National Technical University of Athens
Department of Electrical and Computer Engineering
Data Science and Machine Learning
Course: Distributed Systems
Semester: Winter 2024/25

Chordify: Semester Project in Distributed Systems

Evangelos Chaniadakis
03400279

Anastasios Kanellos
03400251

Dimitra Kallini
03400250

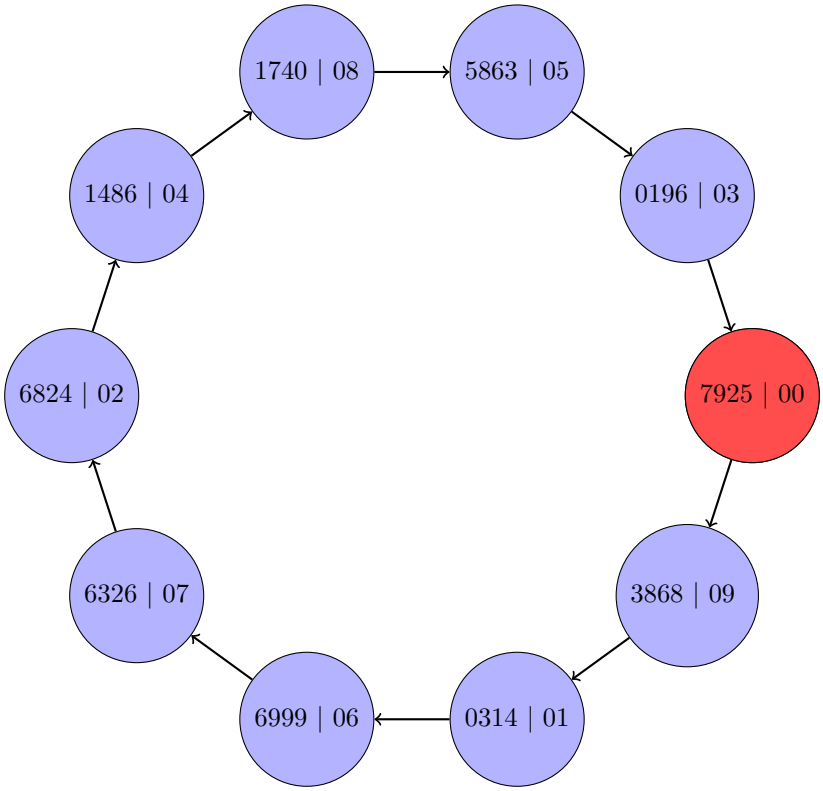
Submission Date
Tuesday 1st July, 2025

1 Introduction

This assignment focuses on the implementation of Chordify, a decentralized peer-to-peer application for song exchange, based on the Chord DHT protocol [1]. The system is designed to offer scalable and fault-tolerant storage by organizing nodes in a ring topology, ensuring that data remain available even in the case of node failures. Each node is responsible for a subset of the key space, and data is replicated across multiple nodes to guarantee consistency and reliability.

2 System Overview

This system is based on the Chord protocol, a distributed hash table (DHT) used for organizing nodes in a decentralized ring structure. Each node is responsible for a subset of the key space, with keys hashed to determine their placement in the ring. The nodes maintain references to their predecessor and successor, allowing them to locate data efficiently and handle node joins and departures.



Data are replicated across multiple nodes for increased reliability and availability, with a configurable replication factor. The system supports two consistency models: chain replication and eventual consistency. Chain replication works by passing updates sequentially through a chain of replicas, while eventual consistency updates the replicas through asynchronous propagation. These two models allow the system to adapt to different application requirements, from strong consistency to higher availability and performance.

The system includes several key operations: inserting a key-value pair, querying data, and deleting key-value pairs. It also provides an overlay network functionality, to collect and visualize the network's topology, which includes node IDs, IP addresses, ports, and key counts.

The system ensures that key transfers during node departures are handled gracefully, updating the network topology to maintain data availability.

