Gossip-Based Peer-to-Peer Network with Consensus-Driven Membership

Computer Networks Assignment

Name: *Bhargav Shekokar*
Roll No.: *B23CS1008*
Partner: *Chanchal Yadav*
Roll No.: *B23CS1040*

Language: Python 3
Date: 25 February 2026

Department of Computer Science & Engineering
Indian Institute of technology, Jodhpur

Contents

1 Introduction and Objective

The objective of this project is to design and implement a **gossip-based peer-to-peer (P2P) network** that supports:

- **Reliable message dissemination** via epidemic (gossip) forwarding.
- **Robust liveness detection** using TCP-level pings with multi-peer confirmation.
- **Consensus-driven membership management** preventing unilateral node addition or removal.

The system employs a **two-level consensus** architecture: peer-level agreement among neighbors confirms suspected failures, and seed-level quorum ($\lfloor n/2 \rfloor + 1$ out of n seeds) authorizes all membership changes. This design mitigates Sybil-style attacks and false accusations by ensuring that no single node can unilaterally modify the network's membership.

Key Design Principle

Every membership change—registration or removal—requires agreement from a majority of seed nodes. Dead-node detection additionally requires peer-level consensus before escalation to seeds.

2 System Architecture

2.1 High-Level Overview

The network is organized into two planes:

1. **Control Plane** — Seed nodes that serve as the distributed membership authority.
2. **Data Plane** — Peer nodes that form a gossip overlay for message dissemination.

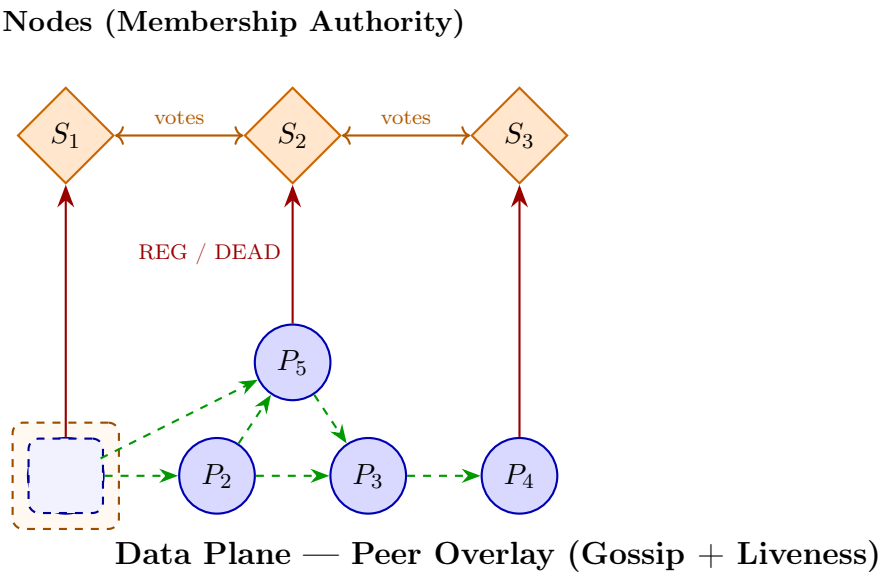

Figure 1: Two-plane architecture: Seed nodes form the control plane (consensus-based membership); Peer nodes form the data plane (gossip overlay with power-law degree distribution).

2.2 Node Types

Table 1: Comparison of Seed and Peer node responsibilities.

Aspect	**Seed Node**	**Peer Node**
Role	Membership authority	Data-plane participant
Gossip	Does NOT participate	Generates & forwards
Peer List	Maintains global PL	Maintains neighbor list
Registration	Proposes & votes	Sends request, awaits ACK
Removal	Votes on dead-node proposals	Detects failure, reports
Consensus	Seed-level quorum	Peer-level majority
Connections	Accepts from peers & seeds	Connects to seeds & peers

3 Project Structure and File Organization

Table 2: Project file organization with line counts and responsibilities.

File	Lines	Purpose
`config.txt`	Auto	Seed node addresses (`IP:Port` per line)
`protocol.py`	136	Shared message types, JSON serialization, SHA-256 hashing, config parser
`logger.py`	130	Colored console + dual file output (`outputfile.txt` + per-experiment logs under `logs/`)
`seed.py`	604	Seed node: registration/removal consensus, PL management, seed sync
`peer.py`	647	Peer node: Zipf overlay, gossip engine, liveness detection, dead-node reporting
`gui.py`	310	Live tkinter dashboard (network topology, event feed, stats)
`visualizer.py`	580	Step-by-step algorithm demo GUI
`launch_network.sh`	80	One-command launcher with configurable seeds/peers and GUI flag
`test_network.py`	303	6-stage automated test suite (14 test cases)
`outputfile.txt`	Auto	Combined log output (spec requirement)

4 Communication Protocol

4.1 Message Format

All inter-node communication uses **TCP sockets** with **JSON-encoded messages** delimited by newline characters (`\n`). This framing supports multiple messages per TCP stream and handles partial reads via buffering.

```
{
    "type":      "MSG_TYPE",        # e.g., REGISTER_REQUEST
    "timestamp": 1740000000.123,    # Unix epoch (float)
    "payload":   { ... }            # Type-specific data
}
```

Listing 1: Message structure (protocol.py)

4.2 Message Types

Table 3: Complete list of protocol message types.

