Final Report Template
Distributed Neural Training Simulation

Final Report for the Distributed Systems Course 2025/26

Colletta Lorenzo
lorenzo.colletta@studio.unibo.it

Giacobbi Domenico Francesco
domenico.giacobbi@studio.unibo.it

Fantilli Giorgio
giorgio.fantilli@studio.unibo.it

July 12, 2026

Distributed Neural Training Simulation (DNTS) is a software framework designed to simulate, analyze, and visualize the training process of Deep Neural Networks (DNN) in a fully decentralized Peer-to-Peer (P2P) environment. Traditional distributed machine learning typically relies on centralized architectures, such as Parameter Servers, which inherently introduce a Single Point of Failure (SPOF) and potential network bottlenecks. DNTS overcomes these limitations by implementing an asynchronous, leaderless architecture based on the Gossip Learning protocol. Within this framework, autonomous nodes process local data partitions and iteratively exchange model weights with randomly selected peers to achieve global consensus and model convergence. Developed entirely in Scala 3 utilizing the Actor Model (via Akka Cluster) and strict Pure Functional Programming principles.

Contents

1 Concept

1.1 Type of Product

The developed system is a hybrid application comprising two main interfaces:

- **Command-Line Interface (CLI):** This serves as the primary control mechanism. Users leverage the CLI to bootstrap cluster nodes (assigning roles such as *seed* or *client*), specify network binding parameters (IP addresses and ports), and load configuration files.
- **Desktop Graphical User Interface (GUI):** A local visualization dashboard that provides real-time, dynamic plotting. It allows users to visually monitor the model's evolution during the simulation, specifically rendering the Decision Boundary, the Loss Function and a Consensus Metric trends across the distributed cluster. It also allows users to dispatch operational commands (e.g., initiate training, pause/resume, stop, or simulate a node crash).

1.2 Use Case Description

- **What is the software doing?** DNTS allows users to define a neural network topology alongside training hyperparameters to launch a fully decentralized training simulation. Within this environment, N independent nodes learn from locally partitioned 2D labeled datasets to solve a classification task. Utilizing an asynchronous Peer-to-Peer (P2P) Gossip Learning algorithm, nodes continuously exchange and average their local weights to converge toward a globally optimized shared model. The software also implements fault-tolerance mechanisms, such as *Passive Gossiping* and local state snapshotting, to seamlessly recover from unexpected node failures.
- **Where are the users?** The primary users are researchers, computer science students, and data scientists studying distributed systems or decentralized machine learning. They can deploy the nodes either locally on a single machine (for testing and simulation purposes) or physically distribute them across a network (e.g., university laboratory workstations or virtual machines communicating via TCP/TLS).
- **When and how do they interact?** Interaction is predominantly required during the initialization phase, writing configuration files and launching the system via CLI. Once the simulation is running, interaction shifts to passive monitoring via the GUI. However, users can actively intervene using GUI commands to inject faults (simulating crashes) or alter the simulation state. The interaction occurs on standard desktop operating systems (Windows, macOS, Linux) utilizing a cross-platform `Fat JAR` executable.
- **Does the system store user data?** The system does not collect or store any personal user data. It exclusively handles application-specific data. Precisely, it

persists synthetic 2D datasets and the serialized state of the neural network. This data is saved locally on the filesystem of each respective node (e.g., as binary files) to enable the crash-recovery mechanisms and guarantee the persistence of the training progress without relying on a centralized database.

1.3 Why is Distribution Needed?

Distribution is an intrinsic requirement to address the core challenges of modern, large-scale machine learning, specifically targeting the following architectural and operational aspects:

- **Elimination of the Single Point of Failure (SPOF):** Traditional distributed machine learning relies heavily on a centralized Parameter Server, which acts as a critical SPOF and a potential network bottleneck. A leaderless P2P architecture ensures high availability and resilience; if a node crashes, the training process continues uninterrupted among the surviving peers.
- **Data Locality and Privacy:** In many real-world scenarios (such as edge computing or healthcare applications), datasets cannot be centralized due to sheer volume constraints or strict privacy regulations. DNTS inherently models this constraint by forcing nodes to learn exclusively from their local data partitions, achieving a global consensus purely through the exchange of model parameters.
- **Computational Parallelism:** Deep learning tasks are highly resource-intensive. Distributing the dataset allows the system to parallelize the heavy computational load of gradient descent (Backpropagation) across multiple autonomous entities, effectively leveraging the computational power of the entire cluster.

