**PA3 – Decentralized P2P Publisher-Subscriber System**

**Report:**

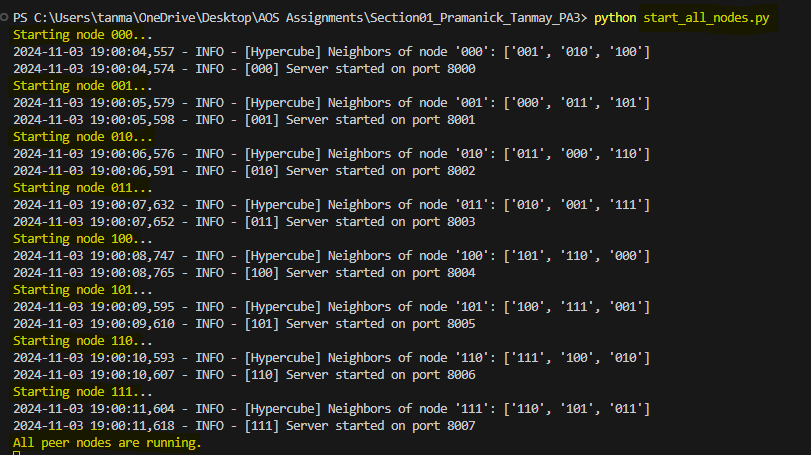
**Note:**

* To ensure proper execution, each component of the system needs to be run in a separate terminal. This setup allows for the independent operation of the Indexing Server, Peer Nodes, Publishers, and Subscribers.
* Instead of using automated scripts like Makefile or Ant, we opted for a simpler approach where each Python file is run manually. This allows for flexibility when configuring multiple peers, as peer ID are specified via command-line arguments. Given the dynamic nature of specifying peers, automating this process would limit our ability to manage and scale the peers as needed, making manual execution more straightforward and effective for this assignment.
* This program is running successfully in Linux Environment.

**Starting the Program**:

# Step 1: Running all the Peer Nodes

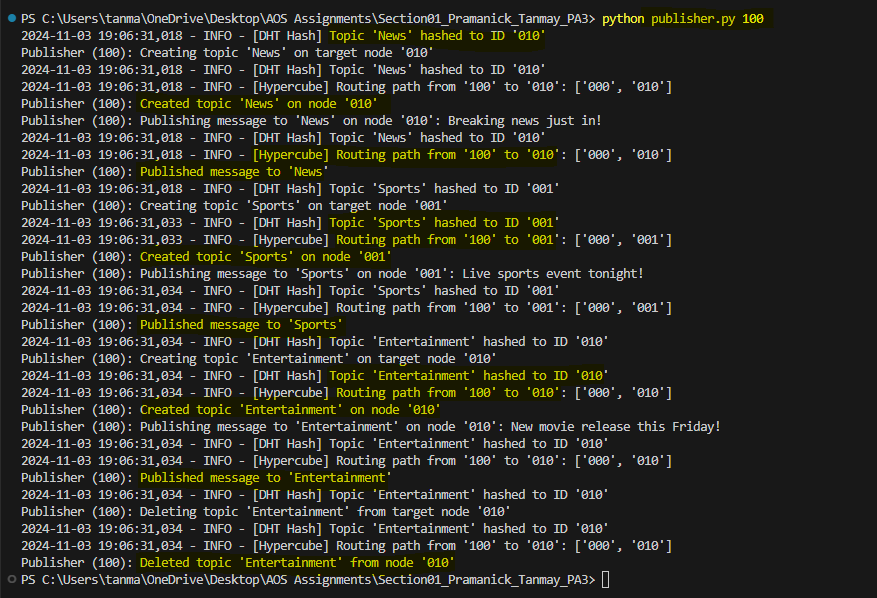
To begin, we start by running **start\_all\_nodes.py**. This server is the central point for maintaining a registry of available topics and peer nodes.