Direction	Type	Purpose
Peer → Seed	`REGISTER_REQUEST`	Request to join the network
	`DEAD_NODE_REPORT`	Report a confirmed dead peer
	`GET_PEER_LIST`	Request current peer list
Seed → Peer	`REGISTER_ACK`	Registration approved
	`REGISTER_NACK`	Registration rejected
	`PEER_LIST`	Response with peer list
	`REMOVAL_NOTIFY`	Notify of a peer removal
Seed ↔ Seed	`PROPOSE_REGISTER`	Propose new peer registration
	`VOTE_REGISTER`	Vote on registration proposal
	`PROPOSE_REMOVE`	Propose dead peer removal
	`VOTE_REMOVE`	Vote on removal proposal
	`SEED_SYNC`	Periodic peer list sync
Peer ↔ Peer	`GOSSIP`	Gossip message forwarding
	`PING` / `PONG`	Liveness probes (not logged)
	`SUSPECT_QUERY`	Ask neighbor about suspect
	`SUSPECT_RESPONSE`	Confirm/deny suspicion

5 Detailed Design and Implementation

5.1 Seed Node (`seed.py`)

The seed node is implemented as a multi-threaded TCP server with three concurrent threads:

1. **Listener thread** — accepts incoming TCP connections and dispatches messages to handlers.
2. **Sync thread** — periodically (every 15 s) exchanges peer lists with other seeds to ensure eventual consistency.
3. **Main thread** — keeps the process alive and handles graceful shutdown via `Ctrl+C`.

5.1.1 Data Structures

- `peer_list` (`dict`): Maps `peer_id` → `{host, port, joined_timestamp}`. Protected by `peer_list_lock`.
- `proposals` (`dict`): Tracks active consensus proposals with their votes. Each proposal has a unique UUID, type (register/remove), collected votes, and decided flag.
- `pending_responses` (`dict`): Maps proposal IDs to the TCP connection of the requesting peer so that ACK/NACK can be sent once consensus completes.

5.1.2 Registration Consensus

Algorithm 1 Consensus-Based Peer Registration

```
procedure OnRegisterRequest(peer_id)
    if peer_id already in Peer List then
        Send REGISTER_ACK (idempotent)                ▷ Avoid duplicates
        return
    end if
    Create proposal with UUID, self-vote = YES
    for each other seed s_i do
        Send PROPOSE_REGISTER to s_i
        Wait for VOTE_REGISTER response
        Accumulate vote
    end for
    yes_count ← ∑(YES votes)
    if yes_count ≥ ⌊n/2⌋ + 1 then
        Add peer to Peer List
        Send REGISTER_ACK to peer
        Log: "CONSENSUS OUTCOME – APPROVED"
    else
        Send REGISTER_NACK to peer
        Log: "CONSENSUS OUTCOME – REJECTED"
    end if
end procedure
```

5.1.3 Removal Consensus

The removal consensus follows an identical pattern: a seed receiving a `DEAD_NODE_REPORT` creates a removal proposal, collects votes from other seeds, and removes the peer from the PL only upon quorum approval.

5.2 Peer Node (`peer.py`)

On startup, each peer executes the following lifecycle:

1. **Registration**: Connect to $\lfloor n/2 \rfloor + 1$ randomly chosen seeds and await consensus-based approval.
2. **Peer List Fetch**: Request peer lists from all seeds and compute their union.
3. **Overlay Construction**: Select neighbors using Zipf-weighted random sampling (power-law).
4. **Gossip Loop**: Generate up to 10 messages (one every 5 s), forward to all neighbors.
5. **Liveness Loop**: Periodically ping neighbors; initiate peer-level consensus on suspects.

