Simplified Gossip Protocol Visualizer

Project Report

Student Names: Arina Zimina, Karina Siniatullina, Egor Agapov

Date: 17.05.2025

1 Introduction

The **Gossip protocol** is a protocol that allows designing highly efficient, secure and low latency distributed communication systems (P2P). The inspiration for its design has been taken from studies on epidemic expansion and algorithms resulting from it.

The gossip protocol is very important in distributed systems because it helps **nodes** (computers, servers, or processes) **share information quickly**, **reliably**, and **without a central coordinator**. Here's why it's critical:

- **Scalability:** Gossip scales really well — even with thousands of nodes — because each node only talks to a few others at a time.
- **Fault tolerance:** Nodes can fail or go offline, but gossip ensures the system can still spread information without depending on any single point.
- **Eventually consistent:** Perfect synchronization is hard in distributed systems, so gossip allows nodes to eventually reach the same state without requiring immediate consistency.
- **Low overhead:** The communication is lightweight and randomized, so it doesn't overload the network.

A few important use cases for gossip protocols:

1. **Membership tracking:** Nodes use gossip to find out which other nodes are alive, dead, or new in the system (example: Amazon DynamoDB).
2. **State dissemination:** Systems like Apache Cassandra use gossip to spread metadata (like schema changes, load info) across all nodes.
3. **Failure detection:** If a node crashes, gossip helps quickly alert the rest of the system so they can reroute traffic or rebalance data.
4. **Blockchain and cryptocurrency networks:** In Bitcoin, Ethereum, and other decentralized networks, gossip spreads new transactions and blocks across peers.

2 Goal

The main goal of this project was to build a visualizer for a simplified gossip protocol to demonstrate how information spreads across nodes in a distributed system. The system simulates message dissemination, convergence detection, and metric monitoring in real time.

Responsibilities:

- **Arina Zimina:** Python backend development using Flask for API creation and HTTP requests processing, gossip protocol implementation using threading for asynchronous message processing, Docker and Docker Compose setup and configuration for containerization and orchestration of services, Prometheus integration for collecting and storing metrics about the system performance.

- **Karina Siniatullina:** Interface design and styling via HTML and CSS, dynamic updating and handling of events and animations in JavaScript, display of nodes and links between them, as well as animation of message propagation over the network via Vis-network library, plotting of graphs and charts via Chart.js library
- **Egor Agapov:** Creating a CI/CD pipeline in GitHub Actions using YAML configuration to run the test, and writing a project report.

3 Methodology

The project follows a microservice-based approach, with each gossip node implemented as an independent service.

System Architecture:

- **Gossip Nodes:** Independent `Flask` servers communicating via gossip protocol.
- **Nginx:** Acts as a load balancer (round-robin) and routes client messages to one of the nodes.
- **Prometheus:** Collects metrics from nodes for graphing.
- **Frontend:** `HTML/JS` interface using `Vis-network` and `Chart.js` to visualize the gossip state and convergence.

Infrastructure Stack:

- **Backend:** `Python 3`, `Flask`, `Prometheus` client, threading
- **Frontend:** `HTML/CSS`, `JavaScript`, `Chart.js`, `Vis-network`
- **Infrastructure:** `Nginx`, `Prometheus`, `Docker`

Planned Flow:

1. Client sends a message via the UI.
2. `Nginx` forwards it to a random gossip node.
3. The node starts spreading the message using gossip algorithm.
4. Each node synchronizes with neighbors until convergence.
5. `Prometheus` periodically asks for metrics from nodes for dynamic graphing.

4 Development of solution

Gossip Node (`gossip_node.py`)

- Receives messages via `/data POST` endpoint.
- Periodically gossips with random peers via `HTTP` requests.
- Tracks state in-memory (`my_data`, `log_data`).
- Records metrics: number of messages, start/end time.

Log Viewer (`log_viewer.py`)

- Web-based interface for observing node states.
- Sends messages to the system and shows convergence chart.
- Includes restart button to manually reset the system state.

