Tapestry with Replication

Shaunak Biswas (2022111024) Abhishek Sharma (2022102004)

November 30, 2025

Abstract

This report presents the design, implementation, and evaluation of a Tapestry peer-to-peer overlay network. Tapestry provides a Decentralized Object Location and Routing (DOLR) interface that enables message routing to endpoints based on identifiers rather than physical addresses. Our implementation follows the protocols described in the original Tapestry research while adding mechanisms for fault tolerance. We implement a three-way replication strategy combined with salted hashing to maintain data availability under node churn. We evaluate the system using a benchmarking suite and demonstrate logarithmic hop scaling, balanced load distribution, and low latency for lookup operations in a simulated environment.

1 Introduction

1.1 Background

Distributed applications require infrastructure that can scale across many nodes, tolerate failures, and route messages without depending on physical locations. Tapestry [1] is a structured peer-to-peer overlay that provides efficient routing to nearby copies of objects or services. The system constructs routing tables based on prefix matching, which ensures that queries reach the closest available replica.

1.2 Problem Statement

The base Tapestry protocol defines routing and object publication mechanisms for dense networks. However, sparse networks or networks with high node turnover present challenges. Standard routing paths can break when nodes join and leave frequently. The basic protocol does not specify how to preserve data when nodes exit the network unexpectedly.

1.3 Contributions

Our implementation extends the base Tapestry specification with several features:

- **Three-way replication:** Each object is stored on three nodes. The system selects two backup nodes from the routing table using random selection.
- **Salted publishing:** To address routing failures in sparse networks, we publish object pointers to multiple root nodes. We generate these roots by hashing the object key with different salt values.

- **Graceful handoff:** When nodes leave voluntarily, they transfer their stored data to neighbors before shutting down.
- **Optimized join protocol:** New nodes populate their routing tables more quickly than the standard approach by attempting to connect with all discovered neighbors.

2 System Architecture

The system consists of autonomous nodes that communicate through Remote Procedure Calls. We implement the system in Go (Golang) for concurrency support and use gRPC for communication between nodes.

2.1 Components

The system has three process types:

1. **Tapestry Node:** Handles routing, stores objects, and provides the DOLR interface.
2. **Cluster Manager:** Spawns node processes and aggregates logs during testing.
3. **Frontend Visualizer:** Displays the network topology and node state through a web interface.

2.2 Technology

We use the following technologies:

- **Language:** Go 1.25
- **Transport:** gRPC over TCP with Protocol Buffers
- **Concurrency:** Goroutines for request handling and background tasks
- **Synchronization:** Read-write locks protect shared data structures

3 RPC Interface

We define the communication protocol using Protocol Buffers. The interface separates routing, maintenance, and data operations.

3.1 Core Routing

The **GetNextHop** RPC accepts a target ID and returns the next node in the routing path. It also indicates whether the current node is the root for the target. This design allows iterative routing, where the client follows the path hop by hop.

3.2 Dynamic Membership

These RPCs maintain the network as nodes join and leave:

- **GetRoutingTable:** Returns the routing table to a joining node. This allows new nodes to initialize their state quickly.
- **AddBackpointer:** Notifies a node that another node points to it in its routing table. This builds the reverse graph used for failure recovery.
- **NotifyLeave:** Informs backpointers when a node leaves voluntarily. Receiving nodes remove the departing node from their routing tables.

3.3 Object Location and Routing

These interfaces handle publishing and retrieving object locations:

- **Publish:** Advertises that an object is available at a publisher node. The message routes toward the root. Each node along the path stores a location pointer mapping the object ID to the publisher.
- **Lookup:** Routes toward the root to find location pointers. If any node has a cached pointer for the object, it returns the pointer immediately.
- **Fetch:** Retrieves the actual data from the publisher after a lookup locates it.
- **Replicate:** Pushes backup copies to selected neighbors. This is our extension for fault tolerance.

4 Core Algorithms

Our implementation follows the routing and location mechanisms from Zhao et al. [1]. We add optimizations for network density and stability.

4.1 Addressing and Routing

Tapestry uses 160-bit identifiers generated by the SHA-1 hash function. We represent these identifiers as 40 hexadecimal digits (base 16). Each node maintains a routing table organized by levels and digits.

The entry at level l and digit d points to the closest node that matches the first l digits and has d as digit $l + 1$. For example, level 4, digit 9 for node 325AE stores the closest node whose ID begins with 95AE.

4.2 Routing Logic

To route a message to ID T, the node follows these steps:

1. **Check for exact match:** If the local node ID equals T, routing is complete.
2. **Find shared prefix:** Calculate the length p of the shared prefix between the local ID and T. Look up level p, digit $T[p]$ in the routing table.

3. **Handle missing entries:** If the entry is empty, use surrogate routing. Try digits $T[p]+1, T[p]+2, \ldots$ (modulo 16) until finding a non-empty entry.

We implemented a check to prevent surrogate loops. If the surrogate digit matches the local node's digit at that level, that node declares itself the surrogate root. This guarantees termination.

4.3 Node Join Protocol

When node N joins, it contacts a bootstrap node and routes to its own ID to find its surrogate root S. The node then copies the routing table from S.

