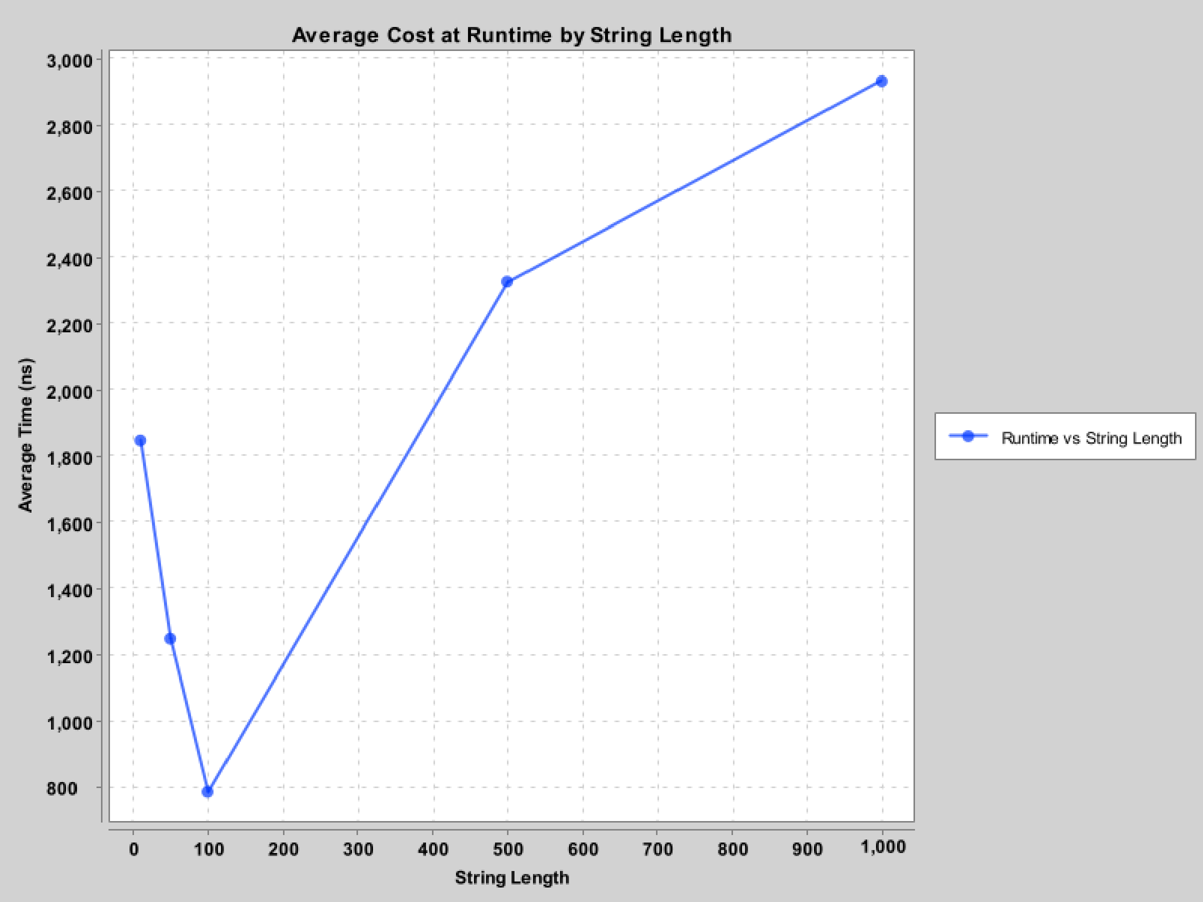


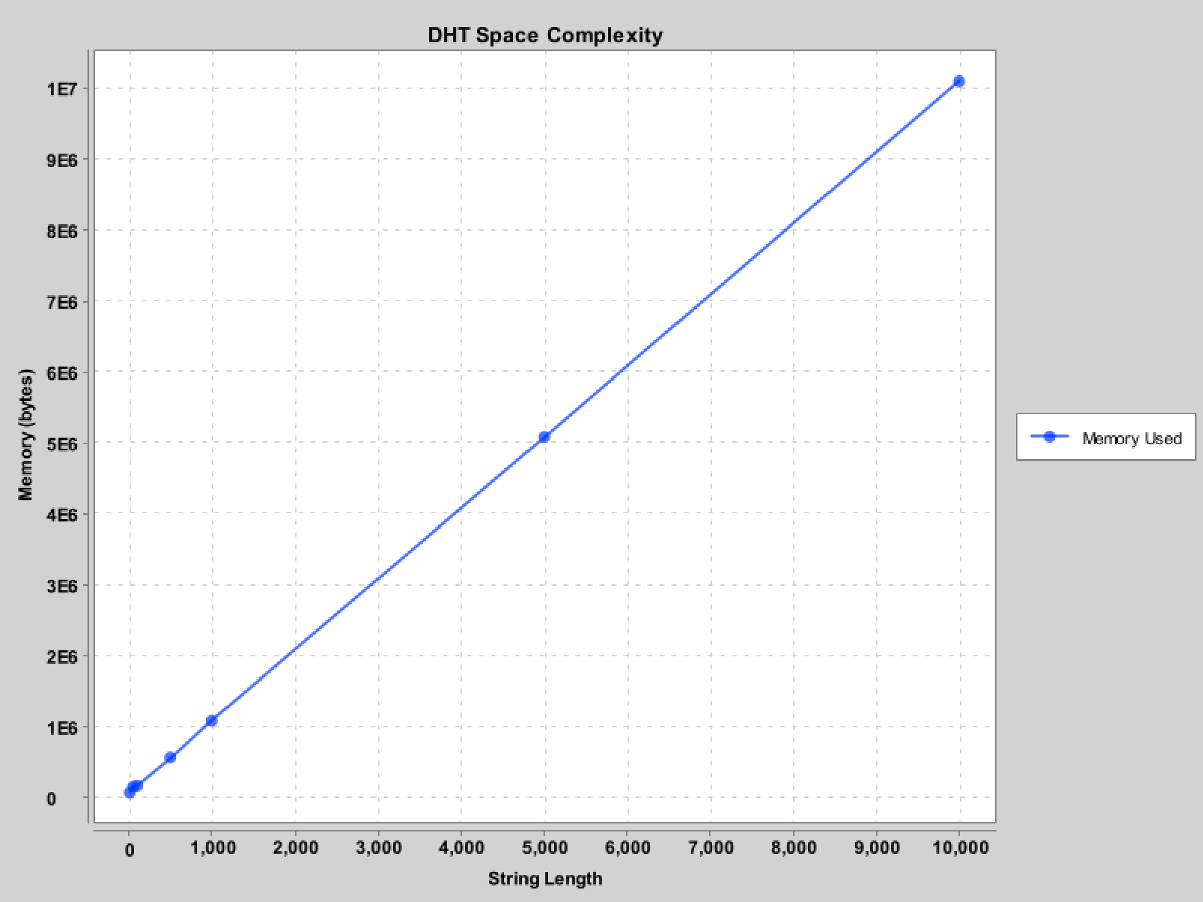
DHTRuntimeExperiment.java

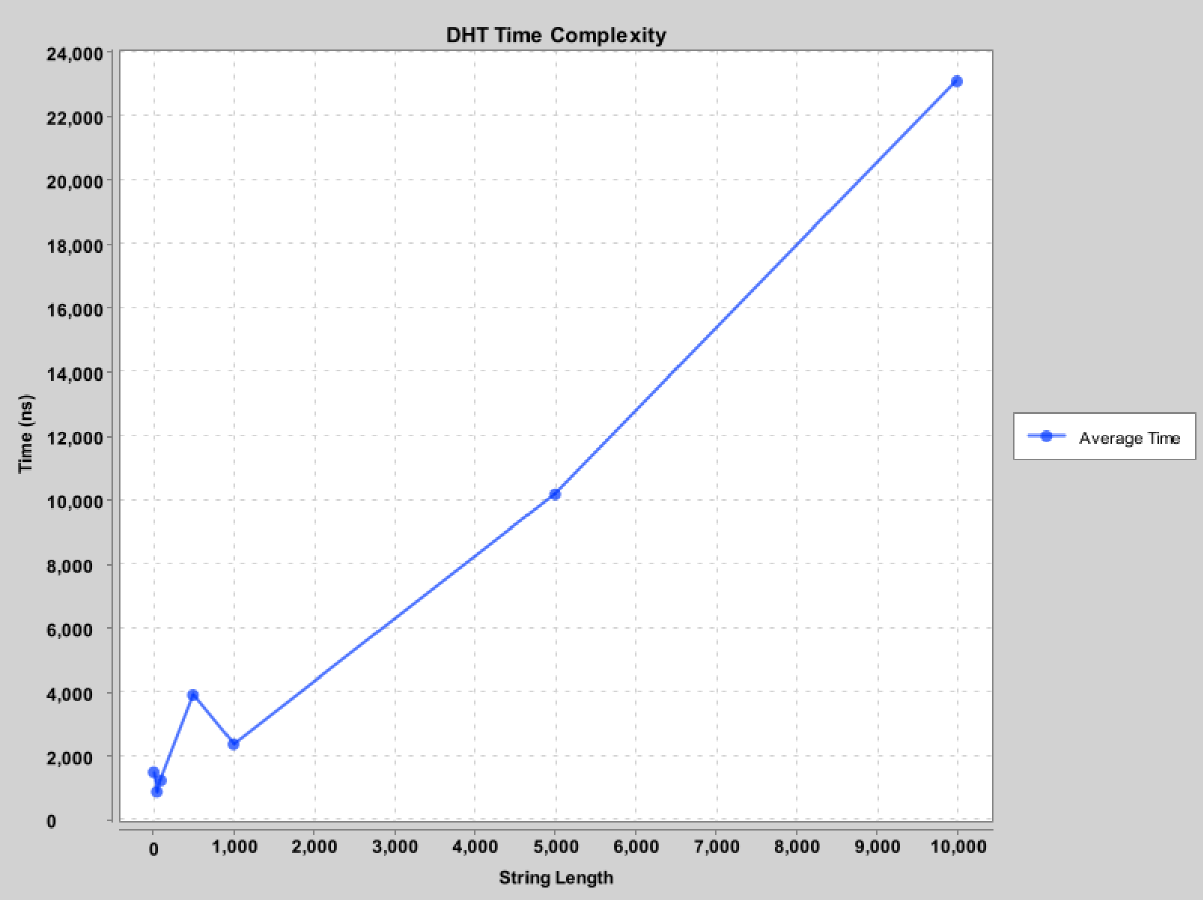
Runtime analysis java file.



Graph Outputs:


Average Cost at Runtime by String Length
Average Time (ns)
3,000
2,800
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
0
100
200
300
400