Web page interface (`logs.html`)

- Main `HTML` template rendered.
- Contains the message input form for sending new messages.
- Includes containers for:
  - Network visualization (`div#network-container`).
  - Message count line chart (`canvas#metricsChart`).
  - Propagation time bar chart (`canvas#timeHistogram`).
- Loads `vis-network` and `Chart.js` from CDN.
- Loads `main.js` for all frontend logic.
- Provides the initial structure for all dynamic visualizations.

Styles (`style.css`)

- Styles the layout and appearance of all UI elements.
- Uses `CSS Grid` to arrange the network and charts.
- Styles the message input form, buttons, and controls.
- Styles the network visualization container.
- Provides responsive design for mobile and desktop screens.
- Adds color coding, animations, and visual feedback for user actions.

Web page functionality (`main.js`)

- Initializes and manages the network visualization using `vis-network`.
- Initializes and updates the message count line chart and propagation time bar chart using `Chart.js`.
- Sends new messages to the backend via `/send-message (POST)`.
- Periodically fetches `/metrics (GET)` to update network state and charts.
- Handles user interactions, error display, and UI feedback.

API Overview

- `/data` (POST) - send new message
- `/state` (GET) - fetch current node state
- `/log` (GET) - fetch gossip log
- `/metrics` (GET) - Prometheus metrics
- `/send-message` (POST) - send a new message to the system

Testing

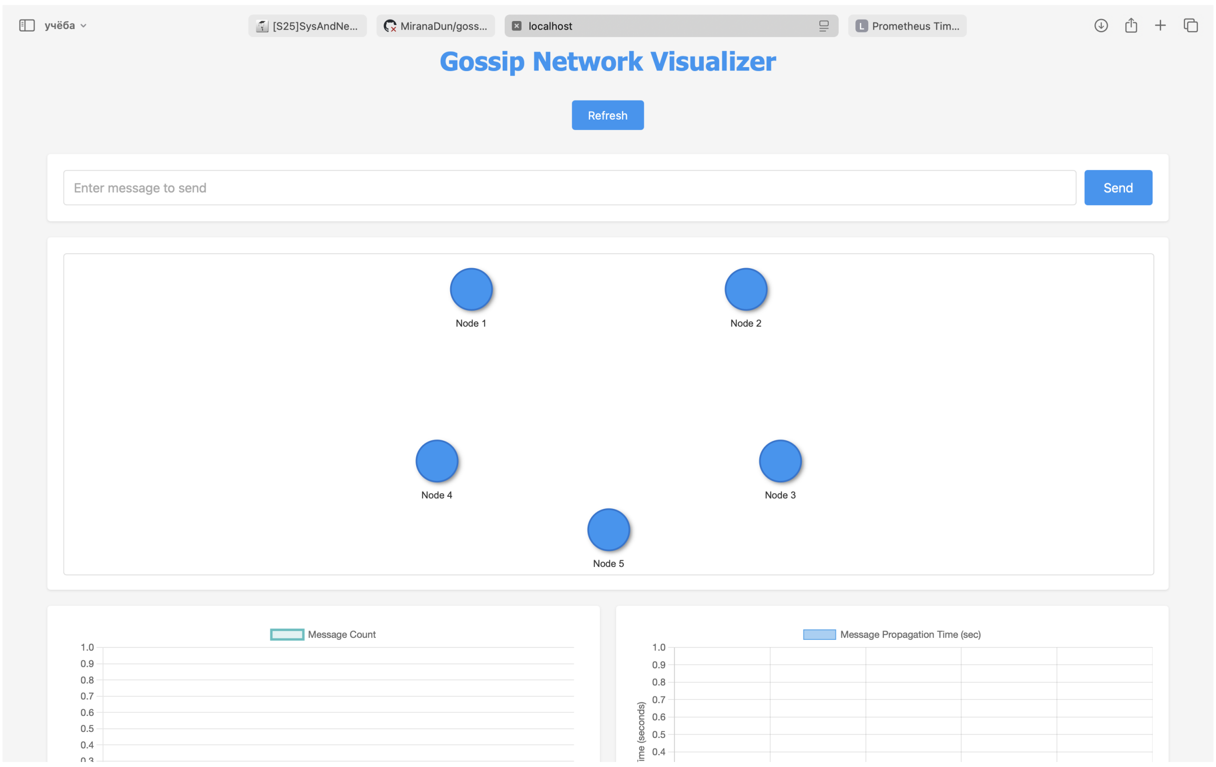
We wrote unit test to verify:

- Nodes are active and respond correctly
- Messages propagate correctly via gossip
- State convergence is reached
- `Prometheus` metrics are accurate

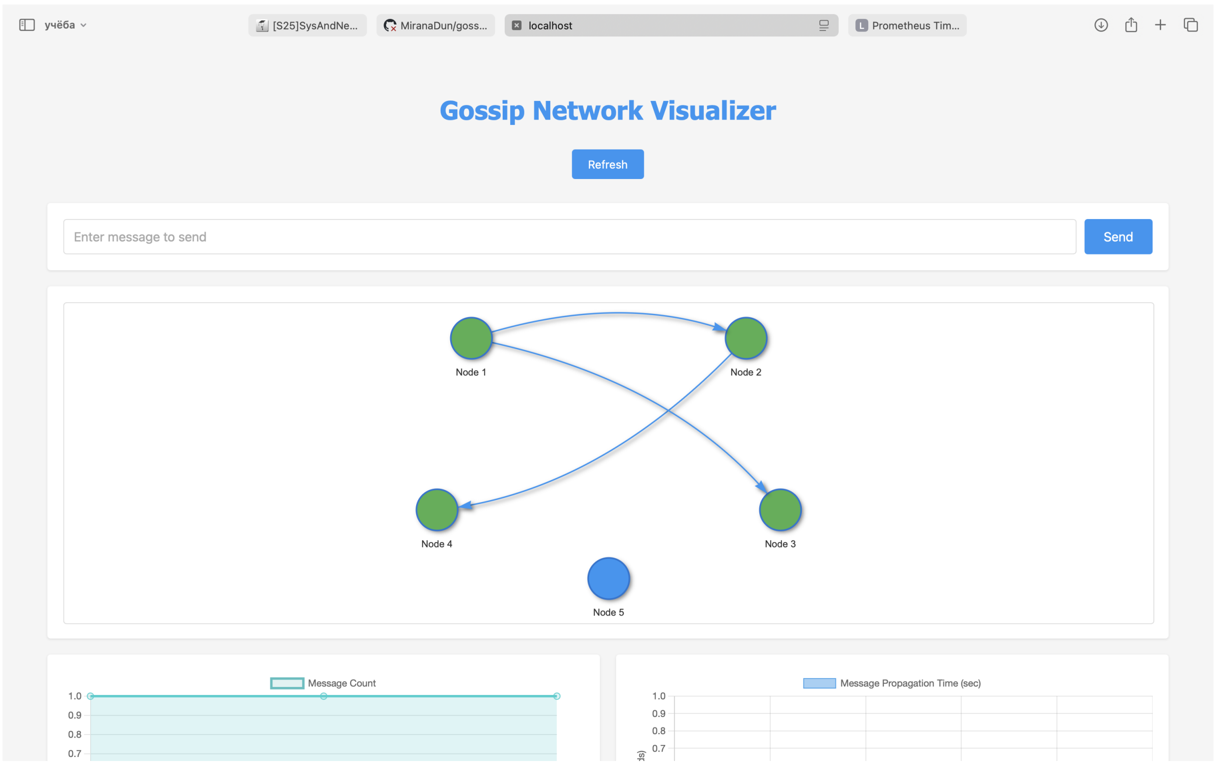
Additionally, we used **GitHub Actions** to automate testing. A `YAML` workflow file was created to spin up multiple gossip nodes using Docker, send test messages through the system, and validate the propagation and convergence behavior. This ensured that the entire system worked correctly in a clean and reproducible environment.

