Decentralised overlay network - Tapestry

Project Report - Team 26

Tuesday, 15 April 2025

Rohan Sridhar (2022101042) Mohammed Faisal (2022101101) Shreyansh (2022111002)

1. Problem Statement

This project implements a decentralized overlay network inspired by Tapestry, enabling scalable, fault-tolerant routing with prefix-based matching. It supports efficient message routing, dynamic node membership, and resilience to node failures for fast lookups in large-scale distributed systems.

2. Framework and Technologies

- **Programming Language:** Go (Golang)
- **Communication Protocol:** gRPC
- **Data Structures:** Prefix-based routing tables and Back pointers
- **Hashing Mechanism:** FNV-A1 hash
- **Storage (Resources):** In-memory storage for node states

2.1. Reasoning Behind Technology Choices

- **Go (Golang) & gRPC**: gRPC provides high-performance, remote procedure calls (RPCs) with built-in support for error handling and serialization using Protocol Buffers. All of us have experience with gRPC in Go from the Assignment.

- **Prefix-Based Routing Tables**: The original Tapestry paper implements a prefix based routing system, using SHA-1, so we are implementing similar to that.

3. Project Functionalities

3.1. Node Insertion:

- Nodes can join the network and establish connections with existing nodes (Populates routing tables and back pointers).
- Each node generates a unique random 64-bit ID using the FNV-A1 hash function.

3.2. Routing:

- Routing method is used to find the root node corresponding to a given key (which can be either Node ID / Object hash) in $O(1)$ hops.

- The routing method uses the prefix-based routing algorithm to find the node responsible for the given key.
- The result is the port of the node with maximum common prefix length with the key.

3.3. Node Deletion:

- Nodes can leave the network gracefully.
- The routing tables and back pointers are updated accordingly in $O(\log^2 n)$ messages.
- The exiting node won't be accessible after exit.

3.4. Add Object:

- Objects are (key, value) pairs (like a **distributed hash table**) that are stored in the network.
- Nodes can add objects to the network, which can then be located from anywhere.

3.5. Object Publish/Unpublish/Find:

- Nodes can publish objects to the network
- Any node can access the object value using their keys after they are published
- An object can be unpublished from anywhere in the network

3.6. Fault Tolerance:

- The system can handle node failures and reconfigure routing tables.
- Even after a node goes down unexpectedly, the system can still function.

3.7. Redundancy:

- The system maintains redundancy by keeping multiple copies of objects, so that even after a node goes down, its objects are accessible from redundant resources.

4. Implementation Details

4.1. Radix, Hash length considerations

- Original Tapestry implementation uses base 16 with a 160-bit hash, which gives 40 digits.
- To simplify the implementation, a 64-bit hash is used with base 4, to give 32 digits.

4.2. Node Insertion:

- New nodes are randomly assigned 64-bit IDs, and are inserted with the help of a bootstrap node.
- The bootstrap node routes the new ID to a (unique) root, that has the **longest common prefix** with the new ID.
- The Routing Table of the new node is obtained by copying the routing table of the node for levels < longest common prefix.
- Higher levels are filled with a **multicast** operation.
- Random assignment of IDs gives $O(\log n)$ nodes that are contacted in the multicast (refer Appendix).

4.3. Routing:

- A prefix-based routing algorithm is used, very similar to a search on a Trie. The Routing tables maintained make up a **Distributed Trie**.
- Since the IDs are 64-bit hashes, the trie descent makes a constant number of hops ($= \log_B H$), where H is the size of the space of IDs, B is the radix of the trie.