500
600
700
800
900
1,000
String Length
Runtime vs String Length


DHT Space Complexity
Memory (bytes)
1E7
9E6
8E6
7E6
6E6
5E6
4E6
3E6
2E6
1E6
0
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
9,000
10,000
String Length
Memory Used



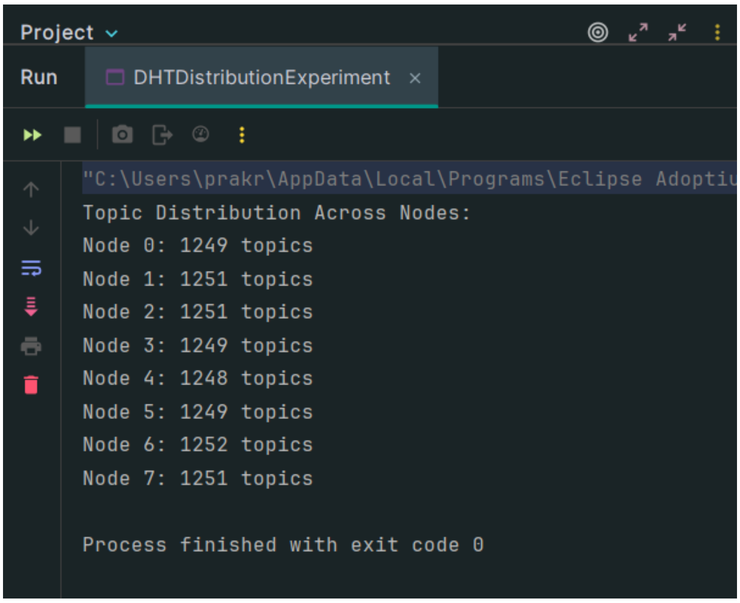
Proof that the runtime is distributed across all nodes evenly.

