Name	**Prakriti Sharma**
Subject	**CS550 Advance Operating Systems**
CWID	**A20575259**
Topic	**PA1**

INDEX

Requirement:

The assignment is a Peer-to-Peer System in JAVA language.

Tools Used:

Maven, Gradle, XYChart, SpringBoot, JAVA

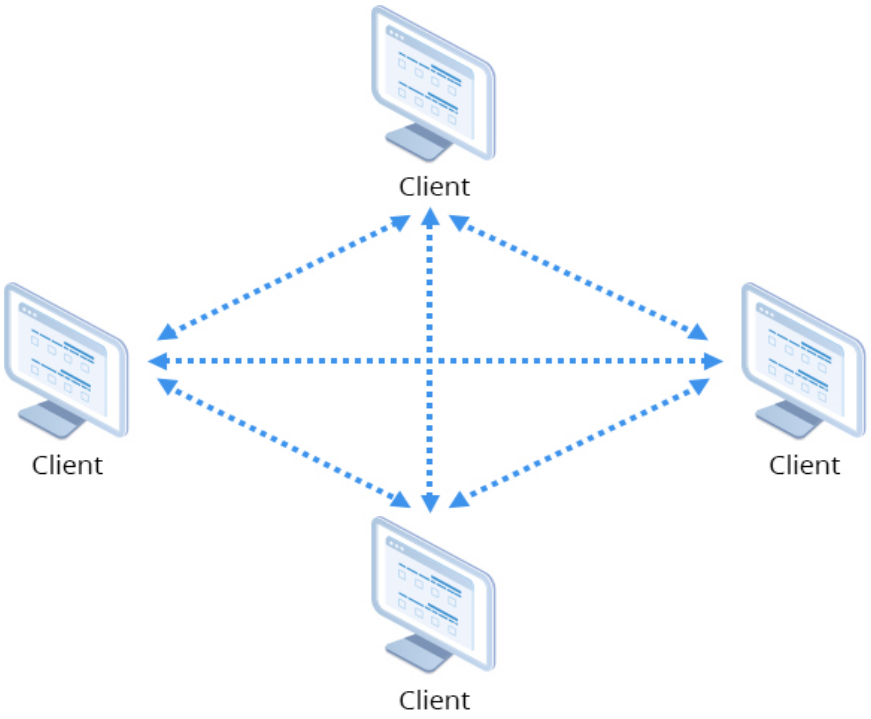
What exactly is a Peer-to-Peer System?

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:

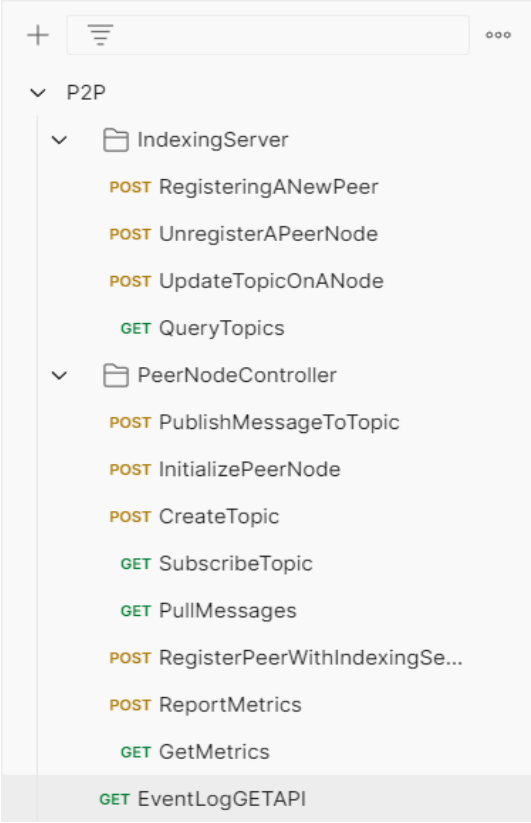
1. Decentralization:
    - Unlike traditional client-server models, there is no central server managing the system. All peers are equally privileged and can communicate directly with each other.
2. Distributed Resources:
    - Resources (such as files, data, or computing power) are distributed among 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



Structure of the APIs:

The requirements of PA2 gives rise to 13 API endpoints.



OVERVIEW

All the above APIs are an extended version of the APIs in Publisher Subscriber System.

The flow of the APIs and a simple dry run through the API cURLs:

Let's go through the flow of your Peer-to-Peer system, understanding each API step-by-step.

1. Initialize Peer Node

API Endpoint: /peer/initialize
Method: POST

cURL:

```
curl -X POST "http://localhost:8080/peer/initialize?indexServerIp=127.0.0.1&indexServerPort=8081"
```

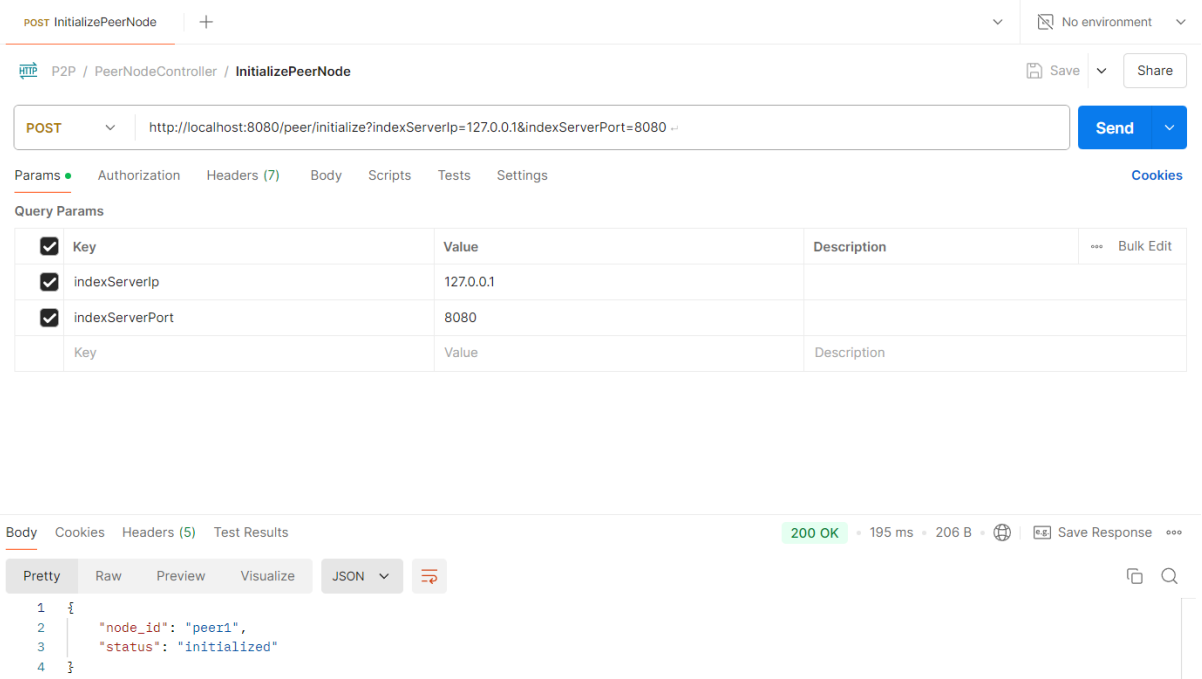
What does the API do?

- Initializes a peer node by assigning it a unique node_id and connecting it to the indexing server.

How does it work?

- The peer node generates a node_id (e.g., "peer1") and stores the URL of the indexing server using the IP and port provided in the request. It returns the node_id to confirm initialization.

Screenshot from postman (Working of the API):



2. Register Peer with Indexing Server

API Endpoint: /peer/register_with_indexing_server
Method: POST

cURL:

```
curl -X POST "http://localhost:8080/peer/register_with_indexing_server"
```

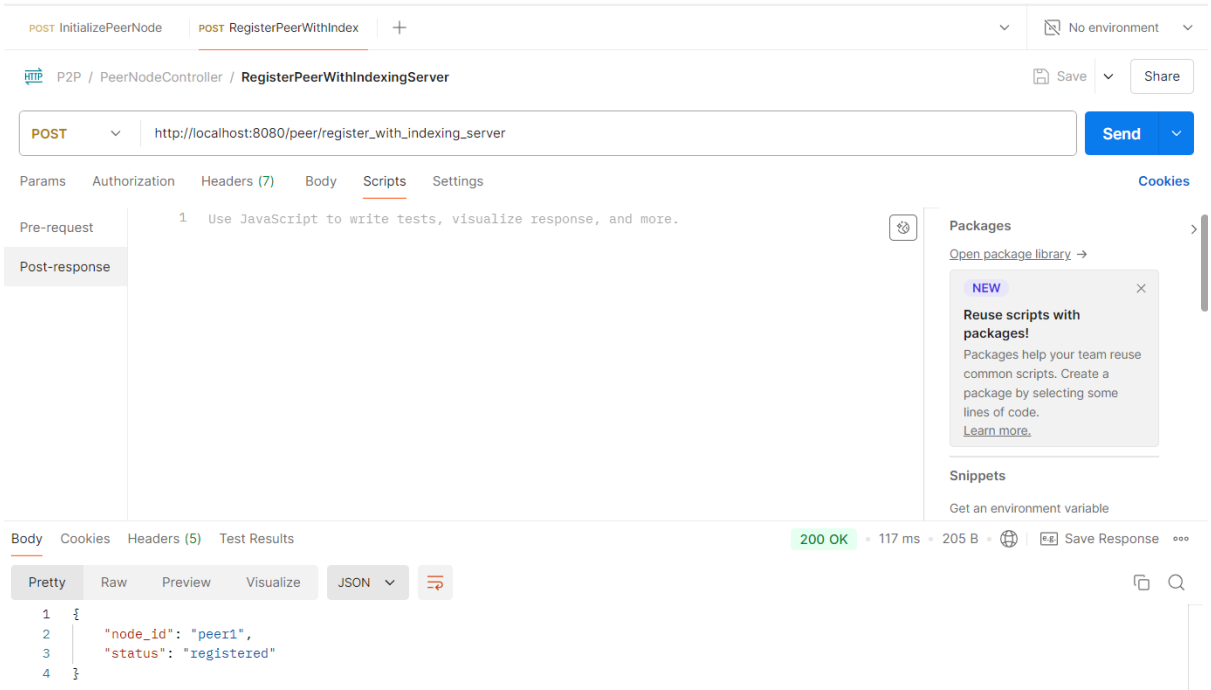
What does the API do?

- Registers the peer node with the central Indexing Server, providing its node_id and topics it is hosting.

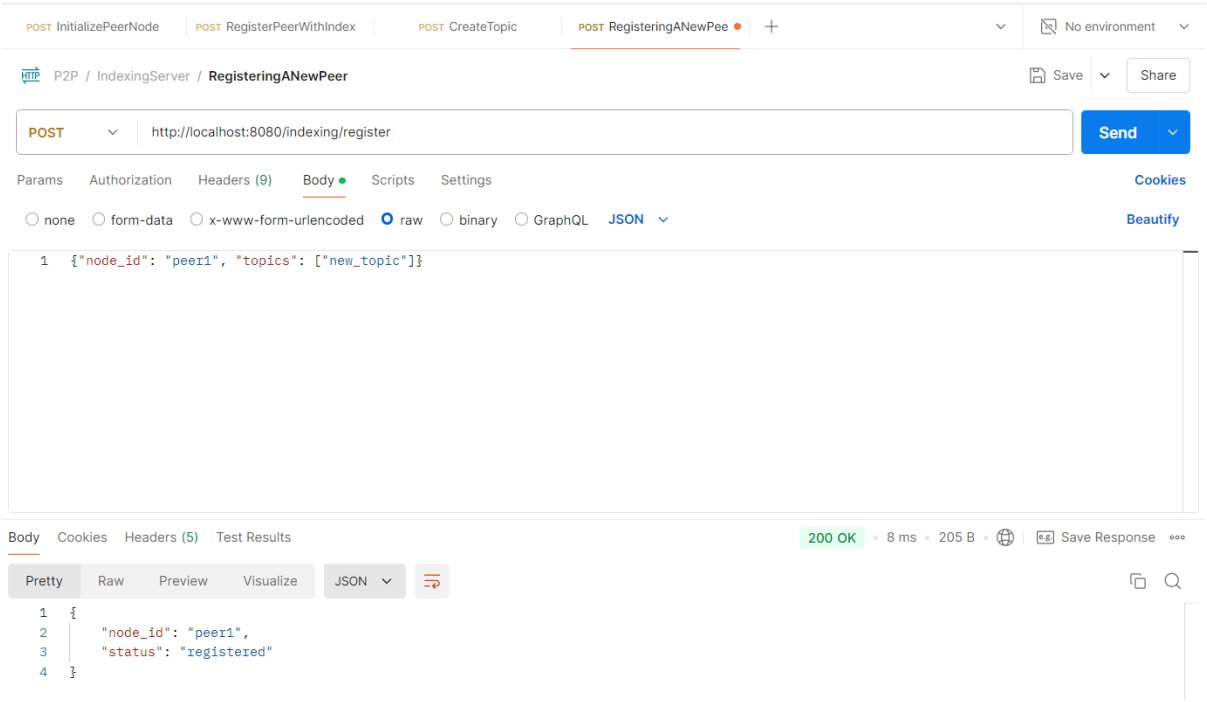
How does it work?

- The peer sends a POST request to the /indexing/register endpoint of the indexing server, passing its node_id and hosted topics. The indexing server stores the peer and its topics, so it can be later queried by other peers.

Screenshot from postman (Working of the API):



The below screenshot shows the postman API for indexing server, to register a peer on indexing server itself.



cURL for the same:

```
curl --location 'http://localhost:8080/indexing/register' \
--header 'Content-Type: application/json' \
--data '{"node_id": "peer1", "topics": ["new_topic"]}'
```

3. Create Topic on Peer Node

API Endpoint: /peer/create_topic
Method: POST

cURL:

```
curl -X POST -d "new_topic" "http://localhost:8080/peer/create_topic"
```

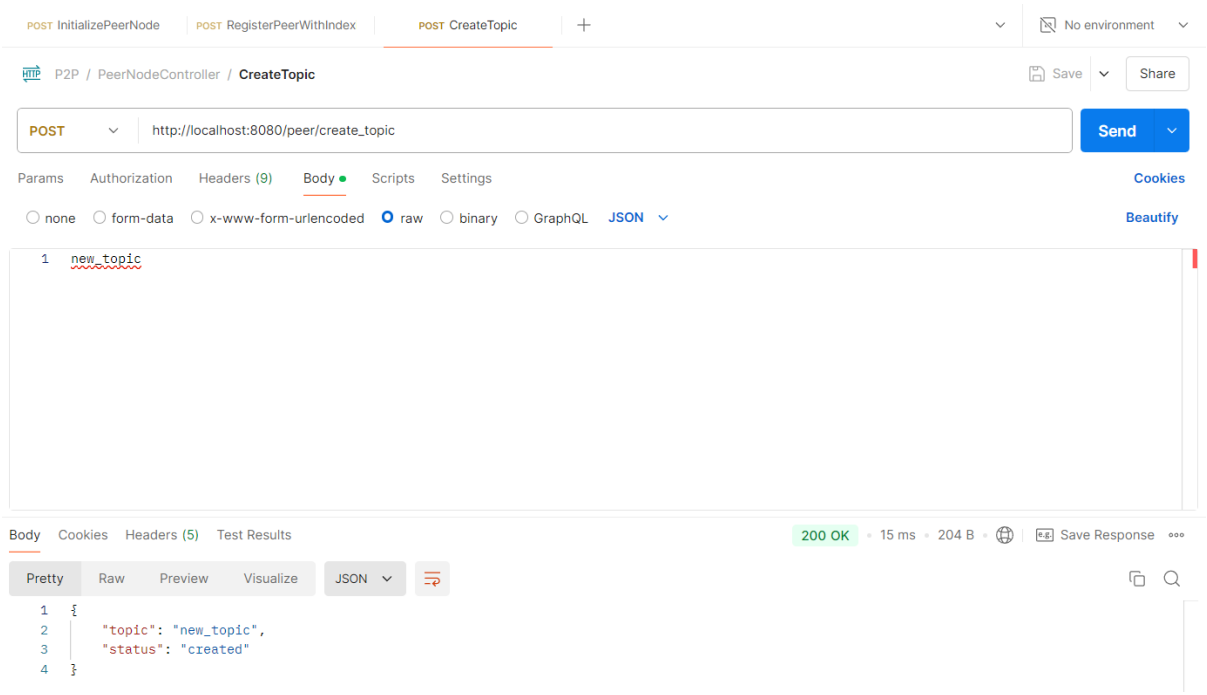
What does the API do?

- Creates a new topic on the peer node and registers this change with the indexing server.

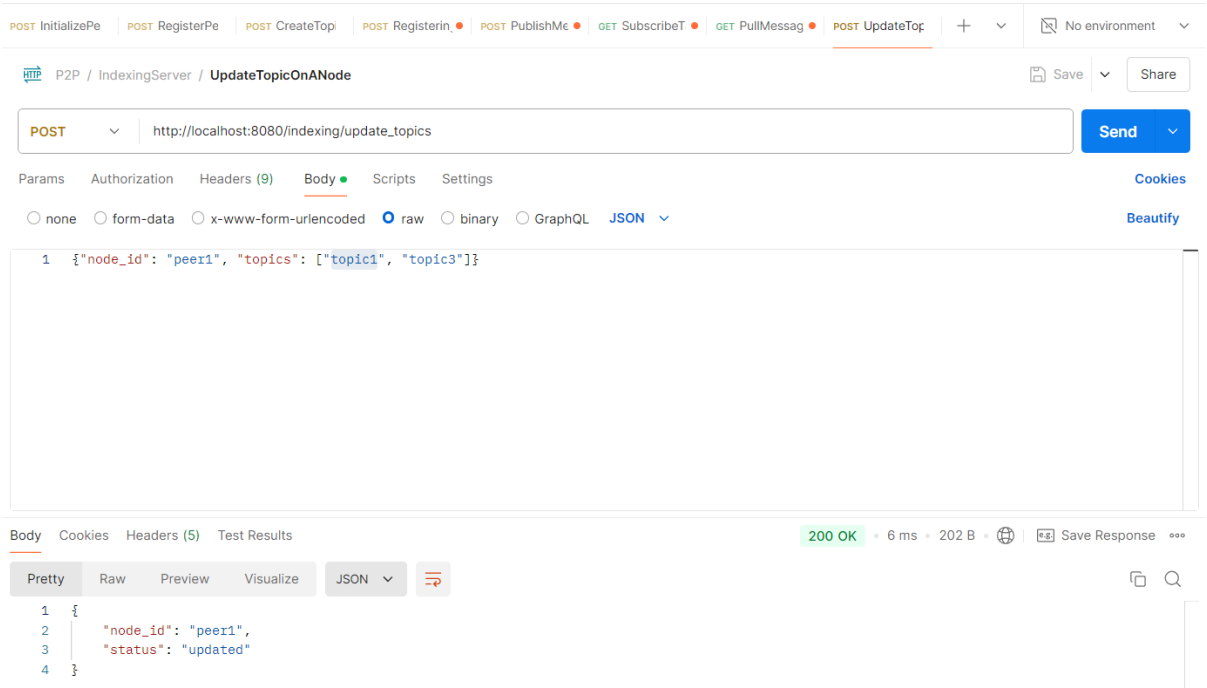
How does it work?

- The peer adds the provided topic to its list of hosted topics and then calls the registerWithIndexingServer method to inform the indexing server of the new topic.

Screenshot from postman (Working of the API):



The below screenshot shows the postman API for indexing server, to update the name of the topic under a peer.



cURL:

```
curl --location 'http://localhost:8080/indexing/update_topics' \
--header 'Content-Type: application/json' \
--data '{"node_id": "peer1", "topics": ["topic1", "topic3"]}'
```

4. Publish Message to Topic

API Endpoint: /peer/publish
Method: POST

cURL:

```
curl -X POST -H "Content-Type: application/json" -d '{"topic":"new_topic","message":"Hello, P2P World"}' "http://localhost:8080/peer/publish"
```

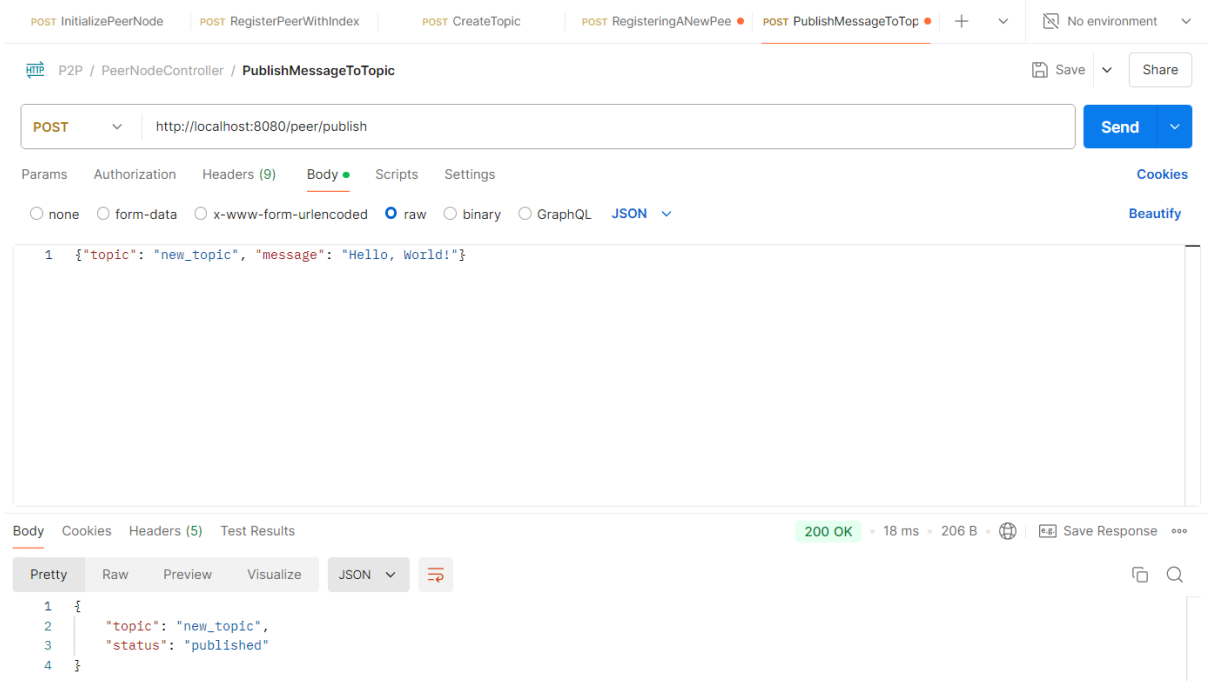
What does the API do?

- Publishes a message to a topic hosted by the peer node.

How does it work?

- The peer checks if it is hosting the topic, and if so, adds the message to its topicMessages map under the corresponding topic. If the topic is not hosted by this peer, it returns an error.

Screenshot from postman (Working of the API):



5. Subscribe to a Topic

API Endpoint: /peer/subscribe/{topic}
Method: GET

cURL:

```
curl -X GET "http://localhost:8080/peer/subscribe/new_topic"
```

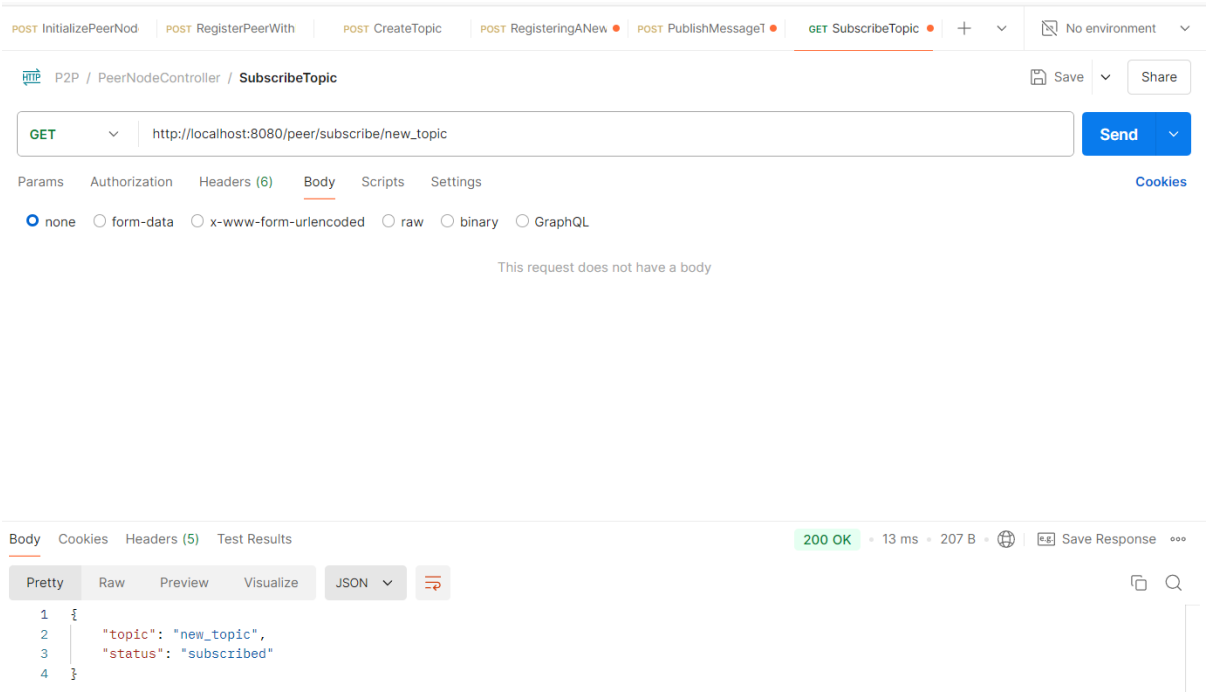
What does the API do?