2 Requirements Elicitation and Analysis

2.1 User Requirements

User requirements define the system from the perspective of the end-user (researchers or students) interacting with the application.

UR1 - Cluster Initialization: The user must be able to bootstrap the distributed cluster and connect new nodes by specifying network parameters (e.g., host IP, ports, and seed nodes) via configuration files or initial CLI arguments.

UR2 - Simulation Setup: Before initiating the training, the user must be able to define the Neural Network topology (number of hidden layers, neurons per layer, activation functions) and learning hyperparameters.

UR3 - Dataset Partitioning: The user must be able to select a specific 2D synthetic dataset (e.g., XOR, Spiral, Ring, Gaussian) which the system will automatically generate and partition across the active nodes.

UR4 - Interactive Simulation Control (GUI): The user must be able to actively control the simulation lifecycle—triggering Start, Pause, Resume, and Stop commands—directly through the Graphical User Interface.

UR5 - Fault Injection (GUI): To observe and test the system's resilience, the user must be able to simulate a node crash (abrupt disconnection) dynamically via the GUI during the training phase.

UR6 - Real-Time Visualization: The user must be provided with a real-time graphical representation of the learning process, including a plot of the evolving Decision Boundary and charts tracking the Loss Function and a Consensus Metric across the cluster.

2.2 Functional System Requirements

These requirements specify the internal computational and distributed behaviors the system must exhibit to satisfy the user requirements.

FR1 - P2P Membership Management: The system must autonomously handle node discovery, join events, and failure detections without relying on a centralized registry, maintaining an updated view of the cluster membership.

FR2 - Asynchronous Gossip Learning: The training logic must execute in a decentralized manner. Nodes must alternate between local computation cycles (Backpropagation on local data) and asynchronous synchronization cycles (Model Averaging) with randomly selected peers.

FR3 - Distributed Consensus Metric: The system must be capable of aggregating a global metric that evaluates the distance/variance between the diverse local models to measure the convergence (consensus) of the distributed network.

FR4 - Pure Functional Backpropagation: The mathematical core of the neural network (Gradient Descent and Backpropagation) must be implemented utilizing pure functions and immutable data structures to avoid race conditions.

FR5 - State Snapshotting & Passive Gossiping: To tolerate faults, nodes must periodically serialize and save their state (Model Weights and Maturity/Epochs) locally. Upon recovery, the system must utilize a weighted "Passive Gossiping" strategy to reintegrate the recovered node without degrading the global consensus.

2.3 Non-Functional Requirements

Non-functional requirements dictate the quality attributes and architectural constraints of the distributed system.

NFR1 - Resilience and Fault Tolerance: The system must be highly available. The failure of one or multiple nodes (up to a critical threshold) must not halt the global training process. Surviving nodes must continue their gossip cycles seamlessly.

NFR2 - Distribution Transparency: The complexity of the network (message routing, serialization, TCP connection drops) must be hidden from the pure domain logic. The mathematical core must remain completely agnostic to the network topology.

NFR3 - Network Efficiency: Given the necessity to transmit entire matrices of weights during the gossip phase, the system must employ highly efficient, low-overhead binary serialization to minimize network latency and bandwidth saturation.

2.4 Implementation Requirements

To satisfy the administrative constraints set by the Distributed Systems course curriculum, and to optimize the limited development time (economic constraint), the system realization is bound to the following choices:

IR1 - Language Paradigm: The system must be developed entirely in Scala 3, rigorously applying Functional Programming (FP) principles. This is an administrative requirement to validate academic proficiency, which also elegantly serves the design need for absolute data immutability during complex mathematical computations.

IR2 - Concurrency Model: The Actor Model must be used as the exclusive concurrency paradigm. Specifically, the system must utilize `Akka Typed` for strictly typed, asynchronous message passing and `Akka Cluster` for distributed membership and failure detection. This is justified by the economic need to rapidly deploy a reliable P2P network within the project deadline, avoiding the prohibitive time cost of writing a custom TCP middleware from scratch.