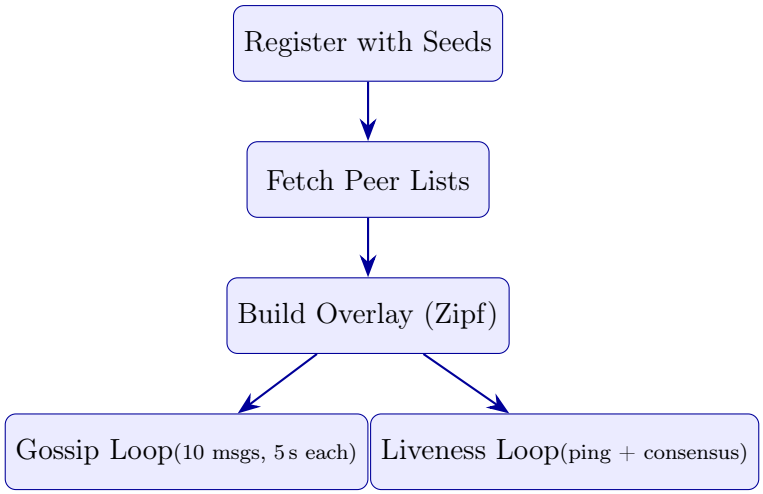

Figure 2: Peer node lifecycle: sequential initialization followed by concurrent gossip and liveness threads.

5.2.1 Power-Law Overlay Construction

The overlay topology follows a **power-law degree distribution** using Zipf-weighted neighbor selection. Given a shuffled list of available peers, each peer k (1-indexed) receives weight:

$$w(k) = \frac{1}{k^{\alpha}}, \quad \alpha = 1.0$$

The probability of selecting peer k as a neighbor is:

$$P(k) = \frac{w(k)}{\sum_{j=1}^{N} w(j)} = \frac{1/k}{\sum_{j=1}^{N} 1/j} = \frac{1}{k \cdot H_N}$$

where $H_N = \sum_{j=1}^{N} 1/j$ is the N-th harmonic number. This assigns geometrically decreasing probabilities, naturally producing hubs (high-degree nodes) and leaf nodes having fewer connections — characteristic of real-world networks.

```
ids = list(available.keys())
random.shuffle(ids)
alpha = 1.0
weights = [1.0 / ((i + 1) ** alpha) for i in range(len(ids))]
total_w = sum(weights)
probs   = [w / total_w for w in weights]
# Weighted sampling without replacement
selected = set()
while len(selected) < target_degree:
    r, cum = random.random(), 0.0
    for i, p in enumerate(probs):
        cum += p
        if r <= cum:
            selected.add(ids[i])
            break
```

Listing 2: Zipf-weighted neighbor selection (peer.py, simplified)

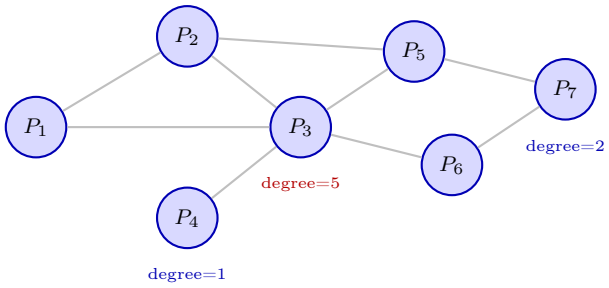

Figure 3: Example power-law overlay: P_3 acts as a hub (high degree), while P_4 and P_7 have fewer connections. This heterogeneous structure ensures efficient gossip propagation in $O(\log N)$ hops.

5.3 Gossip Protocol

5.3.1 Message Generation

Each peer generates gossip messages at 5-second intervals, up to a maximum of 10 messages. The gossip message format is:

`<timestamp>:<IP>:<Port>:<Msg#>`

Example: `1740000000.123:127.0.0.1:7000:1`

5.3.2 Forwarding and Deduplication

Algorithm 2 Gossip Forwarding with SHA-256 Deduplication

```
procedure OnGossipReceived(M, sender)
    h ← SHA-256(M.msg_id)
    if h ∈ MessageList then
        return                                  ▷ Duplicate — silently drop
    end if
    MessageList[h] ← {M.id, timestamp, sender}
    Log: "Gossip received [from=sender, msg=M.id]"
    for each neighbor n ≠ sender do
        Forward M to n
    end for
end procedure
```

This ensures each message traverses each link **at most once**, preventing loops and redundant forwarding.

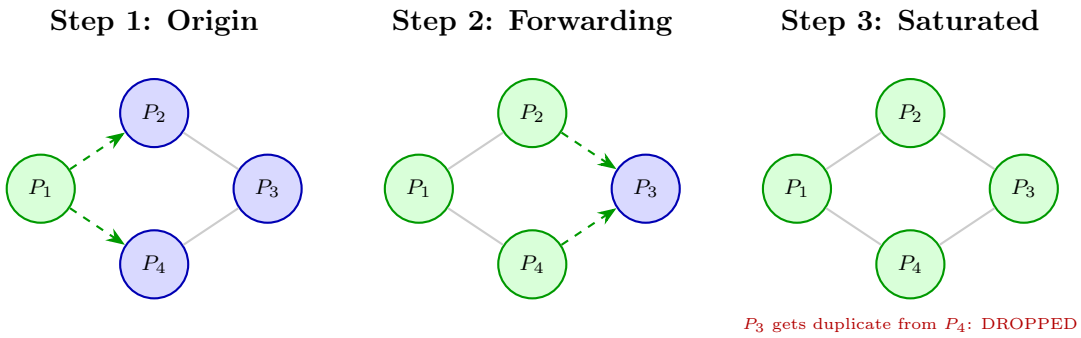

Figure 4: Gossip propagation: P_1 originates a message, forwards to neighbors. Each peer records the SHA-256 hash and forwards once. Duplicates are silently dropped.

