The Whisper Graph: Communication via Probabilistic Field Coherence

OCTA Research Monograph, Refined Full Edition

OCTA Research

December 27, 2025

Abstract

This monograph develops the *Whisper Graph*, a communication substrate in which nodes never exchange discrete messages. Each node maintains a latent probabilistic belief distribution that evolves via weak statistical coupling to its neighbors. Information propagates as spatiotemporal coherence in the network-wide belief field rather than through packetized symbols.

We present a complete and implementation-ready treatment: formal definitions, stability and convergence analysis, multidimensional latent dynamics, information-theoretic perspective, adversarial and detectability models, and a benchmark suite. We provide full reference simulation code in Python for both scalar and multidimensional cases, along with implementation guidance including hardware-layer considerations. A comprehensive RFC-style Whisper Graph Protocol (WGP) specification is included as Appendix A.

The objective is to furnish a rigorous foundation and a practical toolkit for ultra-low-power swarms, stealth networks, and collective robotics built on the Whisper Graph paradigm.

Contents

1 Introduction

Traditional communication systems define explicit messages: sequences of symbols, organized into packets, conveyed across channels. Even when encrypted, such systems expose a rich observable structure: packet sizes, timing, directionality, burst patterns, and routing metadata.

The Whisper Graph proposes a different model in which nodes:

- do not send or receive discrete messages,
- maintain and update internal probabilistic beliefs,
- expose only softly varying, noisy field values,
- jointly shape a global belief field through weak coupling.

The core idea is:

> Nodes do not say *what* they believe. They lean where they believe, and those leanings influence others.

Information is encoded in how belief states across nodes:

- move,
- align,
- persist,
- and fluctuate with structured coherence.

This gives rise to:

- **Stealth:** No packet boundaries or obvious modulation schemes.
- **Ultra-low power:** Nodes emit at low amplitude, with slow update rates.
- **Self-healing:** Failure or removal of nodes naturally relaxes the field.
- **Collective inference:** The network acts as a distributed estimator for latent variables.

We now build the complete formal and practical framework for Whisper Graph systems.

2 Mathematical Foundations

2.1 Network Model and Notation

Let $G = (V, E)$ be a connected undirected graph with $|V| = N$ nodes. Each node $i \in V$ maintains a belief distribution over latent variable $x \in \mathbb{R}^d$:

$$p_i(x, t), \tag{1}$$

where $t \in \mathbb{Z}_{\geq 0}$ indexes discrete time.

Neighbor set:

$$N(i) = \{j \in V : (i,j) \in E\}. \tag{2}$$

We denote:

- $\mu_i(t) \in \mathbb{R}^d$ the mean of $p_i(x,t)$,
- $\Sigma_i(t) \in \mathbb{R}^{d \times d}$ its covariance matrix.

2.2 Gaussian Parametric Family

We focus on Gaussian beliefs for tractability:

$$p_i(x,t) = \mathcal{N}\big(\mu_i(t), \Sigma_i(t)\big). \tag{3}$$

This choice is not mandatory; non-Gaussian families are possible but more complex.

2.3 External Field Emission

External observers and neighbor nodes see only an emitted field:

$$y_i(t) = f\big(p_i(x,t)\big) + \epsilon_i(t), \tag{4}$$

where $\epsilon_i(t)$ is emission noise.

The canonical choice is:

$$y_i(t) = \mu_i(t) + \epsilon_i(t), \qquad \epsilon_i(t) \sim \mathcal{N}(0, \eta_i^2 I_d), \tag{5}$$

with $\eta_i > 0$ controlling the noise floor.

Nodes **never** expose $\Sigma_i(t)$ or full $p_i(x,t)$.