The nodes communicate using sockets, and in eventual consistency mode, updates are propagated asynchronously through multithreading. The server's controller API handles multiple client requests by supporting concurrent connections. A GUI client interface provides users with a convenient way to interact with the system, supporting all operations such as inserting and querying data, visualizing the network, and managing node joins and departures.

3 Experimental Setup

The experiments were conducted in a private AWS cloud environment, comprising a public bastion host and four private VMs. The bastion host serves as the gateway to the network, enabling SSH connections to the private VMs via an internal network. Each VM runs two nodes on separate ports, totaling ten nodes. Once `node.py` is initialized on all nodes, `gui_client.py` is launched to provide system interaction through a graphical interface.

4 Results

4.1 Write Throughput Experiment

Linearizability exhibits decreasing write throughput as replication increases, reflecting the overhead of maintaining strict order and acknowledgment before proceeding across the chain of nodes. Eventual consistency experiences the same trend but with significantly lower latency, as it prioritizes availability and allows asynchronous propagation without waiting for full confirmation from all replicas before proceeding.

Consistency	Linearizability		Eventual Consistency	
Replication Factor	Time Taken (s)	Write Throughput (Keys/sec)	Time Taken (s)	Write Throughput (Keys/sec)
1	3.84	130.2083	3.61	138.5042
3	5.25	95.2381	3.88	128.8660
5	6.21	80.5153	4.11	121.6545

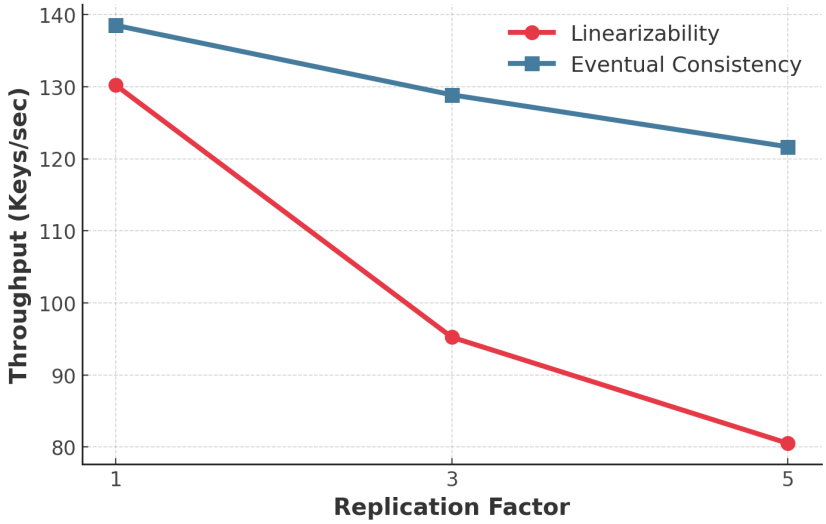


Figure 1: Write Throughput vs. Replication Factor

4.2 Read Throughput Experiment

In chain replication (linearizability), read throughput remains approximately constant because all reads occur at the tail of the chain (the last replica in the sequence). As more nodes are added to the middle, the tail remains unchanged, meaning the read performance does not improve. In contrast, in eventual consistency, reads can be served from any replica. As the replication factor (k) increases, more copies of the data are distributed across the network, increasing the likelihood of accessing a nearby replica and thereby reducing read latency.

Consistency	Linearizability		Eventual Consistency	
Replication Factor	Time Taken (s)	Read Throughput (Queries/sec)	Time Taken (s)	Read Throughput (Queries/sec)
1	4.26	117.3709	4.64	107.7586
3	5.54	90.2527	0.79	632.9114
5	6.52	76.6871	0.29	1,724.1379