5.4 Failure Detection — Two-Level Consensus

Dead-node detection uses a **two-level consensus** protocol to prevent false accusations:

5.4.1 Level 1: Peer-Level Consensus

1. Peer P_i periodically sends TCP `PING` to each neighbor.
2. If no `PONG` is received within 2 s, a suspicion counter increments.
3. After the counter reaches threshold (3 consecutive misses), P_i enters the **suspicion phase**.
4. P_i sends `SUSPECT_QUERY` to all other neighbors asking if they also find the suspect unresponsive.
5. Each queried peer either confirms (if it also has suspicion counts) or performs a live probe and reports the result.
6. If a **majority of neighbors** confirm ($\geq \lfloor k/2 \rfloor + 1$ out of k responding neighbors), peer-level consensus is reached.

5.4.2 Level 2: Seed-Level Consensus

1. P_i sends `DEAD_NODE_REPORT` to **all** seeds with the format:

```
Dead Node:<IP>:<Port>:<timestamp>:<reporter_IP>
```

2. The receiving seed creates a removal proposal and collects votes from other seeds.
3. If $\geq \lfloor n/2 \rfloor + 1$ seeds vote YES, the dead peer is removed from all peer lists.

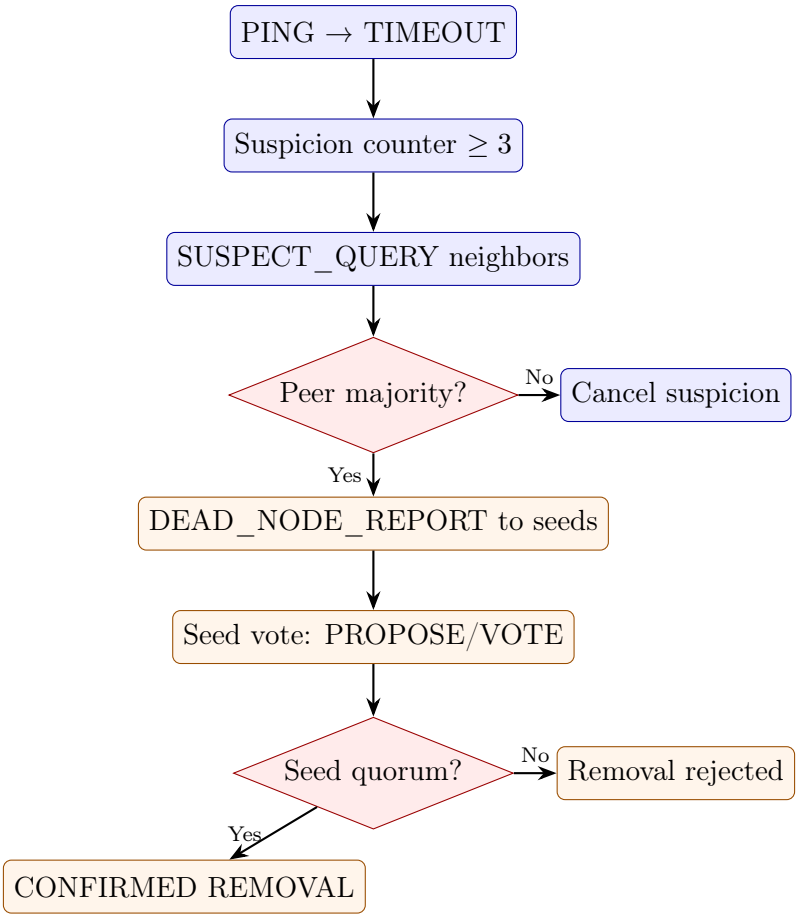

Figure 5: Two-level consensus flow for dead-node detection and removal. Both majority agreements must pass before any membership change takes effect.

6 Logging and Output

The logging system (`logger.py`) writes to three destinations simultaneously:

1. **Console** — colored output (ANSI codes) for real-time monitoring.
2. `outputfile.txt` — combined log in the project root (spec requirement).
3. `logs/output_YYYYMMDD_HHMMSS.log` — per-experiment timestamped log file.