2.4 Definition: Whisper Graph

Definition 2.1 (Whisper Graph)**.** *A* Whisper Graph *is a triple* $(G, \{p_i\}, \{y_i\})$ *consisting of:*

- *a connected graph* $G = (V, E)$*,*
- *node belief distributions* $p_i(x,t)$*,*
- *emission process* $y_i(t)$*,*

such that:

(a) nodes never exchange explicit messages,

(b) updates are driven only by neighbor emissions and internal dynamics,

(c) communication is realized as induced coherence in the joint belief field $\{p_i(x,t)\}_{i \in V}$*.*

3 Belief Relaxation and Update Dynamics

3.1 General Relaxation Rule

Each node i updates its belief using a relaxation operator:

$$p_i(x, t+1) \propto (1-\gamma_i) p_i(x,t) \prod_{j \in N(i)} \phi\big(y_j(t), w_{ij}(t)\big), \tag{6}$$

where:

- $\gamma_i \in [0,1)$ is a memory decay rate,
- $w_{ij}(t) \geq 0$ is an influence weight,
- ϕ is an influence kernel mapping neighbor emissions to likelihood-like factors.

3.2 Linearized Mean Update

For Gaussian beliefs and small update gains, the mean update can be approximated as:

$$\mu_i(t+1) = (1-\alpha_i)\mu_i(t) + \alpha_i \sum_{j \in N(i)} w_{ij}(t) y_j(t), \tag{7}$$

with $0 < \alpha_i \ll 1$.

3.3 Variance Stabilization

To enforce non-zero uncertainty:

$$\Sigma_i(t+1) = (1-\beta_i)\Sigma_i(t) + \beta_i \Sigma_{\min}, \qquad 0 < \beta_i \ll 1, \tag{8}$$

with $\Sigma_{\min} \succ 0$.

Remark 3.1. *Without a covariance floor, the system risks variance collapse, weakening stealth and making the network more detectable via over-coherent emissions.*

4 Coherence as Information Carrier

4.1 Global Mean and Disagreement

Define the global mean:

$$\bar{\mu}(t) = \frac{1}{N} \sum_{i=1}^{N} \mu_i(t). \tag{9}$$

Define disagreement vector:

$$\boldsymbol{\delta}(t) = \begin{bmatrix} \mu_1(t) - \bar{\mu}(t) \\ \vdots \\ \mu_N(t) - \bar{\mu}(t) \end{bmatrix}. \tag{10}$$

4.2 Coherence Index

We define:

$$C(t) = 1 - \frac{\frac{1}{N}\sum_{i=1}^{N} \|\mu_i(t) - \bar{\mu}(t)\|}{\sigma_{\text{noise}}}, \tag{11}$$

with σ_{noise} an effective noise scale (e.g., η).

Interpretation:

- $C(t) \approx 1$: high coherence, nodes agree.
- $C(t) \approx 0$: low coherence, nodes diverge.

4.3 Encoding Dimensions

Information can be encoded in:

- direction of drift of $\bar{\mu}(t)$,
- growth/decay profile of $C(t)$,
- spatial coherence patterns (clusters vs uniform),
- oscillatory phases between nodes.

5 Stability and Convergence

5.1 Matrix Representation

Assume scalar latent ($d = 1$) for clarity, extension to $d > 1$ is by block structure. Stack means into $\boldsymbol{\mu}(t) \in \mathbb{R}^N$.

Let $W(t)$ be row-stochastic with entries $w_{ij}(t)$, and suppose $y(t) = \boldsymbol{\mu}(t) + \boldsymbol{\epsilon}(t)$, with $\boldsymbol{\epsilon}$ the noise vector. For uniform $\alpha_i = \alpha$, we have:

$$\boldsymbol{\mu}(t+1) = (1-\alpha)\boldsymbol{\mu}(t) + \alpha W(t)\left(\boldsymbol{\mu}(t) + \boldsymbol{\epsilon}(t)\right) \tag{12}$$

$$= M(t)\boldsymbol{\mu}(t) + \alpha W(t)\boldsymbol{\epsilon}(t), \tag{13}$$

where:

$$M(t) = (1-\alpha)I + \alpha W(t). \tag{14}$$

5.2 Static Topology Analysis

For static W (time-invariant) and connected G:

Theorem 5.1 (Consensus and Stability)**.** *Let W be a fixed row-stochastic matrix associated with a connected graph. For any $0 < \alpha < 1$, the noiseless update*

$$\boldsymbol{\mu}(t+1) = M\boldsymbol{\mu}(t) \tag{15}$$

converges to a consensus point $c\mathbf{1}$, where c is a scalar determined by the initial condition.