We optimize this process through aggressive discovery. After receiving the surrogate's table, the new node attempts to connect with every unique neighbor found in that table. The routing table logic filters these connections, keeping only the neighbors with lowest latency. This approach builds dense routing tables quickly. In our tests, nodes achieve approximately 1.8 hops for networks of 50 nodes immediately after joining.

5 Replication and Fault Tolerance

The base Tapestry protocol has a limitation: if the root node for an object fails, the object becomes unreachable. We address this through replication.

5.1 Three-way Replication

We require each object to have three copies:

1. **Primary storage:** The local node stores the object in memory.
2. **Select backups:** The node selects two random neighbors from its routing table.
3. **Replicate data:** The node sends the object to both backups via Replicate RPCs.
4. **Publish location:** The primary node publishes its location to the network.

This ensures three copies exist on different physical nodes. If the primary fails, the backups preserve the data.

5.2 Salted Hashing

In sparse networks, the routing path to an object's root may be disconnected from a querying node. We address this by publishing to multiple roots.

Instead of publishing to one root at $Hash(Key)$, we publish to three: $Hash(Key + " - 0")$, $Hash(Key + " - 1")$, and $Hash(Key + " - 2")$. This creates three independent routing paths. During retrieval, if the lookup for the base key fails, the client tries the salted variations. This increases the probability of finding a valid pointer in partitioned networks.

5.3 Client Failover

The Lookup RPC returns a list of publishers (primary plus backups) instead of a single result. The client tries to fetch data from the first publisher. If the connection fails, it tries the next publisher in the list. This approach masks node failures from the application.

5.4 Graceful Exit

When a node leaves voluntarily, it follows this protocol:

1. **Stop accepting writes:** The node stops accepting new storage requests.
2. **Transfer data:** For each stored object, the node selects a new neighbor and transfers the data via Replicate.
3. **Notify backpointers:** The node sends NotifyLeave RPCs to all nodes that point to it.
4. **Shutdown:** The process exits only after completing these steps or timing out.

This prevents data loss when nodes cycle out of the network.

6 Engineering Challenges

Running a distributed system simulation on a single machine requires careful resource management.

6.1 Connection Pooling

In the initial implementation, each RPC created a new TCP connection. With 50 nodes performing frequent operations, this exhausted the operating system's available ports and file descriptors. This caused connection failures and node crashes.

We solved this by implementing a connection pool. The client maintains a map of active gRPC connections keyed by address. Before making a request, it checks if a healthy connection exists. If so, it reuses the connection. Otherwise, it creates a new one. The connections remain open for reuse. This reduced RPC overhead from milliseconds to microseconds and eliminated resource exhaustion. The pool is protected by a read-write mutex, allowing multiple concurrent goroutines to read active connections while ensuring safe insertion of new connections.

6.2 Concurrency Control

The Tapestry node handles incoming requests while running background maintenance tasks. We use read-write locks for the routing table and object store. Since routing lookups are much more frequent than table updates, read locks allow high throughput. During the join phase, we use a semaphore to limit concurrent connection attempts. This prevents CPU overload when processing large routing tables.

6.3 Soft State Maintenance

To handle node crashes, the system uses soft state protocols:

1. **Pointer expiration:** Each location pointer includes a timestamp. A background thread removes entries older than a configured timeout.
2. **Republishing:** Nodes periodically re-advertise their stored objects. This refreshes timestamps throughout the network.
3. **Keep-alives:** Nodes periodically ping their neighbors. If a neighbor is unreachable, the node removes it from the routing table and finds a replacement.

7 Visualization

We developed a web-based interface to demonstrate the overlay topology and node state.

7.1 Architecture

The visualization system consists of:

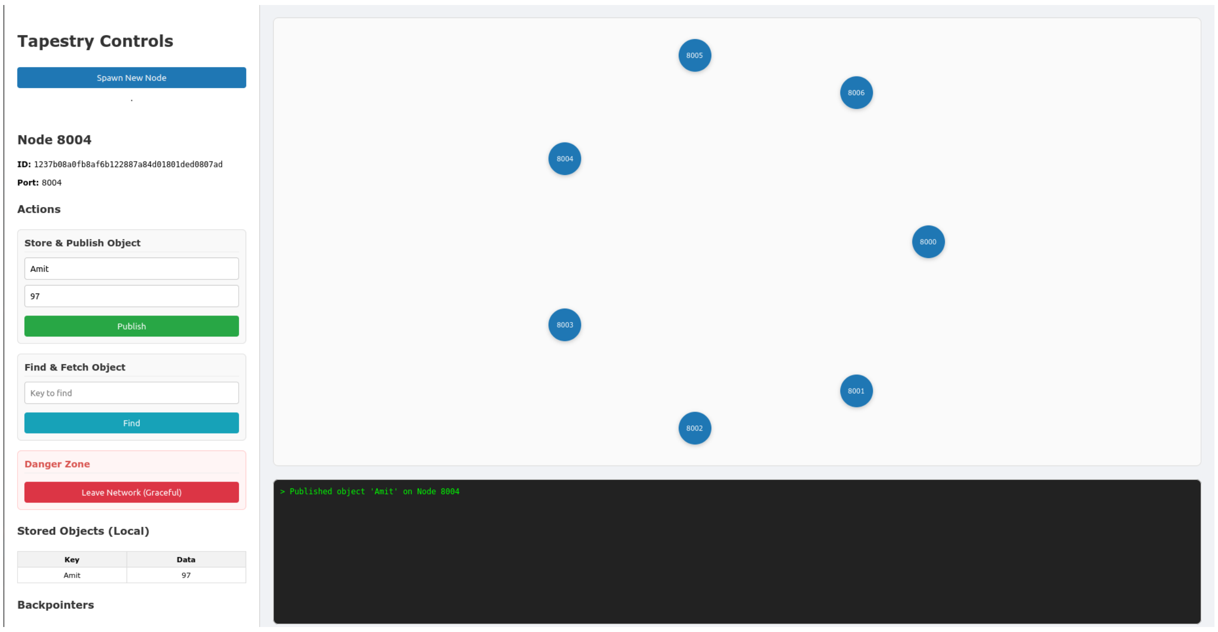
- **Cluster Manager:** Spawns node processes and aggregates logs. It provides an HTTP API to list active nodes.
- **Node HTTP API:** Each node runs an HTTP server that exposes its status, publication, and lookup functions.
- **Web Client:** A browser-based dashboard polls the manager and nodes to render the current state.