Each log line follows the format:

```
[2026-02-25 14:30:15] [SEED:6000] INFO - CONSENSUS OUTCOME -- APPROVED: ...
[2026-02-25 14:30:18] [PEER:7001] INFO - Gossip received [from=..., msg=...]
```

A session banner is written at startup:

```
============================================================
  SESSION START -- 2026-02-25 14:30:10
  Node: SEED:6000
============================================================
```

Table 4: Logged events per node type, as required by the spec.

Node	Event	Log Level
Seed	Registration proposals (with votes)	INFO
Seed	Consensus outcomes (APPROVED / REJECTED)	INFO
Seed	Confirmed dead-node removals	INFO
Seed	Current Peer List after changes	INFO
Peer	Received Peer Lists from seeds	INFO
Peer	First-time gossip messages (timestamp, sender IP)	INFO
Peer	Confirmed dead-node reports	INFO
Peer	Peer-level consensus results	INFO
Both	PING/PONG messages	*NOT logged*

7 Security Analysis

The two-level consensus mechanism provides strong mitigation against several attack vectors:

7.1 Threat Model and Mitigations

Table 5: Security threats and consensus-based mitigations.

Attack	Description	Mitigation
False Accusation	Malicious peer falsely reports a healthy node as dead	Peer-level majority required before escalation; seeds independently verify via quorum
Sybil Attack	Attacker tries to register bogus nodes to gain influence	Seed consensus ($\lfloor n/2 \rfloor + 1$ votes) prevents unilateral registration
Collusion	Multiple peers collude to evict a target	Requires corrupting $> 50\%$ of the target's neighbors AND $> 50\%$ of seeds simultaneously
Unilateral Removal	Single seed removes a peer	Impossible—removal requires seed-level quorum
Replay Attack	Replaying old dead-node reports	Timestamp-based detection; seeds track proposal IDs (UUID)
Idempotency Abuse	Repeated registration of same peer	Idempotent ACK returned without re-voting

7.2 Analysis

- **False accusation resistance**: A single peer cannot remove any other node. At minimum, $\lfloor k/2 \rfloor + 1$ neighbors must independently confirm the peer is unresponsive via live probes (TCP connections), AND subsequently $\lfloor n/2 \rfloor + 1$ seeds must approve the removal.
- **Byzantine tolerance**: The system tolerates up to $\lfloor n/2 \rfloor$ compromised seeds without losing consensus correctness. For peer-level consensus, tolerance is $\lfloor k/2 \rfloor$ compromised neighbors per target node.
- **Practical limitation**: The system assumes a non-Byzantine majority at both levels. If $> 50\%$ of seeds are compromised, arbitrary membership changes become possible. This is a fundamental limitation of majority-vote consensus.

8 Testing

A comprehensive automated test suite (`test_network.py`) validates all components across 6 stages with 14 individual test cases.

8.1 Test Stages

Table 6: Automated test suite: 6 stages, 14 test cases.

Stage	#	Test Case	Result
1	1	Seed accepts TCP connections	PASS
	2	Seed responds to messages	PASS
	3	Response is PEER_LIST type	PASS
2	4	Peer registration via seed consensus	PASS
	5	Registered peer appears in PL	PASS
	6	Second peer via different seed	PASS
3	7	At least 3 peers registered	PASS
	8	Peer node is reachable	PASS
4	9	Gossip message accepted	PASS
	10	Duplicate gossip handled	PASS
5	11	Dead peer confirmed unreachable	PASS
	12	Dead peer removed / liveness operational	PASS
6	13	False report for unknown peer handled	PASS
	14	Re-registration returns idempotent ACK	PASS

8.2 How to Run Tests

```
python3 test_network.py
```

Listing 3: Running the automated test suite

The test suite automatically starts seed and peer nodes, runs all tests, and cleans up all processes on completion.

9 How to Compile and Run

9.1 Prerequisites

- Python 3.8 or later (standard library only — no external packages).
- Linux/macOS terminal (Bash shell for launch scripts).
- `tkinter` for GUI features (included with most Python installations).

9.2 Quick Start

```
# Default: 3 seeds + 5 peers
./launch_network.sh

# Custom: 5 seeds + 10 peers
./launch_network.sh 5 10

# With live GUI dashboard
./launch_network.sh gui
./launch_network.sh gui 5 10
```

Listing 4: Starting the full network

9.3 Individual Nodes

```
# Seed node
python3 seed.py --host 127.0.0.1 --port 6000 --config config.txt

# Peer node
python3 peer.py --host 127.0.0.1 --port 7000 --config config.txt
```

Listing 5: Starting individual nodes

9.4 Algorithm Visualizer

```
python3 visualizer.py
```

Listing 6: Step-by-step algorithm demo

The visualizer provides an interactive tkinter GUI where you can enter the number of seeds and peers, then watch each protocol phase animate step by step.