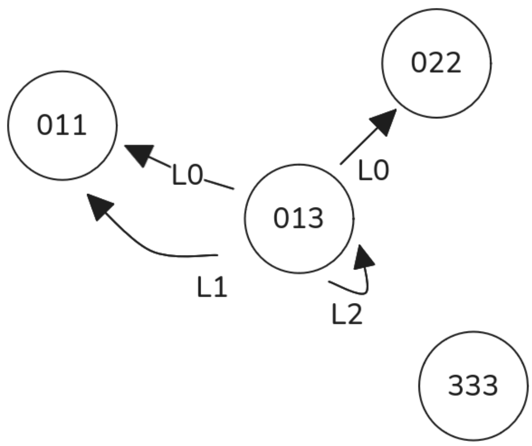

Figure 1: Example connections of a node with ID 013. L_i is the a connection at level i. (Note all connections are not shown for brevity)

4.4. Node Deletion:

- For graceful deletion, a node first identifies the closest node to its own ID, which then serves as its replacement during the deletion process.

- The replacement node updates routing tables and fills any gaps left by the departing node, ensuring continuity and network integrity during deletion.
- Node deletion involves three key steps: **RTUpdate**, to update routing tables; **BPUpdate**, to update backpointers; and **BPRemove**, to remove the departing node from others' backpointers.
- Random assignment of IDs gives an expected $O(\log n)$ nodes to be updated after deletion (refer Appendix).

4.5. Add Object:

- Objects can be inserted into the network from anywhere (like a **distributed hash table**)
- Ensures redundancy by invoking the `StoreObject()` RPC on upto two other nodes (giving a redundancy factor of 3), selected by scanning the routing table. These nodes then replicate the object in their respective local maps.

4.6. Object Publish:

- The `Publish()` function is periodically invoked every 5 seconds in a separate `go` routine.
- It first identifies the root node using `FindRoot()`, which internally calls `Route()`.
- The function then calls the `Register()` RPC on the root node to register itself as a publisher of the object.
- This repeated invocation provides fault tolerance, ensuring that in the event of a failure, a new root is automatically assigned and updated.

4.7. Object Unpublish:

- Removes the object from the node's local `Objects` map.
- Sends an `Unregister()` RPC to the root node, which in turn instructs all other publishers of the object to remove it from their local storage as well.
- This ensures consistency across the system while preserving the desired redundancy.

4.8. Find Object:

- Retrieves the object specified by the user by contacting one of its active publishers.
- Calls the `LookUp()` RPC on the root node to obtain the port number of a live publisher for the requested object.
- It then calls the `GetObject()` RPC on the selected publisher to fetch the object.
- If the object is successfully found, it is returned. Otherwise, a message indicating the absence of the object is displayed.

5. Testing

5.1. Stress Testing

The methods Route(), Insert(), Delete(), Publish(), FindObject(), and Unpublish() were thoroughly validated through automated tests. These tests were committed to a separate git repository and can be reviewed by checking out the testing fork.

5.2. Performance Scaling Results

Reponse times for `Route()` and `Insert()` calls were measured for various sizes of the network.