7.2 User Interaction

The dashboard allows users to manipulate the network. The following figures show the typical lifecycle of data and nodes.

7.2.1 Publishing Objects

Users select an active node and issue a store command. The interface displays the node's state as it stores data and creates replicas.



ID: 1237b08a0fb8af6b122887a84d01801ded0807ad
Port: 8004

Amit
97

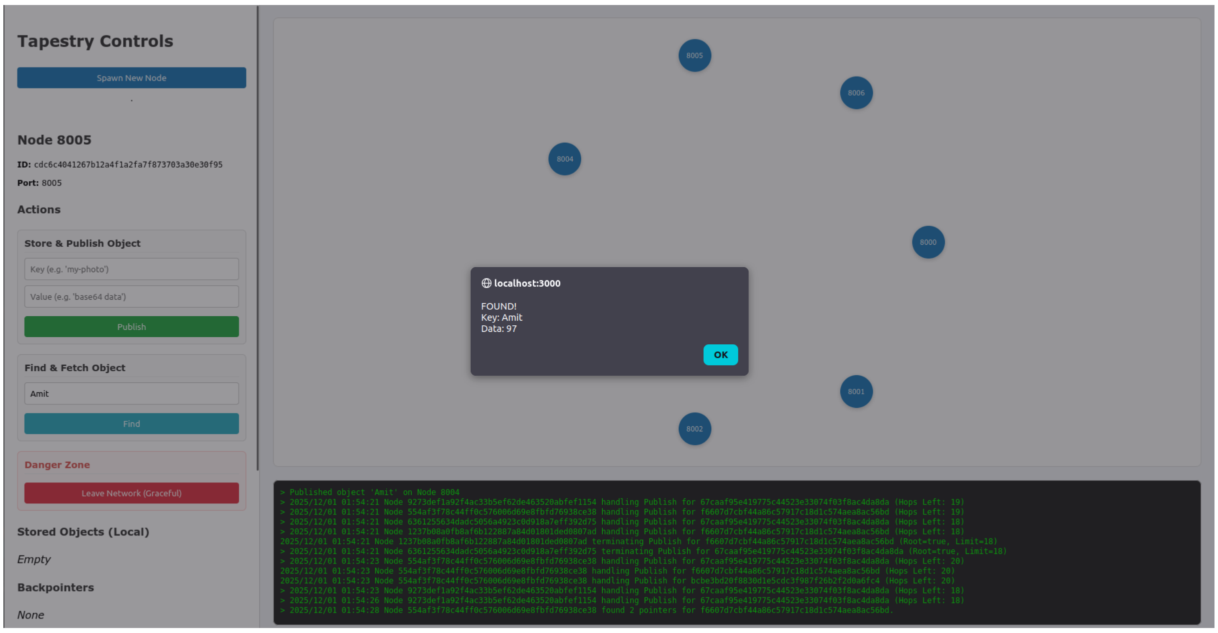
Key	Data
Amit	97

```
> Published object 'Amit' on Node 8004
```

Figure 1: Publication interface showing key-value input and confirmation logs.

7.2.2 Retrieving Objects

Users can select a different node and request a previously published key. The system shows the lookup process and data retrieval.