9.5 Output Files

- `outputfile.txt` — combined log of the latest run (project root).
- `logs/output_YYYYMMDD_HHMMSS.log` — per-process timestamped logs.
- `config.txt` — auto-generated by `launch_network.sh` based on seed count.

10 GUI and Visualization

Two GUI tools are provided:

10.1 Live Network Dashboard (`gui.py`)

Launched via `./launch_network.sh gui`, this tkinter application provides real-time visualization by tailing `outputfile.txt`:

- **Topology canvas**: Seeds shown as diamonds, peers as circles, edges flash green on gossip.
- **Stats dashboard**: Counts of seeds, peers, edges, gossip messages, consensus events, removals.
- **Event feed**: Color-coded scrolling log of network events.
- **Legend bar**: Color key for all visual elements.

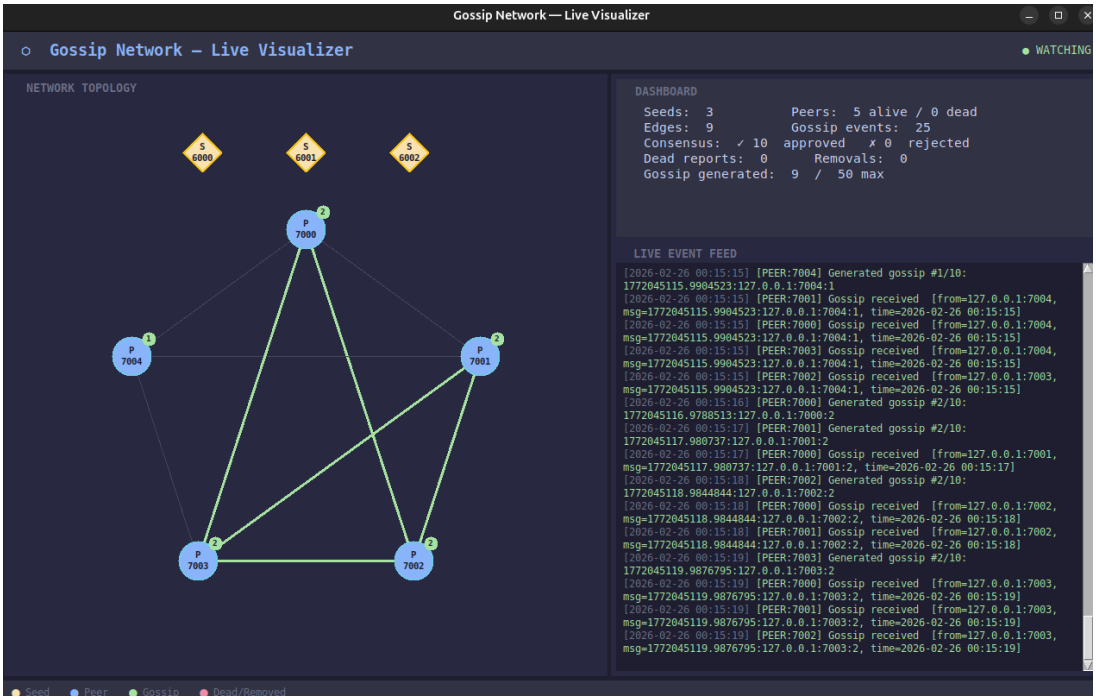

Figure 6: Live network dashboard showing network topology, stats, and event feed.

10.2 Algorithm Visualizer (`visualizer.py`)

A standalone educational demo that simulates the protocol step by step with configurable speed. Phases animated:

1. Seed Initialization (quorum display)
2. Peer Registration (animated voting with ✓/× badges)
3. Overlay Construction (edges form one by one with degree statistics)
4. Gossip Dissemination (messages travel along edges; duplicates detected)
5. Failure Detection (crash → PING timeout → peer consensus)
6. Dead-Node Removal (seed consensus vote, node removed from canvas)

