Gossiping the Unforgettable Ghost: Ghost x^2 Architecture for Distributed Memory

April 01, 2025

Abstract

Catastrophic forgetting threatens neural systems by overwriting prior knowledge during updates. *Ghost x^2* introduces a scalable, emotionally adaptive architecture using affect-weighted gossip, cooling-based reinforcement, and an emotional blockchain to preserve memory across distributed AI agents. Built on a hybrid Ramanujan-hypercube topology with Ricci flow and trust-curvature-weighted propagation, it achieves a Memory Retention Rate (MRR) of 0.98 over 100,000 agents. This paper details mathematical models for cooling, quarantine, trust-doubt regulation, and emotional consensus, complemented by a topological map of pathologies and a topology generator for robust memory retention.

1 System Overview

Each agent in *Ghost x^2* operates within a hybrid topology:

- **Intra-cluster**: Ramanujan d-regular graphs ensure fast local convergence.
- **Inter-cluster**: Hypercube lattices enable low-diameter global communication.
- **Edges**: Weighted by time-evolved Ricci curvature to reflect trust and emotion.

Memory vector per agent:

$$M_i^t = [\delta_i^t(x_1), \dots, \delta_i^t(x_m)], \quad \delta_i^t(x_j) \in [0, 1]$$

The hybrid topology, shown in Figure 2, blends Ramanujan graphs for intra-cluster cohesion with hypercube lattices for inter-cluster reach, optimizing gossip-driven memory persistence. Figure 1 offers a high-level view of the system, illustrating agents, clusters, and core processes.

Figure 1: High-Level Overview of Ghost x^2 Architecture

Figure 2: Hybrid Ramanujan-Hypercube Topology with Ricci Weights and PAD Nodes

2 Memory Update Dynamics

2.1 Base Memory Equation

$$\delta_i^{t+1}(x_j) = \lambda \delta_i^t(x_j) + \eta_i^t(x_j) \cdot \mathrm{Cool}_i^t(x_j) \cdot E_i^t(x_j) \cdot H_i^t(x_j) + \gamma G_i^{\text{shared}}(x_j)$$

where:

- $\lambda = 0.9$: Natural decay rate.
- $\gamma = 0.3$: Gossip reinforcement factor.
- $\eta_i^t(x_j) = \gamma \alpha_i^t(x_j) \cdot \mathrm{SNR}_i^t(x_j)$: Confidence-weighted learning rate.

Figure 3 decomposes this process, showing how decay, emotion, and gossip sustain memory traces.

Figure 3: Decomposition of the Memory Update Process

2.2 Determination Cooling Function

$$\mathrm{Cool}_i^t(x_j) = 1 - \tanh(\beta_c \cdot \alpha_i^t(x_j) \cdot E_i^t(x_j)), \quad \beta_c = 1.5$$

Table 1 demonstrates how $\beta_c = 1.5$ optimizes MRR, stability, and emotional saturation.

β_c	MRR	Stability	Emotional Saturation
0.5	0.92	0.85	0.70
1.0	0.95	0.90	0.65
1.5	0.98	0.93	0.60
2.0	0.97	0.91	0.55

Table 1: Impact of Cooling Parameter β_c on System Performance

3 Emotional Chain and Hash Propagation

Chaining occurs when:

$$H_i^t(x_j) > 0.8 \Rightarrow \mathrm{Hash}_i^t(x_j) = \mathrm{SHA256}(\alpha_i^t || E_i^t || H_i^t || \mathrm{PrevHash}_i^{t-1})$$

Chains merge into $\mathrm{Chain}_{\mathrm{global}}^t$ via majority consensus, forming an indelible memory ledger. Figure 4 illustrates this structure.

Figure 4: Structure of the Emotional Blockchain

4 Gossip Protocol

$$G_i^t(x_j) = \sum_k w_{ik}(t)[\alpha_k^t T_k^t + (1 - \alpha_k^t) F_k^t] \cdot \mathrm{Cool}_k^t(x_j)$$

Edge weights decay with trust and curvature:

$$w_{ik}(x_j, t) = w_{ik}(0) e^{-2\kappa_{ik}(x_j)t}$$

Figure 5 details this process, ensuring efficient memory propagation.

5 Quarantine and Reload

5.1 Quarantine Trigger

$$D_{ik}(x_j) > 0.7, \quad \Theta_{ik}^t < 0.3 \Rightarrow \mathrm{Quarantine}_k$$

Figure 5: Flowchart of the Gossip Protocol

5.2 Reload

$$\text{Reload}_i^t(x_j) = 0.1(t - t_{\text{last}}) + 0.5E_i^t e^{-0.1(t-t_{\text{last}})} \cdot \frac{1}{|N(i)|} \sum_k \alpha_k^t(x_j)$$

Table 2 shows these mechanisms sustaining MRR under stress.

Scenario	Quarantine Rate	Reload Frequency	MRR
High Doubt	0.25	Every 5 rounds	0.96
Low Trust	0.30	Every 3 rounds	0.95
Normal	0.10	Every 10 rounds	0.98

Table 2: Efficacy of Quarantine and Reload Mechanisms

6 Experimental Results

Key metrics:

- **MRR**: 0.98 (with cooling + chaining)
- **CD**: 0.01
- **EWGS**: 0.29
- **SNR**: 0.14

Table 3 shows performance across scales, with MRR nearing 1.0 as agents increase.