IR3 - Architectural Pattern: The codebase must adhere to a strict Layered Architecture (Separation of Concerns), clearly dividing the Pure Functional Core (Domain) from the Stateful Distributed Layer (Actors) and the Boundary Layer (View/GUI).

Rather than an administrative constraint, this emerges as a direct consequence of the system design to ensure full distribution transparency and testability.

2.5 Relevant Distributed System Features

The main priorities are strong decentralization, long-term convergence, robustness to failures, and efficient use of computation and communication. Full distribution transparency is deliberately constrained to preserve complete observability of the simulation.

Distribution Transparency. *Partially relevant.* Low-level network details (such as message routing and physical location) must be abstracted away from the pure mathematical core. However, the distributed behavior must remain visible to the user: delays, topology changes, liveness, and the evolving divergence/convergence of the models are intentionally observable via the monitoring dashboard.

Fault Tolerance and Recoverability. *Highly relevant.* The global training process must survive unexpected node crashes or network partitions. Surviving nodes must continue their local learning and gossip cycles seamlessly. Reconnected nodes must be able to recover their missing state from local snapshots and gracefully reintegrate into the cluster without polluting the global consensus.

Consistency. *Highly relevant (Eventual Consistency).* The system abandons strong, immediate consistency. Nodes' local neural network weights are expected and allowed to diverge during local computation phases. The system relies on an eventual consistency model, utilizing asynchronous Gossip averaging (deterministic conflict resolution) to slowly converge toward a unified global state.

Scalability and Resource Sharing. *Highly relevant.* Nodes share the computational burden and the global dataset by holding local partitions. By adopting a randomized, full-mesh or partial-mesh Gossip communication topology, the system prevents central bandwidth bottlenecks, ensuring that the communication overhead is distributed uniformly to support horizontal scaling.

Performance, Concurrency, and Communication Efficiency. *Highly relevant.* The system must support true concurrent execution: nodes must compute heavy Gradient Descent mathematics while simultaneously serving and receiving network replication messages. Furthermore, because transmitting entire neural network matrices generates massive payloads, optimizing bandwidth efficiency via minimal, custom communication protocols is a strict architectural constraint.

Security and Trust. *Relevant.* DNTS operates over TCP/IP in potentially shared academic environments. Basic transport encryption (mTLS) and cluster access control (password-protected joining and user authentication) are required to prevent unauthorized tampering with the simulation dataset and the consensus process.

Openness, Interoperability, and Heterogeneity. *Relevant.* The architecture must clearly separate the pure algorithmic domain (neural networks) from the distributed runtime logic, allowing future developers to swap out the networking middleware without altering the math. The system must also run seamlessly across heterogeneous commodity operating systems (Windows, macOS, Linux).

Economy and Cost. *Relevant as a constraint.* The project must execute efficiently on standard commodity hardware (e.g., student laptops or basic university laboratory workstations) without relying on expensive, paid managed infrastructure or specialized high-performance centralized clusters.

3 Design

The design of the Distributed Neural Training Simulation (DNTS) framework is driven by the core requirements of high decentralization, resilience, and strict separation of concerns. The system rejects traditional centralized learning paradigms in favor of a hybrid approach that integrates a decentralized Peer-to-Peer (P2P) network topology with a highly disciplined, layered internal architecture within each node.

3.1 Infrastructure

The infrastructure of DNTS is intentionally minimal and strictly decentralized. In the base deployment, the system consists solely of N homogeneous node processes. Each node embeds the complete runtime stack—including the domain core, distributed state management, and TCP networking—without relying on any external infrastructural components such as centralized databases, message brokers, or master coordinators.

Node distribution over the physical or virtual network follows a symmetric P2P model:

- All nodes act as equivalent peers communicating through bidirectional TCP connections, secured via mutual TLS.
- Nodes can be deployed on a single physical machine (e.g., binding to different local ports for development and visual simulation) or distributed across a network.
- The only strict infrastructural requirement is network-level mutual reachability across the configured inter-node endpoints.