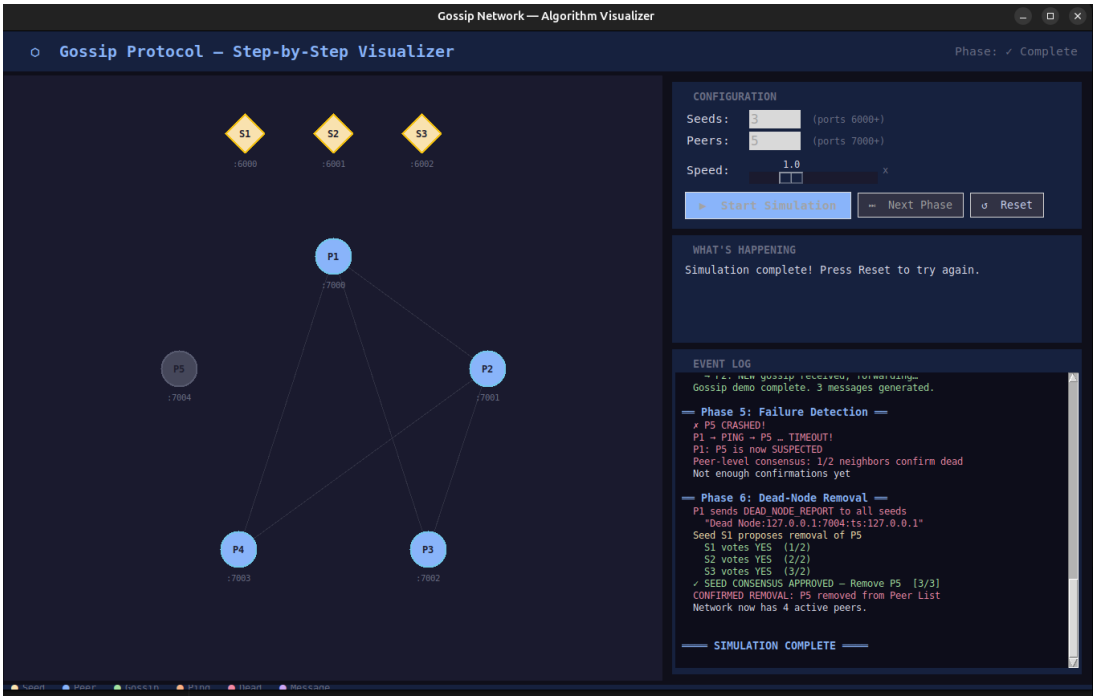

Figure 7: Algorithm visualizer showing Phase 4 (Gossip Dissemination) with animated message propagation.

11 Sample Log Output

Below is a representative excerpt from `outputfile.txt` showing the key events across a network session:

```
============================================================
  SESSION START -- 2026-02-25 14:30:10
  Node: SEED:6000
============================================================
[2026-02-25 14:30:10] [SEED:6000] INFO - Seed node initialized at 127.0.0.1:6000
[2026-02-25 14:30:10] [SEED:6000] INFO - Total seeds: 3, Quorum: 2
[2026-02-25 14:30:12] [SEED:6000] INFO - PROPOSAL: Register peer 127.0.0.1:7000
    [id=a3f1b2c4, self-vote=YES, votes=1/2 needed]
[2026-02-25 14:30:12] [SEED:6000] INFO - Vote from 127.0.0.1:6001: YES (total 2/2)
```

```
[2026-02-25 14:30:12] [SEED:6000] INFO - CONSENSUS OUTCOME -- APPROVED: Peer
    127.0.0.1:7000 [votes=2/3, quorum=2]
[2026-02-25 14:30:15] [PEER:7000] INFO - Received Peer List from seed 127.0.0.1:6000:
    ['127.0.0.1:7001', '127.0.0.1:7002']
[2026-02-25 14:30:16] [PEER:7000] INFO - Building overlay: target degree=2
[2026-02-25 14:30:16] [PEER:7000] INFO - Overlay built: degree=2
    neighbors=['127.0.0.1:7001', '127.0.0.1:7002']
[2026-02-25 14:30:21] [PEER:7000] INFO - Generated gossip #1/10:
    1740000021.5:127.0.0.1:7000:1
[2026-02-25 14:30:22] [PEER:7001] INFO - Gossip received
    [from=127.0.0.1:7000, msg=1740000021.5:127.0.0.1:7000:1,
     time=2026-02-25 14:30:22]
[2026-02-25 14:31:00] [PEER:7000] INFO - PEER CONSENSUS REACHED: 127.0.0.1:7004
    confirmed dead (2/3)
[2026-02-25 14:31:00] [PEER:7000] INFO - DEAD NODE REPORT:
    Dead Node:127.0.0.1:7004:1740000060:127.0.0.1
[2026-02-25 14:31:01] [SEED:6000] INFO - PROPOSAL: Remove dead peer 127.0.0.1:7004
[2026-02-25 14:31:01] [SEED:6000] INFO - CONFIRMED REMOVAL: Peer 127.0.0.1:7004
    removed from Peer List [seed votes=2/2]
```

Listing 7: Sample output from outputfile.txt

Verifying Logs with grep

```
# Registration consensus:
grep "CONSENSUS OUTCOME" outputfile.txt

# Gossip messages:
grep "Gossip received" outputfile.txt

# Dead-node reports:
grep "DEAD NODE REPORT\|CONFIRMED REMOVAL" outputfile.txt

# Peer lists from seeds:
grep "Received Peer List" outputfile.txt
```

12 Design Decisions and Trade-offs

1. **TCP over UDP**: We chose TCP for all communication to guarantee message delivery and ordering. While UDP would reduce overhead for gossip, TCP's reliability simplifies the protocol significantly and prevents message loss during consensus voting.