ID: cdc6c4041267b12a4f1a2fa7f873703a30e30f95
Port: 8005

Amit

Empty

None

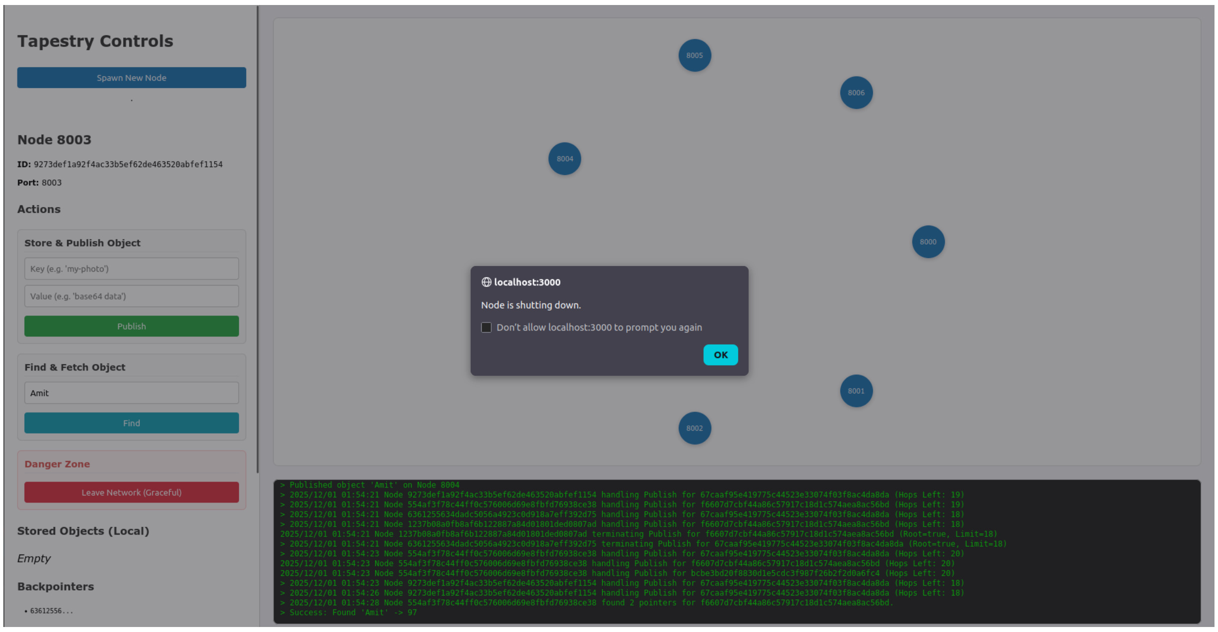
FOUND!
Key: Amit
Data: 97

```
> Published object 'Amit' on Node 8004
> 2025/12/01 01:54:21 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 19)
> 2025/12/01 01:54:21 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 19)
> 2025/12/01 01:54:21 Node 6361255634dadc5056a4923c0d918a7eff392d75 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:21 Node 1237b08a0fb8af6b122887a84d01801ded0807ad handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 18)
2025/12/01 01:54:21 Node 1237b08a0fb8af6b122887a84d01801ded0807ad terminating Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Root=true, Limit=18)
> 2025/12/01 01:54:21 Node 6361255634dadc5056a4923c0d918a7eff392d75 terminating Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Root=true, Limit=18)
> 2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 20)
2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 20)
2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for bcbe3bd20f8830d1e5cdc3f987f26b2f2d0a6fc4 (Hops Left: 20)
> 2025/12/01 01:54:23 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:26 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:28 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 found 2 pointers for f6607d7cbf44a86c57917c18d1c574aea8ac56bd.
```

Figure 2: Fetch interface showing successful object location and retrieval.

7.2.3 Node Exit

The interface provides a button to trigger graceful node departure. Users can observe the node disappear and verify that data handoff occurred.



ID: 9273def1a92f4ac33b5ef62de463520abfef1154
Port: 8003

• 63612556...

Node is shutting down.

```
> Published object 'Amit' on Node 8004
> 2025/12/01 01:54:21 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 19)
> 2025/12/01 01:54:21 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 19)
> 2025/12/01 01:54:21 Node 6361255634dadc5056a4923c0d918a7eff392d75 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:21 Node 1237b08a0fb8af6b122887a84d01801ded0807ad handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 18)
2025/12/01 01:54:21 Node 1237b08a0fb8af6b122887a84d01801ded0807ad terminating Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Root=true, Limit=18)
> 2025/12/01 01:54:21 Node 6361255634dadc5056a4923c0d918a7eff392d75 terminating Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Root=true, Limit=18)
> 2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 20)
2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for f6607d7cbf44a86c57917c18d1c574aea8ac56bd (Hops Left: 20)
2025/12/01 01:54:23 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 handling Publish for bcbe3bd20f8830d1e5cdc3f987f26b2f2d0a6fc4 (Hops Left: 20)
> 2025/12/01 01:54:23 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:26 Node 9273def1a92f4ac33b5ef62de463520abfef1154 handling Publish for 67caaf95e419775c44523e33074f03f8ac4da8da (Hops Left: 18)
> 2025/12/01 01:54:28 Node 554af3f78c44ff0c576006d69e8fbfd76938ce38 found 2 pointers for f6607d7cbf44a86c57917c18d1c574aea8ac56bd.
> Success: Found 'Amit' -> 97
```

Figure 3: Graceful exit showing data handoff before node termination.

8 Evaluation

We developed a benchmarking tool that runs simulations with 20 to 50 nodes. The tool supports different test modes to measure specific performance aspects.

8.1 Routing Efficiency

We measured the average hop count required to locate random IDs as network size increased from 10 to 50 nodes.