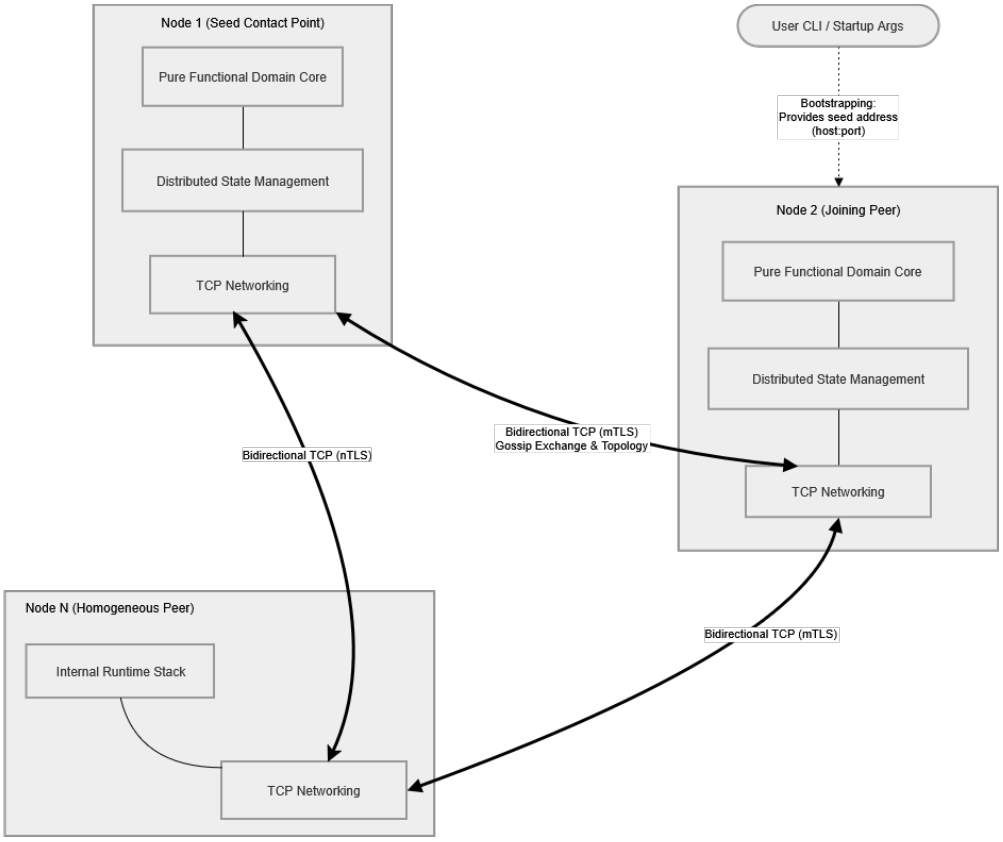

Figure 1: Infrastructure-level node connectivity in the base full-mesh deployment.

Node discovery and topology formation are bootstrap-based and membership-driven, avoiding any centralized DNS or registry:

- **Addressing:** Nodes are explicitly identified over the network using their `host:port` combinations.
- **Bootstrapping:** To join the overlay network, a new node only needs the address of at least one active contact point (a *seed node*). This address is dynamically provided by the user at startup via Command-Line Interface (CLI) arguments, eliminating the need for hard-coded network registries.
- **Membership Maintenance:** Once the initial connection to the seed is established, progressive discovery and cluster-view maintenance are handled autonomously through continuous gossip exchange among peers. However, to preserve the consistency of the distributed dataset, this openness is constrained: once the actual training phase begins, the topology temporarily freezes, rejecting entirely new nodes while selectively allowing known, recovering nodes to rejoin.

3.2 Modelling

Operating over the decentralized infrastructure described above, the system model enforces the Separation of Concerns between the purely mathematical machine learning domain and the distributed runtime execution.

3.2.1 Domain Entities

The core domain is modeled through immutable, pure functional data structures:

- **NNModel:** The mathematical representation of the Neural Network, encapsulating weight matrices, bias vectors, and a *Maturity* indicator (number of epochs trained).
- **Dataset & DataPartition:** The synthetic 2D labeled data points generated for the simulation, structurally divided into local partitions.
- **TopologyConfig:** The hyperparameters defining the network (hidden layers, activation functions, learning rate, etc.).
- **Metrics:** Observability records including Loss values, Decision Boundary coordinates, and the global Consensus Variance.