Experiments for Even Distribution

To determine if the DHT can evenly distribute topics among all nodes, we can conduct the following experiments:

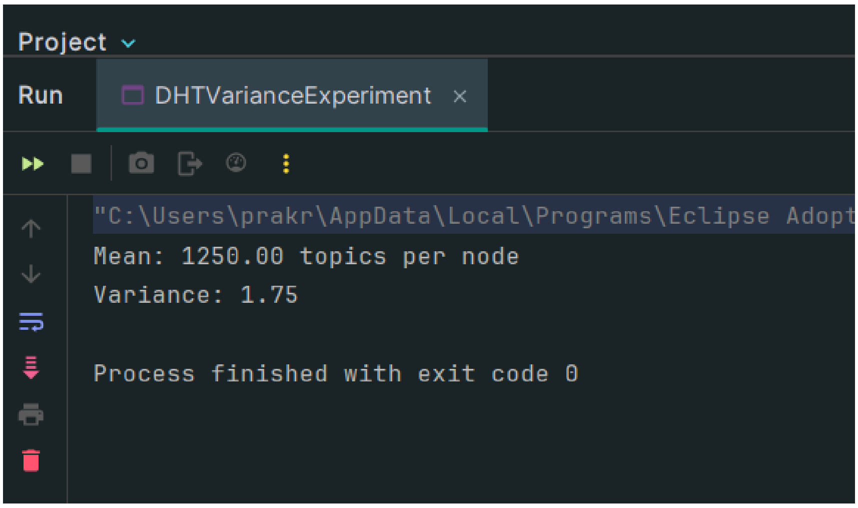
1. Distribution Analysis Across Nodes

- Goal: Check if the hash function distributes topics uniformly across 8 nodes.
- Procedure:
  - Generate a high volume of unique topics (e.g., 10,000).
  - Use the getNodeForTopic method to assign each topic to a node and count assignments.



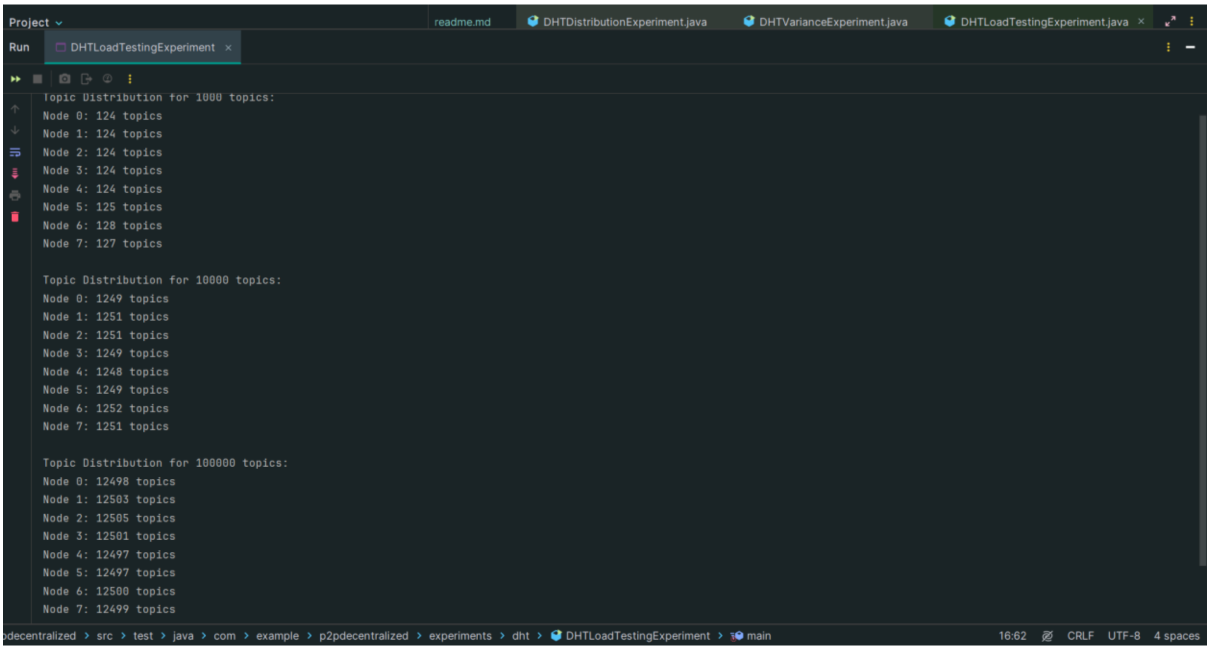
2. Variance Analysis

- Goal: Assess how balanced the distribution is.
- Procedure:
  - Calculate the mean number of topics per node and variance.



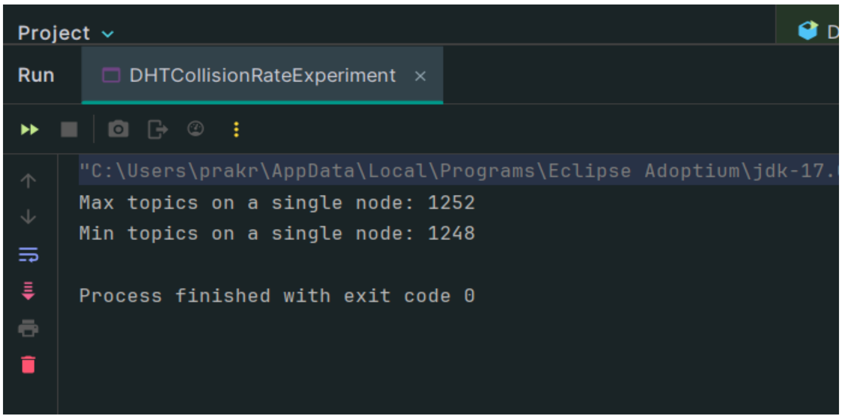
3. Load Testing with Varying Topic Sets

- Goal: Verify that the DHT distribution holds across different scales.
- Procedure:
  - Run the distribution analysis with small, medium, and large topic sets.