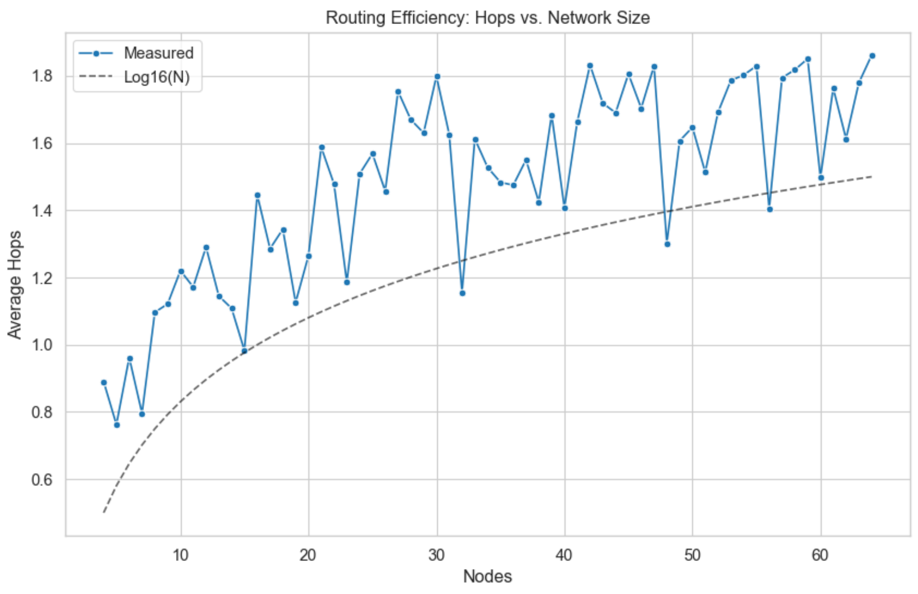


Figure 4: Average hop count versus network size. The measured values closely track the theoretical $O(\log_{16} N)$ bound.

8.2 Load Distribution

We distributed 2000 objects across 16 nodes to evaluate the uniformity of object placement. We analyzed the distribution using four visualizations.

8.2.1 Distribution Histogram

Figure 5 shows the count of objects stored per node.

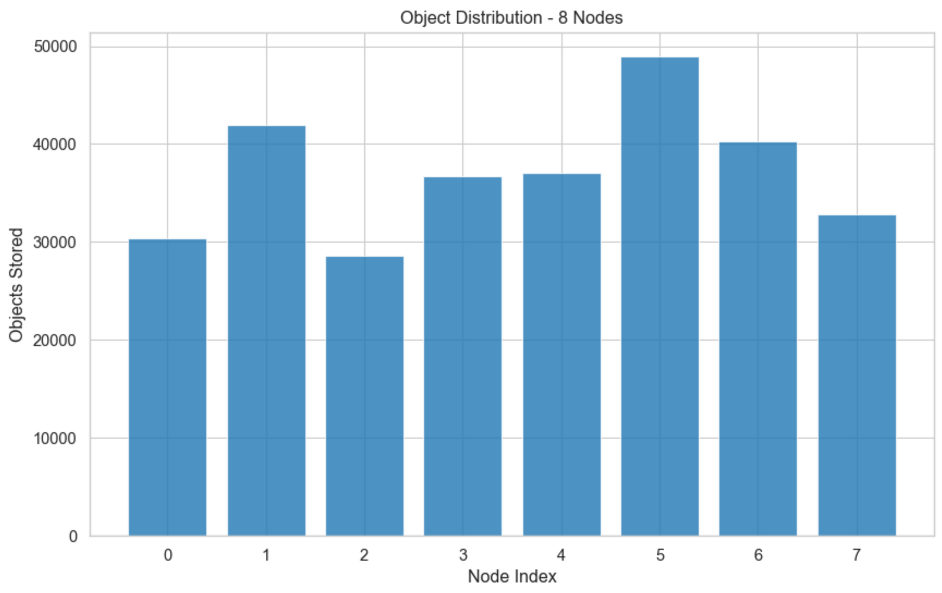


Figure 5: Object distribution across 8 nodes. The relatively flat distribution indicates no single node acts as a hotspot.

8.2.2 Fairness Metrics

We tracked the Gini coefficient, Jain's fairness index, and coefficient of variation as objects were added to the system.