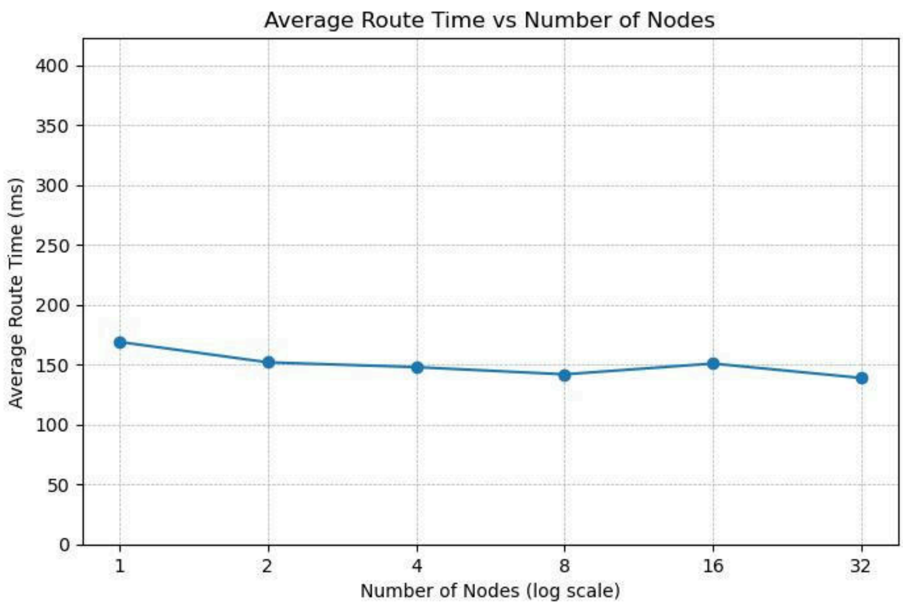

Figure 2: Reponse time for `Route()`

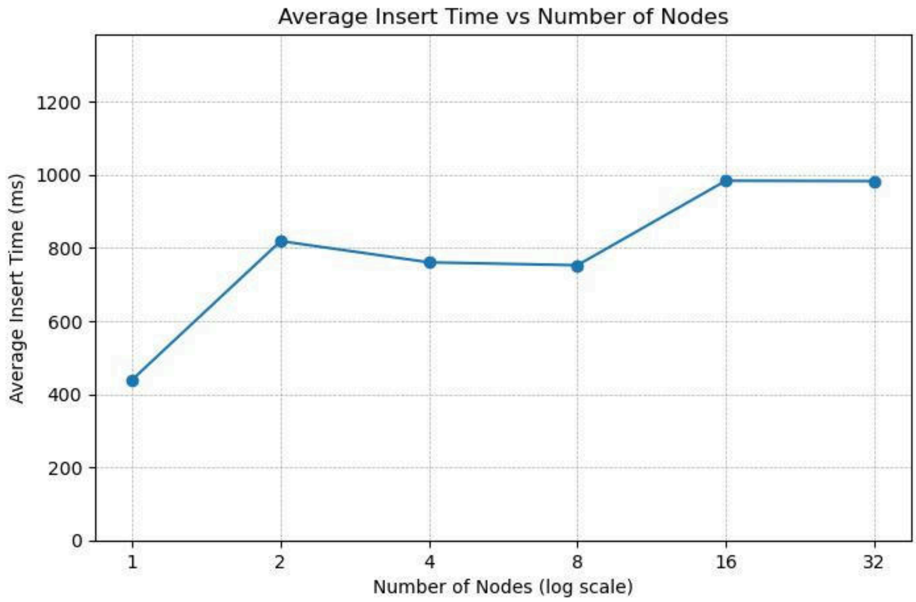

Figure 3: Reponse time for `Insert()`

The plots clearly indicate that `Route()` operates in constant time, whereas `Insert()` exhibits a growth trend that falls between linear and quadratic on the log plot, giving a complexity of $O(\log^2 n)$

Appendix

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

Proof: To get an upper bound on the length of the longest common prefix, it is convenient to assume the strings have inifinite length.

Consider just one string, the longest common prefix (*lcp*) with another random string follows a geometric distribution.

$$\Pr\{\text{lcp} \geq m\} = \frac{1}{|\Sigma|^m}$$

As atleast the first m characters must match.

This is a geometric distribution. Thus the final quantity is the max of n i.i.d. geometric variables (L_i):

$$\Pr\{L_i < m\} = 1 - \frac{1}{|\Sigma|^m}$$

Thus,

$$\Pr\{\max(L_1, ...L_n) < m\} = \prod \Pr\{L_i < m\}$$

$$= \left(1 - \frac{1}{|\Sigma|^m}\right)^n$$

$$\Rightarrow \Pr\{\max(L_1, ...L_n) \geq m\} = 1 - \left(1 - \frac{1}{|\Sigma|^m}\right)^n = f(m)$$

The required probability is:

$$\sum_{i=1}^{\infty} f(i)$$

$$\approx \int_1^{\infty} f(x)dx$$

By approximating

$$1 - \frac{1}{|\Sigma|^x} \approx 1 - e^{-x\left(1 - \frac{1}{|\Sigma|}\right)}$$

And using the standard integral

$$\int_0^1 \frac{1-(1-u)^n}{u} du = \frac{1}{1} + \frac{1}{2} + ... \frac{1}{n} \approx \ln n$$

Gives us $E[\text{lcp}] \approx \log_{|\Sigma|}(n)$