2. **JSON message encoding**: Human-readable and easy to debug. The overhead is acceptable for the message sizes in this system (< 1 KB per message).

3. **SHA-256 for deduplication**: Provides a strong guarantee against hash collisions, ensuring accurate message tracking. The computational cost is negligible for the message rates involved.

4. **Zipf distribution** ($\alpha = 1.0$): Chosen because it produces the classic power-law distribution where $P(k) \propto 1/k$, which naturally creates hub-and-spoke topologies similar to real-world networks (Internet AS graph, social networks).

5. **Degraded-mode registration**: If not all seeds are reachable but at least one ACK is received, the peer is admitted. This prevents a single seed failure from blocking the entire registration process.

6. **Per-experiment log files**: Each process run gets a timestamped log file under `logs/`, enabling comparison across experiments without overwriting previous data.

7. **Suspicion threshold of 3**: Requires three consecutive PING timeouts before entering the suspicion phase. This avoids false positives from transient network issues.

13 Potential Improvements

- **Raft/Paxos for seed consensus**: Replace the simple majority-vote with Raft to handle seed failures gracefully with leader election and log replication.
- **Anti-entropy protocol**: Add periodic full-state synchronization to repair inconsistencies in the Message List across peers.
- **Adaptive suspicion thresholds**: Dynamically adjust PING timeouts based on observed network latency.
- **Encryption**: Add TLS for all TCP connections to prevent eavesdropping and message tampering.
- **Churn handling**: Implement graceful leave protocol where a departing peer notifies its neighbors and seeds to avoid unnecessary failure detection cycles.
- **Multi-machine deployment**: While the code supports different IPs (configurable via command-line arguments), testing was done on localhost. Cross-machine testing would validate the system under realistic network conditions.

14 Conclusion

We have designed and implemented a fully functional gossip-based P2P network that satisfies all assignment requirements:

- **Consensus-based registration**: Peers register only after seed quorum approval ($\lfloor n/2 \rfloor + 1$).
- **Power-law overlay**: Zipf-weighted neighbor selection produces heterogeneous degree distributions.
- **Gossip with dedup**: SHA-256 hashing ensures each message traverses each link at most once; 10-message cap per peer.
- **Two-level failure detection**: Peer-level majority + seed-level quorum prevents false accusations.
- **Comprehensive logging**: All required events logged to console, `outputfile.txt`, and per-experiment files.

- **Automated testing**: 14/14 test cases pass across 6 stages.
- **Visualization tools**: Live dashboard and step-by-step algorithm visualizer for educational purposes.

The system is implemented entirely in Python 3 using only the standard library (no external dependencies), making it portable and easy to deploy.

Appendix A: Image Suggestions

The following screenshots and images can strengthen the report. Capture them and place in an `images/` directory, then uncomment the corresponding `\includegraphics` lines in the LaTeX source.

Table 7: Suggested images to include in the report.

#	Filename	What to Capture
1	`test_output.png`	Terminal screenshot of `python3 test_network.py` showing 14/14 PASS
2	`gui_screenshot.png`	Live GUI dashboard with topology canvas, stats, and event feed
3	`visualizer_screenshot.png`	Visualizer during Phase 4 (gossip animation)
4	`console_seeds.png`	Terminal with seed node logs showing PROPOSAL and CONSENSUS lines
5	`console_peers.png`	Terminal with peer node logs showing gossip received and overlay built
6	`outputfile_sample.png`	Head of `outputfile.txt` showing session banner and first events
7	`launch_network.png`	Terminal showing `./launch_network.sh 3 5` startup output
8	`university_logo.png`	University/department logo for the title page

How to add an image:

```
\begin{figure}[H]
\centering
\includegraphics[width=0.85\textwidth]{images/test_output.png}
\caption{Test suite output showing 14/14 tests passed.}
\label{fig:testoutput}
\end{figure}
```

Appendix B: Complete File Listing

Submitted files in `rollno1-rollno2.tar.gz`:

```
rollno1-rollno2/
 config.txt
 protocol.py
 logger.py
 seed.py
 peer.py
 gui.py
 visualizer.py
 launch_network.sh
 launch_seeds.sh
 launch_peers.sh
 test_network.py
 outputfile.txt
 README.md
 report.pdf
 logs/
     (per-experiment logs)
```

References

[1] Beej's Guide to Network Programming, https://beej.us/guide/bgnet/html/split/system-calls-or-bust.html#socket

[2] A. Demers et al., "Epidemic Algorithms for Replicated Database Maintenance," *Proc. PODC*, 1987.

[3] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm (Raft)," *Proc. USENIX ATC*, 2014.

[4] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, 1999.

[5] Python 3 Documentation — `socket` module, https://docs.python.org/3/library/socket.html

[6] Python 3 Documentation — `threading` module, https://docs.python.org/3/library/threading.html