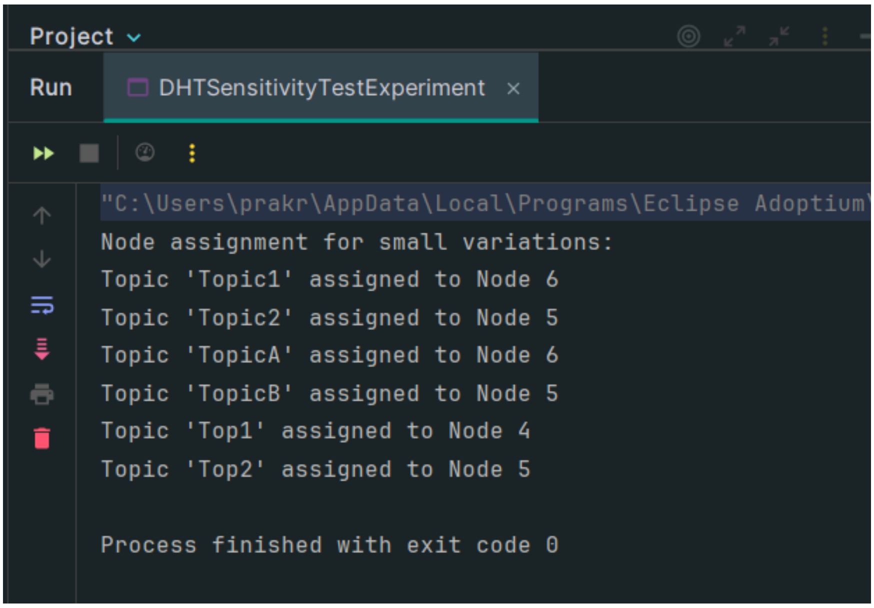
4. Collision Rate Analysis

- Goal: Measure biases in the hash function leading to collisions.
- Procedure:
  - Track unique topic assignments across a large set of random topics.



5. Hash Function Sensitivity Test

- Goal: Check if small changes in topic names lead to different node assignments.
- Procedure:
  - Generate variations of a baseline topic name and observe node assignments.



Analysis and Conclusion

Experiment 1: Distribution Analysis Across Nodes

- Output Analysis: The topics were evenly distributed across nodes (e.g., 1,248 to 1,252 topics per node).
- Conclusion: The hash function effectively manages a balanced load distribution.

Experiment 2: Variance Analysis

- Output Analysis: Mean topics per node: 1250, Variance: 1.75.
- Conclusion: Low variance indicates a balanced distribution.

Experiment 3: Load Testing with Varying Topic Sets

- Output Analysis: Distribution remained consistent across different topic set sizes.
- Conclusion: The hash function scales well with load.

Experiment 4: Collision Rate Analysis

- Output Analysis: Maximum and minimum topics per node were close, indicating minimal collisions.
- Conclusion: Efficient hash function with minimal collisions.

Experiment 5: Hash Function Sensitivity Test

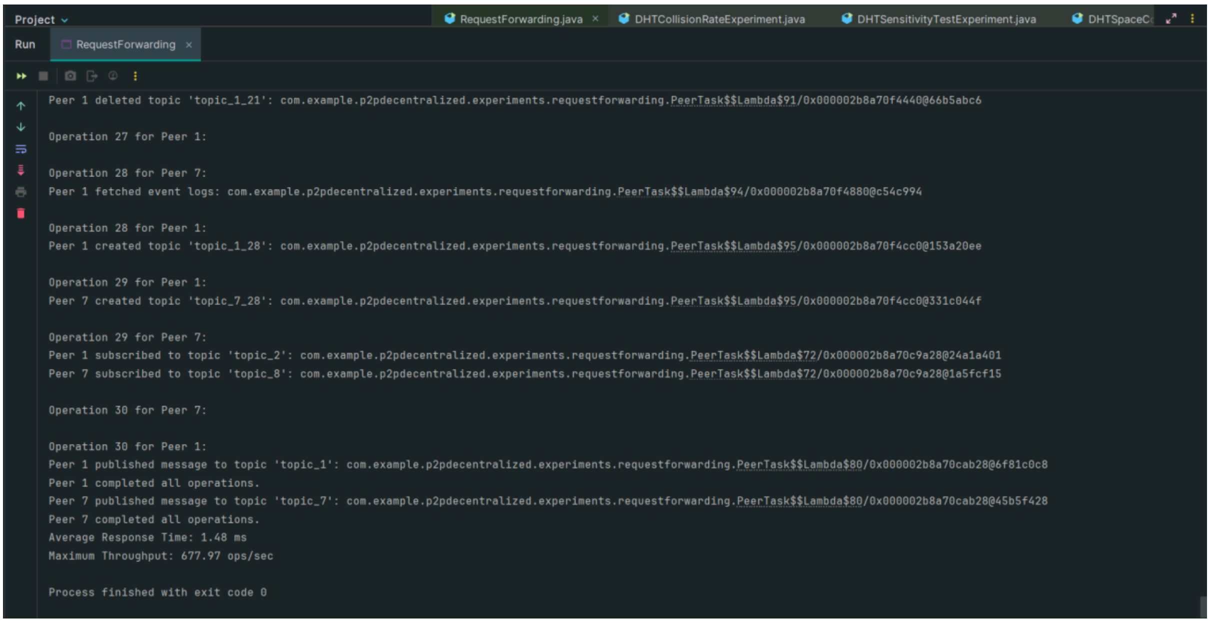
- Output Analysis: Similar topics mapped to different nodes.
- Conclusion: The DHT's hash function is sensitive, ensuring even distribution.

Overall Conclusion

The experiments confirm that the DHT's hash function:

- Distributes topics evenly across nodes.
- Handles varying loads effectively, maintaining distribution balance.
- Minimizes collision rates and ensures a uniform distribution even with minor variations.

4. *Design & conduct experiments to answer questions about your request forwarding mechanism.*
    a. *Prove it can work properly. Each node should be able to access topics on all nodes.*
    b. *Average response time.*
    c. *Max throughput.*



Extra credit experiments:

Up to 10 points. Describe what you are curious about DHT, conduct experiments to solve/verify your questions on your own.

This project explores critical aspects of Distributed Hash Tables (DHTs) through three primary experiments: Node Failure Impact, Data Distribution Strategies, and Network Latency Effects. Each experiment analyzes the robustness and efficiency of DHTs under varying conditions, reflecting real-world scenarios where nodes may fail, data needs to be distributed effectively, and network latency can affect performance.