Proof. W has simple eigenvalue 1 with right eigenvector $\mathbf{1}$, and all other eigenvalues λ_k satisfy $|\lambda_k| < 1$. Then $M = I + \alpha(W - I)$ has eigenvalues:

$$\lambda_k(M) = 1 + \alpha(\lambda_k(W) - 1). \tag{16}$$

For $\lambda_1(W) = 1$, we get $\lambda_1(M) = 1$. For $|\lambda_k(W)| < 1$, it follows that $|\lambda_k(M)| < 1$ for $k > 1$. Thus all non-consensus modes decay, and the solution converges to the eigenspace spanned by $\mathbf{1}$, i.e., consensus. □

5.3 Lyapunov Disagreement Decay

Define $V(t) = \|\boldsymbol{\delta}(t)\|^2$. Under mild assumptions, one can show:

$$\mathbb{E}[V(t+1)] \leq (1 - \kappa)\mathbb{E}[V(t)] + c, \tag{17}$$

for some $\kappa > 0$ and $c \geq 0$, implying exponential convergence to a noise-limited coherence.

6 Multidimensional Latent Dynamics

Consider a latent state $x(t) \in \mathbb{R}^d$ evolving as:

$$x(t+1) = Ax(t) + \xi(t), \tag{18}$$

with $A \in \mathbb{R}^{d \times d}$ and process noise $\xi(t)$.

Each node i can maintain a prediction:

$$\mu_i^{\text{pred}}(t+1) = A\mu_i(t), \tag{19}$$

then observe neighbor emissions and relax toward them:

$$\mu_i(t+1) = (1 - \alpha_i)\mu_i^{\text{pred}}(t+1) + \alpha_i \sum_{j \in N(i)} w_{ij}(t) y_j(t). \tag{20}$$

This results in a distributed, message-less approximation to a Kalman filter where:

- each node acts as a local filter,
- the global population behaves like a redundant estimator,
- no explicit measurement messages are ever transmitted.

7 Information-Theoretic Perspective

7.1 Simplified Scalar Model

Consider a scalar latent $x^\star \in \mathbb{R}$, constant in time. Each node emits:

$$y_i(t) = x^\star + n_i(t), \tag{21}$$

with $n_i(t) \sim \mathcal{N}(0, \eta^2)$ i.i.d.

If a central observer collected all $y_i(t)$ across N nodes and T steps, the Fisher information is:

$$\mathcal{I}(x^\star) = \frac{NT}{\eta^2}, \tag{22}$$

and the Cramér–Rao bound is:

$$\mathrm{Var}(\hat{x}) \geq \frac{\eta^2}{NT}. \tag{23}$$

The Whisper Graph does not explicitly centralize these samples; instead, the belief field implicitly aggregates them via local coupling.

7.2 Effective Capacity

Effective capacity is limited by:

- noise level η,
- update rate α,
- graph topology and mixing time,
- additional jitter and variance floors for stealth.

Qualitatively:

$$\text{capacity} \propto \frac{\text{coherence strength} \times \text{update rate}}{\text{noise power} \times \text{mixing time}}. \tag{24}$$

8 Adversarial and Security Model

8.1 Threat Classes

We distinguish:

- **Passive observer:** Records $y_i(t)$, attempts to infer presence of communication and latent state.
- **Active perturber:** Injects noise or biases into some emissions.
- **Compromised node:** Gains control of internal state at node i, can emit arbitrary values.

8.2 Security Goals

- **Stealth:** Hard to distinguish Whisper activity from environmental noise.
- **Robustness:** Coherence degrades gracefully under perturbations.
- **Deception resistance:** Compromised nodes have bounded influence due to trust weighting.