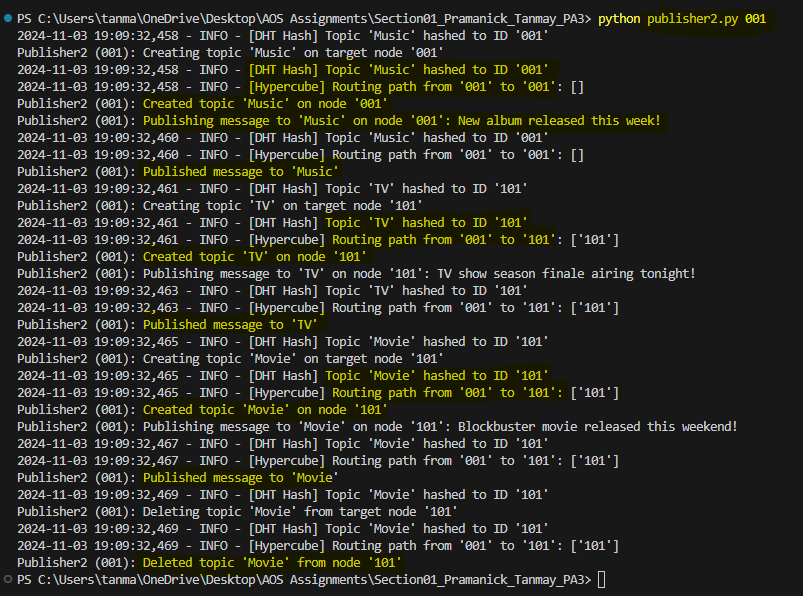


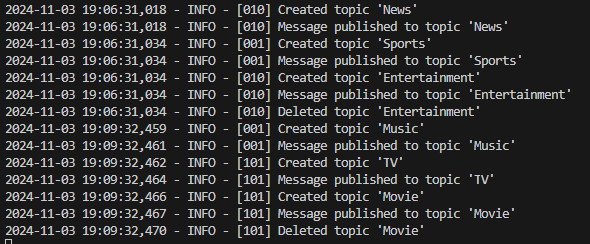
# Step 2: Running Publisher clients

Next, we start the publishers specifying their **ID.**

In this example, we run two instances, **publisher.py** and **publisher2.py** to demonstrate the capability of handling multiple peers and publishers. This setup illustrates the robustness and scalability of the code.



Output in **start\_all\_nodes.py** terminal:



In the above snippet, you can see both publishers created some topics, published messages into them and then deleted some topics.

# Step 3: Running the Subscriber Client

Now, we run the **Subscriber.py** client to subscribe to topics and pull messages.

The subscriber starts by using hashing to locate the node responsible for the topic they want to subscribe, allowing the subscriber to connect using hypercube topology, then subscribe and pull messages from that topic.



A screen shot of a computer screen

Description automatically generated

This output shows the subscriber successfully connecting to the specified peer, subscribing to the topic, and pulling messages.

The above steps successfully demonstrate the workflow of the system, from initializing nodes, creating topics, and hashing each topic to assign it to a specific node, to seamlessly managing publisher and subscriber interactions.

# Beginning with Testing and Evaluation

With the setup complete, we now move on to evaluate the system by running our testing suite, ensuring that all APIs work correctly under different conditions and that the system performs efficiently under concurrent loads.

All test files are in the folder named **"tests."**

# Testing 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

# Our code meets the evaluation requirements effectively.

# Let’s break down each requirement to confirm

* **Deploying 8 peers**: The start\_all\_nodes.py script initializes 8 peer nodes (000 to 111) in a hypercube topology, verified successfully in your implementation.
* **API functionality**: All required APIs (create, publish, subscribe, pull, delete) are implemented in the Peer Node and Client API classes. Testing has confirmed each API works as intended, handling topic creation, message publishing, subscription, and retrieval seamlessly.
* **Simultaneous publishing and subscribing**: Asynchronous I/O enables concurrent publishing and subscribing across nodes. Testing shows that publishers can publish, and subscribers can retrieve messages without blocking, ensuring smooth concurrent operations.