- Subscribes the peer to a topic by querying the indexing server to find which node is hosting the topic.

How does it work?

- The peer sends a request to the indexing server to find the node hosting the topic. If the topic is hosted by another peer, it forwards the subscription request to the correct peer node. If the current peer is hosting the topic, it adds itself as a subscriber.

Screenshot from postman (Working of the API):



6. Pull Messages from a Topic

API Endpoint: /peer/pull_messages/{topic}
Method: GET

cURL:

```
curl -X GET "http://localhost:8080/peer/pull_messages/new_topic"
```

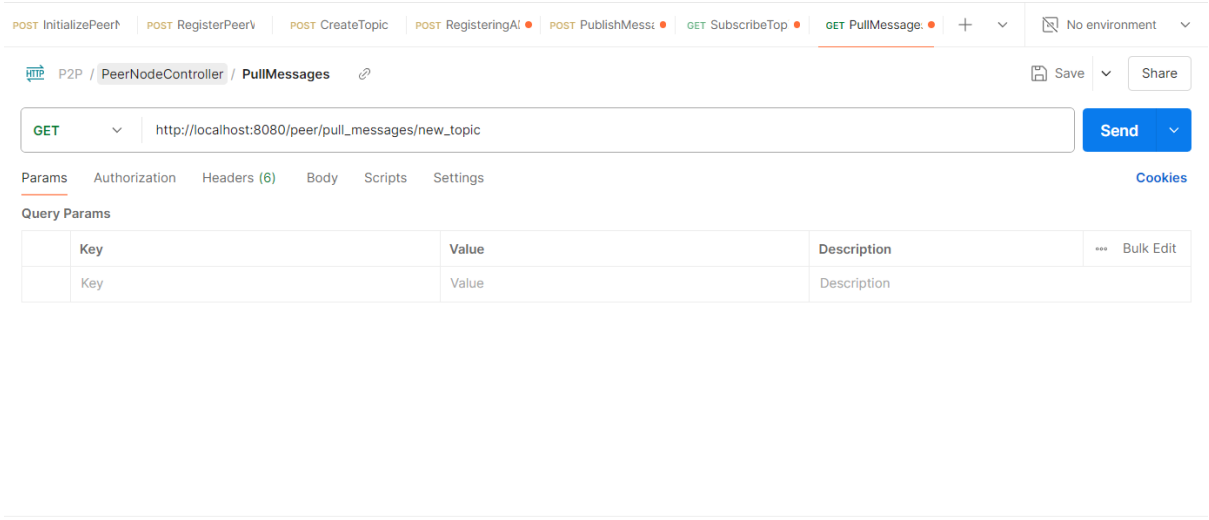
What does the API do?

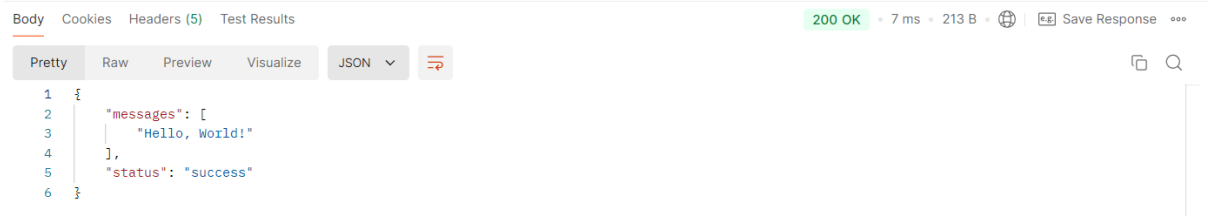
- Pulls messages from a specific topic hosted by the peer node.

How does it work?

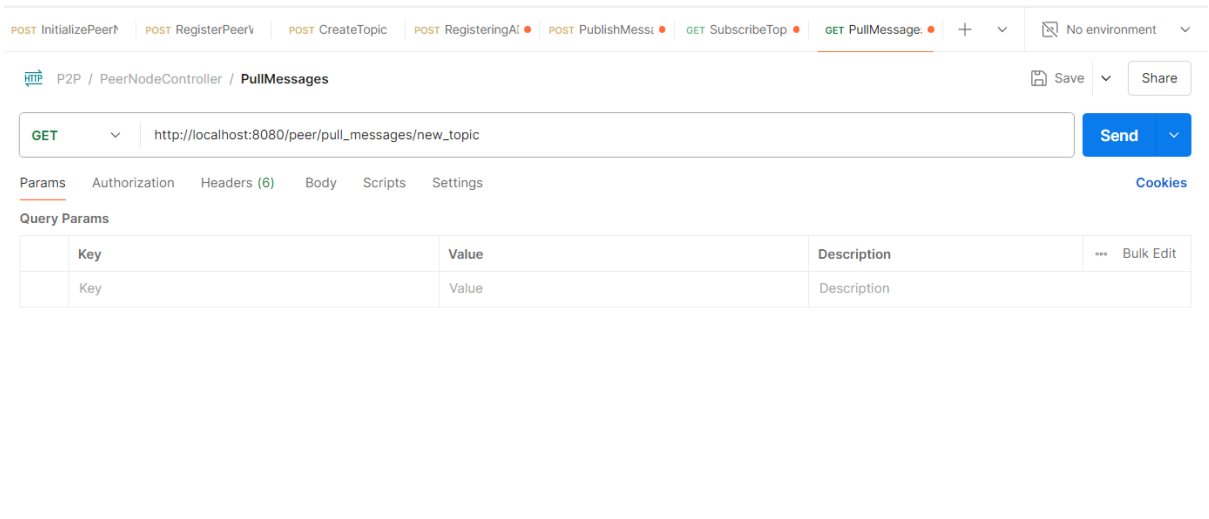
- The peer checks if it hosts the topic, retrieves all messages for that topic from its topicMessages map, and then clears the list after the messages are returned.

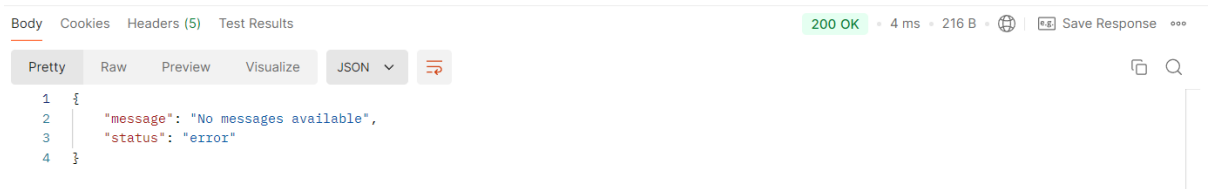
Screenshot from postman (Working of the API):





The message, once read, should not be displayed again.





7. Unregister Peer Node

API Endpoint: /indexing/unregister
Method: POST

cURL:

```
curl -X POST -H "Content-Type: application/json" -d '{"node_id":"peer1"}'
"http://localhost:8081/indexing/unregister"
```

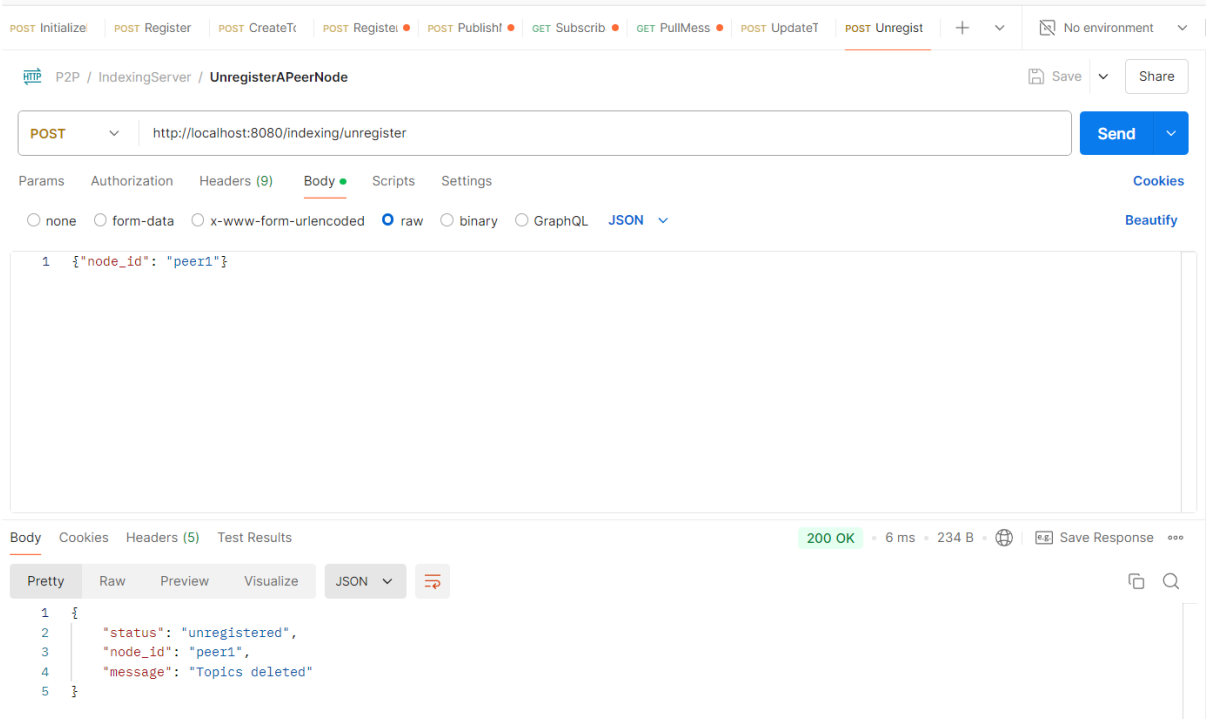
What does the API do?

- Unregisters a peer node from the indexing server.

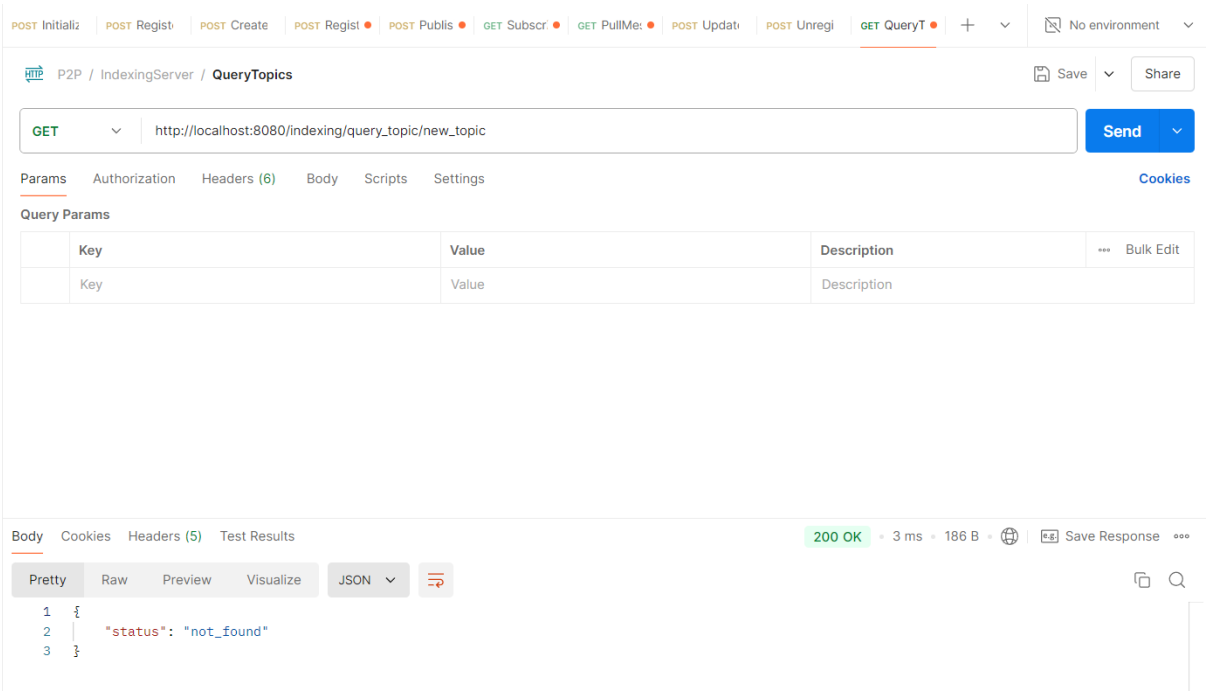
How does it work?

- The indexing server removes the peer node from its list of registered peers and attempts to migrate its topics to another peer, if possible. If no other peers are available, the topics are deleted.

Screenshot from postman (Working of the API):



The below screenshot shows a proof that the topics were deleted.



8. Query Topic from Indexing Server

API Endpoint: /indexing/query_topic/{topic}
Method: GET

cURL:

```
curl -X GET "http://localhost:8081/indexing/query_topic/new_topic"
```

What does the API do?

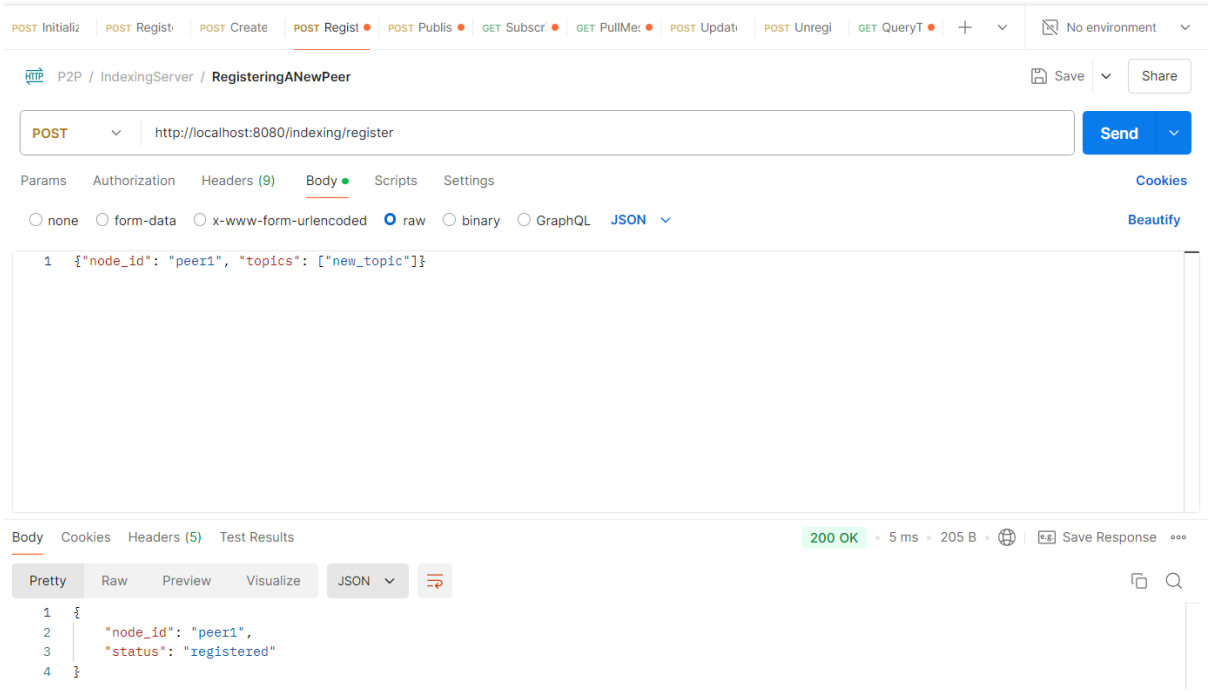
- Queries the indexing server to find out which peer node is hosting a specific topic.

How does it work?

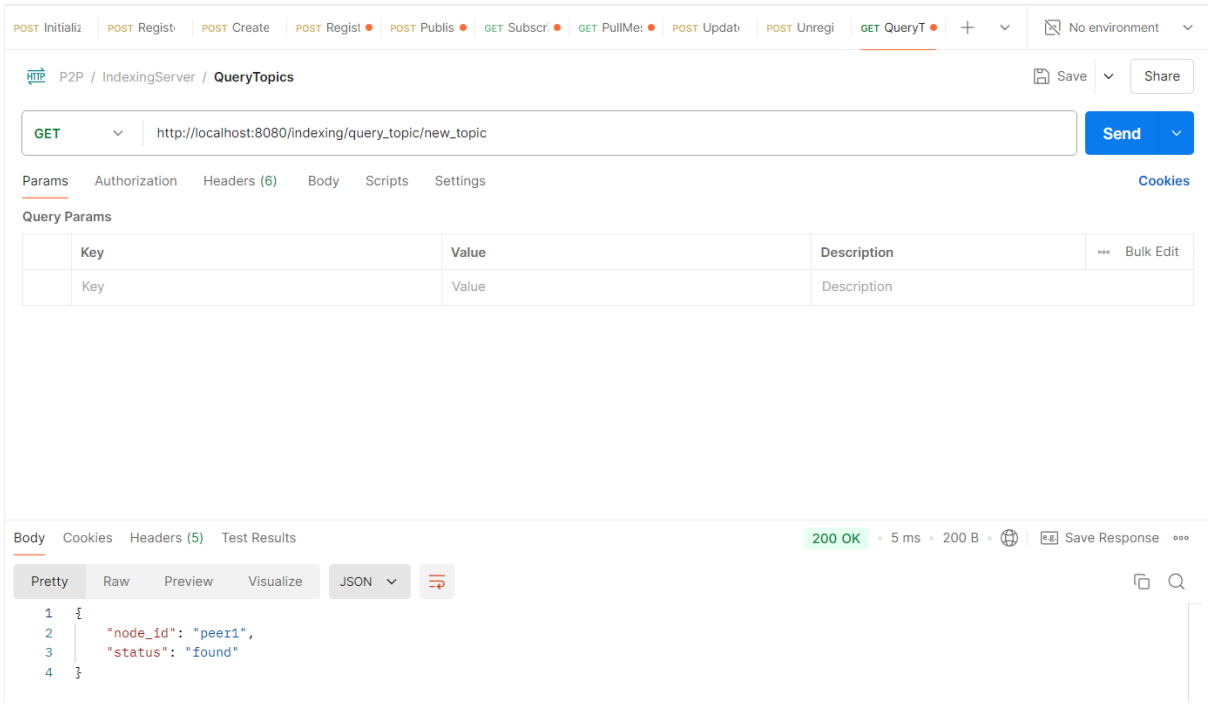
- The indexing server searches through its map of peers and topics. If a peer is hosting the requested topic, it returns the node_id of that peer; otherwise, it returns a "not_found" status.

Screenshot from postman (Working of the API):

Registering a new peer with topic to query later:



The query:



9. Report Metrics to Indexing Server

API Endpoint: /peer/report_metrics
Method: POST

cURL:

```
curl -X POST -H "Content-Type: application/json" -d '{"latency":10, "bandwidth":500}' "http://localhost:8080/peer/report_metrics"
```

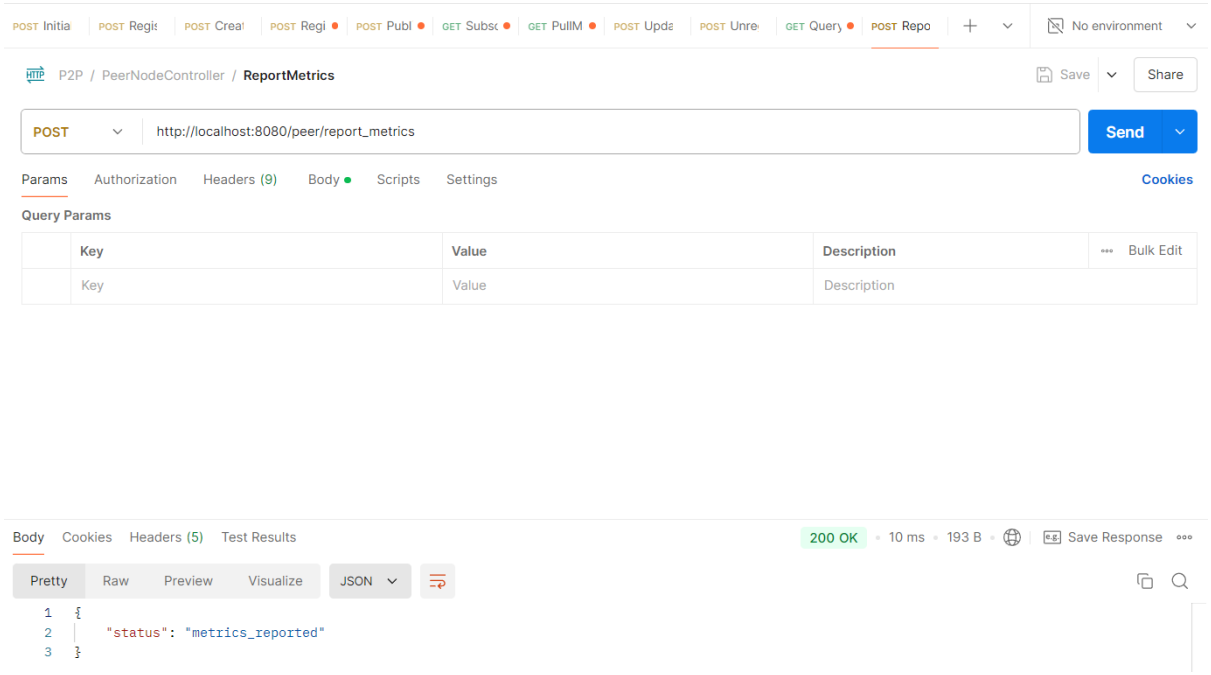
What does the API do?

- Reports performance metrics (e.g., latency, bandwidth) from the peer to the indexing server.

How does it work?

- The peer sends a report to the indexing server, which may store or display the metrics as part of a monitoring system.

Screenshot from postman (Working of the API):



10. Get Event Log of Peer

API Endpoint: /peer/event_log
Method: GET

cURL:

```
curl -X GET "http://localhost:8080/peer/event_log"
```

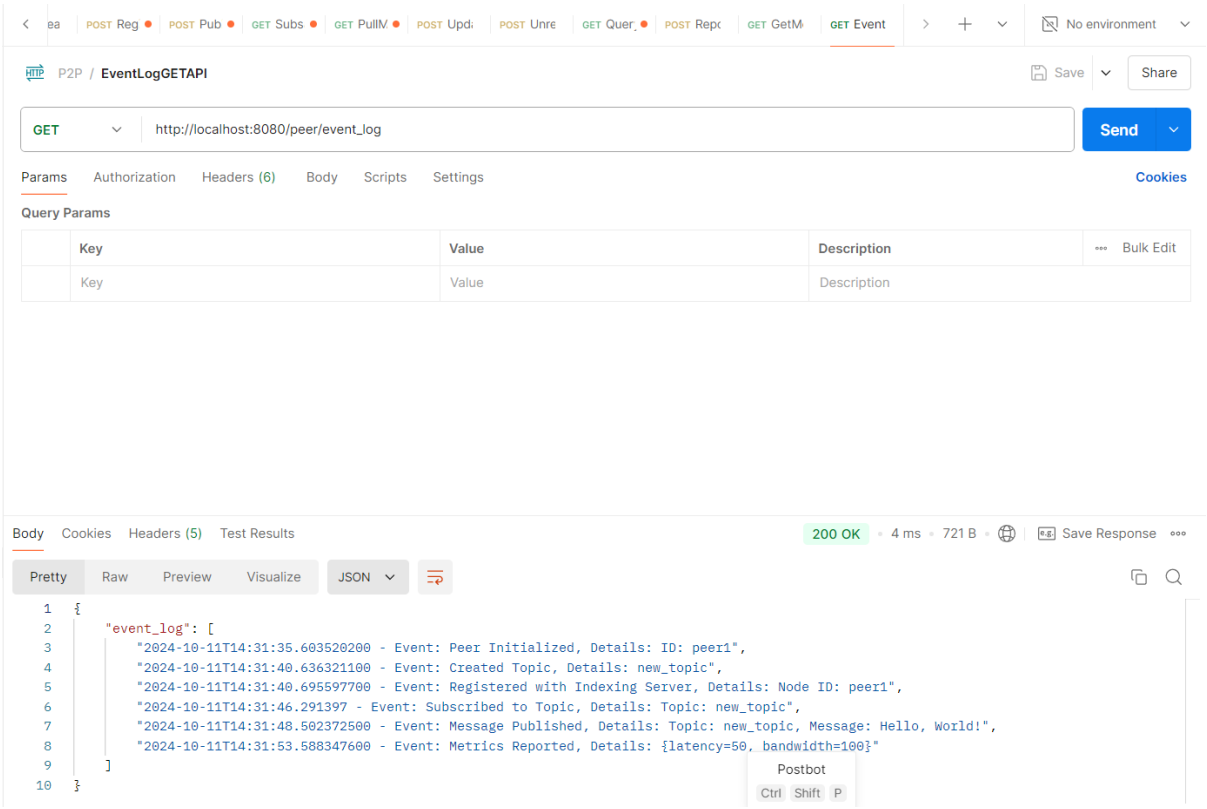
What does the API do?