8.3 Trust Weight Adaptation

Nodes can maintain a coherence score $\kappa_{ij}(t)$ for neighbor j, e.g., based on how often $y_j(t)$ aligns with local predictions. Then:

$$w_{ij}(t+1) = \rho w_{ij}(t) + (1-\rho)\tilde{w}_{ij}(t), \tag{25}$$

where $\tilde{w}_{ij}(t)$ is normalized over $N(i)$ and proportional to $\kappa_{ij}(t)$, and $0 < \rho < 1$ controls adaptation speed.

9 Detectability and Forensics

9.1 Correlation-Based Detection

An external analyst might compute cross-correlations:

$$R_{ij}(\tau) = \mathbb{E}\big[y_i(t)y_j(t+\tau)\big], \tag{26}$$

and test against a null hypothesis of independent noise. Whisper operations that create coherence will induce nonzero R_{ij}.

9.2 Stealth Strategies

To reduce detectability:

- maintain low update rates α,
- keep coherence subtle and time-limited,
- embed emissions within plausible physical processes (e.g., sensor jitter),
- vary η and $\Sigma_{\min}$ to mimic environmental statistics.

There is a balance between usable coherence and stealth.

10 Simulation Framework and Reference Code

10.1 Scalar Whisper Graph Simulation

We reproduce and slightly refine the scalar reference simulation.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
whisper_graph_sim_scalar.py

Scalar Whisper Graph simulation:
- Random geometric topology
- Gaussian beliefs (mean + variance)
- One biased source node
"""

import math
import random
from dataclasses import dataclass, field
from typing import List, Dict

# --------------------------
# Global Configuration
# --------------------------

N_NODES: int = 50
AREA_SIZE: float = 1.0
NEIGHBOR_RADIUS: float = 0.3

ALPHA: float = 0.03
BETA: float = 0.02
GAMMA: float = 0.001
ETA: float = 0.03
SIGMA_MIN: float = 0.05

N_STEPS: int = 500
SOURCE_NODE: int = 0
SOURCE_TARGET: float = 1.0
SOURCE_STRENGTH: float = 0.01

RANDOM_SEED: int = 17


@dataclass
class NodeState:
    idx: int
    x: float
    y: float
    neighbors: List[int] = field(default_factory=list)
    mu: float = 0.0
    sigma2: float = 1.0
```

```
    alpha: float = ALPHA
    beta: float = BETA
    gamma: float = GAMMA
    eta: float = ETA
    sigma_min: float = SIGMA_MIN

    def emit_field(self) -> float:
        noise = random.gauss(0.0, self.eta)
        return self.mu + noise


class WhisperGraphSim:
    def __init__(self, n_nodes: int = N_NODES):
        random.seed(RANDOM_SEED)
        self.nodes: List[NodeState] = []
        self._init_positions(n_nodes)
        self._init_neighbors()
        self.history_mu: List[List[float]] = []
        self.history_coherence: List[float] = []

    def _init_positions(self, n_nodes: int):
        for i in range(n_nodes):
            x = random.random() * AREA_SIZE
            y = random.random() * AREA_SIZE
            node = NodeState(idx=i, x=x, y=y)
            self.nodes.append(node)

    def _init_neighbors(self):
        # geometric neighbors
        for i, ni in enumerate(self.nodes):
            for j, nj in enumerate(self.nodes):
                if i == j:
                    continue
                dx = ni.x - nj.x
                dy = ni.y - nj.y
                dist = math.hypot(dx, dy)
                if dist <= NEIGHBOR_RADIUS:
                    ni.neighbors.append(j)
        # ensure connectivity
        for i, ni in enumerate(self.nodes):
            if not ni.neighbors:
                best_j = None
                best_d = 1e9
                for j, nj in enumerate(self.nodes):
                    if i == j:
                        continue
```