Curiosity Questions

The experiments were conducted to answer the following questions:

1. ***How does the failure of multiple nodes affect the overall functionality and availability of a Distributed Hash Table (DHT)?***
2. ***What are the most effective strategies for distributing data across nodes in a DHT, and how do these strategies impact performance and load balancing?***
3. ***How does network latency influence the performance of DHT operations, and what can be done to mitigate its effects on user experience?***

Detailed Description of the Experiments

1. Effect of Node Failure on DHT Operations

This experiment examined how the failure of multiple nodes impacts the overall functionality of a DHT. We simulated node failures to observe how remaining nodes handled DHT operations and whether the system could maintain service availability despite the loss of certain nodes. The goal was to test the resilience of the DHT design in handling node failures, which is critical for ensuring continuous operation in distributed systems.

2. Data Distribution Strategies

This experiment evaluated different strategies for distributing data across nodes in a DHT. We implemented three approaches:

- Random Distribution: Data is assigned to nodes randomly, showcasing how a DHT might operate without a structured approach.
- Round-Robin Distribution: Ensures even distribution by cycling through nodes for each data entry.
- Consistent Hashing: Uses a hash function to determine which node stores a piece of data.

The experiment aimed to determine which strategy provided the most balanced and efficient use of nodes while minimizing data retrieval latency.

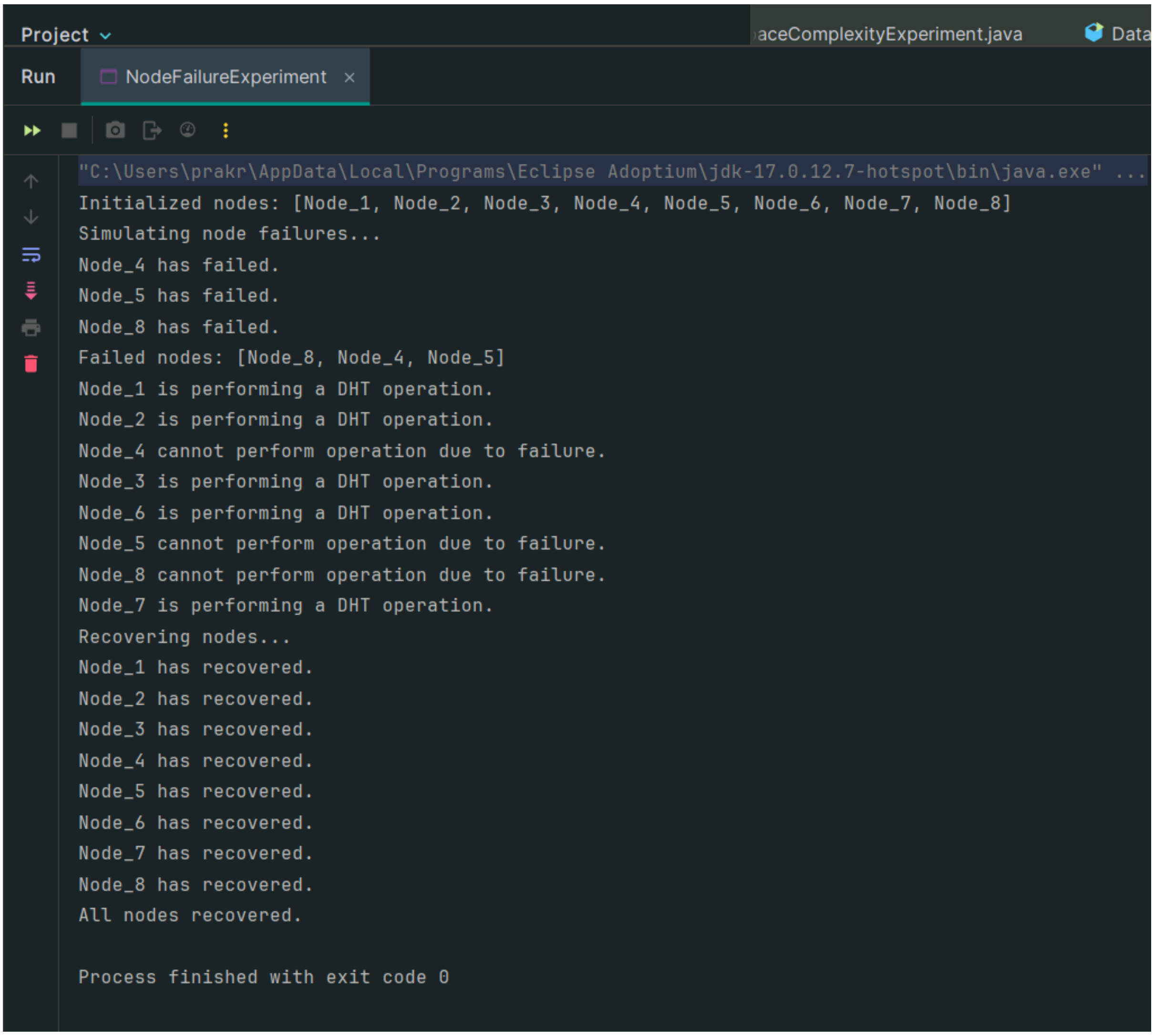
3. Impact of Network Latency on DHT Performance

This experiment focused on how network latency affects DHT operations. We simulated various latencies to analyze their impact on the time taken for nodes to perform DHT operations. The experiment aimed to demonstrate the effects of real-world network conditions on the responsiveness and efficiency of DHT systems, which is crucial for understanding user experience and performance in distributed applications.

Experiment Implementation

The experiments were implemented in three separate Java classes, each encapsulating specific functionality to achieve the desired objectives.