3.2.2 Entity-to-Infrastructure Mapping

Given the pure P2P architecture, there is no centralized database or master server holding the "ground truth." Entities map to the infrastructural nodes symmetrically:

- **Dataset Mapping:** The global dataset exists only conceptually. In practice, it is physically partitioned during initialization, and each Node securely stores exactly $1/N$ of the data on its local filesystem.
- **Model Mapping:** Every Node holds a complete, local replica of the `NNModel` within its memory space. The functional `NNModel` entity is explicitly wrapped and managed by a stateful infrastructural component (the `ModelActor`, detailed in Section 3.3).
- **Metrics Mapping:** Local metrics are generated on each Node and continuously pushed to the boundary layer (GUI) for visualization, without being persisted to a global database.

3.2.3 System State

The overall state of a single DNTS node comprehends both domain and distributed dimensions:

- **Persistent/Recoverable State:** The local `NNModel` weights, its current *Maturity*, and the allocated `DataPartition`. This state is periodically serialized to the local disk to allow crash recovery.

- **Volatile/Runtime State:** The internal status of the training state machine (e.g., *Idle, Training, Paused*), the current gradient descent calculations, and the local view of the active P2P cluster membership.

3.2.4 Messages and Events

Interaction within and between nodes is modeled entirely through strongly-typed, asynchronous message passing (Algebraic Data Types):

- **Commands (Directives):** Used to trigger specific actions. Examples include `StartTraining`, `PauseSimulation`, `CrashNode` (from GUI to internal layers), and `UpdateModel` (from Trainer to Model).
- **Network Messages (Gossip & Consensus):** Exchanged inter-node over TCP. The primary payload is the `GossipModelPush`, containing a serialized `NNModel` replica sent to a random peer, which triggers a `MergeWeights` computation on the receiver's end.
- **Domain Events:** Emitted when the system state materially changes. Examples include `EpochCompleted` (triggering metric collection) or `TopologyChanged` (when the underlying cluster layer detects a node joining, leaving, or crashing).

3.3 Architecture

The internal architecture of a single DNTS node is structured to govern complexity and isolate side-effects. It follows a strict *Layered Architectural Style* consisting of three distinct macro-layers:

1. **Pure Functional Core (Domain Layer):** This layer encapsulates the entire mathematical and algorithmic domain of the deep neural network. It includes linear algebra primitives (immutable matrices and vectors), network topology configurations, backpropagation algorithms, and deterministic model-averaging operators. This layer is completely pure: it possesses zero dependencies on network libraries, actor frameworks, or I/O mechanisms, ensuring perfect unit-testability.
2. **Stateful Distributed Layer (Actor System Layer):** Positioned above the core, this layer manages concurrency, network communication, and mutable application state. Implemented using `Akka Typed`, it models the node's lifecycle and distributed behaviors through a specialized hierarchy of actors that isolate mutable states and communicate exclusively via asynchronous message passing. The primary actors include:
    - *RootActor & ClusterManager:* Bootstrap the hierarchy and handle native membership events, failure detection, and dynamic fault-tolerance policies.
    - *DiscoveryActor:* Utilizes the Akka Receptionist to dynamically index services and maintain an actively updated routing table of reachable peers.

- *GossipActor:* Drives the P2P networking protocol, asynchronously exchanging local model snapshots with remote peers.
- *ConfigurationActor:* Orchestrates the propagation and validation of the training hyperparameters across the joining nodes before the simulation starts.
- *DatasetDistribution & Consensus Actors:* Manage the initial asynchronous dataset scattering from the seed to clients, and compute global network deviation metrics.
- *TrainerActor:* The local execution engine that delegates data batches to the pure mathematical core for Gradient Descent.
- *ModelActor (Single Source of Truth):* Safely encapsulates the mutable neural network weights and biases, guaranteeing atomic and sequential updates.
- *MonitorActor:* Acts as the bridge to the Boundary Layer, aggregating telemetry and emitting periodic updates to the GUI.
- *AuthActor:* Handles user authentication and authorization, including user registration, credential verification, access token issuance and validation, and active session management.

3. **Boundary Layer (Infrastructure & View Layer):** This outermost layer manages user interaction and environment boundaries. It consists of the CLI parser, configuration loaders (`application.conf`), and the Swing based Graphical User Interface (GUI) responsible for rendering dynamic plots (Decision Boundaries, Loss curves, and Consensus metrics) without blocking the underlying distributed computation.