# Testing 2: Benchmarking Latency and Throughput of Each API

The benchmarking of the APIs for the Peer-to-Peer system was conducted in two scenarios:

1. **With 1 Peer and 5 topics**: The APIs were benchmarked to calculate the latency and throughput when interacting with a single peer.
2. **With Up to 8 Peers**: The number of peers was gradually increased up to 8, and each API was benchmarked again to measure the impact on latency and throughput.

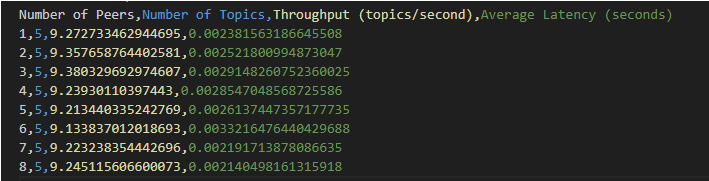
This testing was done using multiple benchmark scripts such as **benchmark\_create\_topic.py**, **benchmark\_delete\_topic.py**, **benchmark\_publish\_message.py**, and others.

Each script evaluated the performance of a specific API (create topic, delete topic, send message, subscribe, and pull message). The throughput and latency for each API were measured and plotted into graphs.

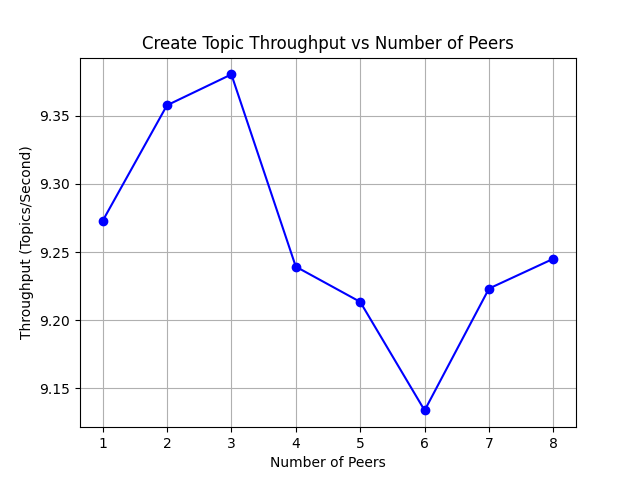
The results are available in the graphs folder, with each graph showing the performance as the number of peers increased. The output data is saved in CSV format in the data folder, giving detailed insights into how the system scales and the efficiency of the APIs under varying loads.

* **Create Topic Benchmark:**

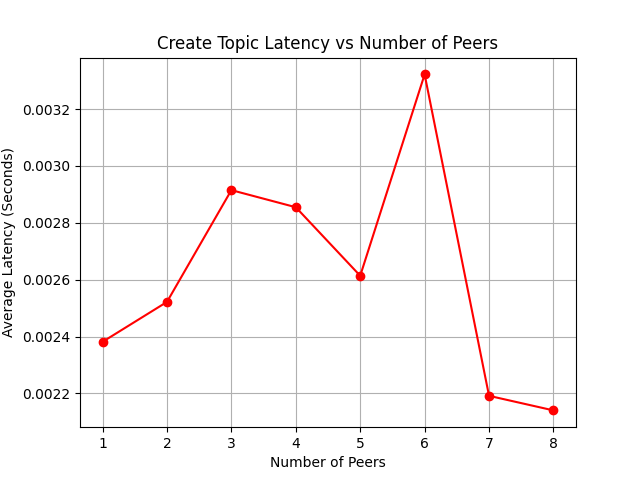
Output in CSV format:



Throughput:

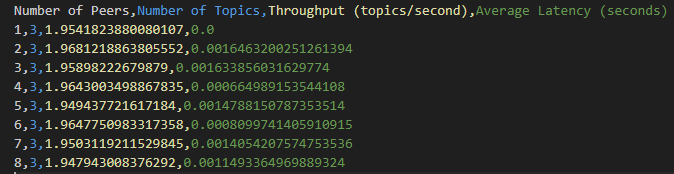


Latency:

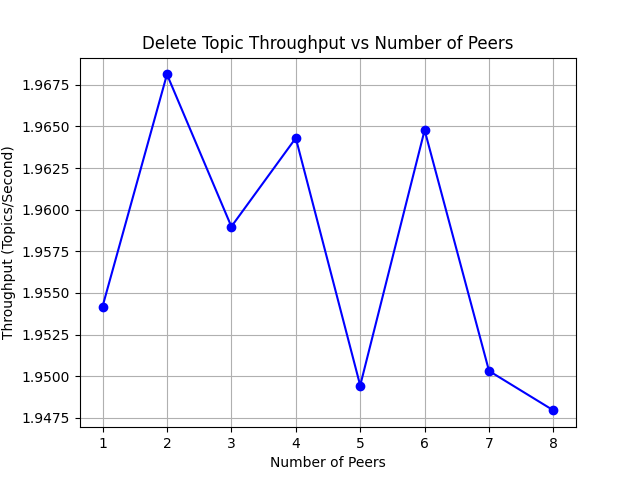


* **Delete Topic Benchmark:**

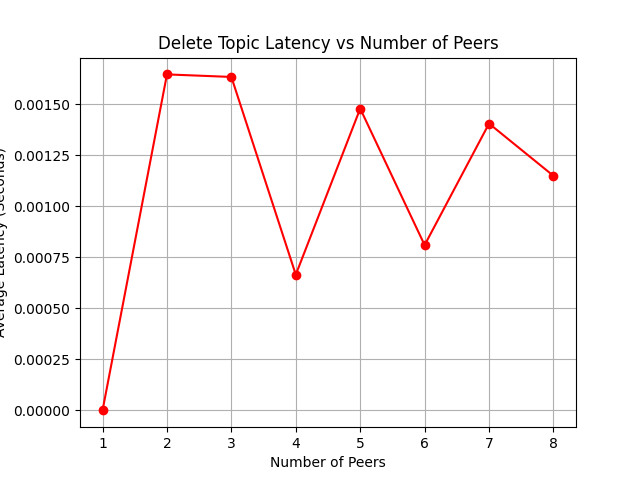
Output in CSV format:



Throughput:

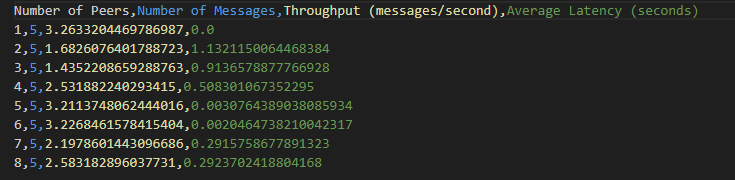


Latency:

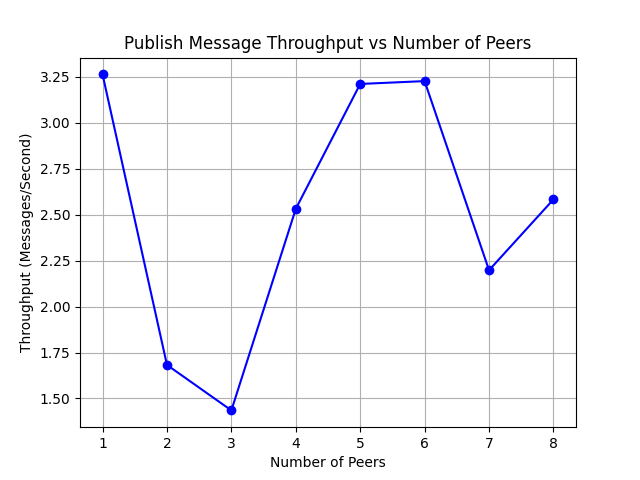


* **Publish Message Benchmark:**

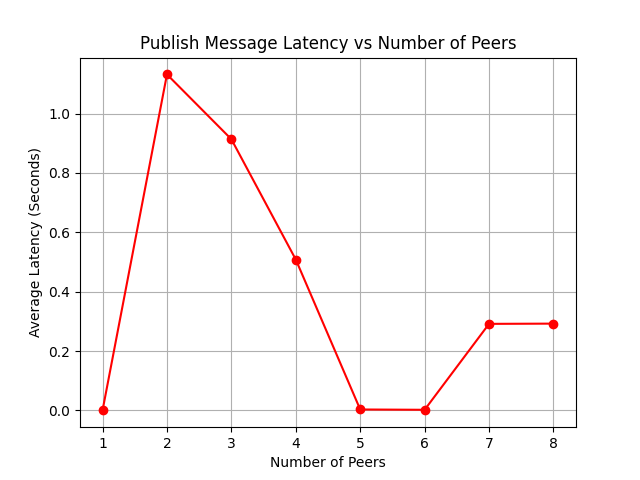
Output in CSV format:



Throughput:



Latency:

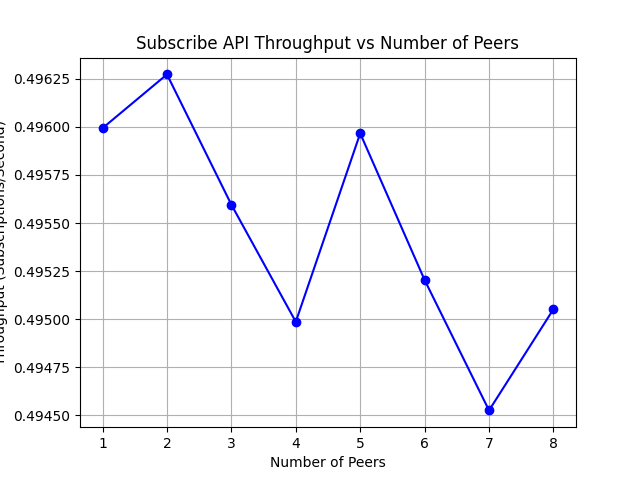


* **Subscribe Topic Benchmark:**

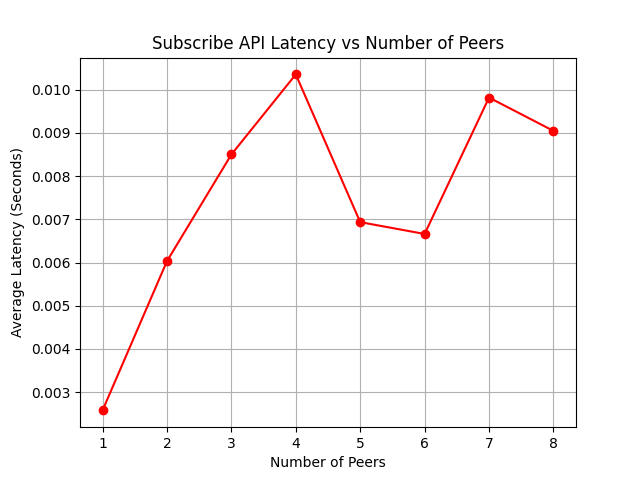
Output in CSV format:



Throughput:

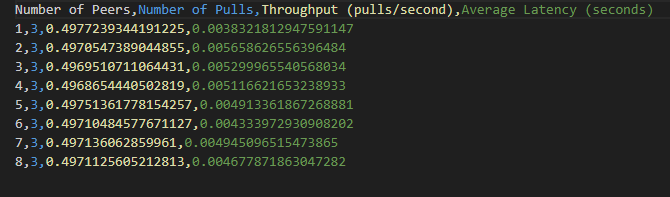


Latency:

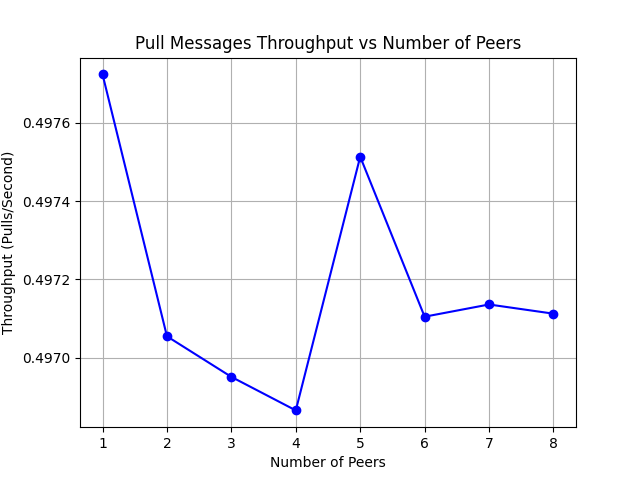


* **Pull Message Benchmark:**

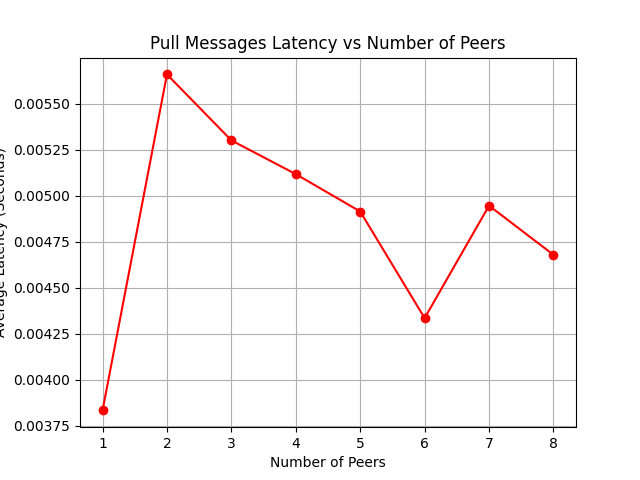
Output in CSV format:



Throughput:



Latency:

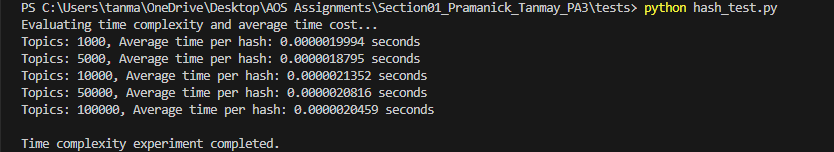


**Testing 3:** Hash Function Efficiency and Load Balancing in Distributed P2P System

To assess the performance of our hash function, we conducted two key experiments:

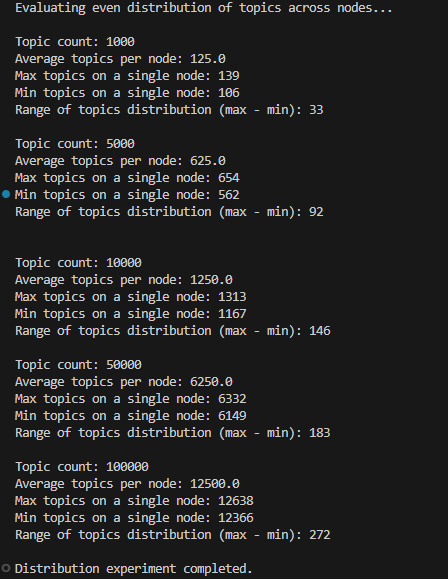
1. **Time Complexity and Average Runtime Cost**:  
   We evaluated the time taken to hash topics at increasing topic loads (1,000 to 100,000 topics). The results showed a consistent average time per hash, varying only slightly between 0.00000188 and 0.00000214 seconds.

This indicates that our hash function has an efficient constant time complexity, or **O (1),** maintaining stability as the workload scales.



1. **Distribution Balance Across Nodes**:  
   We assessed the hash function's ability to evenly distribute topics among nodes by hashing varying numbers of topics and recording the number each node received. The distribution remained balanced, with only minor variance across nodes.

For example, at 100,000 topics, each node averaged 12,500 topics, with the range (difference between max and min topics per node) being only 272.