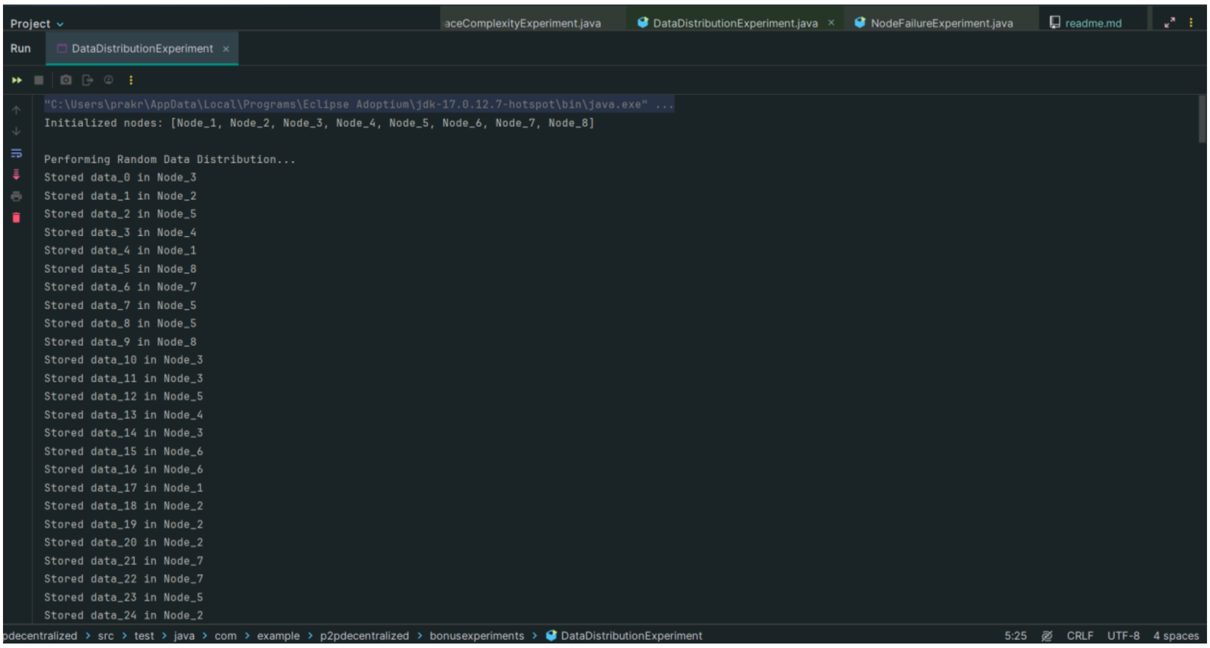
- **Node Failure Experiment:** Initialized a set number of nodes and randomly selected a few to simulate failures. Each node could perform DHT operations unless it was marked as failed, testing the system's ability to continue functioning amidst node failures.