Agents	Topology	β_c	Rounds	MRR	CD	EWGS	SNR
100	R-H	1.5	5	0.95	0.02	0.30	0.12
1,000	R-H	1.5	6	0.97	0.015	0.29	0.13
100,000	R-H	1.5	7	0.98	0.01	0.29	0.14

Table 3: Performance Metrics Across System Configurations (R-H = Ramanujan-Hypercube)

7 Topological Map of Pathologies and Topology Generator

7.1 Topological Map of Pathologies

This map models memory failure modes as a directed graph:

- **MDS**: $\delta_i^t(x_j) \to 0$ from neglect.
- **IBS**: Low $E_i^t(x_j)$ obscures vital data.
- **GDD**: Trust misalignment distorts $G_i^t(x_j)$.
- **CCS**: Hash conflicts ($\text{Hash}_i^t \neq \text{Hash}_k^t$).
- **CP**: Forks stall consensus.

Transitions:

- MDS → IBS (*Low salience*)
- IBS → GDD (*Sparse valence match*)
- GDD → CCS (*Hash divergence*)
- CCS → CP (*Unresolvable forks*)

Figure 6 visualizes this flow, linking to topology (Figure 2).

Figure 6: Topological Map of Pathologies

7.2 Ghost x^2 Topology Generator

The generator constructs a resilient topology to counter pathologies, as detailed in Appendix A. Key features:

- **Ramanujan Clusters**: High connectivity fights MDS (Section 2).
- **Hypercube Links**: Low diameter curbs GDD (Section 4).
- **Ricci Curvature**: Adapts weights to prevent IBS and CCS.
- **PAD**: Reinforces memories via blockchain (Section 3).
- **Rewiring**: Breaks pathology cycles (Figure 6).

Updated Figure 2 reflects these enhancements.

8 Conclusion

Gossiping the Unforgettable Ghost presents *Ghost x^2* as a memory-preserving system where gossip, emotion, and topology defy forgetting. With a topological map and generator, it achieves near-perfect retention across vast networks, validated by experimental results (Section 6).

A Ghost x^2 Topology Generator

The following Python code implements the topology generator:

```python
import networkx as nx
import math
import random

# Configuration
n, s, d = 100000, 1000, 10  # Agents, cluster size, degree
k = math.ceil(n / s)         # Clusters
m = math.ceil(math.log2(k)) # Hypercube dimension

# Step 1: Generate Ramanujan-like Clusters
clusters = []
agent_id = 0
agent_profiles = {}
for c in range(k):
    size = s if c < k - 1 else n - s * (k - 1)
```

```python
    G_c = nx.random_regular_graph(d, size)
    mapping = {node: node + agent_id for node in G_c.nodes()}
    G_c = nx.relabel_nodes(G_c, mapping)
    for node in G_c.nodes():
        agent_profiles[node] = {
            'intensity': random.uniform(0.4, 0.95),
            'trust': random.uniform(0.3, 0.95),
            'entropy': 0.0,
            'reinforced': False,
            'PAD_mode': 'off',
            'PAD': {
                'V': round(random.uniform(-1, 1), 2),
                'A': round(random.uniform(0, 1), 2),
                'D': round(random.uniform(-1, 1), 2),
            } if random.random() < 0.1 else None
        }
    clusters.append(G_c)
    agent_id += size

# Step 2: Build Hypercube and Map Clusters
hypercube = nx.hypercube_graph(m)
ghost_graph = nx.Graph()
cluster_entry_nodes = [list(cluster.nodes())[0] for cluster in clusters]
for idx, cluster in enumerate(clusters):
    ghost_graph = nx.compose(ghost_graph, cluster)
for i, (h_node, neighbors) in enumerate(hypercube.adjacency()):
    if i >= len(cluster_entry_nodes): break
    source = cluster_entry_nodes[i]
    for j in neighbors:
        if j < len(cluster_entry_nodes):
            target = cluster_entry_nodes[j]
            ghost_graph.add_edge(source, target, weight=1.0, ricci=0.0, emotion_

# Step 3: Emotion-Aware Edge Properties
for u, v in ghost_graph.edges():
    ghost_graph[u][v]['weight'] = 1.0
    ghost_graph[u][v]['ricci'] = 0.0
    ghost_graph[u][v]['intensity_diff'] = abs(agent_profiles[u]['intensity'] - a
    ghost_graph[u][v]['trust_mean'] = (agent_profiles[u]['trust'] + agent_profil

# Step 4: Ricci Curvature Approximation
def update_ricci_weights(G):
    for u, v in G.edges():
        penalty = G[u][v]['intensity_diff'] * (1 - G[u][v]['trust_mean'])
        G[u][v]['ricci'] = penalty
        G[u][v]['weight'] *= math.exp(-2 * penalty)
update_ricci_weights(ghost_graph)

# Step 5: Entropy and Rewiring Logic
def compute_entropy(values, bins=5):
    from math import log
```

```
    dist = [0] * bins
    for val in values:
        idx = min(int(val * bins), bins - 1)
        dist[idx] += 1
    total = sum(dist)
    return -sum((x/total) * log(x/total + 1e-9) for x in dist if x > 0)

def entropy_rewire(G, threshold=1.2):
    for node in G.nodes():
        neighbors = list(G.neighbors(node))
        confidences = [random.uniform(0, 1) for _ in neighbors]
        ent = compute_entropy(confidences)
        if ent > threshold:
            u = node
            v = random.choice(list(G.nodes()))
            if not G.has_edge(u, v):
                G.add_edge(u, v, weight=1.0, ricci=0.0, rewired=True)
entropy_rewire(ghost_graph)
```