- Retrieves the event log of the peer node, showing all significant events like initialization, topic creation, subscription, etc.

How does it work?

- The peer node returns its internal event log as a list, providing details of important actions taken since it was initialized.

Screenshot from postman (Working of the API):



11. Get Metrics from Indexing Server

API Endpoint: /indexing/metrics
Method: GET

cURL:

```
curl -X GET "http://localhost:8081/indexing/metrics"
```

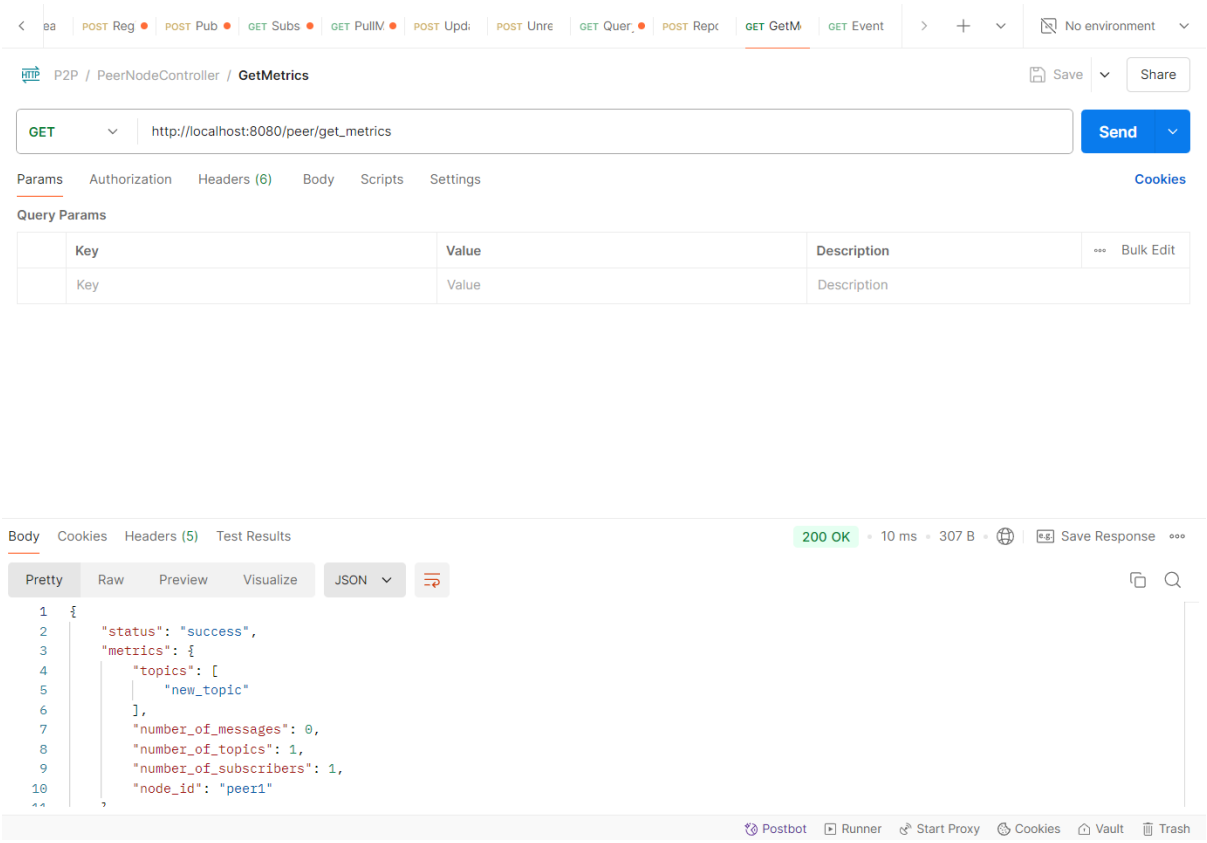
What does the API do?

- Retrieves the performance metrics collected by the indexing server from all peer nodes.

How does it work?

- The indexing server returns a map of metrics reported by each peer, showing data like latency and bandwidth usage for network health monitoring.

Screenshot from postman (Working of the API):



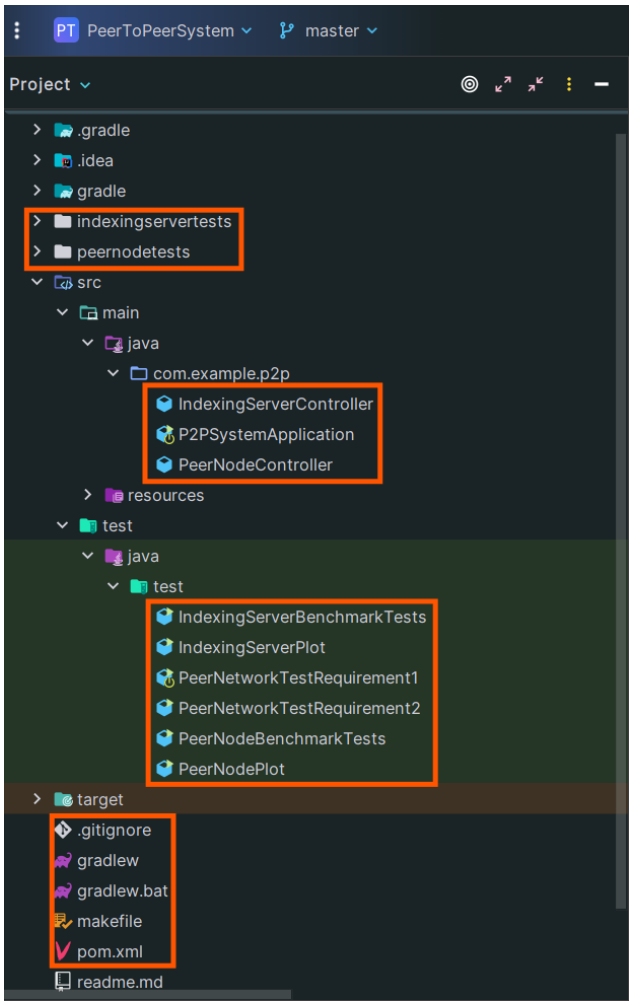
Summary Flow:

1. Initialize the peer node with /peer/initialize.
2. Create topics on the peer node with /peer/create_topic.
3. Register the peer with the indexing server using /peer/register_with_indexing_server.
4. Publish messages to the topics with /peer/publish.
5. Subscribe to topics using /peer/subscribe/{topic}.
6. Pull messages from the topic using /peer/pull_messages/{topic}.
7. Unregister the peer when it shuts down using /indexing/unregister.
8. Query topics from the indexing server using /indexing/query_topic/{topic}.
9. Report metrics using /peer/report_metrics.
10. Monitor events using /peer/event_log and network metrics using /indexing/metrics.

This flow allows peers to interact, share topics, and exchange messages in a decentralized fashion.

The structure of the project files are as follows:

The PeerNodeController and IndexingServerController are central components in a Peer-to-Peer (P2P) system, each with distinct roles. The PeerNodeController handles the operations of individual peer nodes, allowing them to publish messages, subscribe to topics, and communicate with other peers via an indexing server. It facilitates topic creation, message retrieval, and peer-to-peer messaging. Peers are also responsible for reporting metrics and logging significant events, such as topic updates or system shutdowns, while maintaining lists of hosted topics, subscribers, and messages. On the other hand, the IndexingServerController acts as the central registry for the entire P2P network, maintaining a list of all available topics and the peers hosting them. It enables peers to discover which node hosts a particular topic, ensuring efficient routing of requests for subscription or message publishing. This controller orchestrates the discovery and registration of topics across the network, allowing peers to dynamically join and leave while ensuring that topics and their corresponding hosting nodes are kept up to date. Together, these two controllers create a scalable and dynamic environment for peer-to-peer communication.



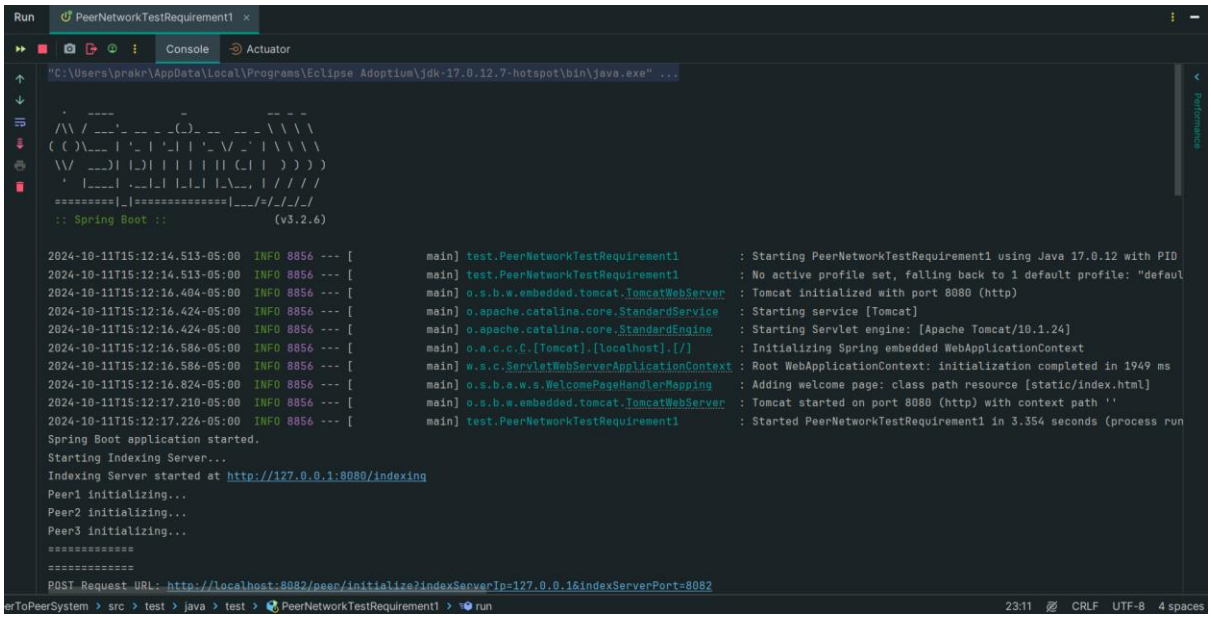
Testing Results

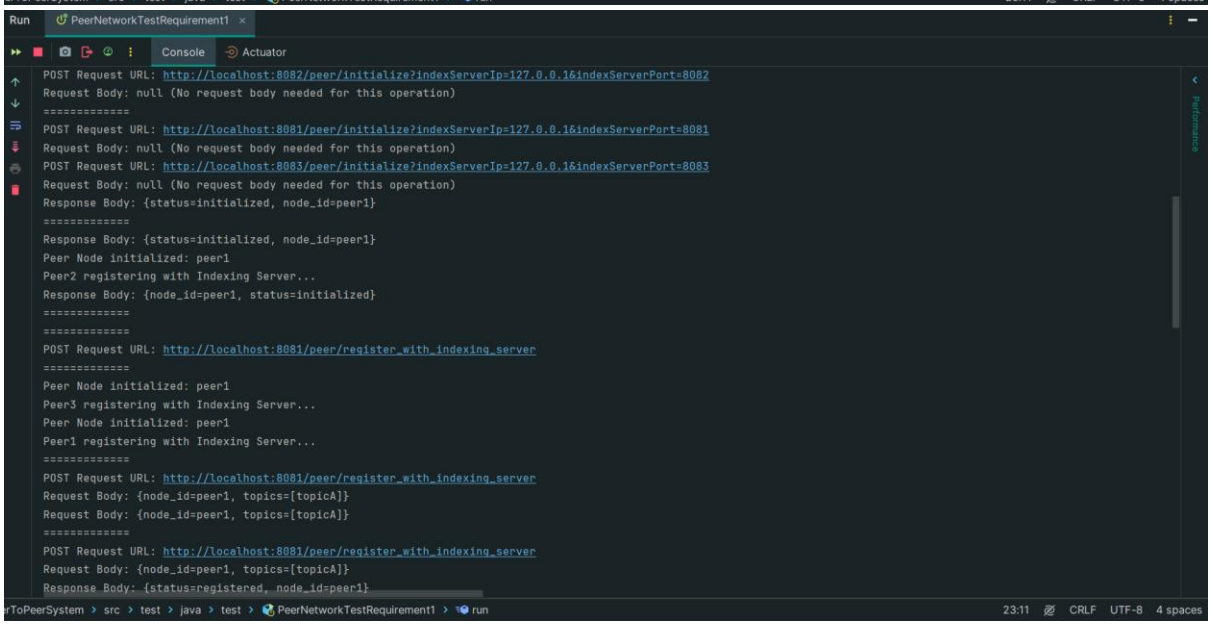
1. Deploying at least 3 peers and 1 indexing server. They can be set up on the same machine or different machines.
Ensure all APIs are working properly.
Ensure multiple peer nodes can simultaneously publish and subscribe to a topic.

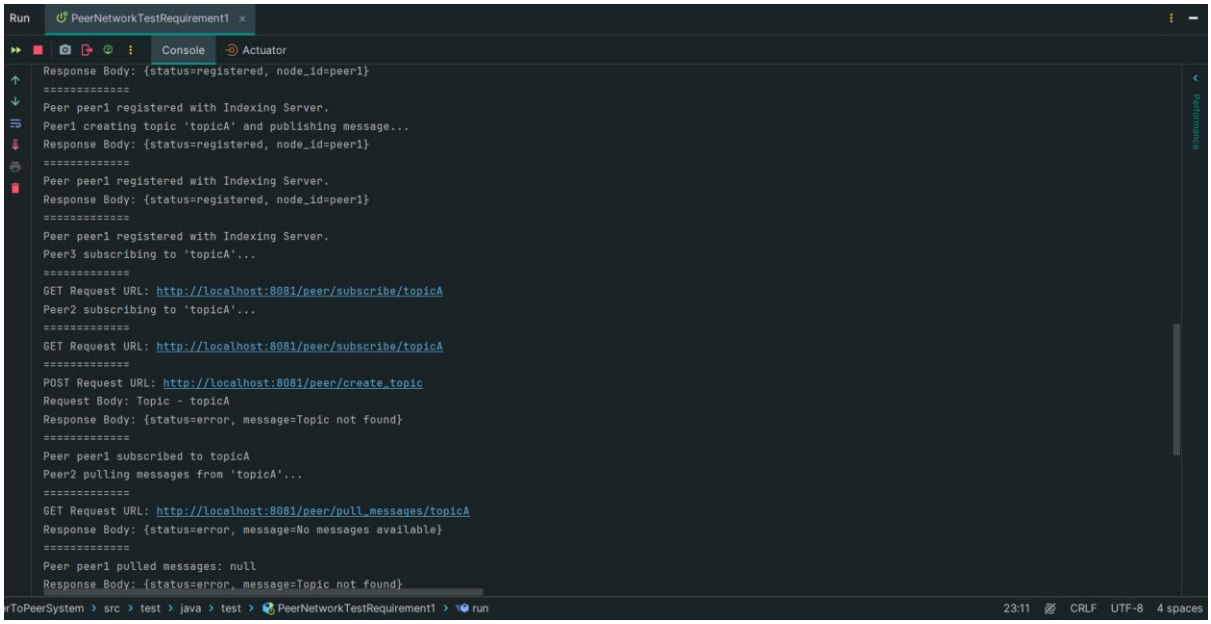
This requirement is fulfilled by the PeerNetworkTestRequirement1 file.

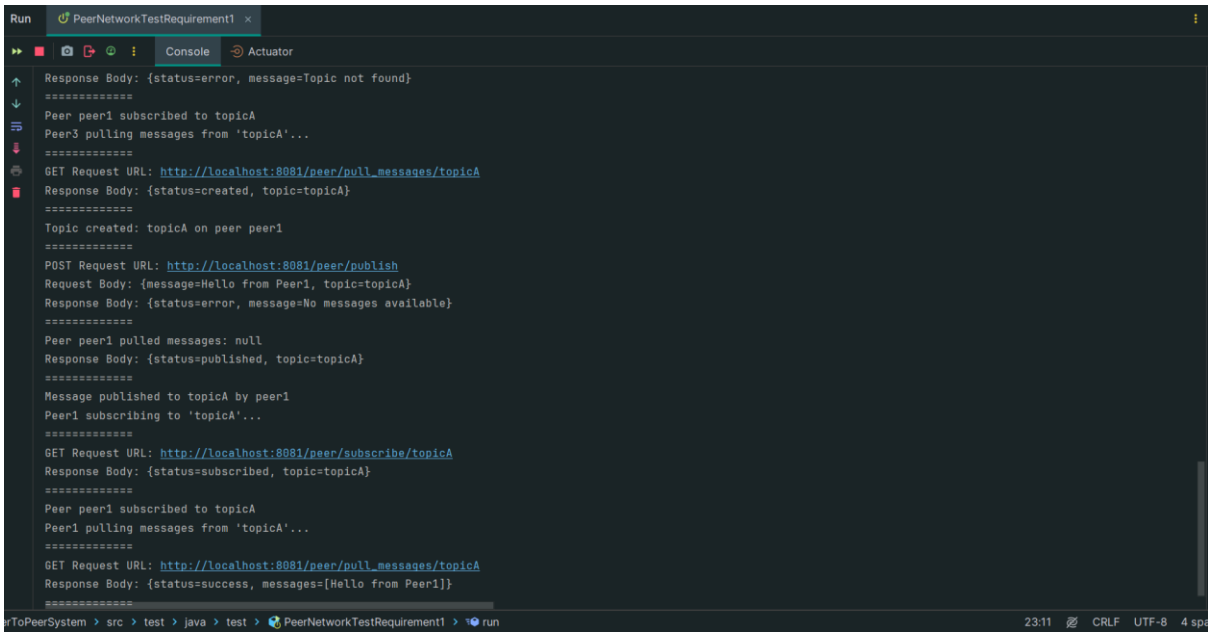
To run this file, the steps will be mentioned in readme.md file.

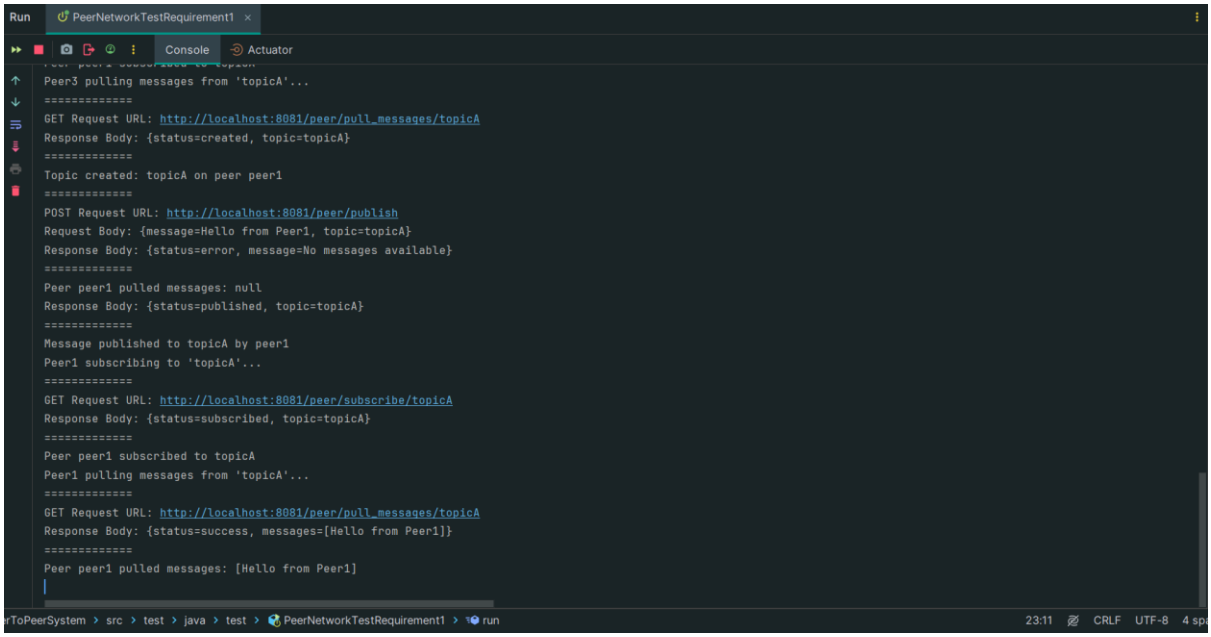
Below is the output of the file.






Run
PeerNetworkTestRequirement1
Console
Actuator
Response Body: {status=registered, node_id=peer1}
Peer peer1 registered with Indexing Server.
Peer1 creating topic 'topicA' and publishing message...
Response Body: {status=registered, node_id=peer1}
Peer peer1 registered with Indexing Server.
Response Body: {status=registered, node_id=peer1}
Peer peer1 registered with Indexing Server.
Peer3 subscribing to 'topicA'...
GET Request URL: http://localhost:8081/peer/subscribe/topicA
Peer2 subscribing to 'topicA'...
GET Request URL: http://localhost:8081/peer/subscribe/topicA
POST Request URL: http://localhost:8081/peer/create_topic
Request Body: Topic - topicA
Response Body: {status=error, message=Topic not found}
Peer peer1 subscribed to topicA
Peer2 pulling messages from 'topicA'...
GET Request URL: http://localhost:8081/peer/pull_messages/topicA
Response Body: {status=error, message=No messages available}
Peer peer1 pulled messages: null
Response Body: {status=error, message=Topic not found}
src
test
java
test
PeerNetworkTestRequirement1
run
23:11
CRLF
UTF-8
4 spaces


Run
PeerNetworkTestRequirement1
Console
Actuator
Response Body: {status=error, message=Topic not found}
Peer peer1 subscribed to topicA
Peer3 pulling messages from 'topicA'...
GET Request URL: http://localhost:8081/peer/pull_messages/topicA
Response Body: {status=created, topic=topicA}
Topic created: topicA on peer peer1
POST Request URL: http://localhost:8081/peer/publish
Request Body: {message=Hello from Peer1, topic=topicA}
Response Body: {status=error, message=No messages available}
Peer peer1 pulled messages: null
Response Body: {status=published, topic=topicA}
Message published to topicA by peer1
Peer1 subscribing to 'topicA'...
GET Request URL: http://localhost:8081/peer/subscribe/topicA
Response Body: {status=subscribed, topic=topicA}
Peer peer1 subscribed to topicA
Peer1 pulling messages from 'topicA'...
GET Request URL: http://localhost:8081/peer/pull_messages/topicA
Response Body: {status=success, messages=[Hello from Peer1]}
src
test
java
test
PeerNetworkTestRequirement1
run
23:11
CRLF
UTF-8



2.Measuring the average response time when multiple peer nodes are concurrently querying topics from the indexing server node.