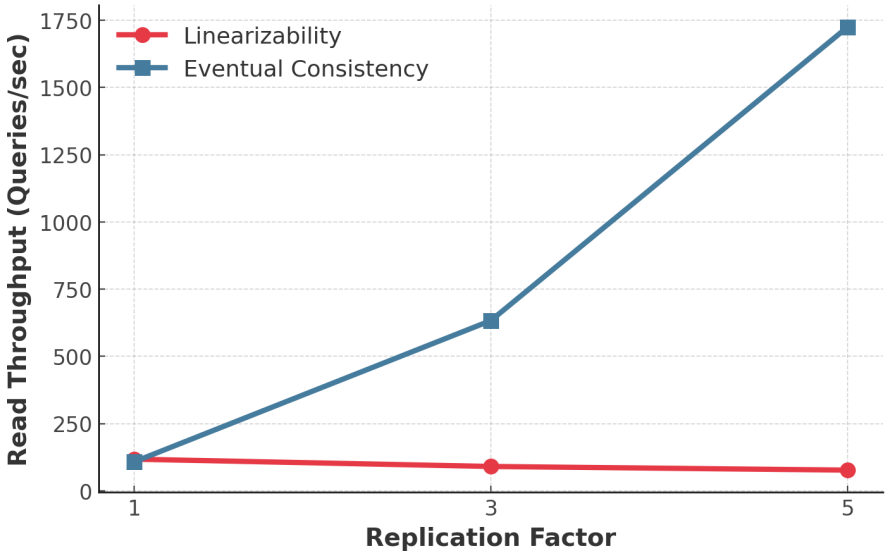


Figure 2: Read Throughput vs. Replication Factor

4.3 Freshness Experiment

Below, we compare system responses collected after this experiment execution under linearizability and eventual consistency. The system processes insert & query requests, storing query results for the two discrete consistency types to later detect discrepancies between their values. In Table 1 we highlight missing or extra values in each consistency model, demonstrating how consistency levels influence data propagation and potential inconsistencies.

The results align with our expectations, showing that linearizability maintains more fresh values compared to eventual consistency. This is due to the stronger consistency requirements of linearizability, which ensures that all nodes have the most up-to-date data. In contrast, eventual consistency allows for stale data to exist temporarily across nodes, leading to fewer fresh values. Consequently, while eventual consistency improves throughput and scalability, it sacrifices data freshness to some extent.

Line	Key	Linearizability	Eventual Consistency	Missing Values
41	Hey Jude	542, 596, 506, 502	542	506, 502, 596
42	Hey Jude	542, 596, 506, 502	542	506, 502, 596
43	Hey Jude	542, 596, 506, 502	542	506, 502, 596
44	Respect	528, 522, 566	528, 522	566
45	What's Going On	531, 529	531	529
46	Hey Jude	542, 596, 506, 502	542	506, 502, 596
47	Hey Jude	542, 596, 506, 502	542	506, 502, 596
48	Hey Jude	542, 596, 506, 502	542	506, 502, 596
50	Respect	528, 522, 566	528, 522	566
51	What's Going On	531, 529	531	529
52	Hey Jude	542, 596, 506, 502	542	506, 502, 596
53	Respect	528, 522, 566	528, 522	566
54	Hey Jude	542, 596, 506, 502	542	506, 502, 596
55	What's Going On	531, 529	531	529
56	Hey Jude	542, 596, 506, 502	542, 596	506, 502

Table 1: Logging Differences Between Linearizability and Eventual Consistency

Using `"Hey Jude"` as an example, we can observe that under linearizability, all intended values `"542, 596, 506, 502"` were consistently present, while eventual consistency frequently lagged, capturing only the initial value `"542"`. This pattern highlights the delayed propagation in eventual consistency, where replicas eventually converge but may temporarily lack recent updates. On line 56, we can see a partial convergence, where an additional value `"596"` appears, indicating gradual synchronization over time.

5 Conclusion

The implementation of Chordify demonstrates the trade-offs between linearizability and eventual consistency in a decentralized peer-to-peer system. In linearizability, an update is acknowledged only after it has been successfully propagated to all nodes, resulting in higher latency and reduced throughput, while in eventual consistency availability and scalability are prioritized, although with temporary data inconsistencies.

References

[1] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.