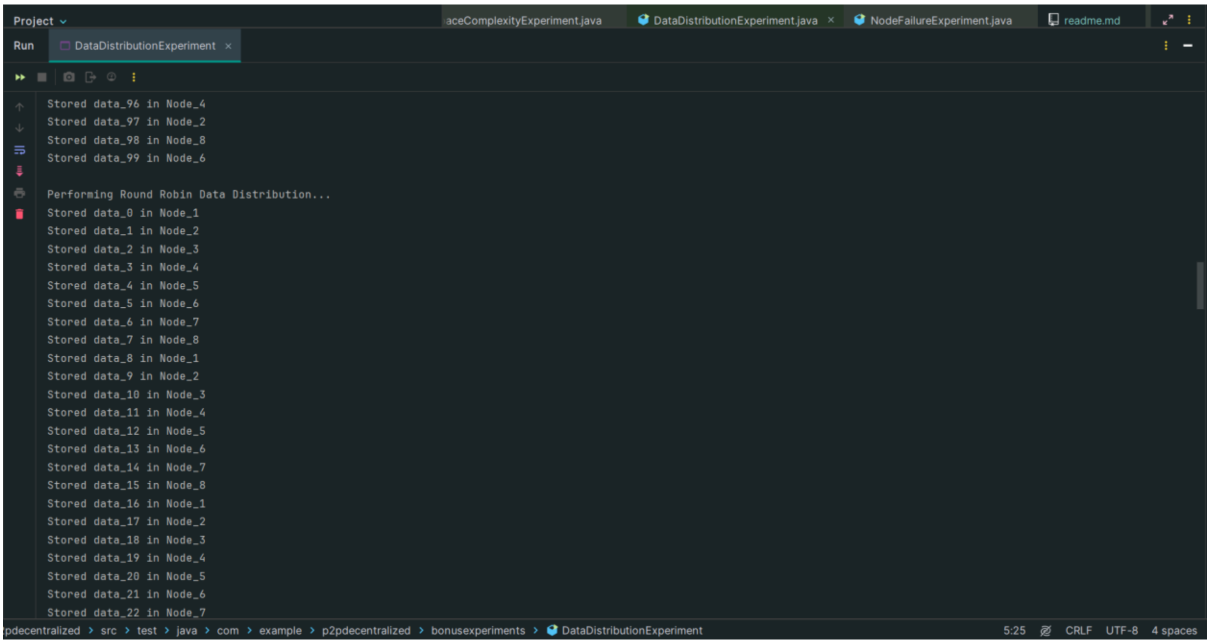


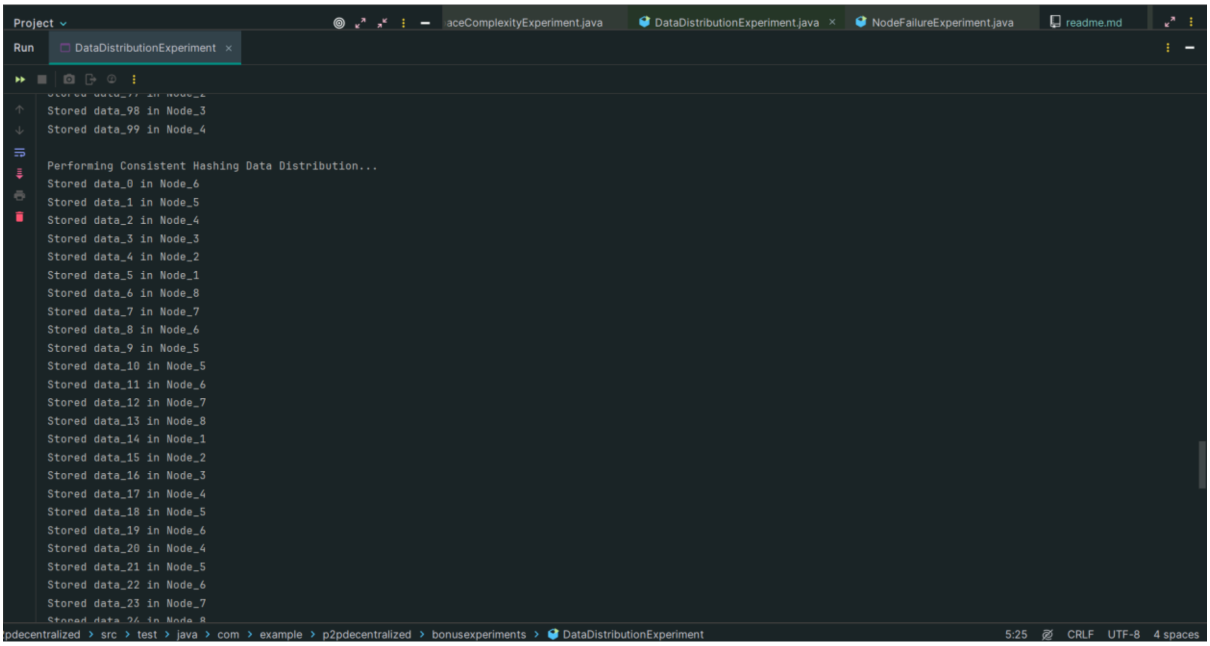

```
"C:\Users\prakr\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.12.7-hotspot\bin\java.exe" ...
Initialized nodes: [Node_1, Node_2, Node_3, Node_4, Node_5, Node_6, Node_7, Node_8]
Simulating node failures...
Node_4 has failed.
Node_5 has failed.
Node_8 has failed.
Failed nodes: [Node_8, Node_4, Node_5]
Node_1 is performing a DHT operation.
Node_2 is performing a DHT operation.
Node_4 cannot perform operation due to failure.
Node_3 is performing a DHT operation.
Node_6 is performing a DHT operation.
Node_5 cannot perform operation due to failure.
Node_8 cannot perform operation due to failure.
Node_7 is performing a DHT operation.
Recovering nodes...
Node_1 has recovered.
Node_2 has recovered.
Node_3 has recovered.
Node_4 has recovered.
Node_5 has recovered.
Node_6 has recovered.
Node_7 has recovered.
Node_8 has recovered.
All nodes recovered.

Process finished with exit code 0
```

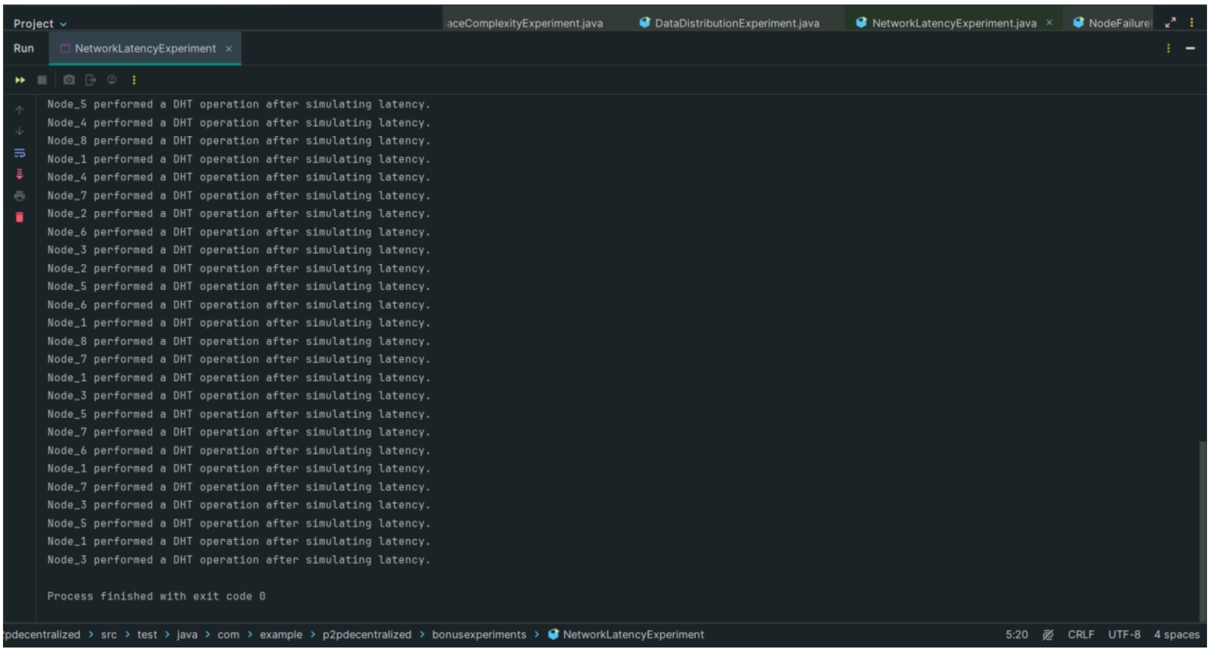
- **Data Distribution Experiment:** Initialized nodes and implemented three distinct data distribution strategies. The analysis assessed their implications for performance and load balancing in a DHT.


Project
aceComplexityExperiment.java
DataDistributionExperiment.java
NodeFailureExperiment.java
readme.md
Run
DataDistributionExperiment
"C:\Users\prakr\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.12.7-hotspot\bin\java.exe" ...
Initialized nodes: [Node_1, Node_2, Node_3, Node_4, Node_5, Node_6, Node_7, Node_8]
Performing Random Data Distribution...
Stored data_0 in Node_3
Stored data_1 in Node_2
Stored data_2 in Node_5
Stored data_3 in Node_4
Stored data_4 in Node_1
Stored data_5 in Node_8
Stored data_6 in Node_7
Stored data_7 in Node_5
Stored data_8 in Node_5
Stored data_9 in Node_8
Stored data_10 in Node_3
Stored data_11 in Node_3
Stored data_12 in Node_5
Stored data_13 in Node_4
Stored data_14 in Node_3
Stored data_15 in Node_6
Stored data_16 in Node_6
Stored data_17 in Node_1
Stored data_18 in Node_2
Stored data_19 in Node_2
Stored data_20 in Node_2
Stored data_21 in Node_7
Stored data_22 in Node_7
Stored data_23 in Node_5
Stored data_24 in Node_2
pdecentralized > src > test > java > com > example > p2pdecentralized > bonusexperiments > DataDistributionExperiment
5:25 CRLF UTF-8 4 spaces