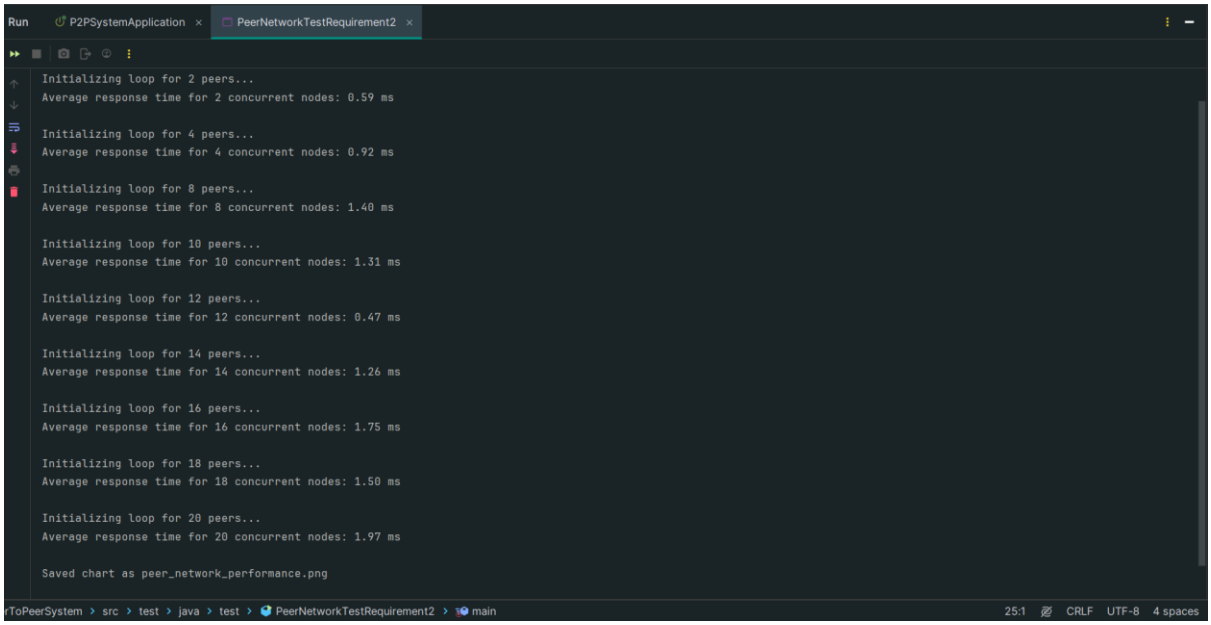
a. Varying the number of concurrent peer nodes (N) and observe how the average response time changes (different peer nodes: 2, 4, 8)
b. The number of requests per node is a fixed number, such as 1000.
c. You need to tweak the indexing server so it holds at least 1 million topics information (you may change this number if needed)

Graph your results

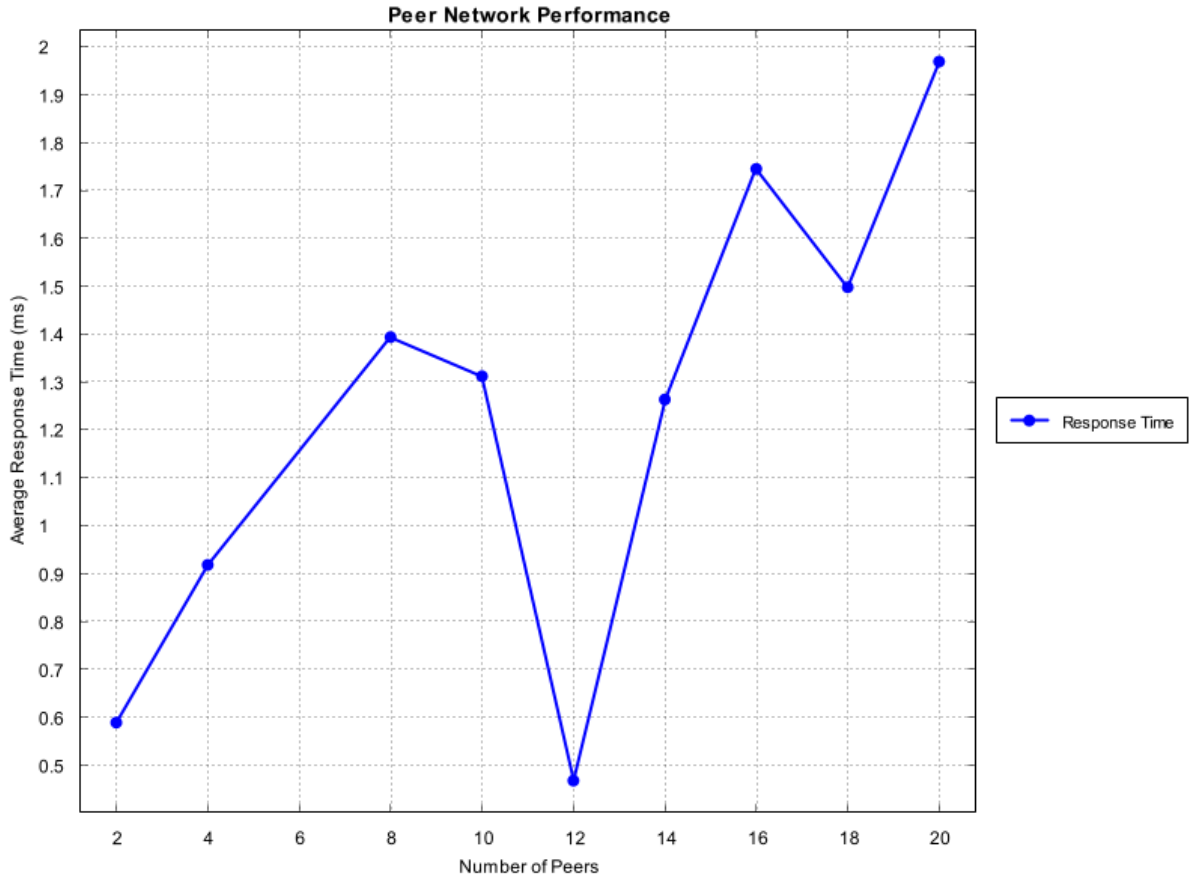
For this evaluation, I have used the following scaled down configuration so that the program does not take much time to execute as it was taking 3 hours on my local.

// Constants for indexing server URL, number of requests per node, and configuration for concurrent nodes and topics
private static final String INDEXING_SERVER_URL = "http://localhost:8080/indexing"; // Base URL for the indexing server
private static final int REQUESTS_PER_NODE = 1000; // Number of requests each peer node will make
private static final int[] CONCURRENT_NODES = {2, 4, 8, 10, 12, 14, 16, 18, 20}; // Array of peer node counts to test with
private static final int NUM_TOPICS = 1000; // Number of topics to be pre-registered in the indexing server

Below is the output for the evaluation:



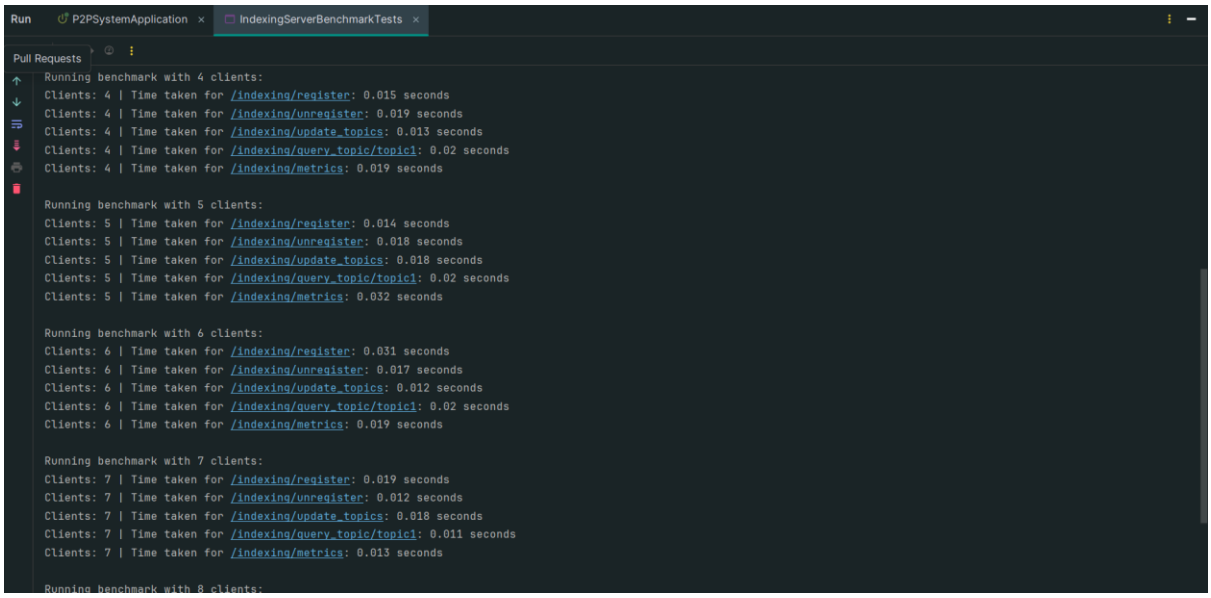
The graph of the above output is as follows:

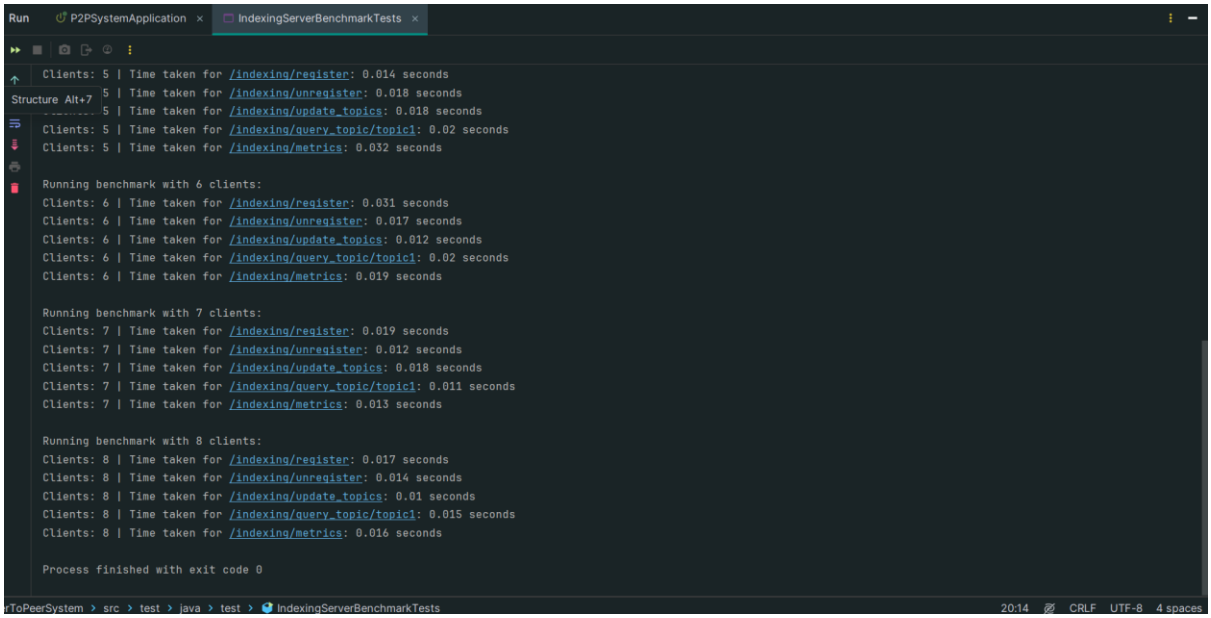


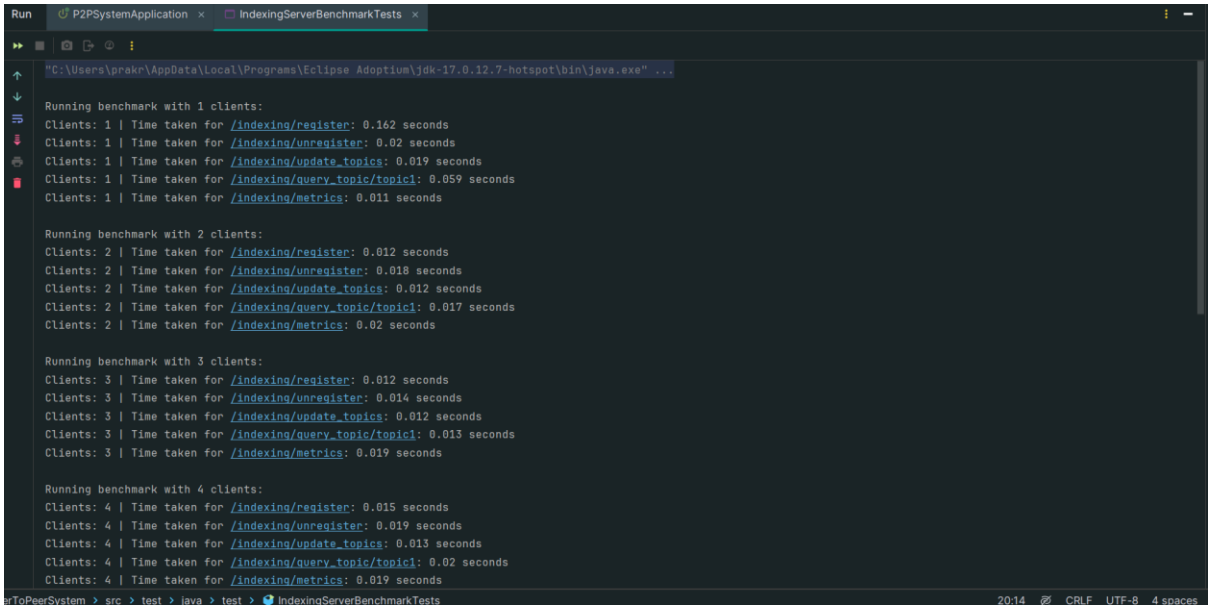
3. Similar to PA1, in PA2 you need to benchmark the latency and throughput of each API.
    a. Starting with 1 peer and 1 indexing server. Benchmark each API.
    b. Increase the number of peers to at least 8. Repeat step a.
    c. Graph your results

For this evaluation, the code is contained by the file IndexingServerBenchmarkTests and the PeerNodeBenchmarkTests. The graphs are plotted by IndexingServerPlot and PeerNodePlot respectively. Below are the screenshots of the running file and below that the graph outputs are pasted for latency and throughput of each API.

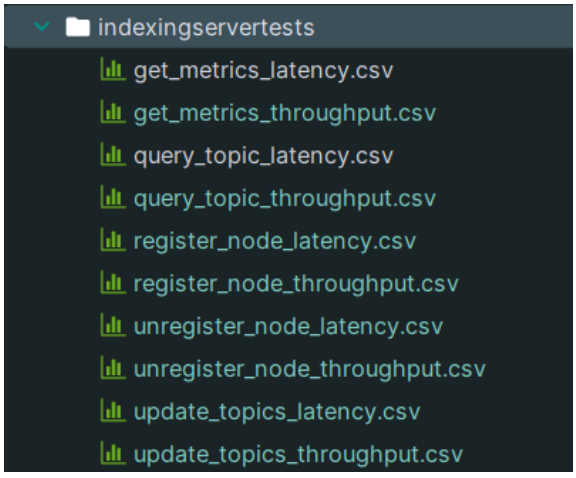
IndexingServerBenchmarkTests.java:





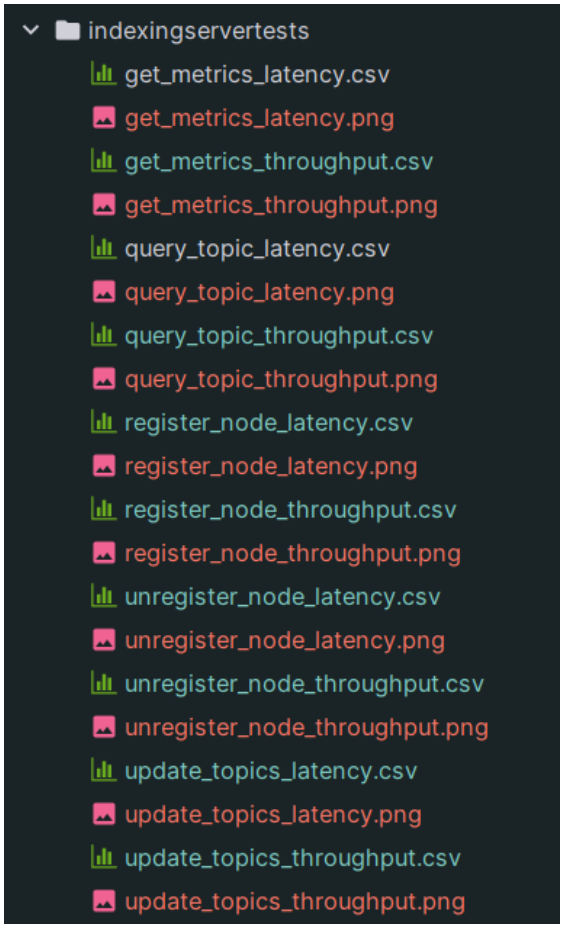


The outputs are saved in csv files located at /indexingservertests :

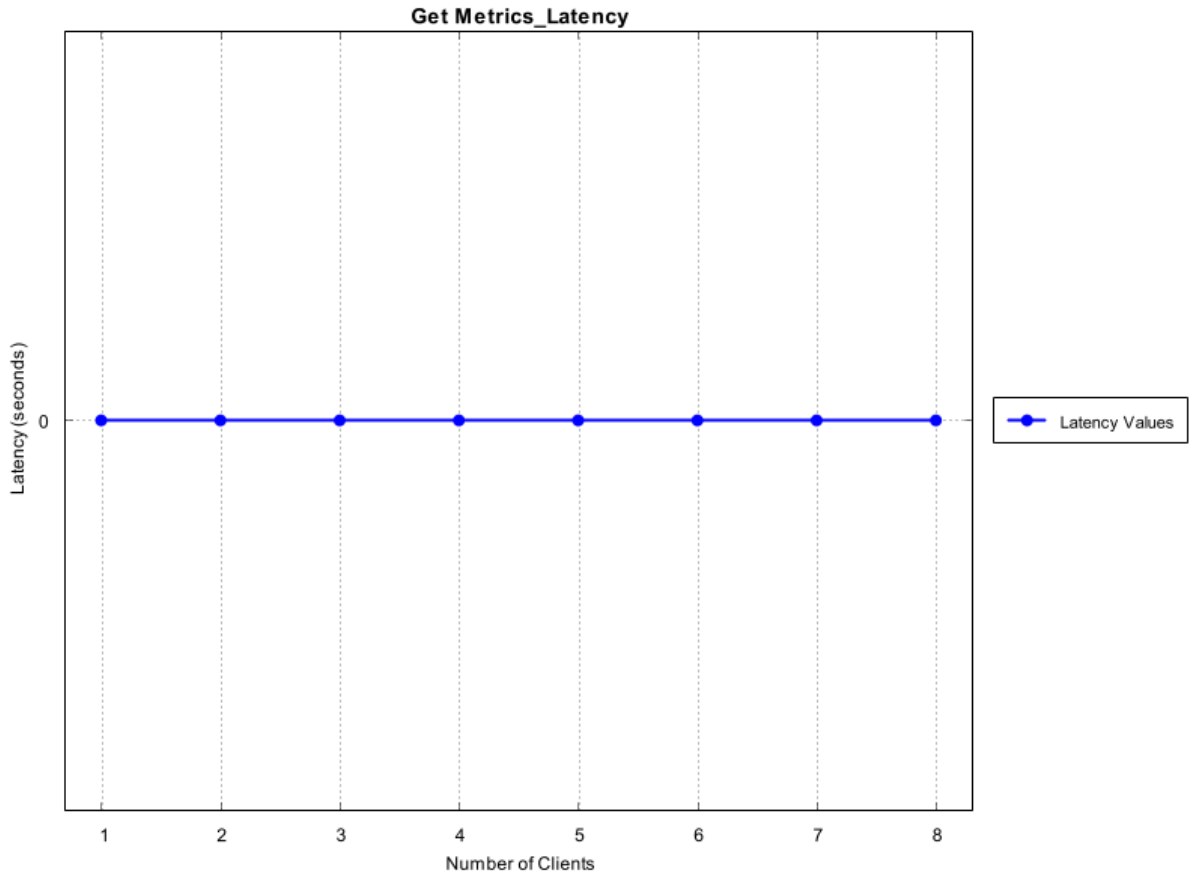


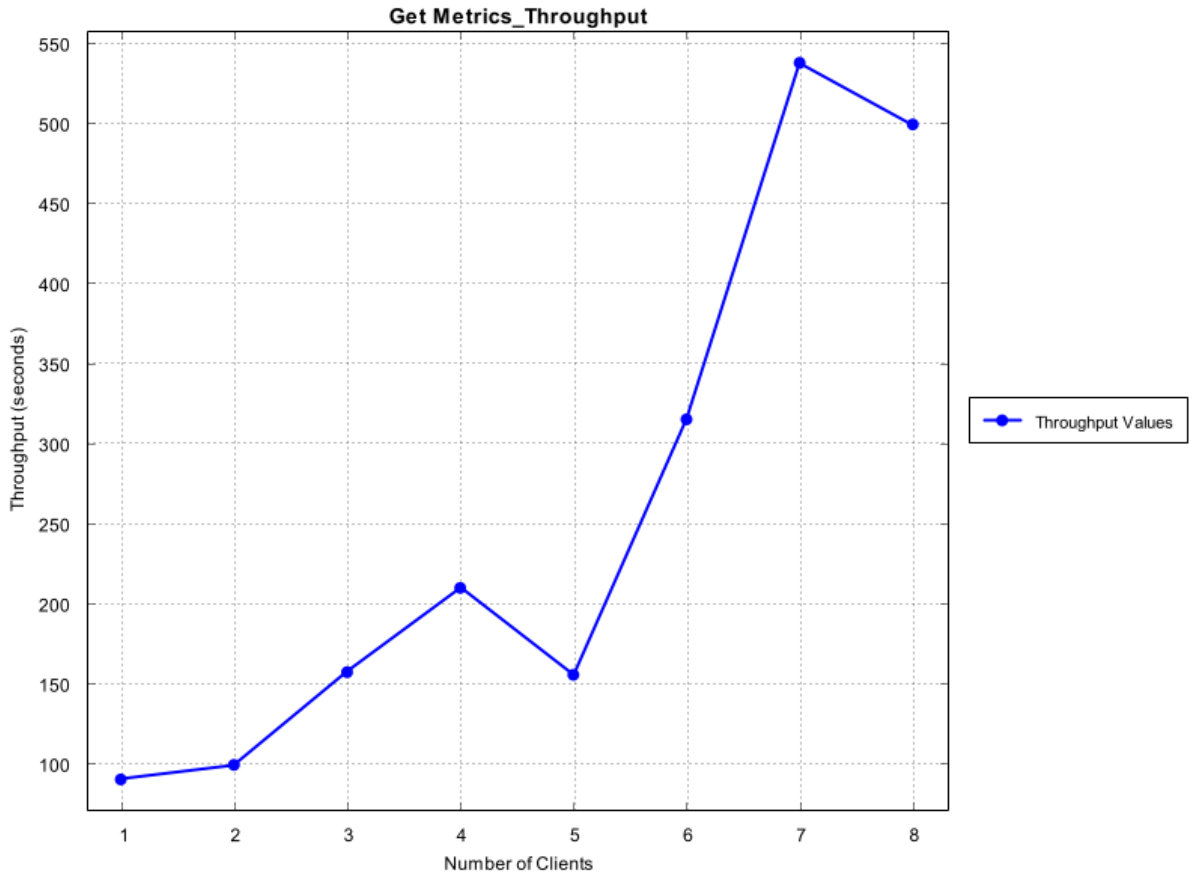
IndexingServerPlot.java:

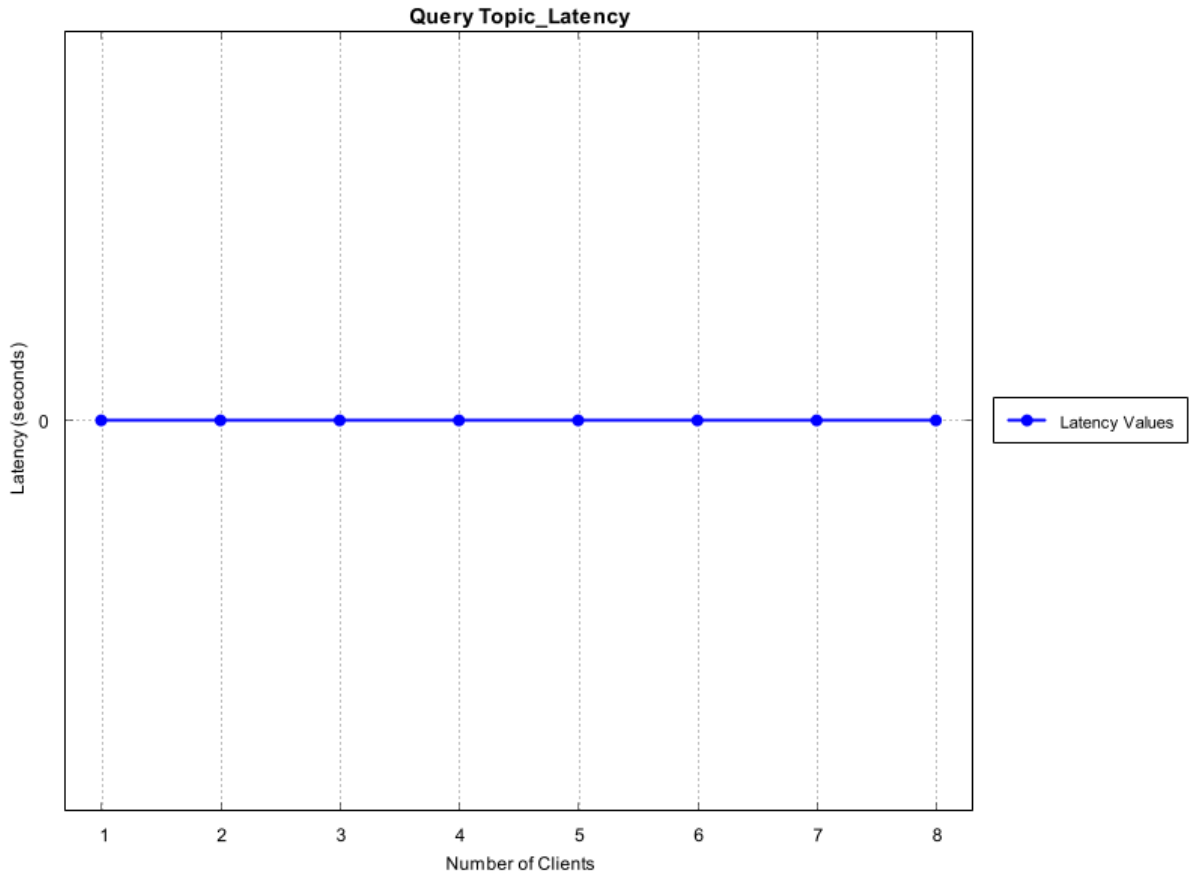
Saved images of graphs:

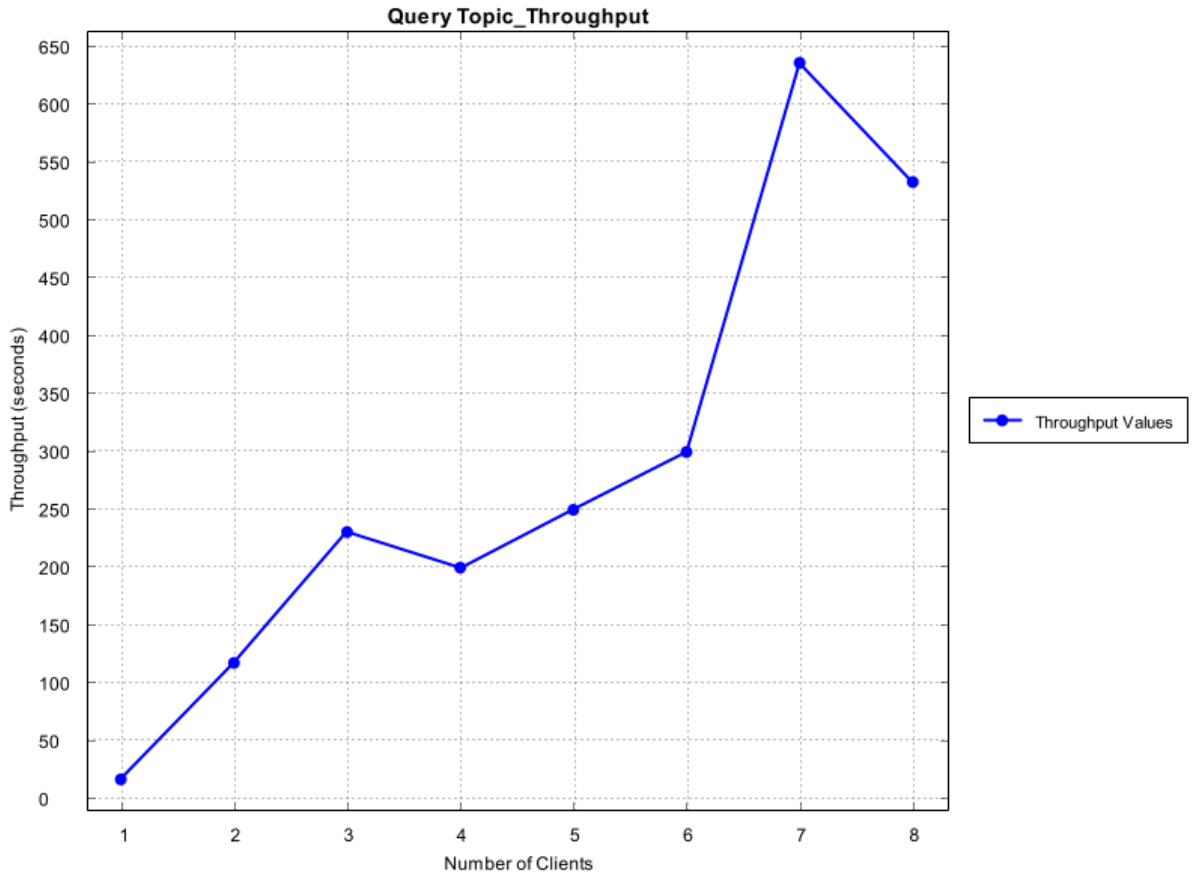


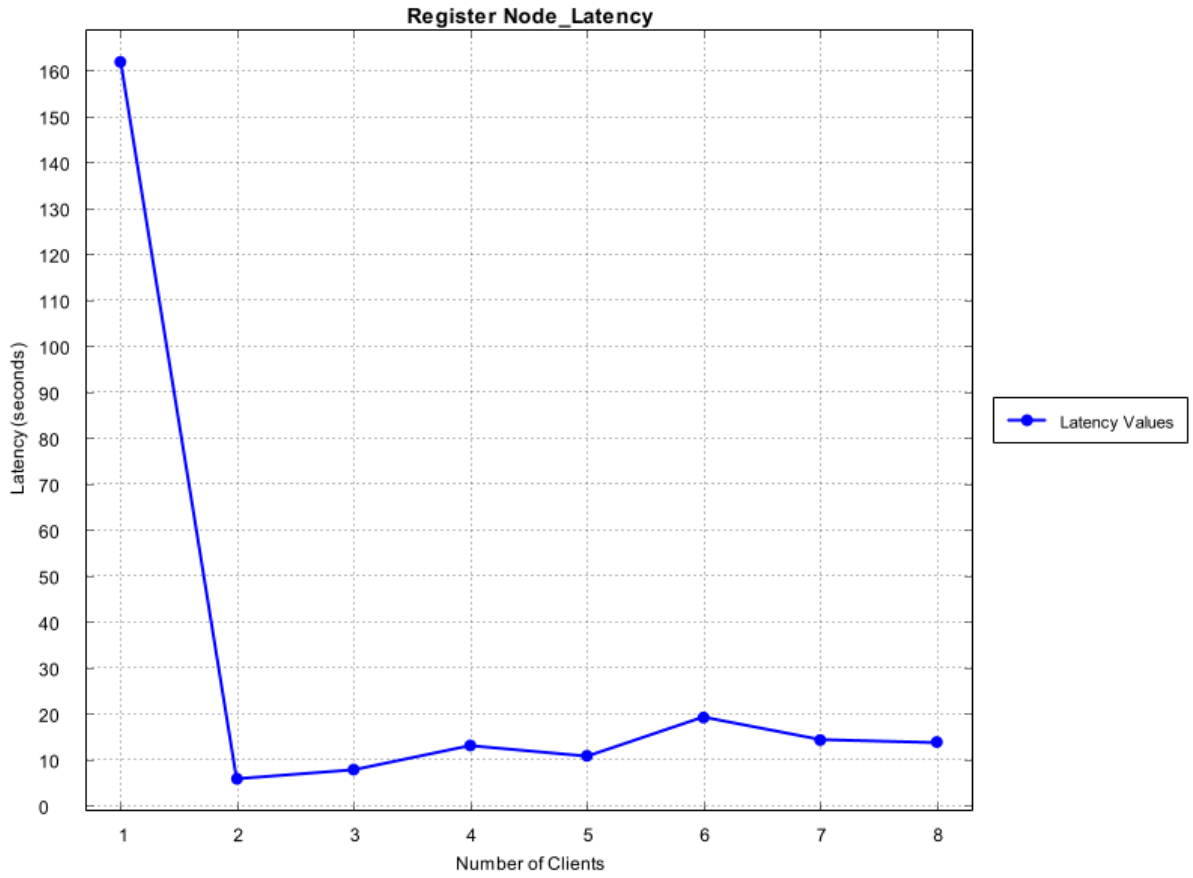
Pasted graph images:

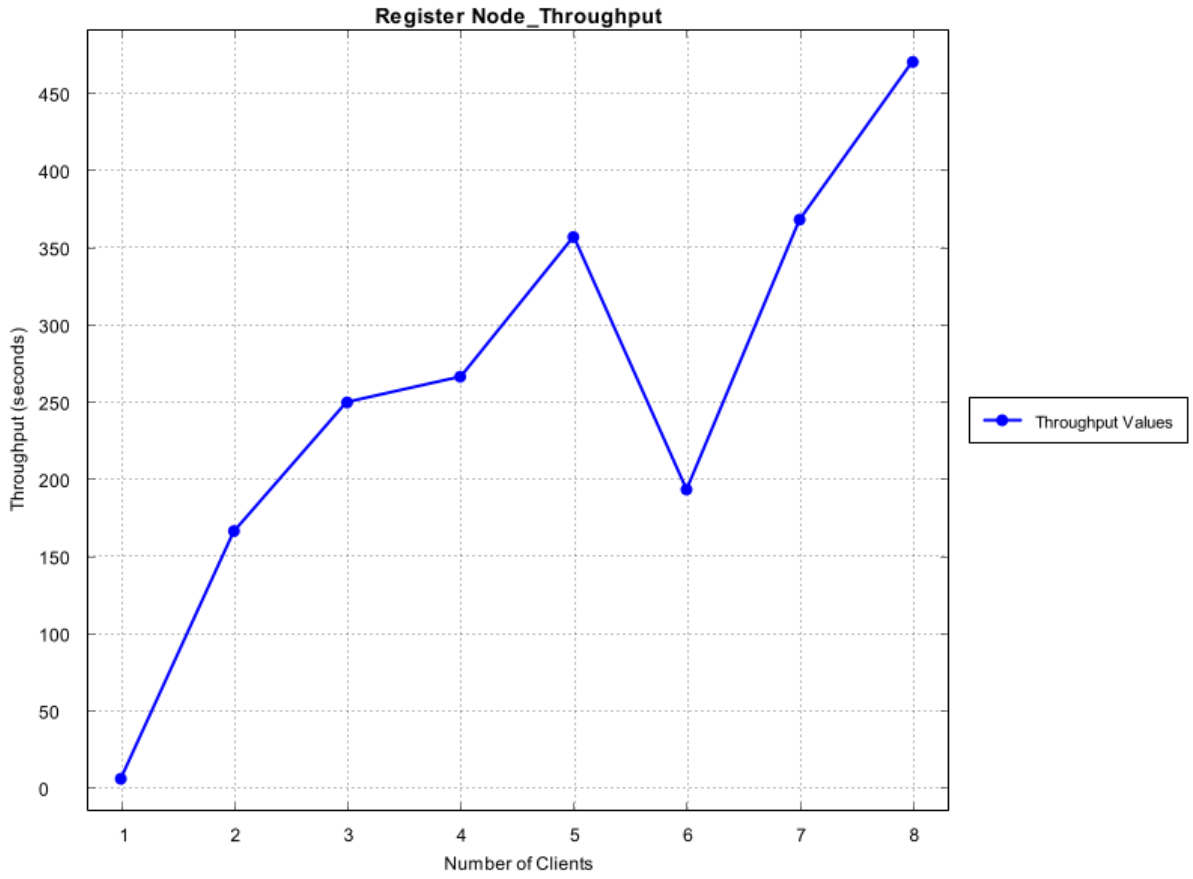

Get Metrics_Latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
Latency Values

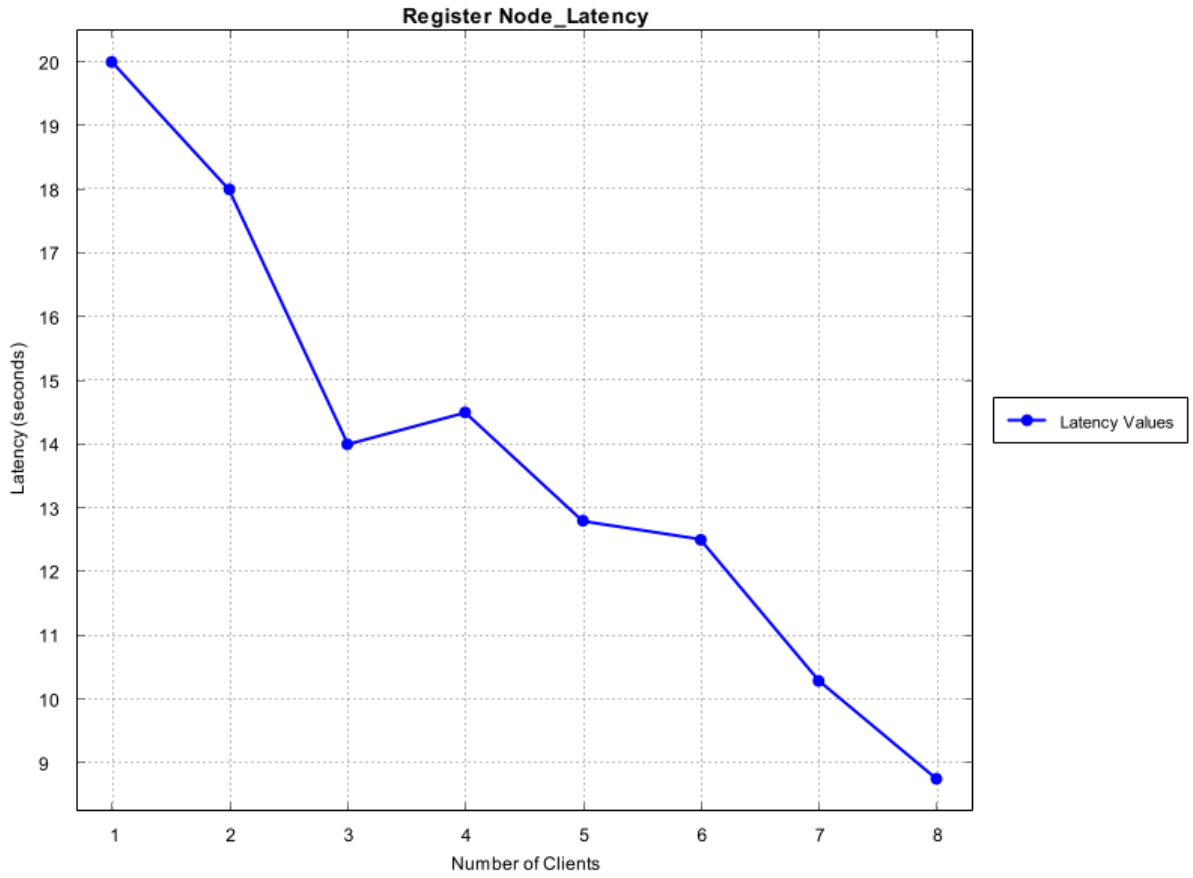

Get Metrics_Throughput
Throughput (seconds)
550
500
450
400
350
300
250
200
150
100
1
2
3
4
5
6
7
8
Number of Clients
Throughput Values

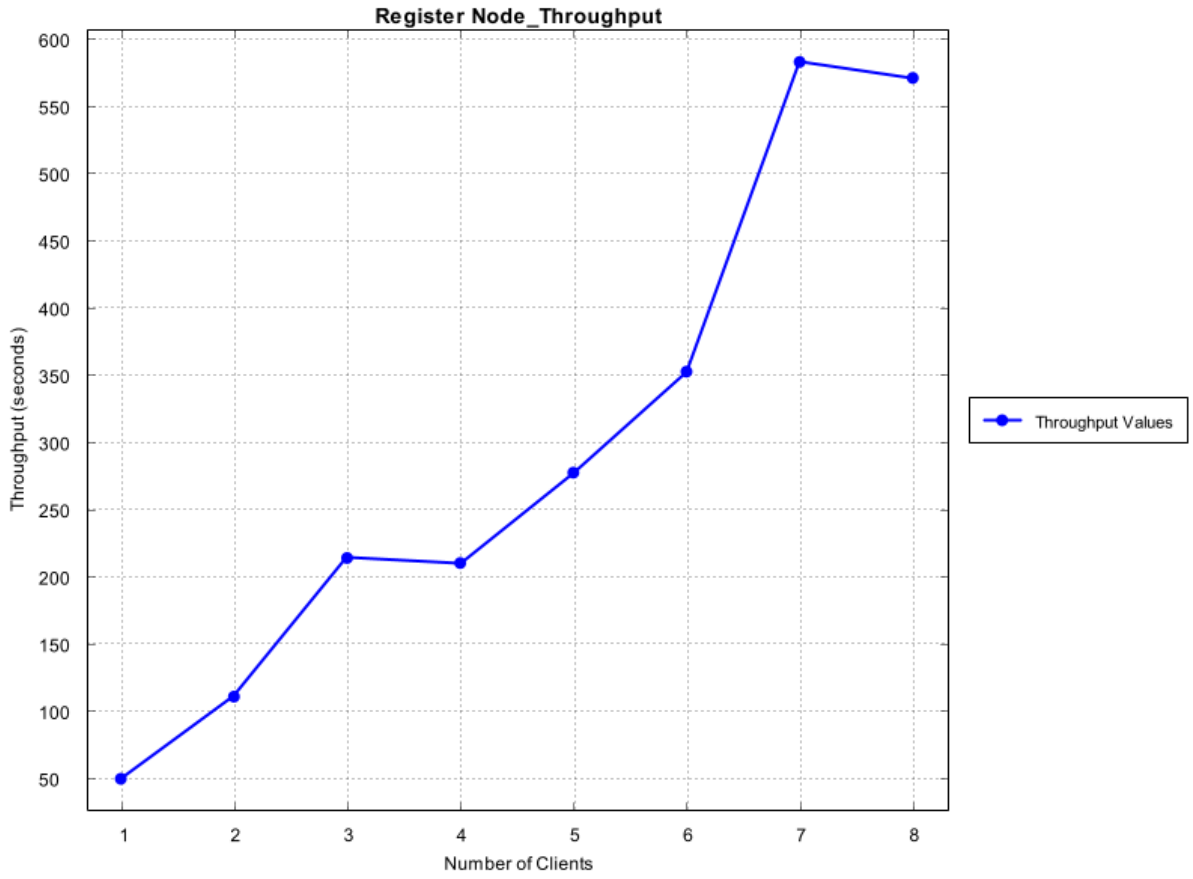

Query Topic_Latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
Latency Values

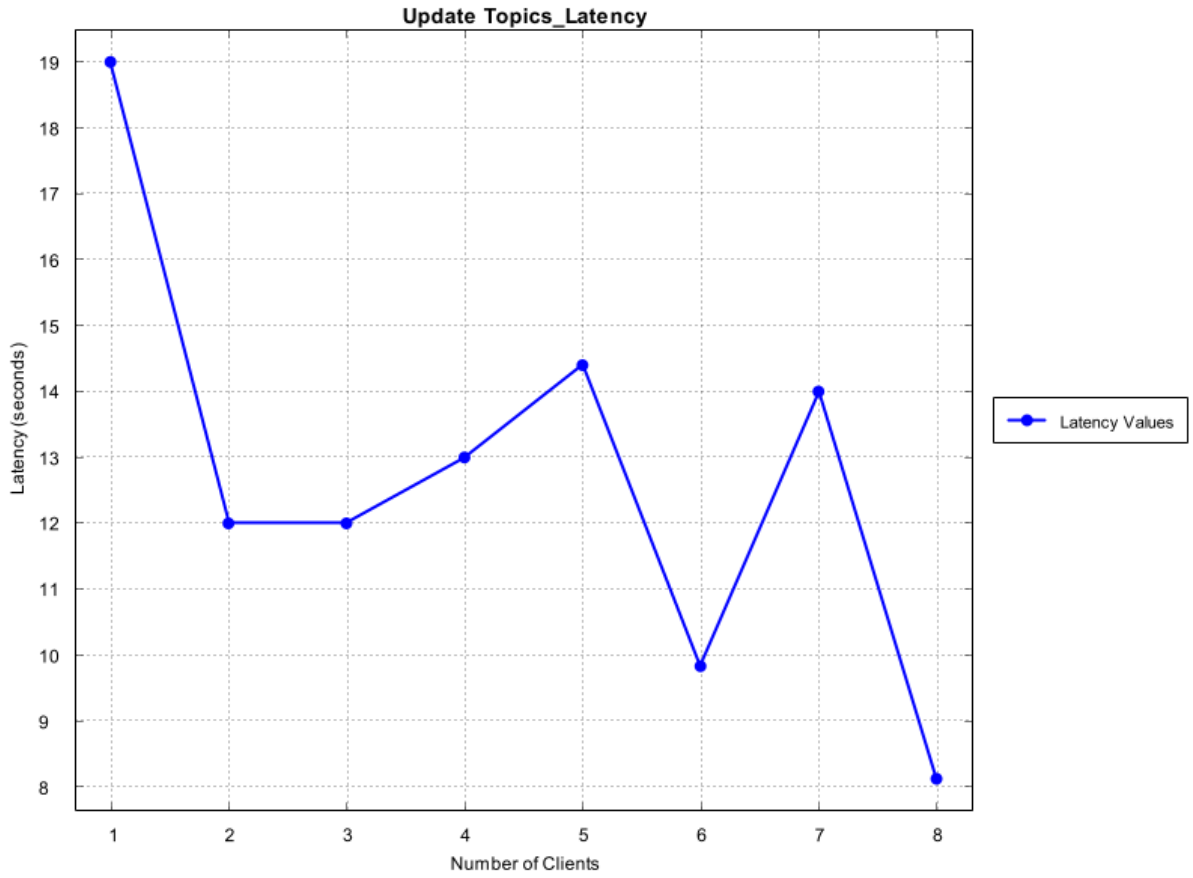

Query Topic_Throughput
Throughput (seconds)
650
600
550
500
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
Number of Clients
Throughput Values

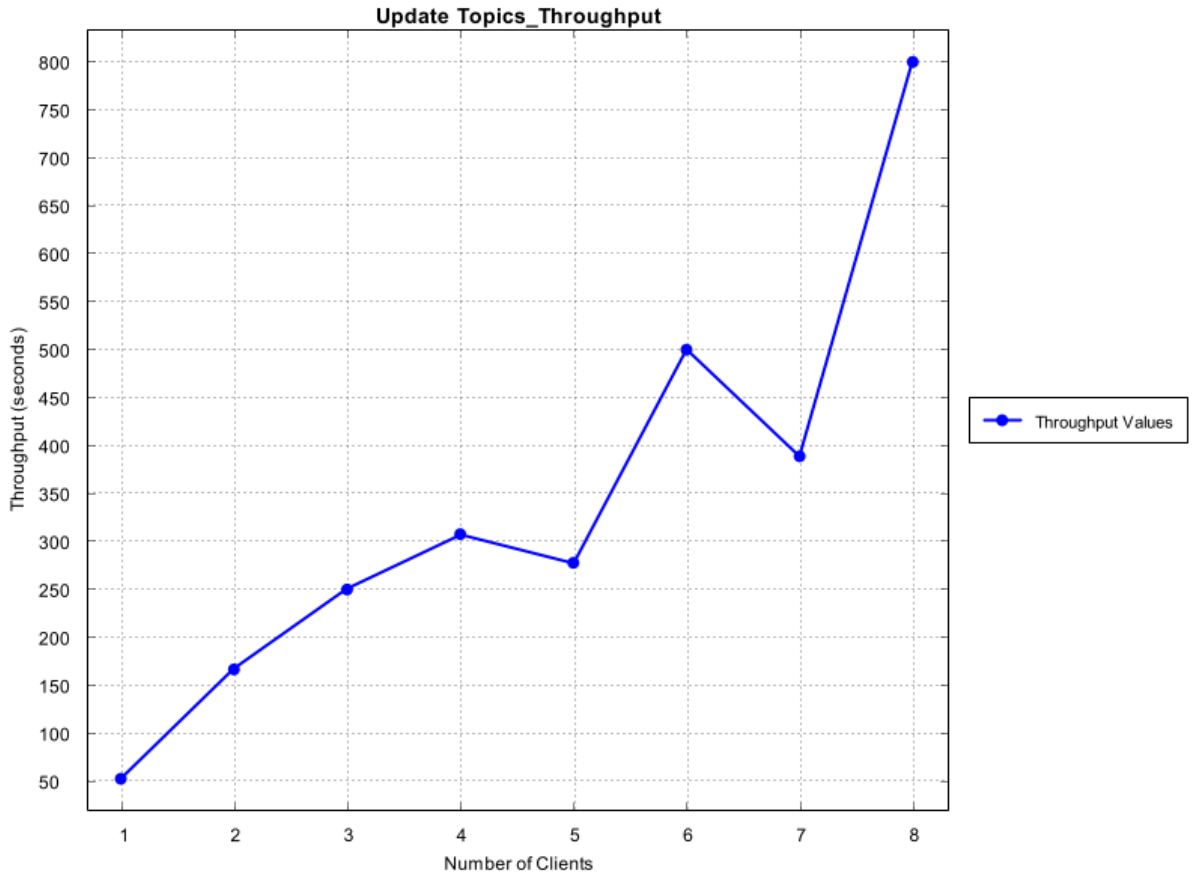

Register Node_Latency
Latency (seconds)
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
1
2
3
4
5
6
7
8
Number of Clients
Latency Values