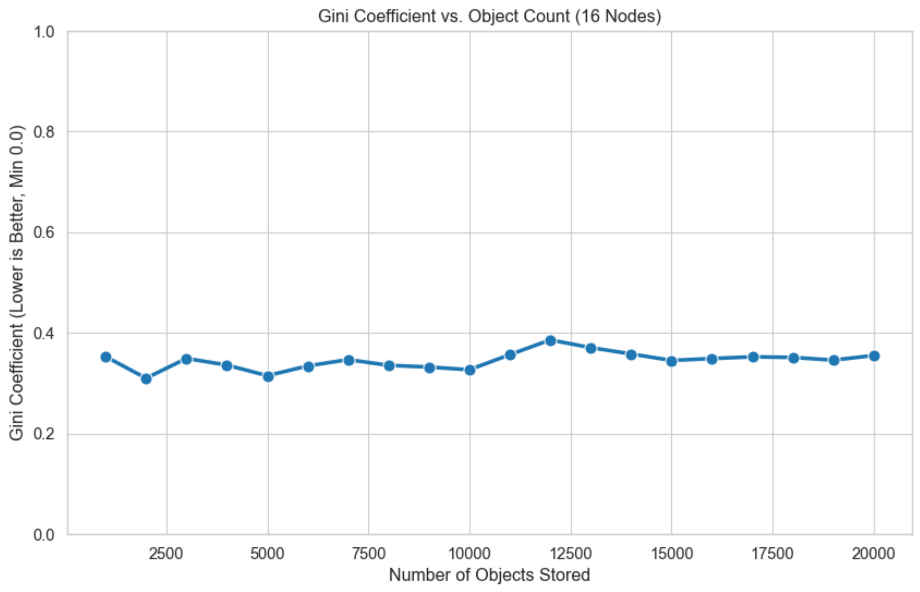


Figure 6: Gini coefficient convergence. The coefficient stabilizes below 0.4, indicating balanced load distribution.

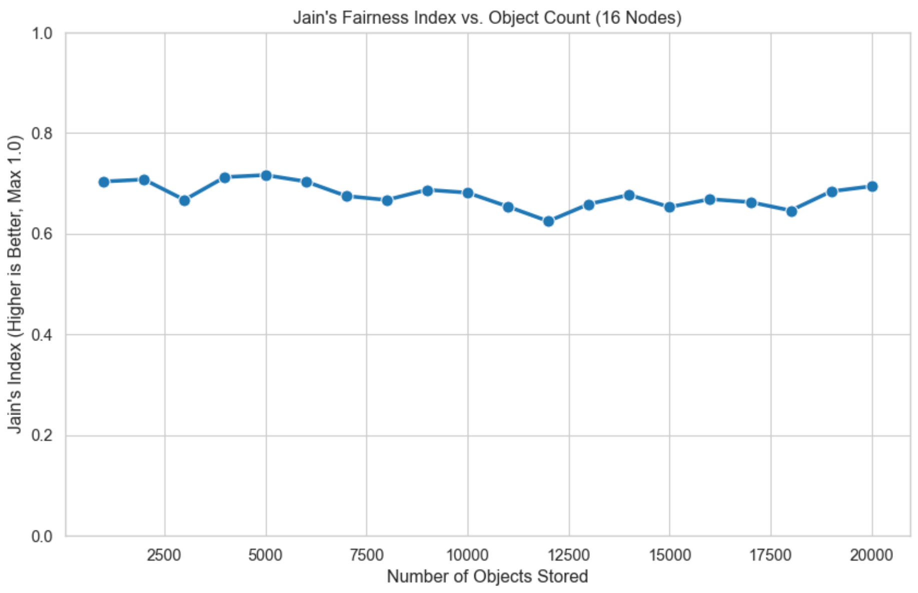


Figure 7: Jain's fairness index above 0.7 confirming minimal load variation between nodes.

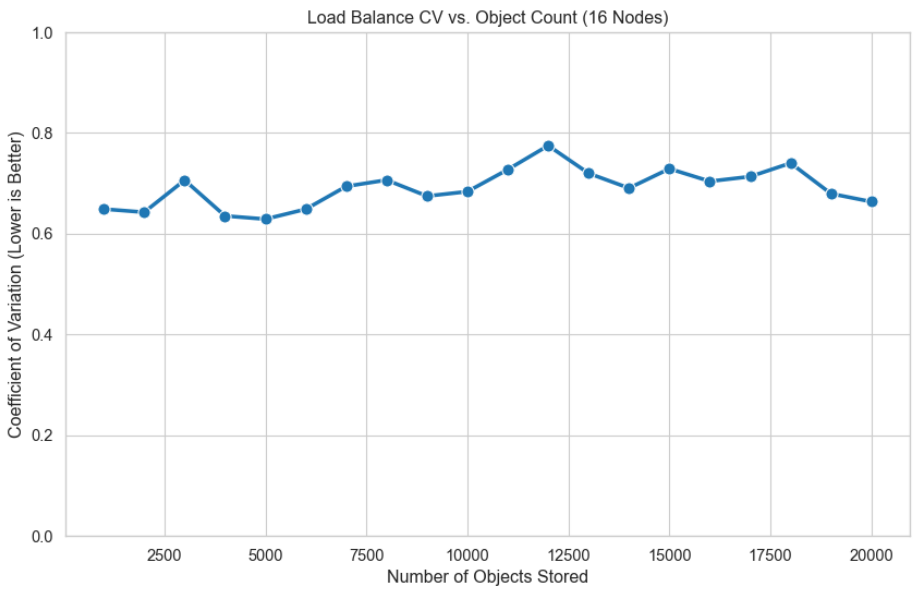


Figure 8: Coefficient of variation showing minimal load variation between nodes.

8.3 System Performance

We tested the cluster under high concurrency. Fifty workers performed mixed read and write operations simultaneously.