```
                    dx = ni.x - nj.x
                    dy = ni.y - nj.y
                    dist = math.hypot(dx, dy)
                    if dist < best_d:
                        best_d = dist
                        best_j = j
                if best_j is not None:
                    ni.neighbors.append(best_j)

    def _compute_coherence(self) -> float:
        mus = [n.mu for n in self.nodes]
        mean_mu = sum(mus) / len(mus)
        avg_dev = sum(abs(m - mean_mu) for m in mus) / len(mus)
        sigma_noise = ETA if ETA > 0 else 1.0
        coherence = 1.0 - (avg_dev / sigma_noise)
        return coherence

    def step(self, t: int):
        emissions: Dict[int, float] = {}
        for n in self.nodes:
            emissions[n.idx] = n.emit_field()

        # source node bias
        src = self.nodes[SOURCE_NODE]
        src.mu = (1.0 - SOURCE_STRENGTH) * src.mu + SOURCE_STRENGTH *
            SOURCE_TARGET

        new_mu: Dict[int, float] = {}
        new_sigma2: Dict[int, float] = {}

        for n in self.nodes:
            if not n.neighbors:
                m_next = (1.0 - n.alpha) * n.mu
            else:
                neighbor_vals = [emissions[j] for j in n.neighbors]
                neighbor_mean = sum(neighbor_vals) / len(neighbor_vals)
                m_next = (1.0 - n.alpha) * n.mu + n.alpha * neighbor_mean

            # memory decay and variance floor
            m_next = (1.0 - n.gamma) * m_next
            s2_next = (1.0 - n.beta) * n.sigma2 + n.beta * (n.sigma_min **
                2)

            new_mu[n.idx] = m_next
            new_sigma2[n.idx] = s2_next
```

```
        for n in self.nodes:
            n.mu = new_mu[n.idx]
            n.sigma2 = new_sigma2[n.idx]

        self.history_mu.append([n.mu for n in self.nodes])
        self.history_coherence.append(self._compute_coherence())

    def run(self, n_steps: int = N_STEPS):
        for t in range(n_steps):
            self.step(t)

    def dump_summary(self):
        print("=== Whisper Graph Scalar Simulation Summary ===")
        print(f"Nodes: {len(self.nodes)}")
        print(f"Steps: {len(self.history_mu)}")
        print(f"Source node: {SOURCE_NODE}, target={SOURCE_TARGET}")
        if self.history_coherence:
            print(f"Final coherence: {self.history_coherence[-1]:.4f}")
        mus_final = [n.mu for n in self.nodes]
        avg_mu = sum(mus_final) / len(mus_final)
        print(f"Final average mu: {avg_mu:.4f}")