5 Results' overwiew

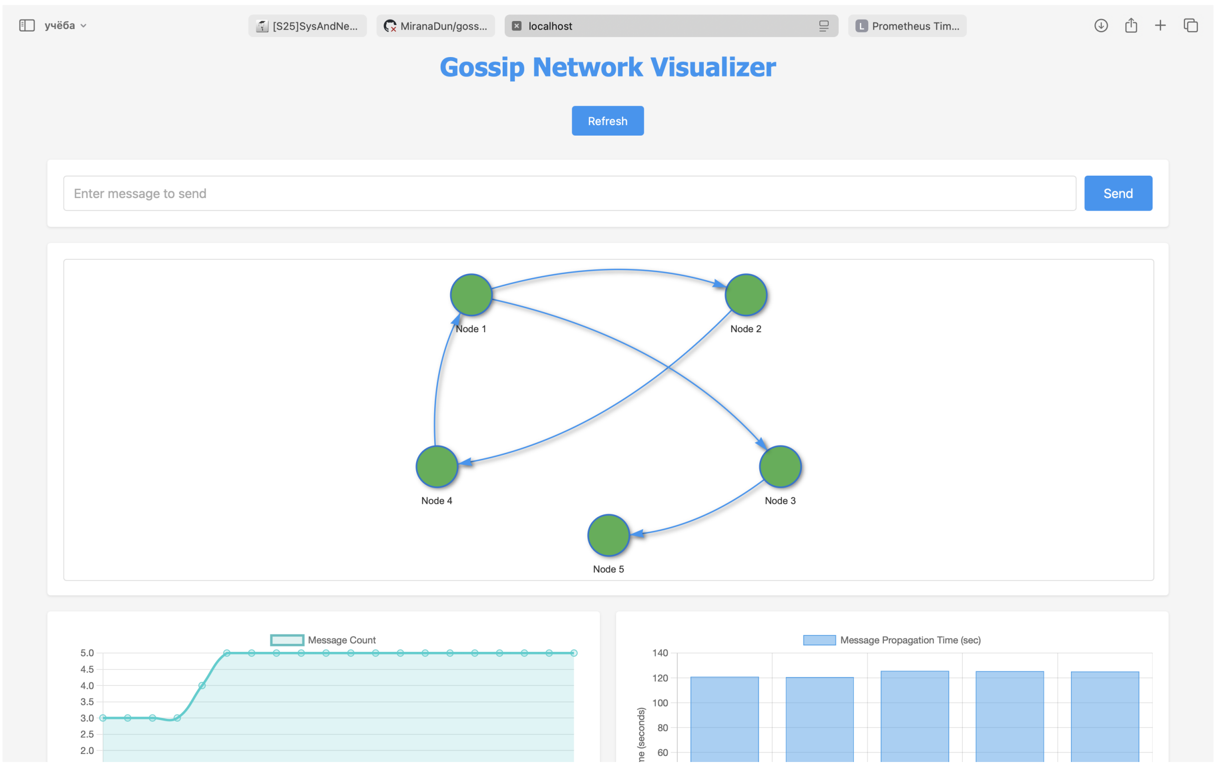
Visualization Start Page:

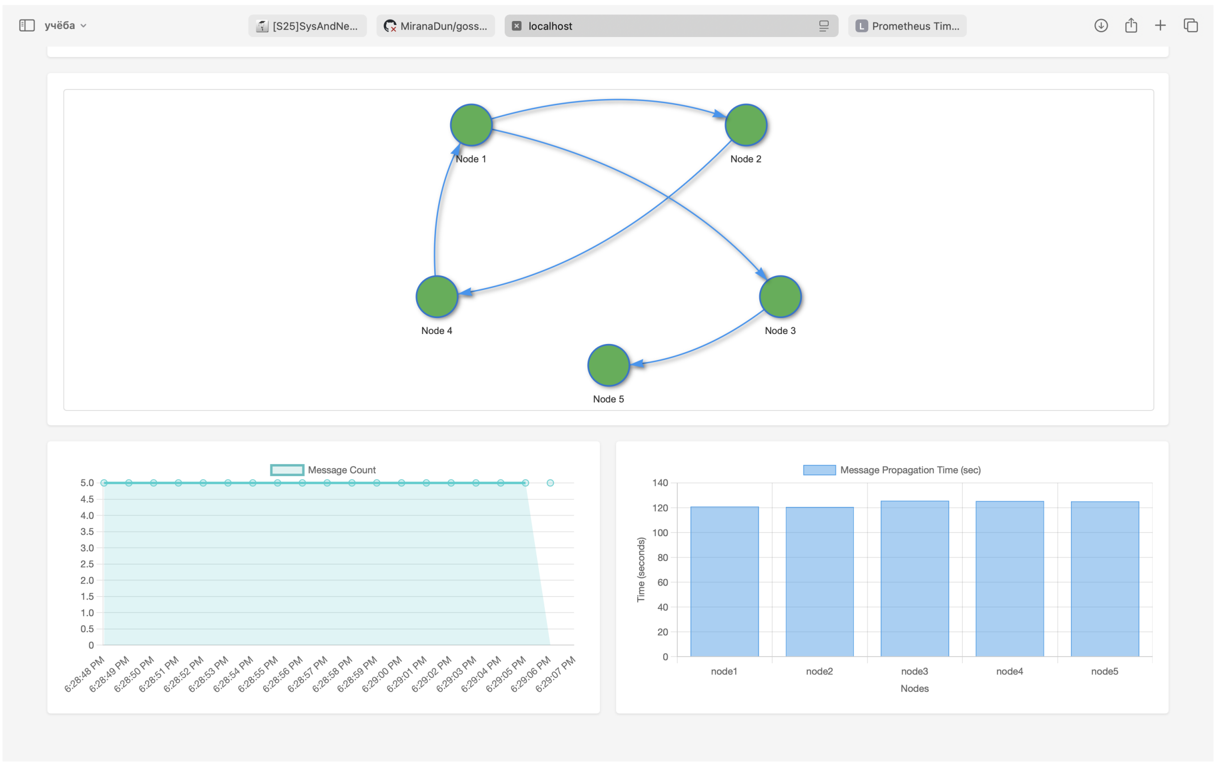


The process of synchronizing a message between nodes:



End of message synchronization and metrics output:





6 Difficulties faced / New skills acquired

Difficulties

- Challenges with handling timestamps for convergence graph generation.
- Difficulties in choosing appropriate delays between message transmissions to maintain synchronization accuracy.
- The convergence graph sometimes does not update automatically — we had to add a manual restart button.
- When sending a new message, the previous graph is not cleared.
- In general — debugging asynchronous behavior and visualizing the state of nodes was quite challenging.

New Skills Acquired

- Working with `Prometheus` and collecting real-time metrics.
- Developing asynchronous microservices using `Flask`.
- Integrating graphs and visualizations with backend APIs.
- Configuring `Nginx`.

7 Conclusion

This project helped us better understand the core mechanics behind gossip-based communication and how it applies in modern distributed systems. We successfully implemented a simplified but functional version of such a system, including message dissemination, convergence detection, and real-time metric monitoring.

Despite a few unresolved issues in the UI refresh logic, the system works as intended and can demonstrate the convergence behavior of gossip protocols. In the future, we would like to improve the UI responsiveness and enhance metric reporting for better user experience and performance evaluation.

Code Repository and Resources:

- **Repository:** `https://github.com/MiranaDun/gossip-app`
- **Demo:**