

Simplified Gossip Protocol

Project Report

Student Names: Arina Zimina, Karina Siniatullina,
Adelina Karavaeva, Egor Agapov

Date: 29.04.2025

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 meta-data (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 Methods

Our implementation of the Gossip Protocol consists of a modern web application with a clear separation between backend and frontend components. The system architecture is designed to be scalable, maintainable, and provides real-time visualization of the gossip protocol simulation.

2.1 Architecture

The project follows a client-server architecture:

- **Backend:** Implemented in Python using FastAPI framework, providing RESTful API endpoints for simulation control and data retrieval
- **Frontend:** Built with React.js, offering a modern, interactive user interface with:
 - Real-time visualization of node states and message propagation

- Interactive controls for simulation parameters
- Animated transitions using Framer Motion for smooth node state changes
- Responsive design for various screen sizes
- **Communication:** REST API with CORS support for seamless frontend-backend interaction

2.2 Simulation Implementation

The core simulation logic is implemented in the following components:

- **Node Class:** Represents individual nodes in the network with properties for:
 - Node identification
 - Data storage
 - Active/inactive states
 - Alive/dead states
- **Gossip Algorithm:** Implements the core protocol with features:
 - Random peer selection
 - Concurrent message exchange using threading
 - Fault tolerance simulation (1% node failure probability)
 - Node recovery simulation (5% recovery probability)
 - Convergence detection

2.3 Convergence and Metrics

The system tracks several important metrics:

- **Convergence:** Detected when all alive nodes have the same data
- **Round History:** Each round's state is saved, including:
 - Node states (active/inactive, alive/dead)
 - Flagging whether the message has reached the node
 - Message exchanges
 - Convergence status
- **Simulation Logs:** Detailed JSON logs for each simulation round

2.4 Tools and Technologies

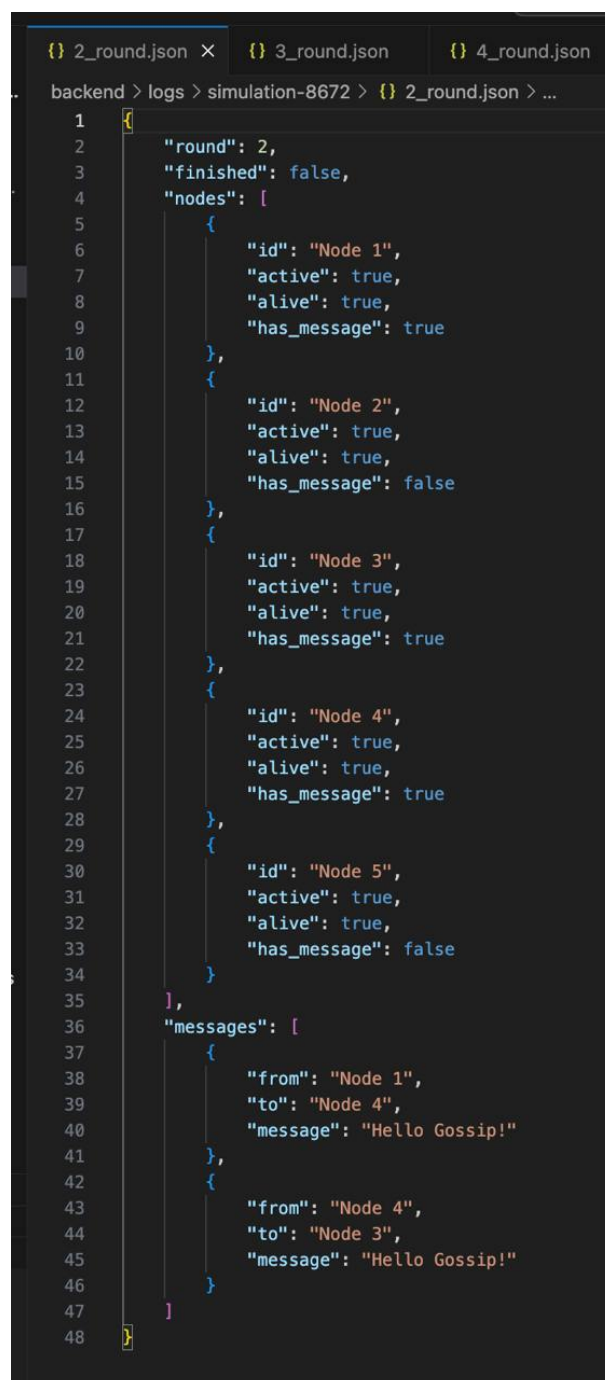
- **Backend:** Python, FastAPI, threading
- **Frontend:**
 - React.js for component-based UI development
 - Framer Motion for smooth animations and transitions
 - Modern CSS for responsive design