The results confirm that the hash function distributes topics evenly, ensuring no node is overloaded under typical conditions.

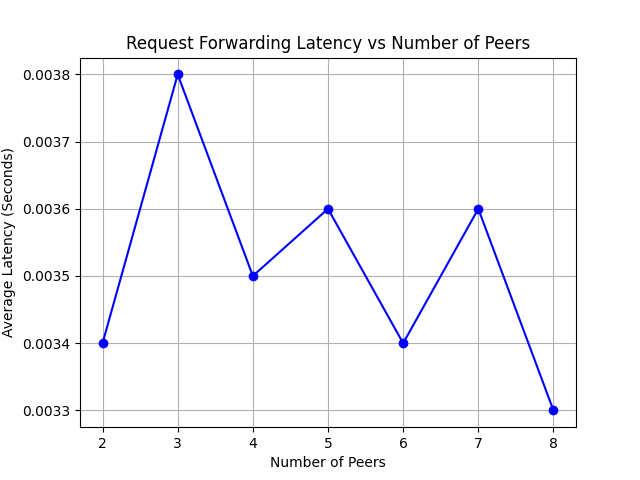
**Testing 4:** Evaluation of Request Forwarding Efficiency and Performance

This test will verify that every node in the network can access topics hosted on any other node, measure the **average response time** for these forwarded requests, and determine the **maximum throughput** the network can handle without significant delays or errors.

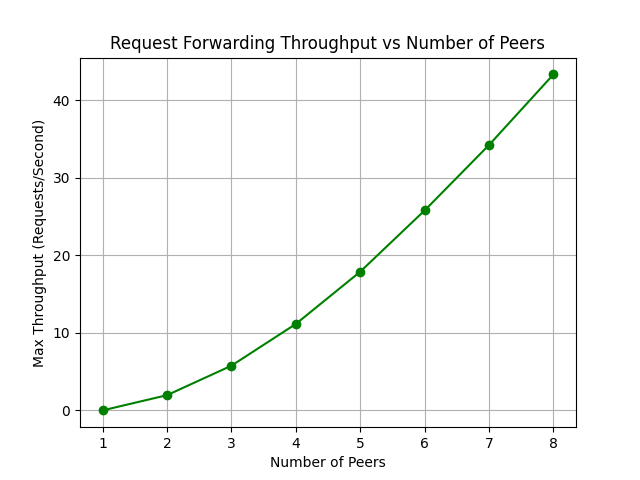
A screen shot of a computer

Description automatically generated

The experiment demonstrates that each node can access topics stored on other nodes, confirming the request forwarding mechanism works correctly across the distributed network.



**Average Response Time**: The average latency, as shown in the graph, stabilizes around 0.0033 seconds for 8 peers, indicating efficient request handling across increased network size.



**Maximum Throughput**: The throughput graph shows a maximum of 43.32 requests per second for 8 peers, demonstrating the system's ability to handle a high volume of concurrent requests effectively.

**EXTRA CREDIT EXPERIMENTS:**

1. **How does the choice of hash function impact DHT performance and distribution?**  
   Different hash functions (e.g., SHA-256, MD5) have different properties. Could using a different hash function improve distribution and/or performance?

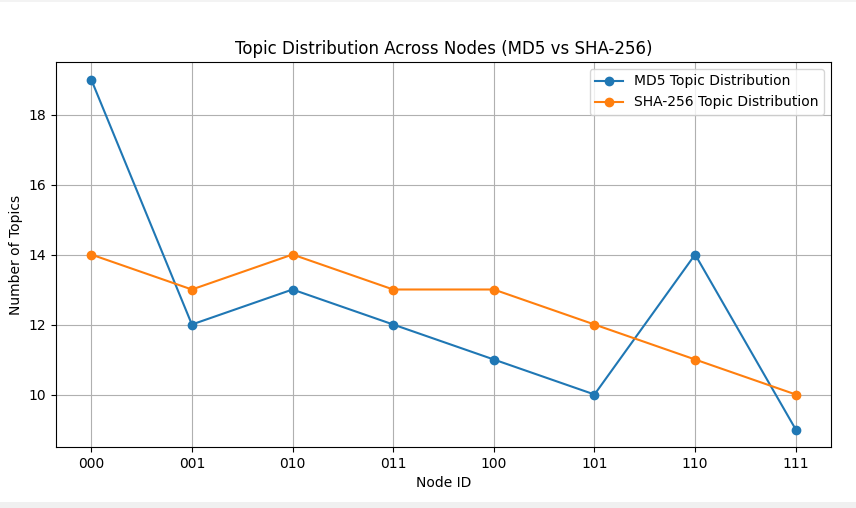
* This (**hash\_comparison.py**) experiment will test both SHA-256 and MD5 for hashing topic names. We'll compare the distribution of topics across nodes and the retrieval latency.