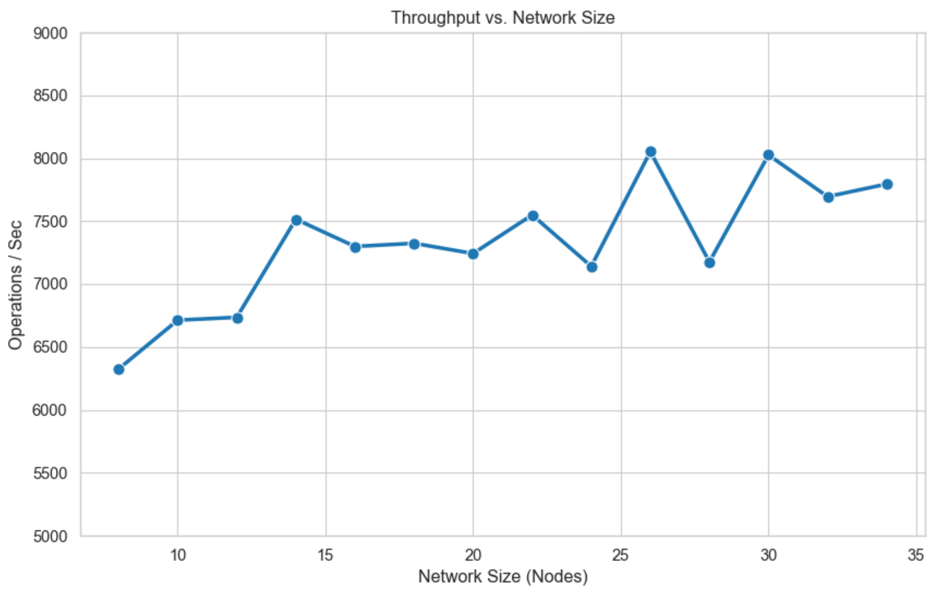


Figure 9: System throughput

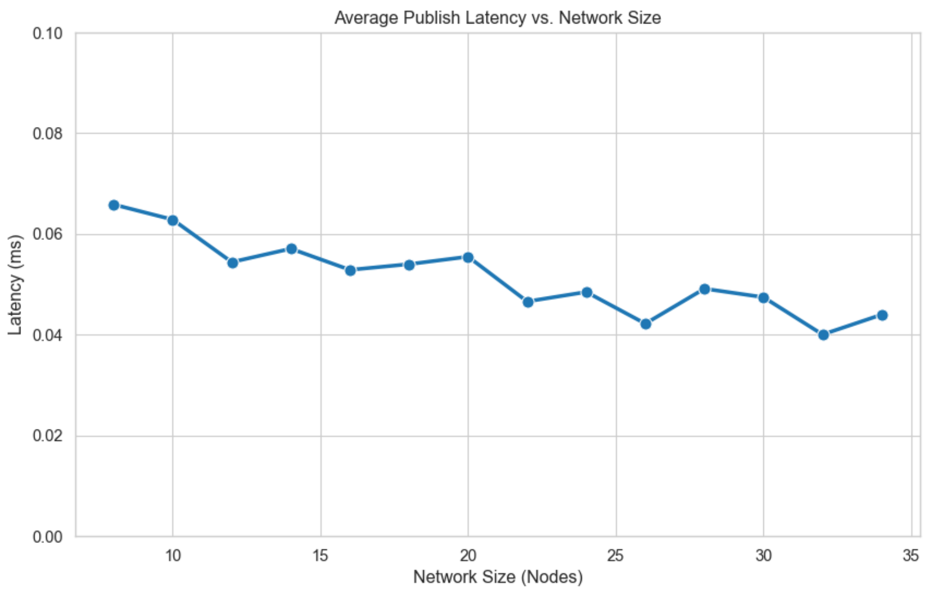


Figure 10: Publish latency averaging 0.05ms.

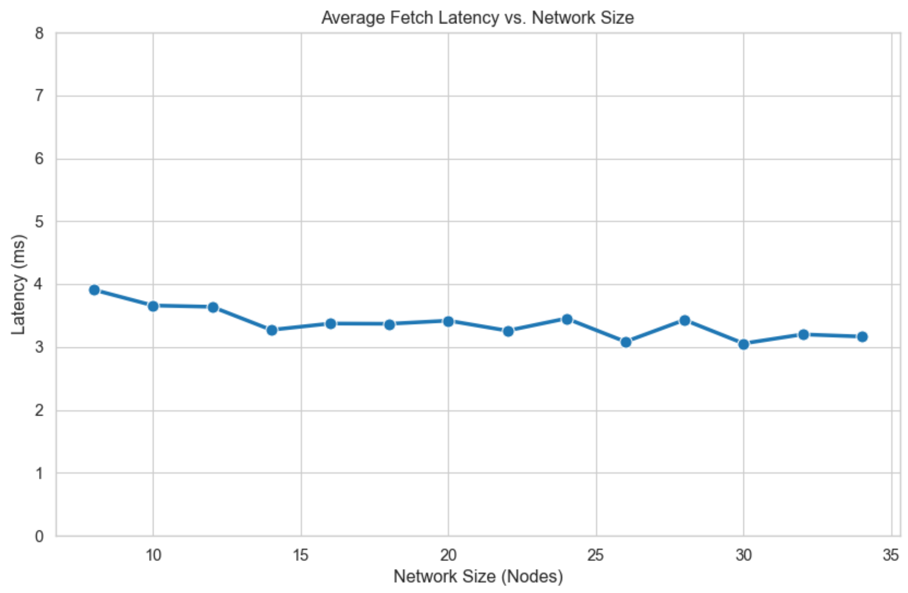


Figure 11: Fetch latency averaging 3.5ms. The lower latency confirms that queries often find cached pointers at intermediate hops.