if __name__ == "__main__":
    sim = WhisperGraphSim()
    sim.run()
    sim.dump_summary()
```

Listing 1: Scalar Whisper Graph simulation

10.2 Multidimensional Whisper Graph Simulation

We now provide a vector-valued ($d = 2$) simulation where each node tracks a 2D latent.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
whisper_graph_sim_vector.py

Vector (2D) Whisper Graph simulation:
- Each node tracks mu in R^2
- Same topology as scalar case
"""

import math
import random
from dataclasses import dataclass, field
```

```
from typing import List, Dict, Tuple

D: int = 2  # latent dimension

N_NODES: int = 40
AREA_SIZE: float = 1.0
NEIGHBOR_RADIUS: float = 0.35

ALPHA: float = 0.02
BETA: float = 0.02
GAMMA: float = 0.001
ETA: float = 0.04
SIGMA_MIN: float = 0.05

N_STEPS: int = 400
SOURCE_NODE: int = 0
SOURCE_TARGET: Tuple[float, float] = (1.0, -0.5)
SOURCE_STRENGTH: float = 0.01

RANDOM_SEED: int = 23


@dataclass
class VecNodeState:
    idx: int
    x: float
    y: float
    neighbors: List[int] = field(default_factory=list)
    mu: List[float] = field(default_factory=lambda: [0.0] * D)
    sigma2: float = 1.0
    alpha: float = ALPHA
    beta: float = BETA
    gamma: float = GAMMA
    eta: float = ETA
    sigma_min: float = SIGMA_MIN

    def emit_field(self) -> List[float]:
        vals = []
        for i in range(D):
            noise = random.gauss(0.0, self.eta)
            vals.append(self.mu[i] + noise)
        return vals


class WhisperGraphSimVector:
    def __init__(self, n_nodes: int = N_NODES):
```

```
        random.seed(RANDOM_SEED)
        self.nodes: List[VecNodeState] = []
        self._init_positions(n_nodes)
        self._init_neighbors()
        self.history_mu: List[List[List[float]]] = []
        self.history_coherence: List[float] = []

    def _init_positions(self, n_nodes: int):
        for i in range(n_nodes):
            x = random.random() * AREA_SIZE
            y = random.random() * AREA_SIZE
            node = VecNodeState(idx=i, x=x, y=y)
            self.nodes.append(node)

    def _init_neighbors(self):
        for i, ni in enumerate(self.nodes):
            for j, nj in enumerate(self.nodes):
                if i == j:
                    continue
                dx = ni.x - nj.x
                dy = ni.y - nj.y
                dist = math.hypot(dx, dy)
                if dist <= NEIGHBOR_RADIUS:
                    ni.neighbors.append(j)
        for i, ni in enumerate(self.nodes):
            if not ni.neighbors:
                best_j = None
                best_d = 1e9
                for j, nj in enumerate(self.nodes):
                    if i == j:
                        continue
                    dx = ni.x - nj.x
                    dy = ni.y - nj.y
                    dist = math.hypot(dx, dy)
                    if dist < best_d:
                        best_d = dist
                        best_j = j
                if best_j is not None:
                    ni.neighbors.append(best_j)

    def _compute_coherence(self) -> float:
        # vector coherence: avg norm deviation from mean
        mus = [n.mu for n in self.nodes]
        mean_mu = [sum(m[i] for m in mus) / len(mus) for i in range(D)]
        devs = []
        for m in mus:
```

```
            dx = m[0] - mean_mu[0]
            dy = m[1] - mean_mu[1]
            devs.append(math.hypot(dx, dy))
        avg_dev = sum(devs) / len(devs)
        sigma_noise = ETA if ETA > 0 else 1.0
        coherence = 1.0 - (avg_dev / sigma_noise)
        return coherence

    def step(self, t: int):
        emissions: Dict[int, List[float]] = {}
        for n in self.nodes:
            emissions[n.idx] = n.emit_field()

        # source node bias towards 2D target
        src = self.nodes[SOURCE_NODE]
        for k in range(D):
            src.mu[k] = (1.0 - SOURCE_STRENGTH) * src.mu[k] \
                        + SOURCE_STRENGTH * SOURCE_TARGET[k]

        new_mu: Dict[int, List[float]] = {}
        new_sigma2: Dict[int, float] = {}

        for n in self.nodes:
            if not n.neighbors:
                m_next = [(1.0 - n.alpha) * v for v in n.mu]
            else:
                # average neighbors in R^2
                sum_vec = [0.0] * D
                for j in n.neighbors:
                    for k in range(D):
                        sum_vec[k] += emissions[j][k]
                neighbor_mean = [sv / len(n.neighbors) for sv in sum_vec]

                m_next = []
                for k in range(D):
                    val = (1.0 - n.alpha) * n.mu[k] + n.alpha *
                       neighbor_mean[k]
                    # memory decay
                    val = (1.0 - n.gamma) * val
                    m_next.append(val)

            s2_next = (1.0 - n.beta) * n.sigma2 + n.beta * (n.sigma_min **
                2)
            new_mu[n.idx] = m_next
            new_sigma2[n.idx] = s2_next
```

```
        for n in self.nodes:
            n.mu = new_mu[n.idx]
            n.sigma2 = new_sigma2[n.idx]

        self.history_mu.append([n.mu[:] for n in self.nodes])
        self.history_coherence.append(self._compute_coherence())

    def run(self, n_steps: int = N_STEPS):
        for t in range(n_steps):
            self.step(t)

    def dump_summary(self):
        print("=== Whisper Graph Vector Simulation Summary ===")
        print(f"Nodes: {len(self.nodes)}")
        print(f"Steps: {len(self.history_mu)}")
        print(f"Source node: {SOURCE_NODE}, target={SOURCE_TARGET}")
        if self.history_coherence:
            print(f"Final coherence: {self.history_coherence[-1]:.4f}")
        mus_final = [n.mu for n in self.nodes]
        mean_final = [sum(m[k] for m in mus_final) / len(mus_final) for k
           in range(D)]
        print(f"Final average mu: {tuple(round(v, 4) for v in mean_final)}
           ")


if __name__ == "__main__":
    sim = WhisperGraphSimVector()
    sim.run()
    sim.dump_summary()
```