Register Node_Throughput
Throughput (seconds)
0
50
100
150
200
250
300
350
400
450
1
2
3
4
5
6
7
8
Number of Clients
Throughput Values

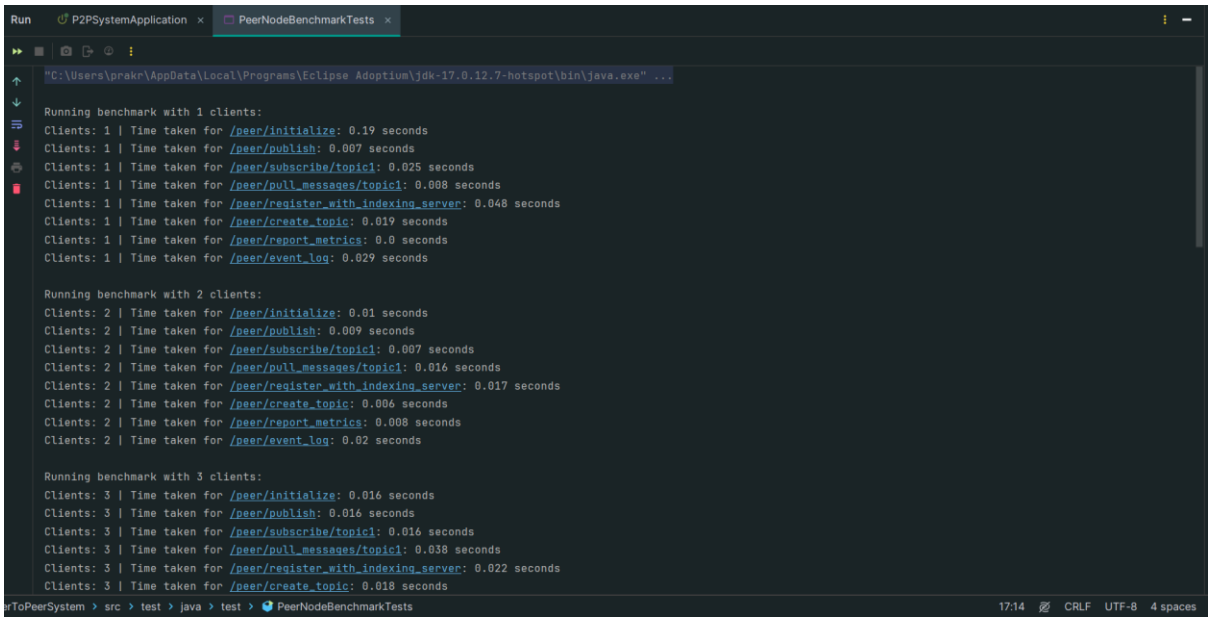

Register Node_Latency
Latency (seconds)
20
19
18
17
16
15
14
13
12
11
10
9
1
2
3
4
5
6
7
8
Number of Clients
Latency Values

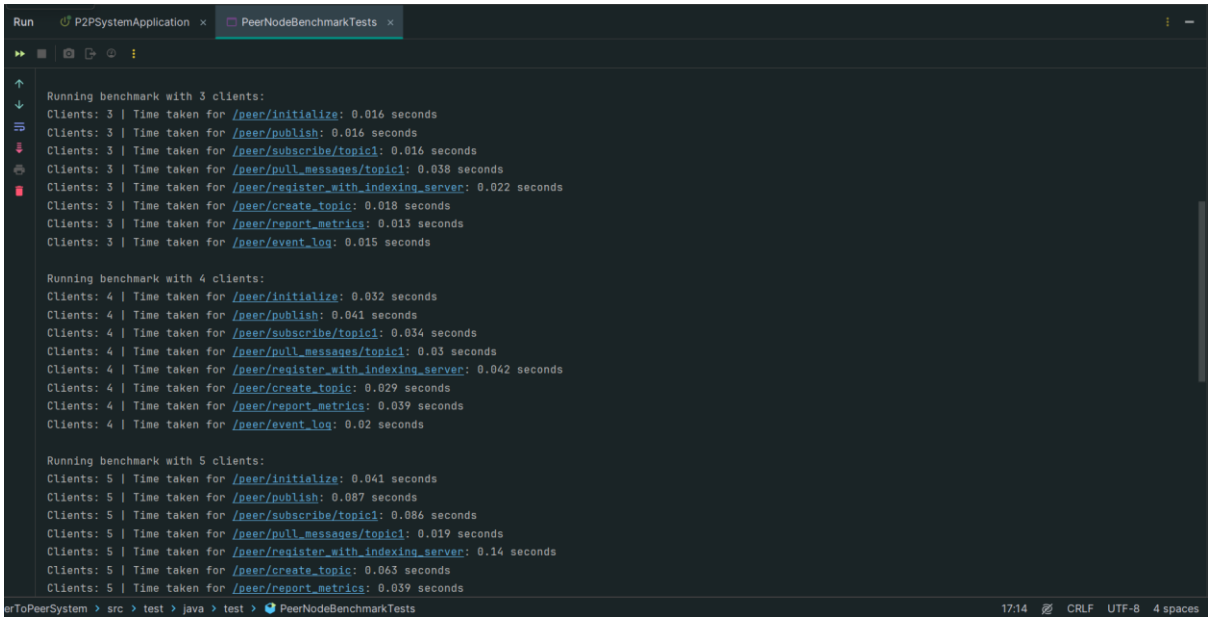

Register Node_Throughput
Throughput (seconds)
600
550
500
450
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8
Number of Clients
Throughput Values

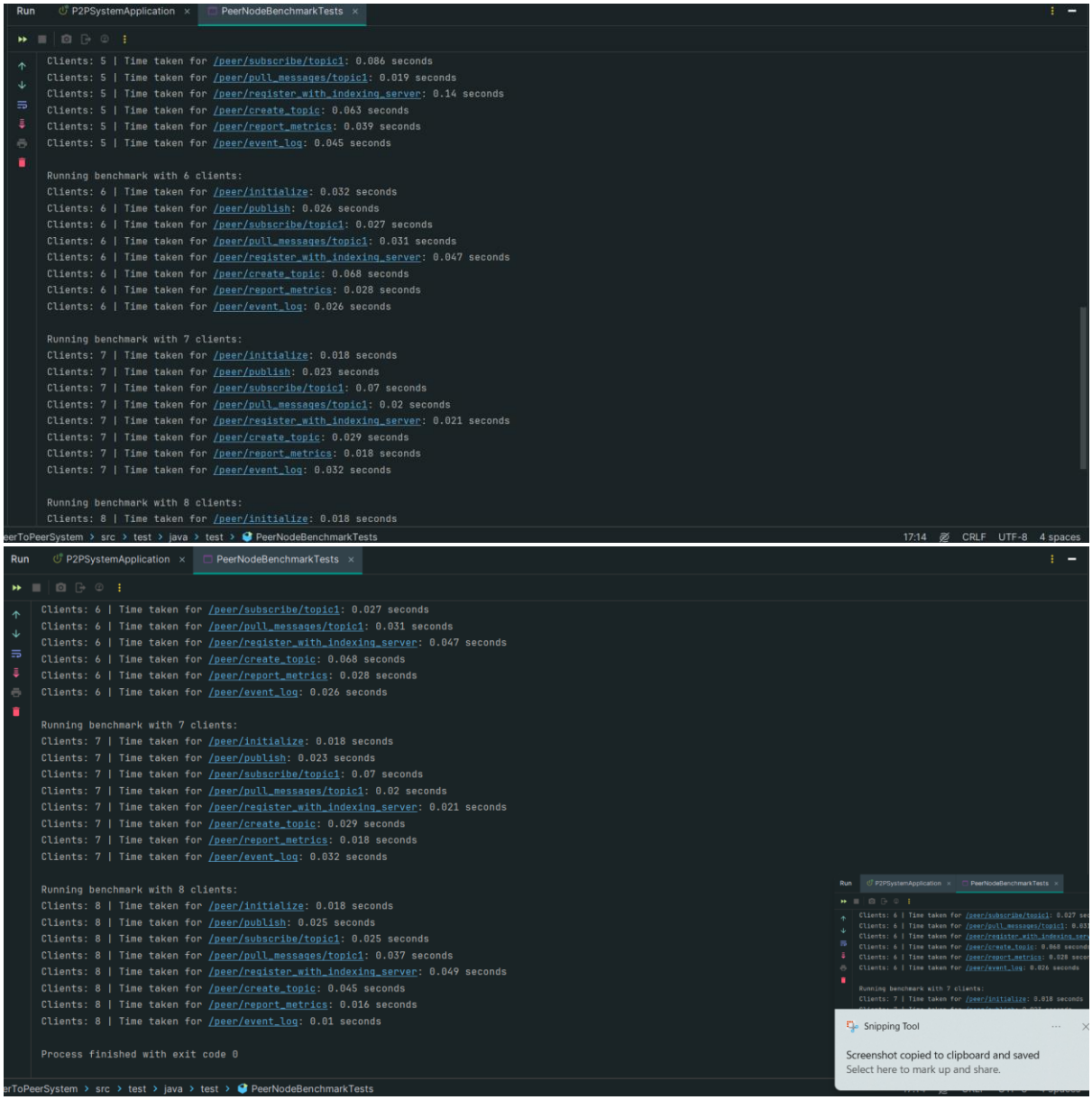

Update Topics_Latency
Latency (seconds)
Number of Clients
Latency Values
19
18
17
16
15
14
13
12
11
10
9
8
1
2
3
4
5
6
7
8


Update Topics_Throughput
Throughput (seconds)
Number of Clients
Throughput Values
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8

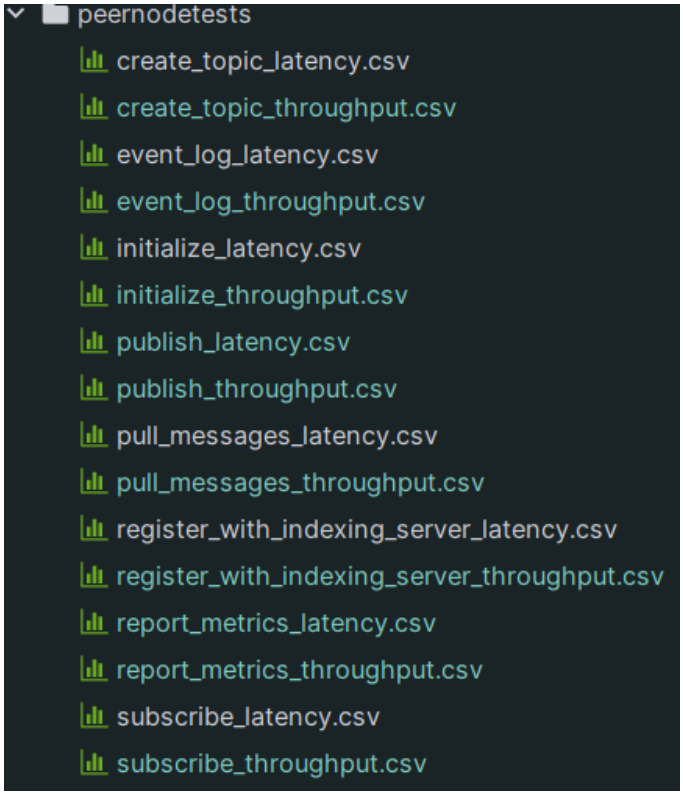
PeerNodeBenchmarkTests.java:



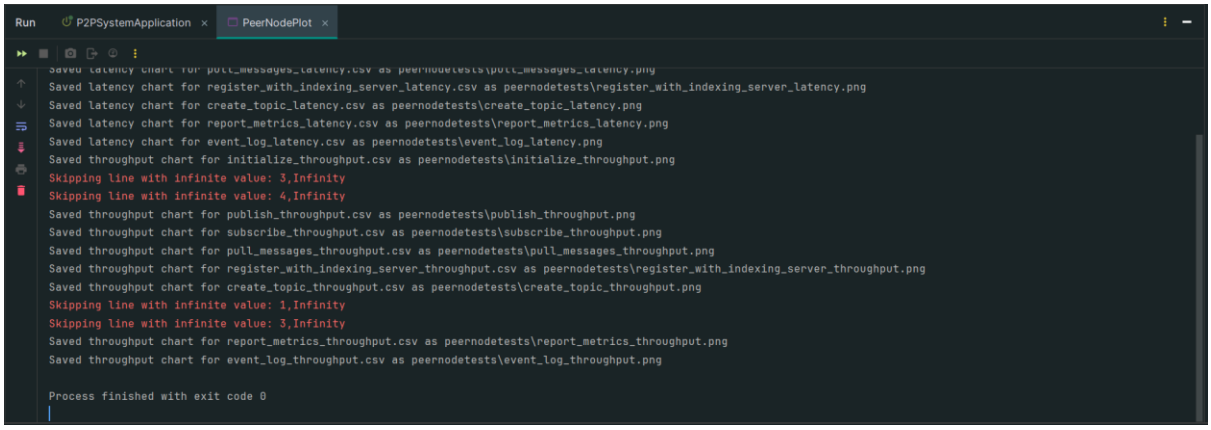




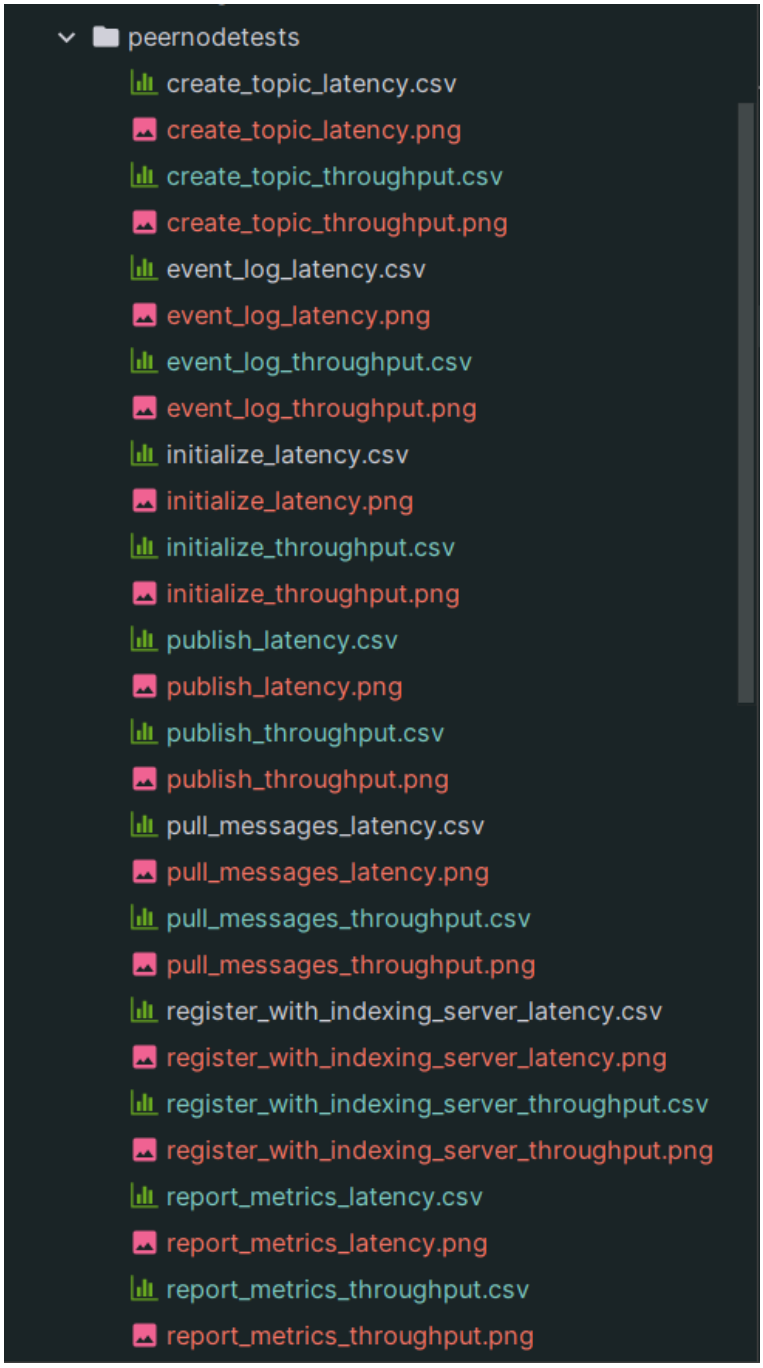
The outputs are saved in csv files located at /peernodetests :



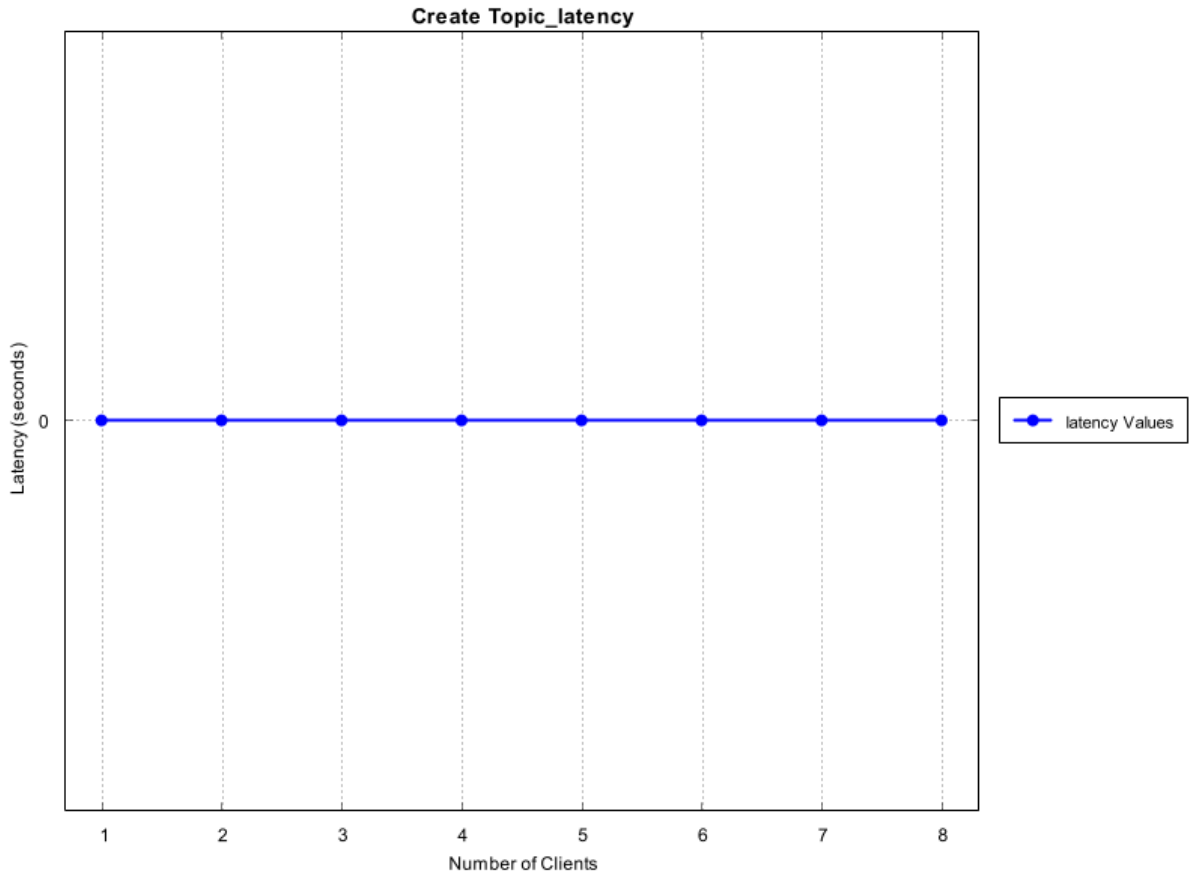
PeerNodePlot.java:

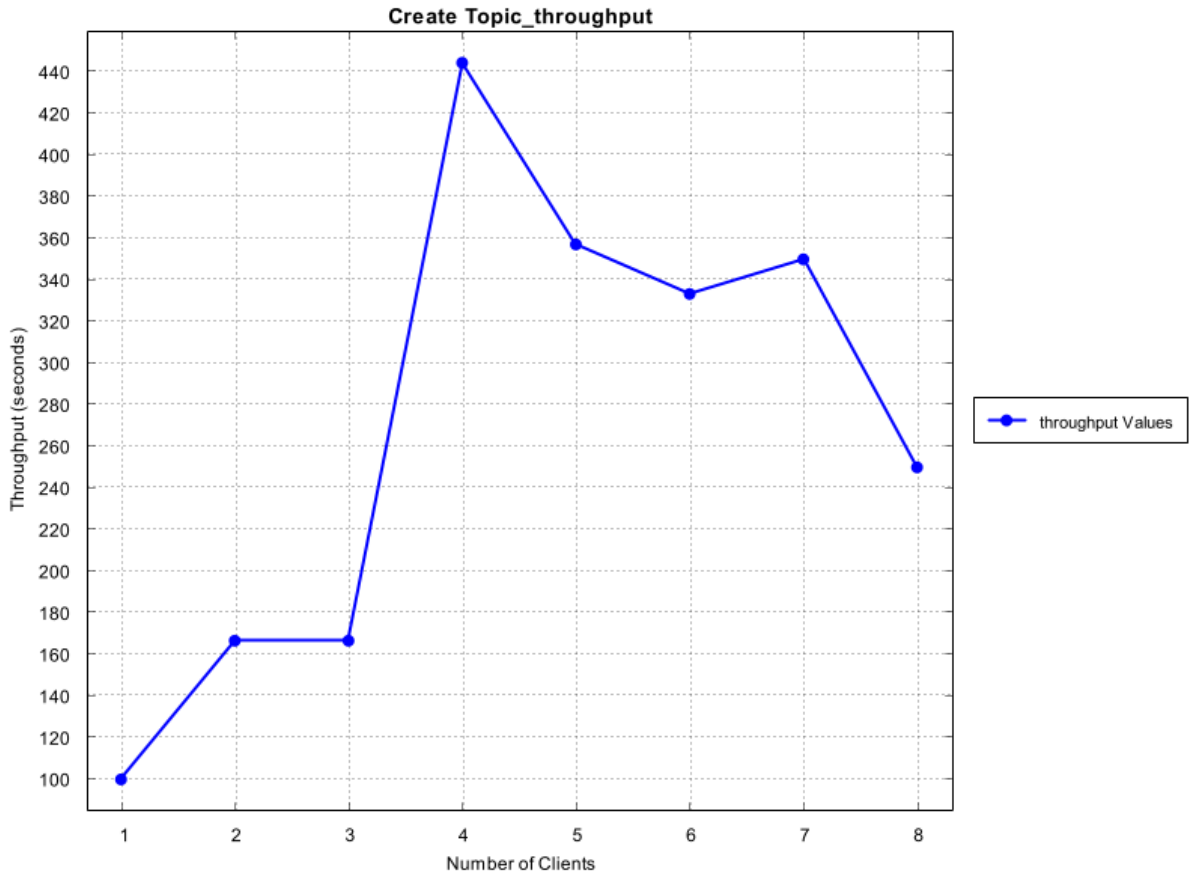


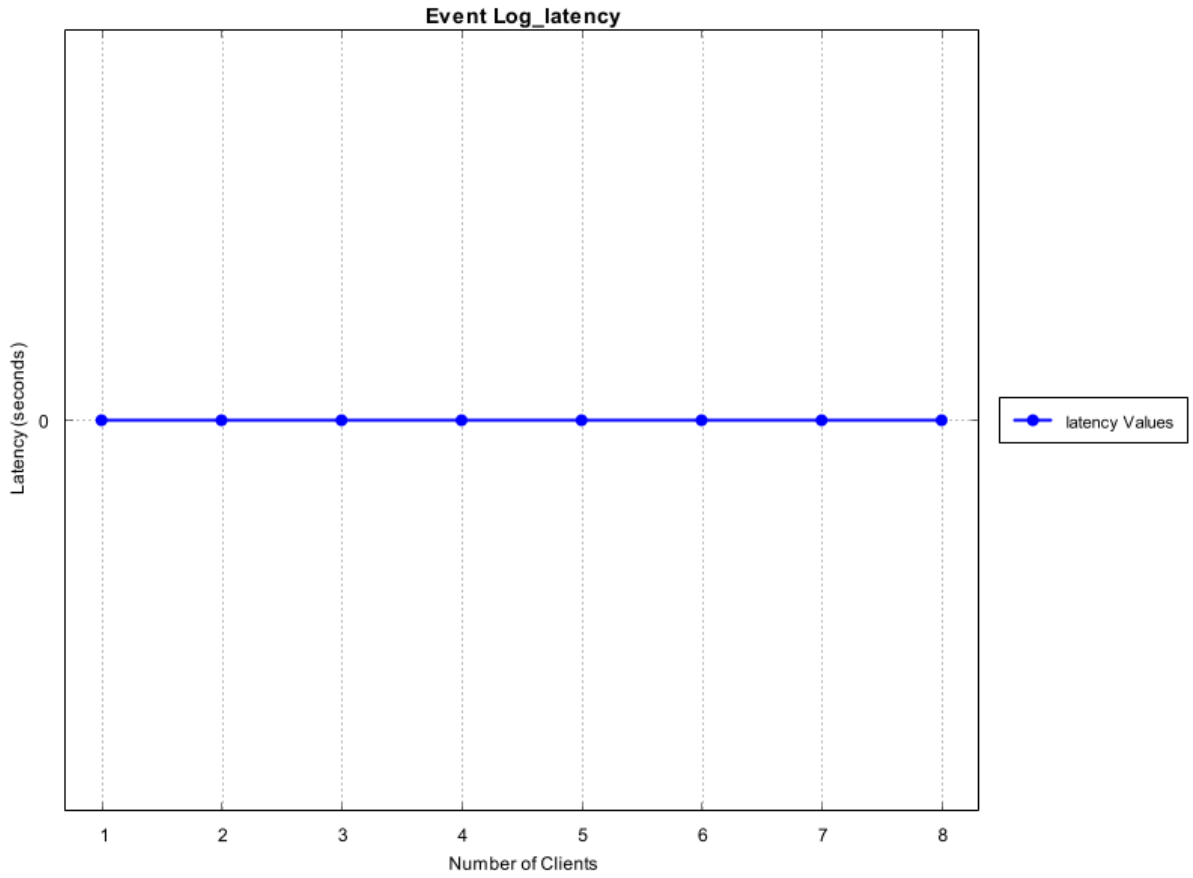
Saved images of graphs:

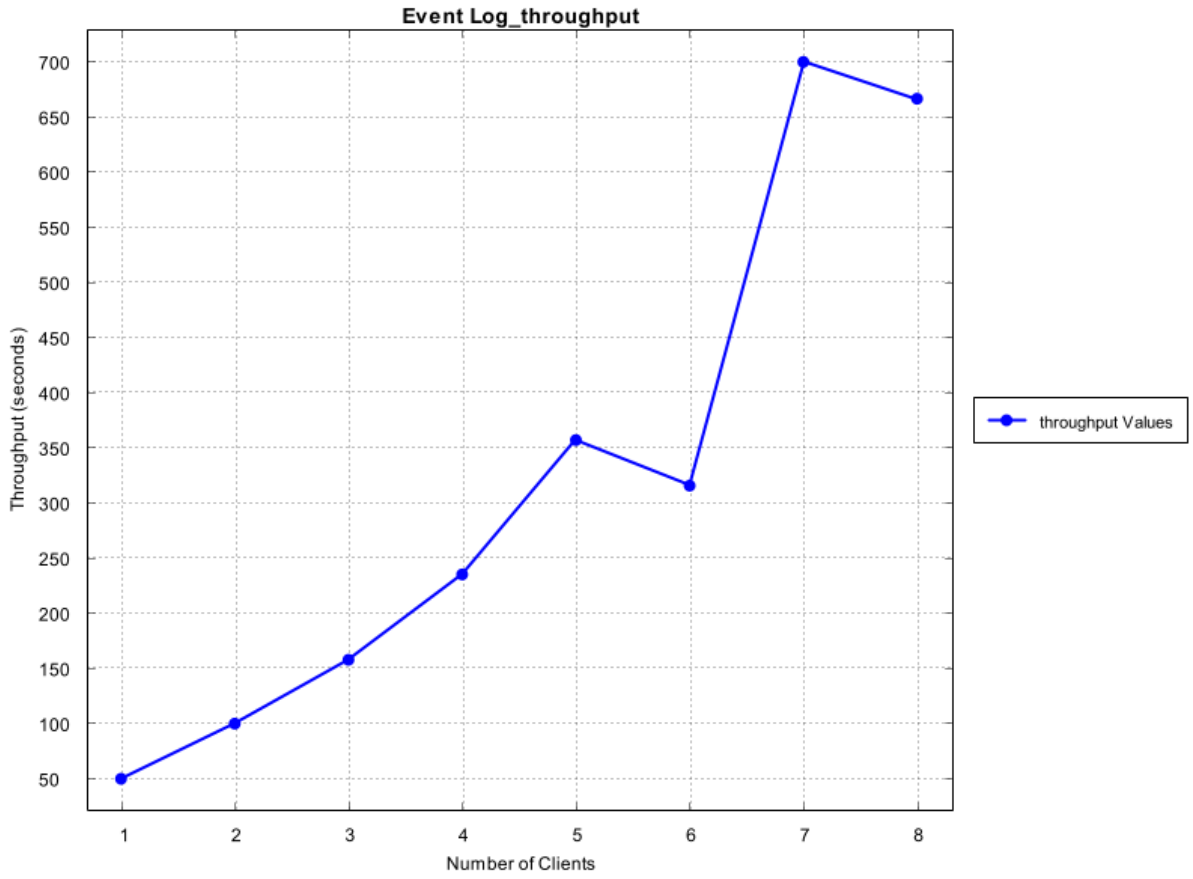


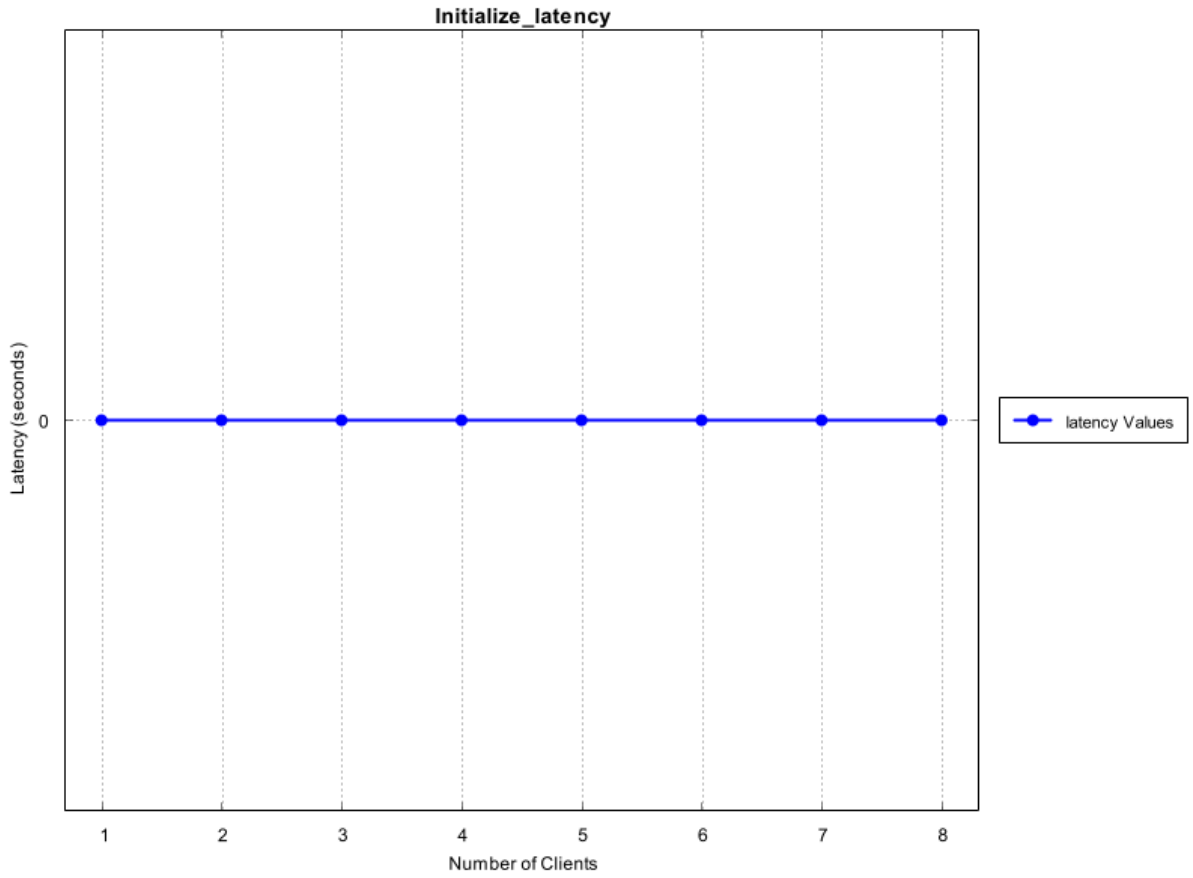
Pasted graph images:

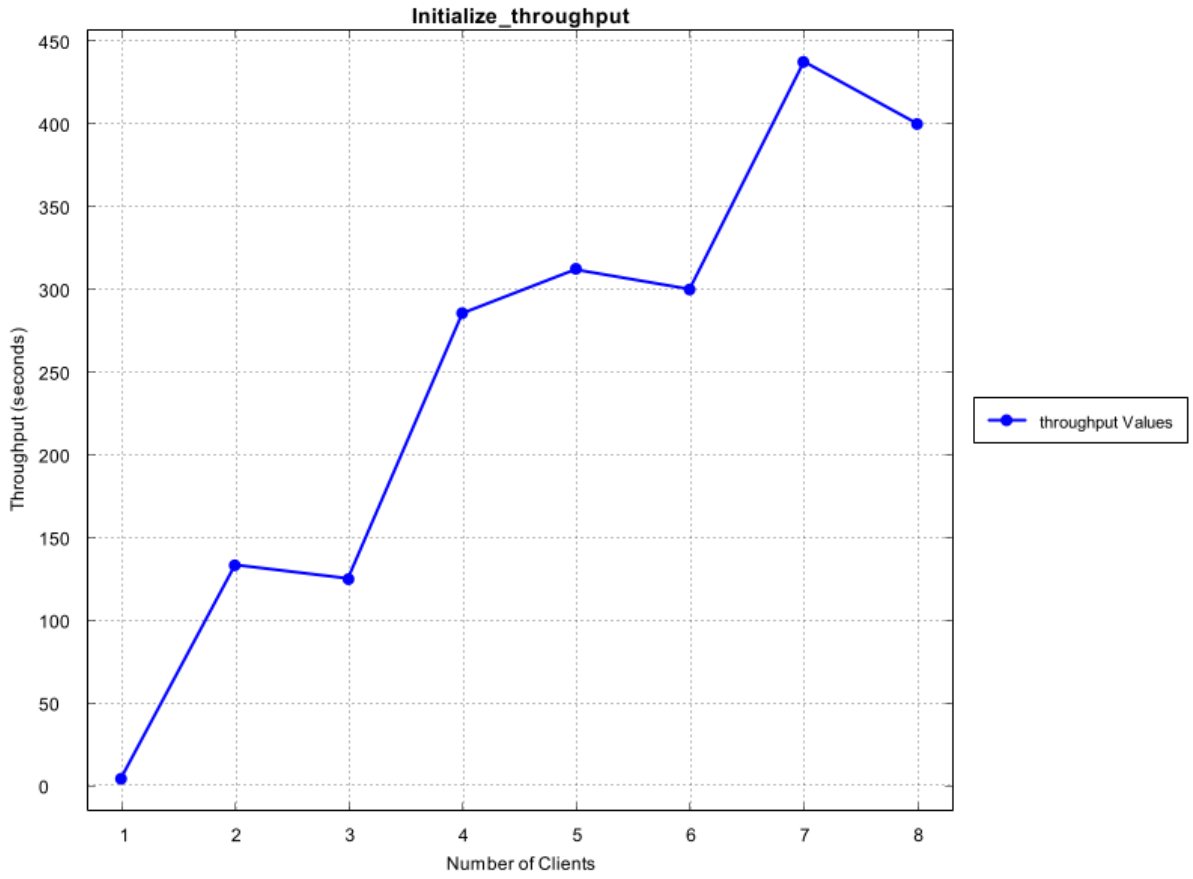

Create Topic_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values

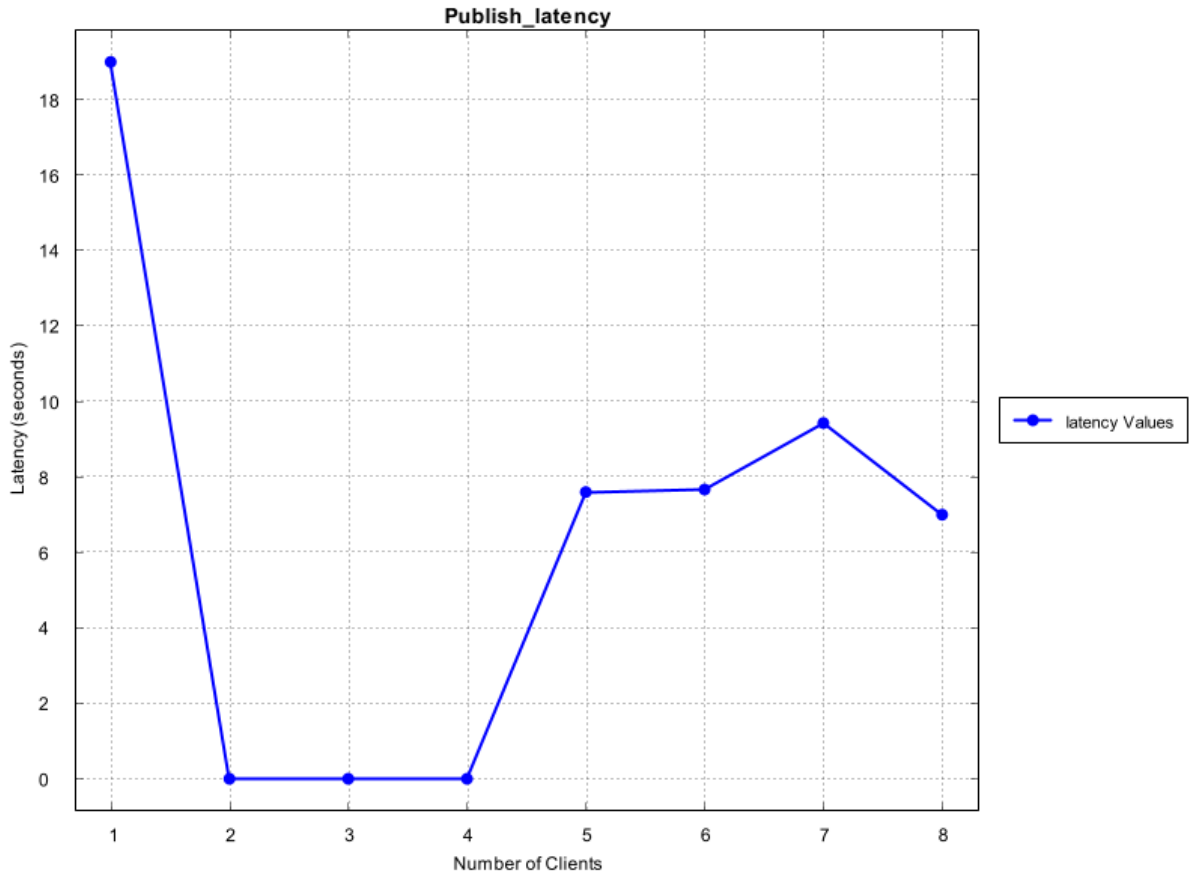

Create Topic_throughput
Throughput (seconds)
100
120
140
160
180
200
220
240
260
280
300
320
340
360
380
400
420
440
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

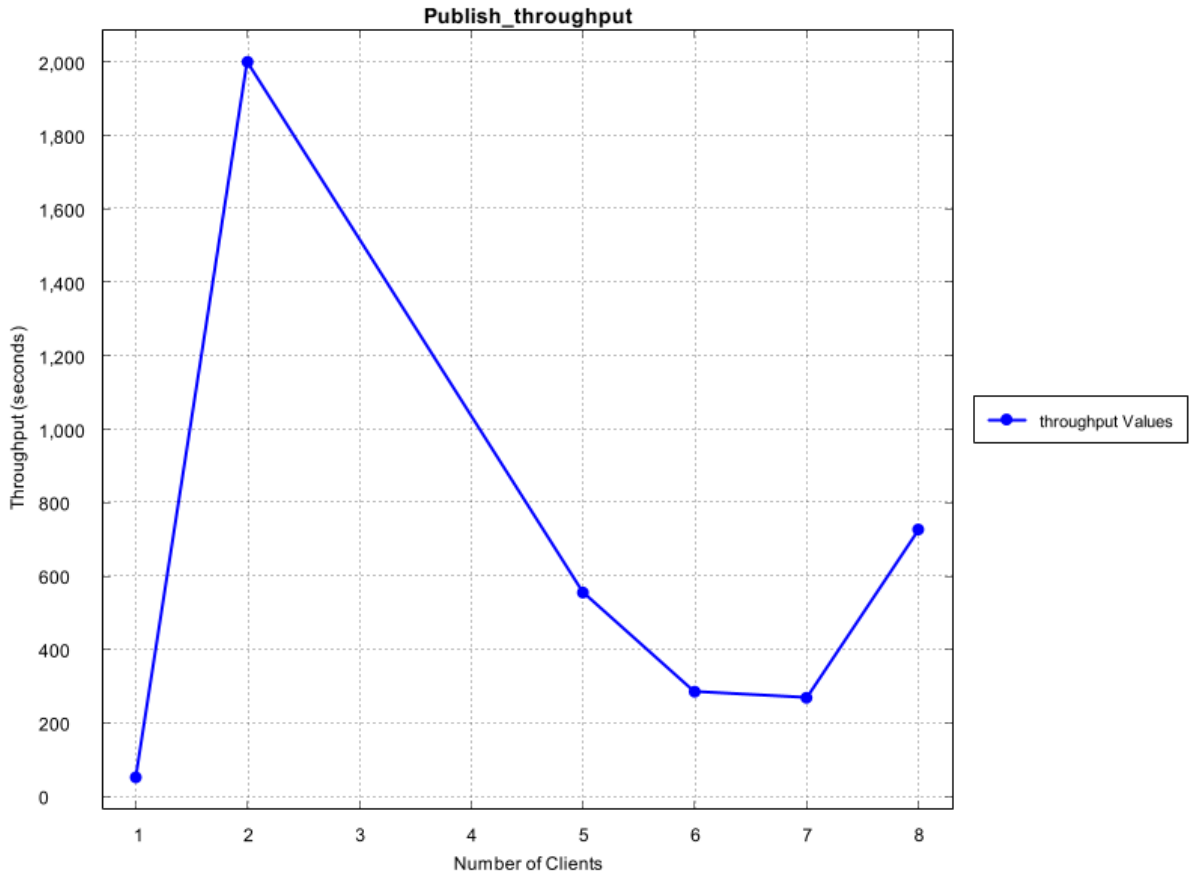

Event Log_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values

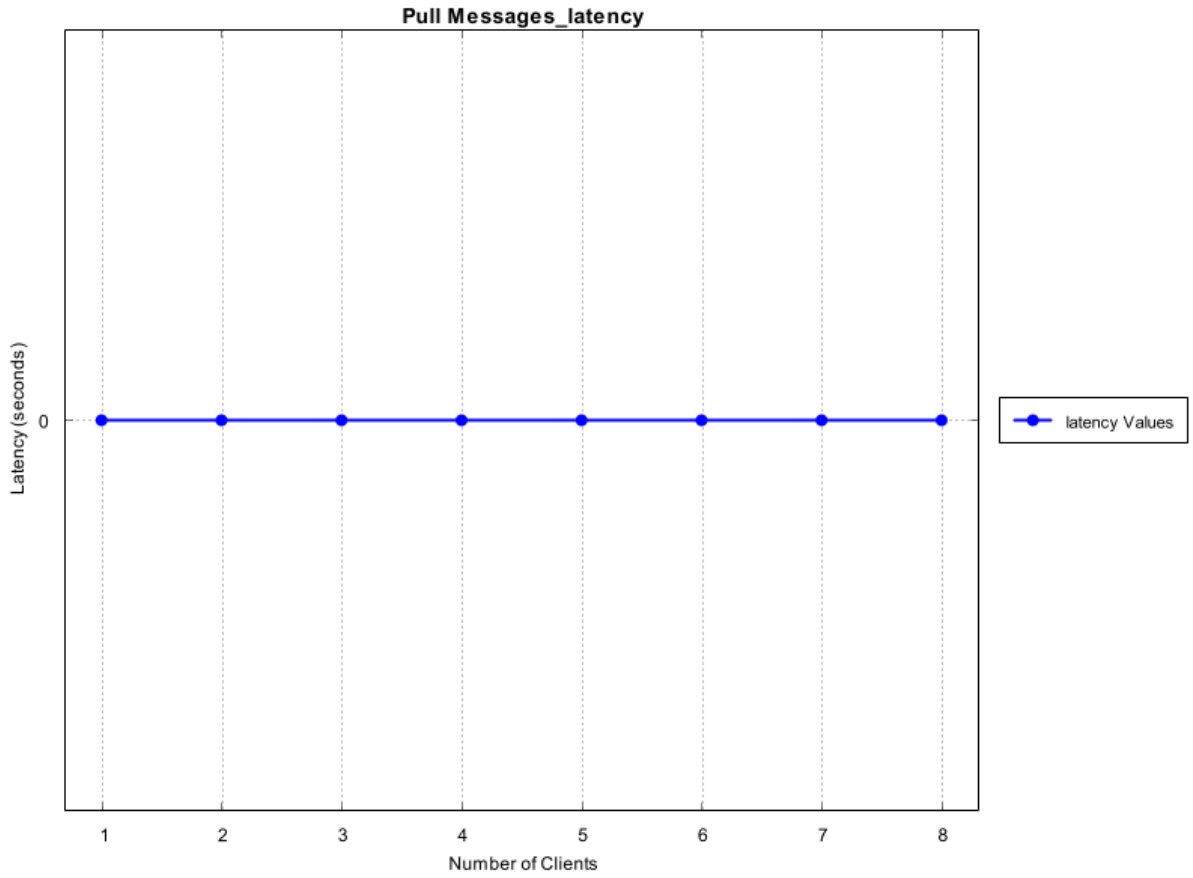

Event Log_throughput
Throughput (seconds)
700
650
600
550
500
450
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