8.4 Fault Tolerance

We tested the system's ability to handle failures by randomly terminating nodes during operation.

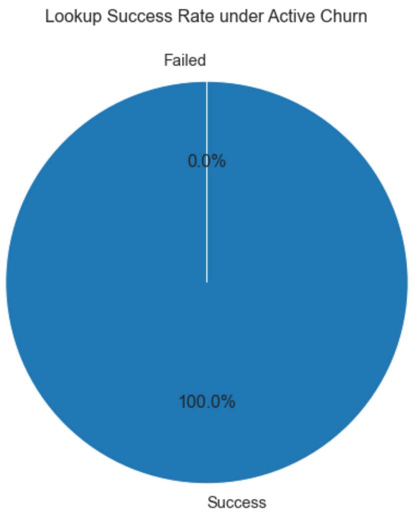


Figure 12: Lookup success rate under node churn. Despite continuous failures, the system maintained 100% success rate. This validates that salted replication masks node failures effectively.

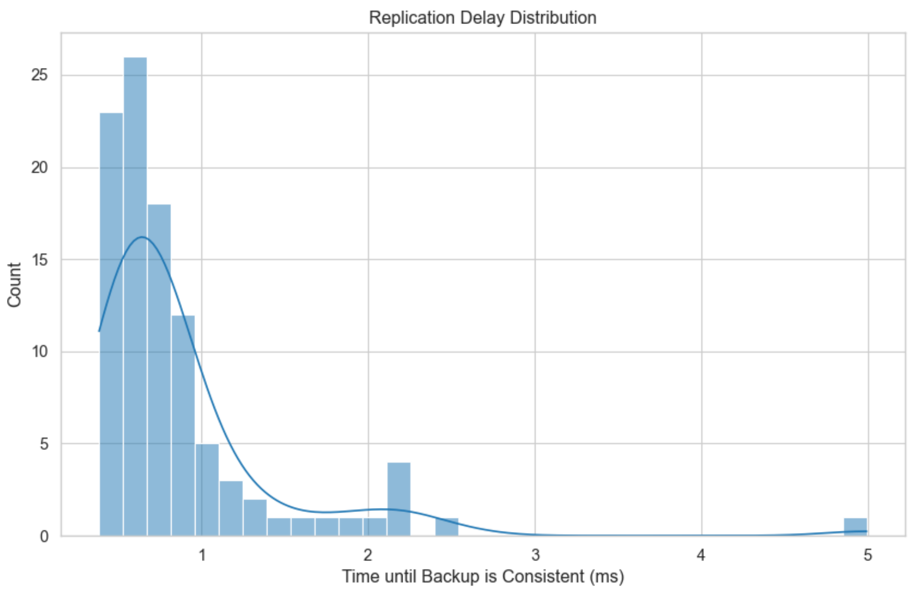


Figure 13: Replication delay showing most replications complete within 3ms. This ensures a small vulnerability window where data exists on only one node.

9 Conclusion

We have implemented a resilient Tapestry overlay network in Go. Our extensions to the base protocol include salted replication and graceful handoff. These mechanisms address the vulnerabilities of sparse networks. Our evaluation demonstrates that the system achieves logarithmic scalability, balanced load distribution, and high availability under node churn. The engineering optimizations were necessary to achieve stable performance in a resource-constrained simulation environment.

A Analysis of Routing Depth

Claim: The expected length of the longest common prefix among n random strings is $O(\log_{|\Sigma|} n)$, where Σ is the alphabet.

Proof: Consider strings with alphabet size $B = |\Sigma|$. Let X represent the length of the longest common prefix between two random strings. The probability that two strings match up to length k is:

$P(X \geq k) = \frac{1}{B^k}$

For n random strings, let M_n be the maximum prefix length. For $M_n \geq k$, at least one pair must share a prefix of length k. The number of pairs is $\binom{n}{2} \approx \frac{n^2}{2}$. Using the union bound:

$P(M_n \geq k) \leq \frac{n^2}{2} \cdot \frac{1}{B^k}$

To find k where this probability is small (less than 1):

$\frac{n^2}{2B^k} \approx 1 \implies B^k \approx n^2 \implies k \approx 2 \log_B n$

Therefore, the expected depth in the Tapestry mesh is proportional to $\log_B n$. For base 16, this explains the observed logarithmic scaling where hops grow slowly with cluster size.

References

[1] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, Jan. 2004.

[2] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," *UC Berkeley Technical Report*, UCB/CSD-01-1141, April 2001.