Listing 2: Vector Whisper Graph simulation (2D latent)

11 Implementation Architecture and Hardware

11.1 Node Software Modules

A reference software architecture for a Whisper node:

- **Belief Engine:** Maintains μ_i, Σ_i; implements update rules.
- **Field Sampler:** Observes analog emissions from neighbors.
- **Noise Injector:** Applies emission noise η_i.
- **Trust Manager:** Adapts weights $w_{ij}(t)$.
- **Scheduler:** Controls update interval dt.

11.2 Physical Carriers

Examples:

- sub-GHz radios broadcasting minimal analog-like beacons,
- BLE advertising signals with jittered RSSI patterns,
- optical intensity flicker (LED arrays),
- ultrasonic modulation under noise constraints.

Care must be taken that the carrier does not reintroduce explicit packet semantics at the observable layer.

12 Benchmark Suite

12.1 Core Benchmarks

1. **Source Bias Propagation**: time to reach target coherence C_{target}.
2. **Eavesdropper Impact**: coherence loss when a probing node is introduced.
3. **Compromised Node Deception**: shift in global mean under targeted attacks.
4. **Mobility**: re-coherence after dynamic topology changes.

Each benchmark records:

- $C(t)$ curves,
- global mean trajectories,
- detectability metrics.

13 Figures

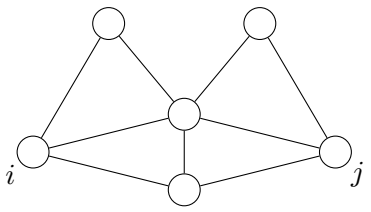

Figure 1: Example local neighborhood in a Whisper Graph.

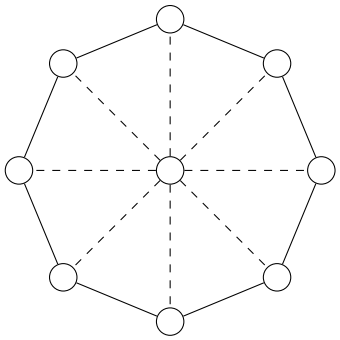

Figure 2: Ring topology with a central integrating node.

14 Ethical Considerations

Whisper Graph systems enable communication that is intentionally hard to detect and analyze. This can be beneficial (privacy, resilience, censorship evasion) but also poses serious misuse potential.

Deployers should:

- align usage with applicable law and regulation,
- evaluate human rights and civil liberties impact,
- avoid deployments that facilitate oppression or harm,
- maintain appropriate oversight and auditing.

15 Conclusion

We have developed the Whisper Graph from concept to full mathematical and implementation detail. Nodes refrain from exchanging messages, instead shaping a shared probabilistic field through continuous, noisy leaning. Coherence in this field is the carrier of information.

We have defined the core dynamics, analyzed stability, connected the system to distributed filtering and information theory, introduced a rigorous adversarial and detectability model, provided complete reference simulations, and specified a full protocol.

This forms a foundation for concrete implementations of ultra-low-power swarms, stealth networks, and collective robotics under the Whisper Graph paradigm.

A Appendix A: Whisper Graph Protocol (WGP) RFC Specification

A.1 Status of This Memo

This memo specifies the Whisper Graph Protocol (WGP), an experimental communication protocol wherein nodes do not exchange explicit messages. Distribution is unlimited.