- WebSocket for real-time updates
- **Development:** Git for version control
- **Testing:** Dedicated test suite for protocol verification

3 Results

3.1 Logs

Our logs are collected every round. At the beginning we write the number of the round, then a checkbox whether the exchange of information has been completed (final round), then a description of the state of each node in the algorithm and then a list of messages exchanged by the nodes in this round.



```
{} 2_round.json X {} 3_round.json {} 4_round.json
backend > logs > simulation-8672 > {} 2_round.json > ...
1  {
2    "round": 2,
3    "finished": false,
4    "nodes": [
5      {
6        "id": "Node 1",
7        "active": true,
8        "alive": true,
9        "has_message": true
10     },
11     {
12       "id": "Node 2",
13       "active": true,
14       "alive": true,
15       "has_message": false
16     },
17     {
18       "id": "Node 3",
19       "active": true,
20       "alive": true,
21       "has_message": true
22     },
23     {
24       "id": "Node 4",
25       "active": true,
26       "alive": true,
27       "has_message": true
28     },
29     {
30       "id": "Node 5",
31       "active": true,
32       "alive": true,
33       "has_message": false
34     }
35   ],
36   "messages": [
37     {
38       "from": "Node 1",
39       "to": "Node 4",
40       "message": "Hello Gossip!"
41     },
42     {
43       "from": "Node 4",
44       "to": "Node 3",
45       "message": "Hello Gossip!"
46     }
47   ]
48 }
```

4 Discussion

4.1 What has been achieved

- The basic logic of the algorithm is fully implemented
- The ability to select the number of nodes in the algorithm has been implemented
- At the end of each round, nodes generate comprehensible logs and share them among themselves
- A simple and clear web page design and interface has been implemented

4.2 What has not been achieved

- We have not been able to deal with the comprehensibility and accessibility of the web interface for a large number of nodes (e.g. starting from 100)
- It was not possible for us to implement data transmission simulating real network delays

Despite these limitations, the core functionality of the project works reliably, and the simulation successfully demonstrates the intended behavior of the gossip protocol. The project serves as a solid foundation for further improvements and scaling.

4.3 What could be improved

- **Scalability:** The current implementation could be optimized to better handle simulations involving hundreds or thousands of nodes, both in terms of performance and UI clarity.
- **Realism:** Introducing configurable network delays and message loss rates would make the simulation more realistic and useful for testing fault tolerance.

5 References

1. What is the Gossip Protocol: <https://academy.bit2me.com/en/what-is-gossip-protocol/>
2. To visualize people in the algorithm: <https://codepen.io/MAW/full/RaXRdE/>
3. Gossip Protocol: https://en.wikipedia.org/wiki/Gossip_protocol
4. Gossip Protocol implementation: <https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html>
5. The basic idea behind the Gossip Protocol: <https://neerc.ifmo.ru/wiki/index.php?title=Gossip->
6. FastAPI Documentation: <https://fastapi.tiangolo.com>
7. React Documentation: <https://react.dev>
8. CSS Animations: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animations/Using_CSS_animations
9. Motion Animation Library: <https://motion.dev>