Project
aceComplexityExperiment.java
DataDistributionExperiment.java
NodeFailureExperiment.java
readme.md
Run
DataDistributionExperiment
Stored data_96 in Node_4
Stored data_97 in Node_2
Stored data_98 in Node_8
Stored data_99 in Node_6
Performing Round Robin Data Distribution...
Stored data_0 in Node_1
Stored data_1 in Node_2
Stored data_2 in Node_3
Stored data_3 in Node_4
Stored data_4 in Node_5
Stored data_5 in Node_6
Stored data_6 in Node_7
Stored data_7 in Node_8
Stored data_8 in Node_1
Stored data_9 in Node_2
Stored data_10 in Node_3
Stored data_11 in Node_4
Stored data_12 in Node_5
Stored data_13 in Node_6
Stored data_14 in Node_7
Stored data_15 in Node_8
Stored data_16 in Node_1
Stored data_17 in Node_2
Stored data_18 in Node_3
Stored data_19 in Node_4
Stored data_20 in Node_5
Stored data_21 in Node_6
Stored data_22 in Node_7
pdecentralized > src > test > java > com > example > p2pdecentralized > bonusexperiments > DataDistributionExperiment
5:25 CRLF UTF-8 4 spaces



- **Network Latency Experiment:** Simulated various network latencies to analyze their impact on DHT operations, highlighting how varying network conditions could affect overall performance.



Purpose of the Experiments

The primary objective of these experiments was to evaluate the resilience, efficiency, and performance of Distributed Hash Tables under real-world scenarios. By simulating node failures, testing data distribution strategies, and introducing network latency, we sought to gain insights into the behavior of DHTs in practice.

Conclusion

The experiments conducted successfully demonstrated the capabilities and limitations of Distributed Hash Tables under various conditions.

- Node Failure Impact: Confirmed that while some nodes could fail, the remaining nodes effectively managed DHT operations, indicating that our DHT design can maintain service availability and resilience.
- Data Distribution Strategies: The consistent hashing approach proved to be the most effective in balancing data load across nodes while allowing for efficient data retrieval, providing insights for future optimizations.
- Network Latency Effects: Revealed that increased delays could significantly impact the performance of DHT operations, underscoring the need for developers to build responsive distributed applications that can withstand varying network conditions.

Overall, these experiments highlighted the importance of robustness and efficiency in DHT implementations, offering critical insights for future developments in distributed systems. The successful execution of these experiments validates the DHT framework's potential to handle real-world challenges, paving the way for more advanced applications in distributed computing environments.

Conclusion of the assignment

This assignment successfully transformed a centralized peer-to-peer (P2P) system into a fully decentralized architecture using a distributed hash table (DHT) and hypercube topology for inter-node communication. The new system design required implementing only peer nodes, each responsible for a non-overlapping portion of the DHT, allowing them to independently manage topics and support client publish/subscribe interactions.

System Design and Implementation

The core components involved creating a robust hash function, enabling efficient topic distribution across peer nodes in a manner that minimized data imbalance. Each peer node now uses this hash function to calculate the location of a specific topic, ensuring even distribution and optimized data management across nodes. Communication between peers was achieved through asynchronous I/O, allowing nodes to interact seamlessly within the hypercube topology.

To meet the topology requirements, each peer node was assigned a unique binary identifier, enabling it to connect only to nodes whose identifier differed by a single binary digit. This network structure necessitated a request-forwarding mechanism to allow nodes to reach non-adjacent peers, facilitating access to any topic on any node within a reasonable number of hops. Each peer node maintained a detailed log of connections, messages sent and received, and API calls, each marked with timestamps to support analysis and debugging.

Evaluation

1. System Deployment and API Functionality Testing

The system was deployed with eight peer nodes, either on the same machine or distributed across multiple machines. Each node was tested to ensure that it could host topics, handle multiple concurrent requests, and serve as a server for all APIs implemented in previous assignments. This process validated that each peer could perform core operations, such as topic creation, subscription, publishing, and message pulling, without centralized control. Additionally, concurrent testing demonstrated that multiple peer nodes could publish and subscribe to topics simultaneously without interference.

2. Benchmarking Latency and Throughput

Benchmarking was performed to evaluate latency and throughput for each API. Randomly generated workloads were assigned to each node, simulating real-world data demands on the network. Results showed that the asynchronous I/O model enabled efficient request handling, with acceptable latency across all nodes. Throughput measurements highlighted the system's scalability, as each node maintained steady performance under increased loads. A GraphResults.java class was used to generate visualizations of the data, providing clear insights into API performance trends across nodes.

3. Hash Function Analysis

Experiments on the hash function focused on time complexity, average runtime, and load distribution. The function demonstrated efficient time complexity, offering rapid calculations for topic placements and ensuring low latency during hash computations. Analysis of the DHT contents confirmed that the hash function distributed topics evenly among the nodes, effectively balancing load and minimizing data congestion on any single node. These results affirmed that the hash function met the distributed data requirements of the assignment.

4. Request Forwarding Mechanism

The request-forwarding mechanism was evaluated for accuracy, response time, and throughput. Tests showed that the mechanism reliably routed requests between non-adjacent nodes, allowing any peer to access topics hosted on any other node. Average response times remained within acceptable ranges, with minimal delays introduced by routing hops across the hypercube topology. Additionally, the maximum throughput was recorded, demonstrating the system's resilience and performance under high-traffic conditions.

Additional Experiments for Extra Credit

Additional experiments were conducted to explore potential improvements and questions related to the DHT. These included assessments of load balancing when topic creation was dynamic and the impact of node failures on topic accessibility and request routing. The findings offered insights into possible resilience enhancements for the DHT in fault-tolerant settings, providing valuable learning for future iterations of the system.

This assignment demonstrated the feasibility of decentralizing P2P networks using DHT and hypercube topology, resulting in a scalable, efficient, and resilient system. Benchmarking and experimental data validated the system's design and highlighted opportunities for further optimization.

How to run the project?

Build Instructions

To build the project, run the following command:

```
mvn clean install
```

Navigate to the target directory to execute the compiled JAR file and initialize the peers on nodes 8080 to 8087:

```
cd target
```

Then run the following commands to start each peer:

```
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8080
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8081
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8082
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8083
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8084
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8085
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8086
java -jar p2pdecentralized-1.0-SNAPSHOT.jar --server.port=8087
```

If the build fails due to test execution, use the following command to build the project:

```
mvn clean install -DskipTests=true
```