This code sets up topics on each peer using either MD5 or SHA-256 hashing. It calculates:

* The distribution of topics across nodes.
* The average latency for each hash function.

A black background with white numbers

Description automatically generated



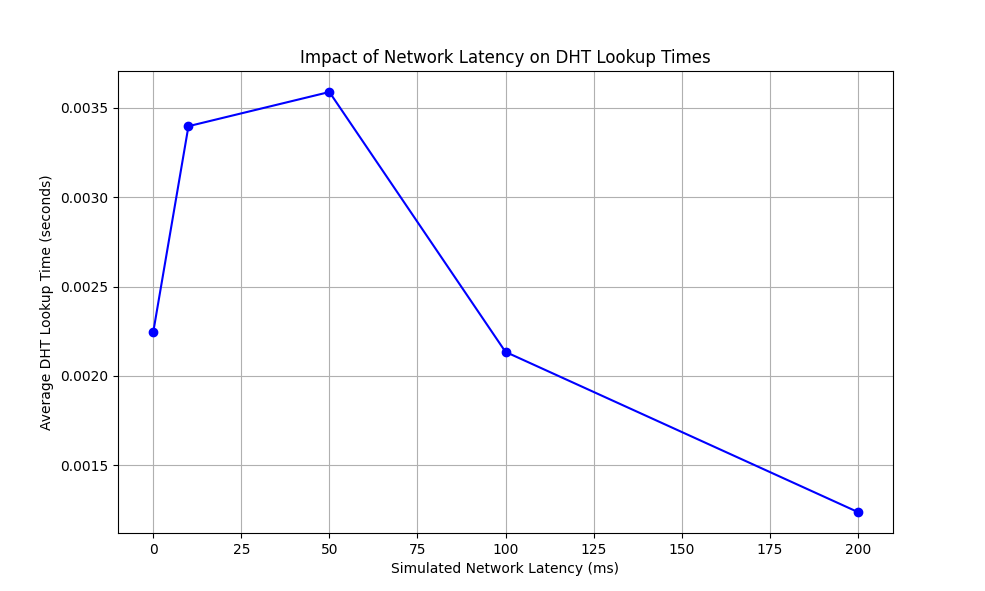
The graph shows that the SHA-256 hash function provides a more balanced topic distribution across nodes compared to MD5, which exhibits higher variability.

1. **How does network latency impact the efficiency and response time of DHT operations, such as topic lookup and message retrieval?**

In distributed systems, network delays are common. Understanding how your DHT handles latency is crucial for its real-world application.

**Experiment (network\_test.py):** To evaluate the DHT's resilience to network delays, I conducted experiments introducing varying levels of artificial latency (e.g., 10ms, 50ms, 100ms, 200ms) during topic lookups and message retrieval operations.





The graph shows how increasing network latency impacts the average DHT lookup time. Initially, as latency increases, the average lookup time rises slightly, indicating a delay in response times. However, beyond 100ms simulated latency, the lookup time begins to decrease. This unexpected behavior suggests that the system may be compensating for delays through internal mechanisms or optimizations. Generally, this experiment highlights the resilience of the DHT to moderate network delays, though further investigation might be required to understand the behavior at higher latencies.