A.2 Conventions

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** are to be interpreted as described in RFC 2119.

A.3 Architecture Overview

WGP defines:

- internal node belief states over latent variables,
- emission functions mapping belief to noisy field values,
- neighbor coupling rules for probabilistic relaxation,
- requirements that prevent reintroduction of message-like semantics.

A.4 Node State Variables

Each node i maintains:

$$S_i(t) = \{p_i(x,t), \mu_i(t), \Sigma_i(t), \alpha_i, \gamma_i, \eta_i, W_i(t)\}, \tag{27}$$

where:

- $p_i(x,t)$: internal belief distribution,
- $\mu_i(t)$: mean parameter,
- $\Sigma_i(t)$: covariance parameter,
- α_i: update rate,
- γ_i: memory decay,
- η_i: emission noise amplitude,
- $W_i(t)$: neighbor weight vector $(w_{ij}(t))_{j \in N(i)}$.

A.5 Emission Function

At each step t, node i computes:

$$y_i(t) = \mu_i(t) + \epsilon_i(t), \tag{28}$$

where $\epsilon_i(t)$ is zero-mean noise with variance η_i^2.

Nodes **MUST** ensure:

- emission noise is non-zero,
- emissions are not discretized into explicit symbols at the WGP layer.

A.6 Belief Update Rule

Nodes **MUST** implement a relaxation rule of the form:

$$\mu_i(t+1) = (1-\alpha_i)\mu_i(t) + \alpha_i \sum_{j \in N(i)} w_{ij}(t) y_j(t), \tag{29}$$

with $\sum_{j \in N(i)} w_{ij}(t) = 1$.

Nodes **MUST** enforce a covariance floor:

$$\Sigma_i(t+1) \succeq \Sigma_{\min} \succ 0. \tag{30}$$

A.7 Prohibited Behaviors

Nodes **MUST NOT**:

- transmit explicit messages or packets at the WGP semantics layer,
- encode information via abrupt on/off patterns or bursts that create clear symbol boundaries,
- expose the full belief distribution $p_i(x,t)$ to peers,
- collapse variance to zero or near-zero.

A.8 Recommended Parameter Ranges

Parameter	Description	Typical Range
α_i	update rate	0.001–0.05
γ_i	memory decay	0.0001–0.02
η_i	emission noise	1–5% of value scale
$\Sigma_{\min}$	covariance floor	small SPD matrix
dt	update period	20–200 ms

A.9 Security Considerations

WGP is intentionally stealth-oriented. Implementations **SHOULD**:

- maintain variance floors and noise levels appropriate to the environment,
- avoid deterministic or highly periodic emission patterns,
- implement trust adaptation to reduce influence of anomalous nodes.

A.10 Interoperability

Implementations **MAY** use any physical carrier capable of real-valued noisy emission, provided the WGP semantics are preserved and the behavior satisfies all above requirements.

B Appendix B: Additional Experiments

B.1 Multi-Source Competition

Benchmark network behavior when multiple nodes act as independent sources with conflicting targets. Key metrics:

- regional clustering of beliefs,
- time-varying dominance patterns,
- ability to maintain partial coherence.

B.2 Topology Rewiring

Simulate dynamic edge creation and deletion. Measure:

- recovery time to steady coherence,
- robustness of global convergence under mobility.

C Appendix C: Extended TikZ Figures

D Acknowledgements

Developed under OCTA Research initiatives in probabilistic communication, distributed inference, and stealth networks.

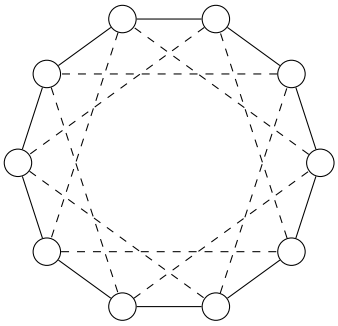

Figure 3: Dense Whisper Graph ring with additional dashed shortcuts for faster mixing.