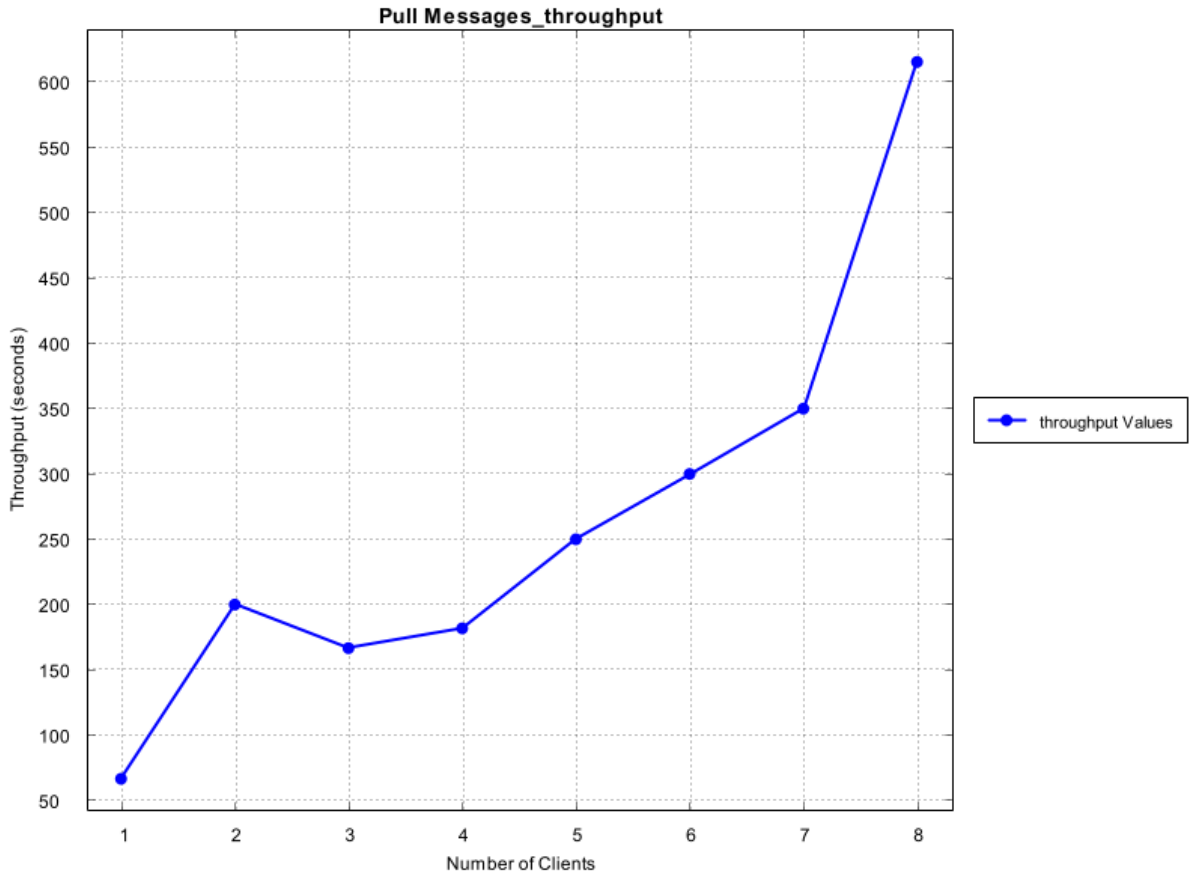

Initialize_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values

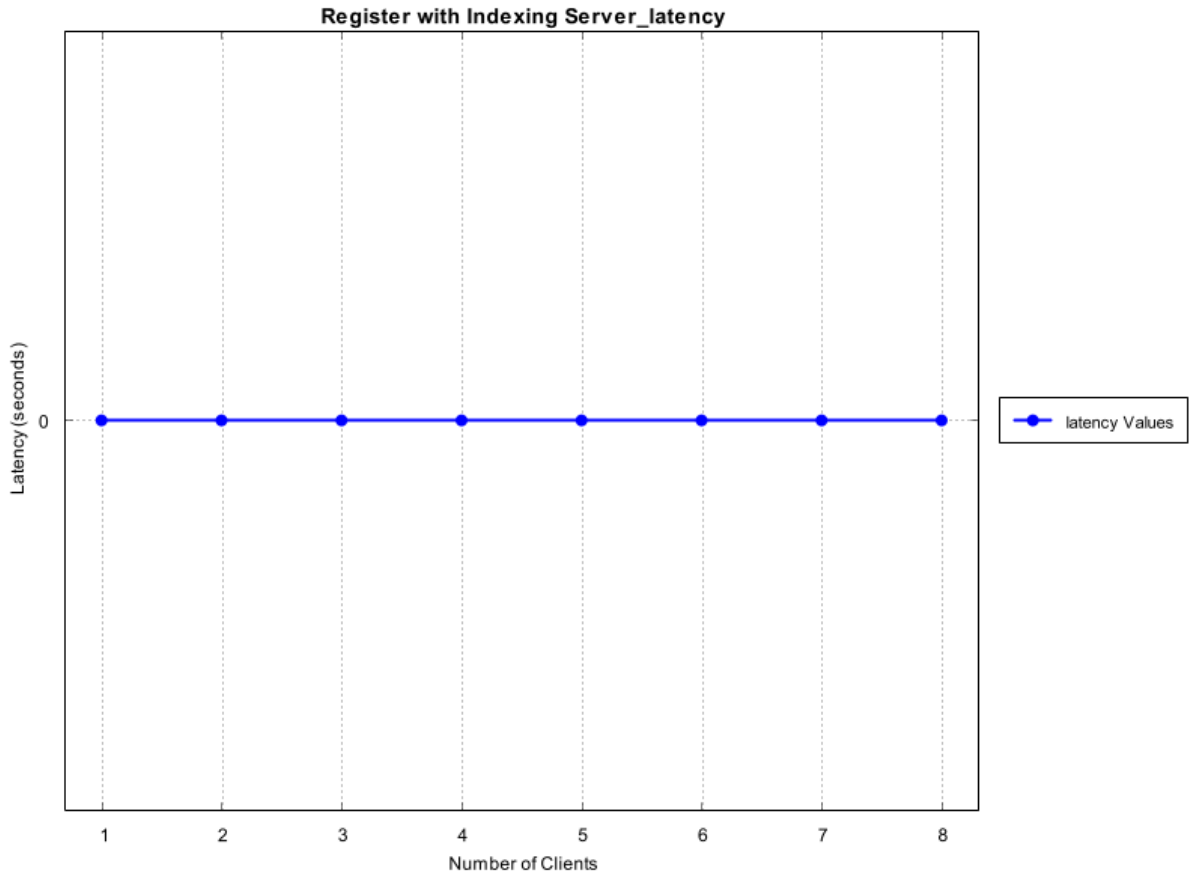

Initialize_throughput
Throughput (seconds)
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

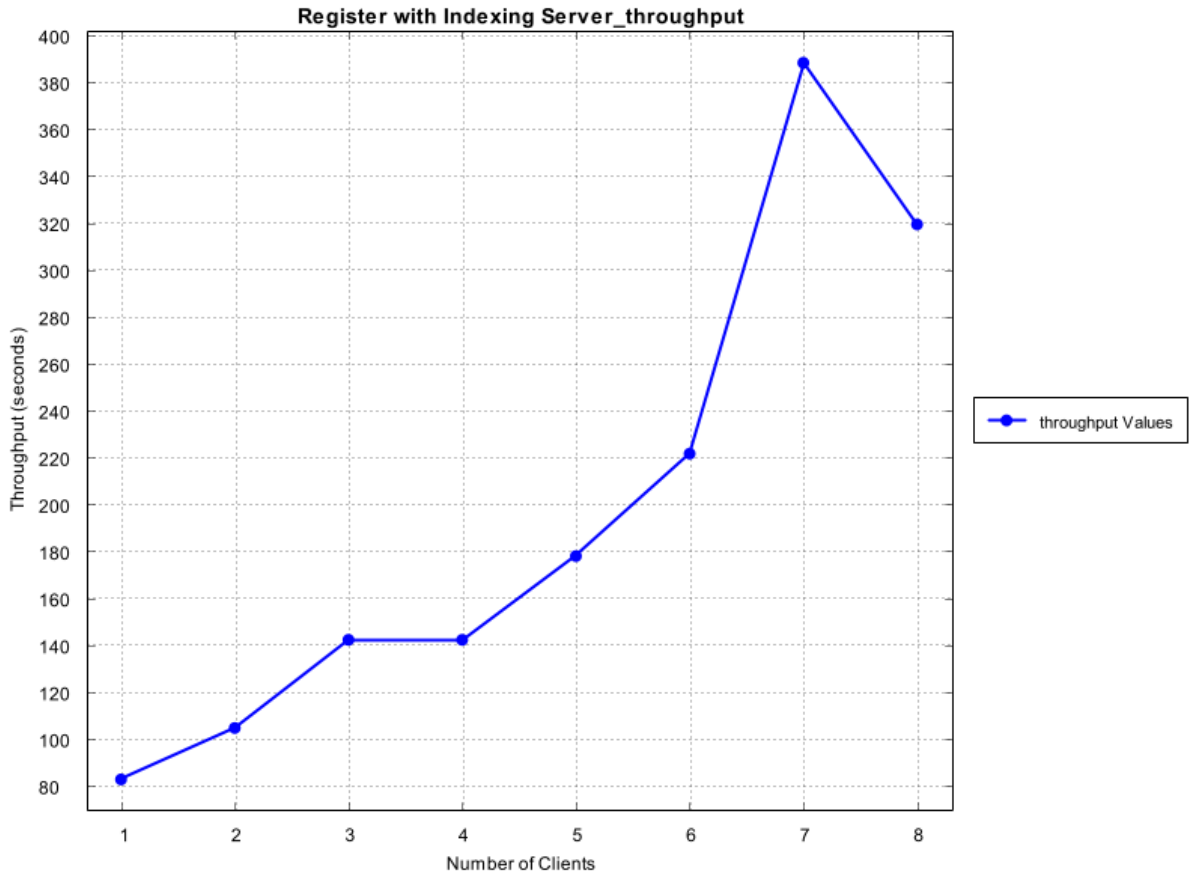

Publish_latency
Latency (seconds)
0
2
4
6
8
10
12
14
16
18
1
2
3
4
5
6
7
8
Number of Clients
latency Values

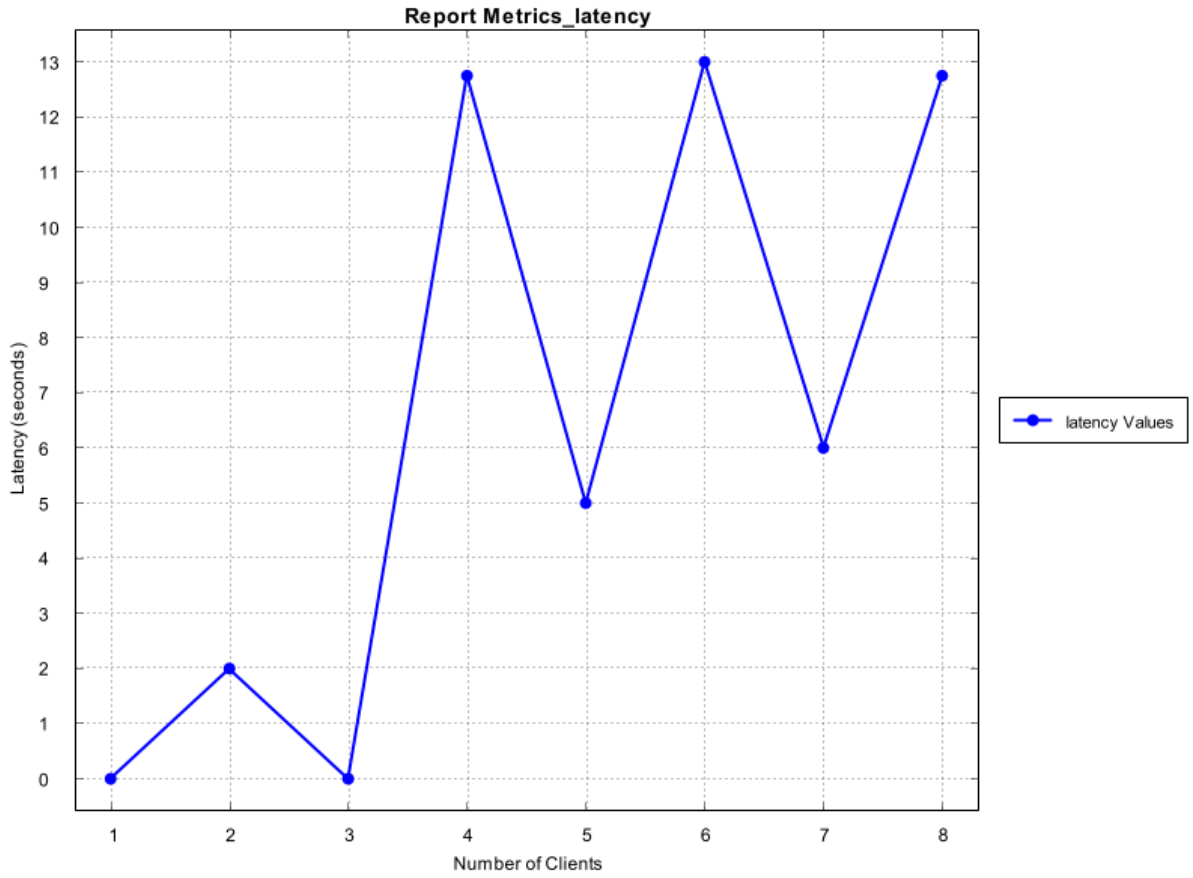

Publish_throughput
Throughput (seconds)
0
200
400
600
800
1,000
1,200
1,400
1,600
1,800
2,000
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

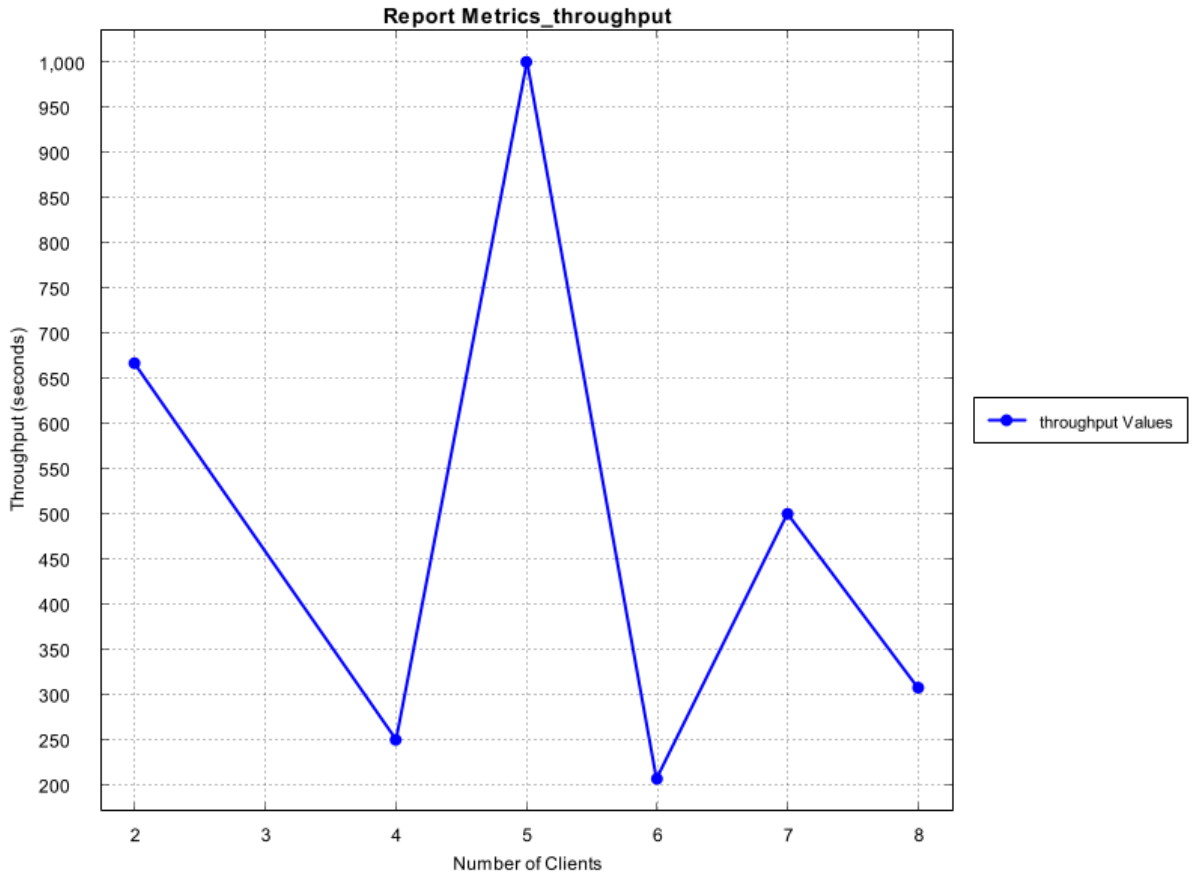

Pull Messages_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values

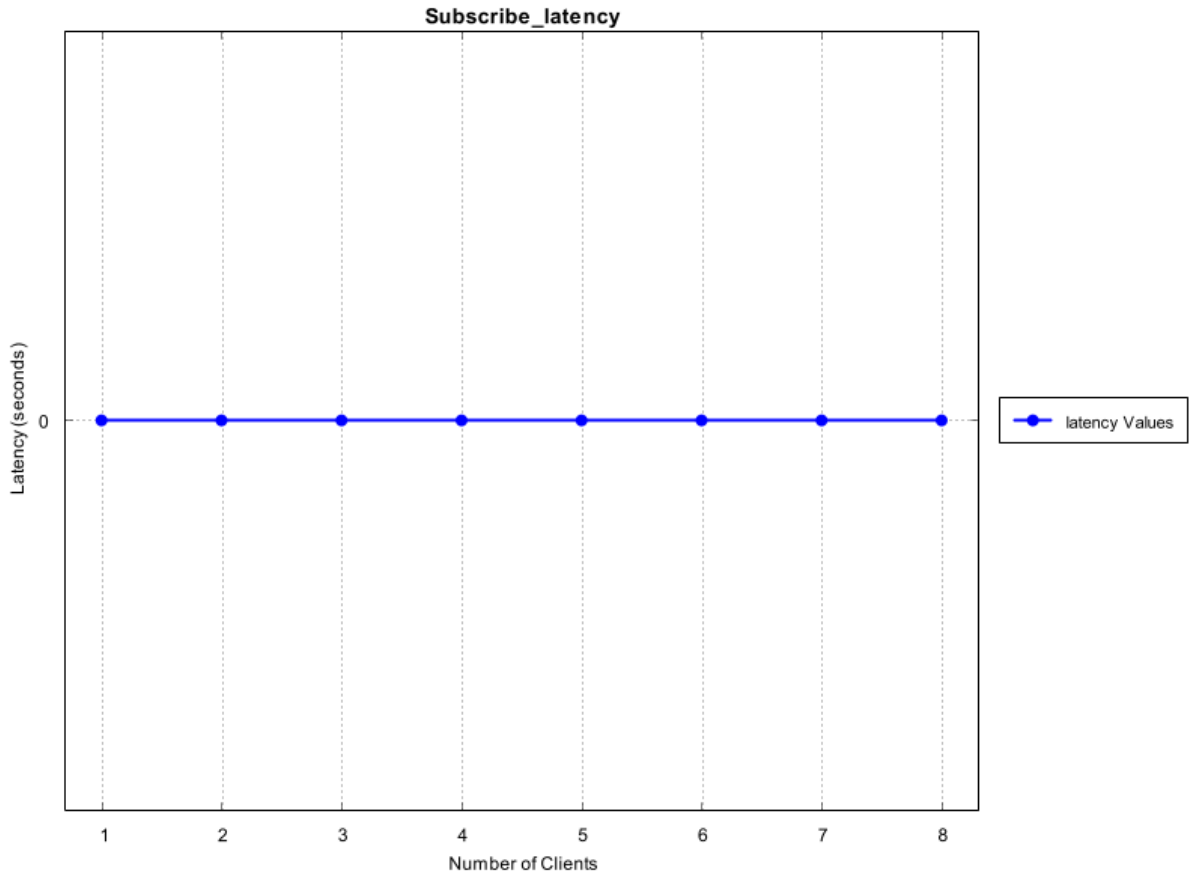

Pull Messages_throughput
Throughput (seconds)
600
550
500
450
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

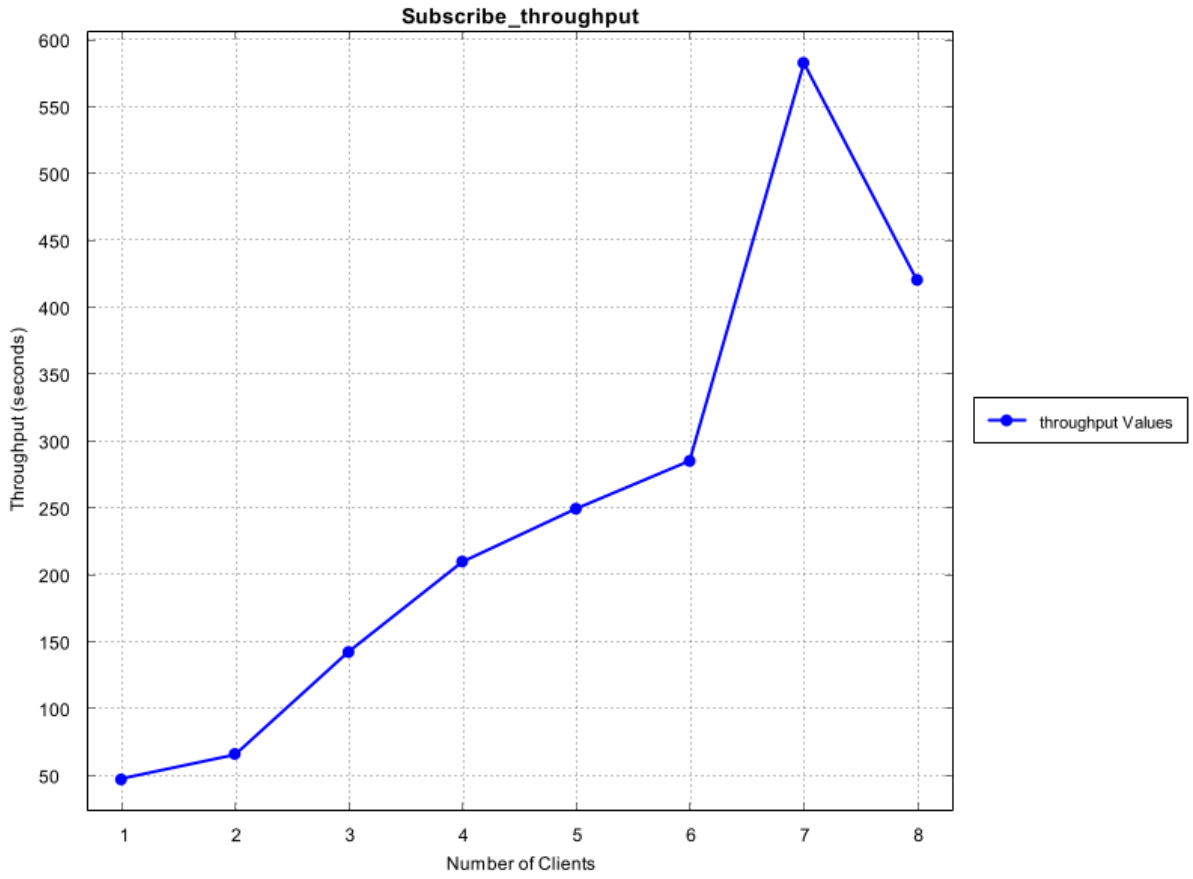

Register with Indexing Server_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values


Register with Indexing Server_throughput
Throughput (seconds)
400
380
360
340
320
300
280
260
240
220
200
180
160
140
120
100
80
1
2
3
4
5
6
7
8
Number of Clients
throughput Values


Report Metrics_latency
Latency (seconds)
Number of Clients
latency Values
0
1
2
3
4
5
6
7
8
9
10
11
12
13
1
2
3
4
5
6
7
8


Report Metrics_throughput
Throughput (seconds)
Number of Clients
throughput Values
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1,000
2
3
4
5
6
7
8


Subscribe_latency
Latency (seconds)
0
1
2
3
4
5
6
7
8
Number of Clients
latency Values


Subscribe_throughput
Throughput (seconds)
600
550
500
450
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8
Number of Clients
throughput Values

Conclusion

The project successfully implemented a robust Peer-to-Peer (P2P) messaging system using Spring Boot, focusing on efficient communication, fault tolerance, and high availability. Throughout the development process, several key features were integrated to enhance system performance and user experience.

1. ***Architecture and Design****:* The architecture was designed to facilitate direct communication between peers while maintaining a decentralized model. This structure ensured scalability, allowing the system to adapt to varying loads and the addition of new peers without compromising performance.
2. ***PeerNode and Indexing Server Controllers****:* The core components, PeerNodeController and IndexingServerController, were implemented to handle node registration, topic management, and message routing. These controllers provided a foundation for efficient data flow within the network, ensuring messages were correctly directed and delivered.
3. ***Latency and Throughput Measurements***: The implementation included mechanisms to benchmark the system's latency and throughput through comprehensive testing. CSV files were utilized to store the metrics, enabling visualization of performance through graphs generated using XChart. This facilitated a deeper understanding of the system's responsiveness under different loads.
4. ***Network Monitoring and Analytics***: A dashboard was developed to monitor network health in real time. It provided insights into active peers, message latency, and bandwidth usage, which are critical for maintaining system performance and reliability.
5. ***Fault Tolerance and High Availability****:* Techniques such as heartbeat mechanisms for peer failure detection and automatic recovery were incorporated. Data replication strategies were also implemented to ensure message availability even in the event of node failures, significantly enhancing the overall robustness of the system.
6. ***Testing and Validation****:* Comprehensive testing was performed to validate the functionality of the APIs, ensuring that the system could handle concurrent requests effectively. The results showed that the system maintained acceptable response times even with multiple concurrent peers, proving its reliability and efficiency.

In summary, the project demonstrated the potential of P2P architectures in creating resilient and scalable messaging systems. The integration of monitoring tools, fault tolerance mechanisms, and performance metrics provided a comprehensive solution that not only met the project's objectives but also laid the groundwork for future enhancements. Moving forward, further exploration into advanced features such as end-to-end encryption, improved user interfaces, and additional monitoring capabilities could further enhance the system's capabilities and user experience.

How to run the project?

Clone the repository from the below link or (unzip the file from canvas)

https://github.com/prakriti31/PeerToPeerSystems/tree/master

Now build the pom.xml file using the following command.

```
mvn clean install
```

Now trigger all the test files either one by one or use

```
mvn test
```

Below are the pre-requisites to run each testing file.

To run the IndexingServerTests, you need to initialize the P2PSystemApplication.java file to run the spring boot server on port 8080.
Then you can trigger the indexing server plot file to fetch the graphs from the csvs.
Similar approach applies to PeerNodeTests and PeerNodePlot respectively.

Now to trigger requirements1 file, you need to kill the procedure running on port 8080 and initialize 3 peers on 2 ports using the following commands.

1. Navigate to the project.
2. Navigate to cd target/ from root.
3. java -jar p2p-system-0.0.1-SNAPSHOT.jar --server.port=8081
4. java -jar p2p-system-0.0.1-SNAPSHOT.jar --server.port=8082
5. java -jar p2p-system-0.0.1-SNAPSHOT.jar --server.port=8083

Similarly, to trigger the requirements2 file, you need to do the same for 20 peers, i.e 20 ports from 8081 to 8100 respectively.

To initialize the APIs, run the P2PSystemApplication.java and use the cURLs provided.