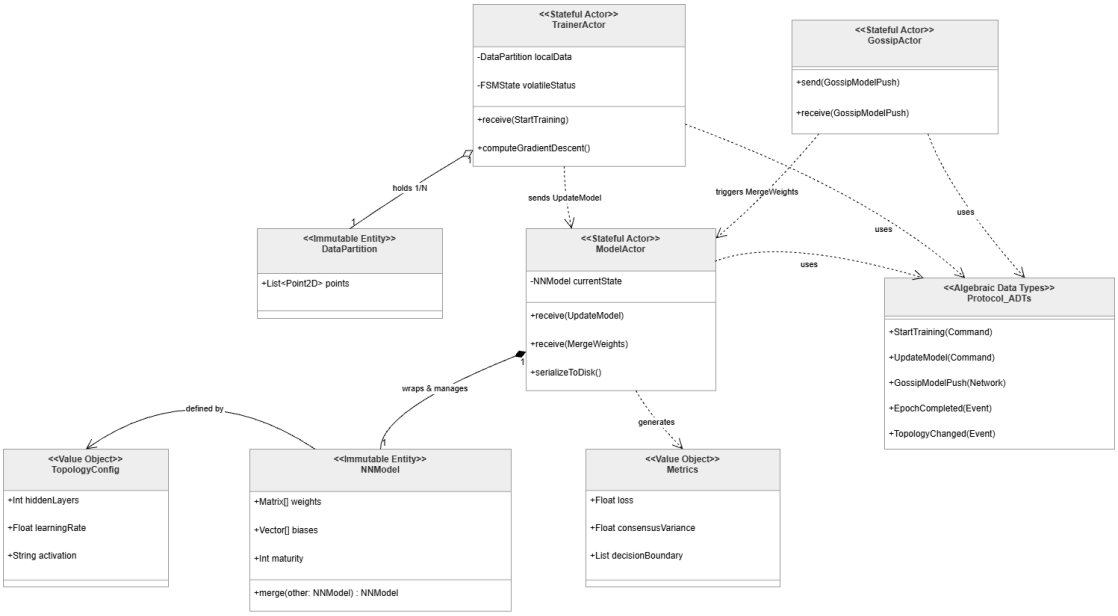

Figure 2: UML Class Diagram highlighting the mapping between the Pure Functional Domain entities (e.g., `NNModel`, `Dataset`) and the Stateful Distributed Layer (e.g., `ModelActor`, `TrainerActor`).

3.4 Interaction

The interaction paradigm in DNTS is fundamentally based on **Asynchronous Message Passing**, adhering to the Actor Model (`Akka Typed`). This approach entirely eliminates the need for traditional shared-memory synchronization primitives, delegating all coordination to the exchange of immutable messages.

To satisfy the diverse requirements of the system's lifecycle—from initial bootstrap to continuous training and observability—the architecture implements a combination of distinct distributed interaction patterns, meticulously mapped to specific actor behaviors.

3.4.1 Topology and Discovery: Publish-Subscribe

Rather than relying on static IP configurations or a centralized registry, node discovery is managed through a **Publish-Subscribe** interaction pattern via the Akka Receptionist.

- **Publish:** Upon joining the cluster, each node's `GossipActor` and `ConsensusActor` publish their references to a distributed topic (Service Key).
- **Subscribe:** The `DiscoveryActor` subscribes to these keys, asynchronously receiving listing updates whenever a peer joins, leaves, or crashes. This ensures that

every node maintains a dynamically updated, eventually consistent routing table of active peers without active polling.

3.4.2 Initialization: Point-to-Point Distribution

During the brief bootstrapping phase, the system exhibits a transient, asymmetric interaction to distribute the workload. In this phase, the Seed node generates the global synthetic dataset and uses direct **Point-to-Point** messaging (`DatasetDistributionActor`) to slice and distribute exactly $1/N$ of the data to each joining Client node. Once this initialization phase is complete, this hierarchical interaction is dismantled, and the system transitions into a pure, symmetric P2P topology.

3.4.3 Continuous Training: Push-based Gossip & Eventual Consistency

The core inter-node communication avoids blocking or strong consensus algorithms, which would impose severe latency bottlenecks when synchronizing high-dimensional matrices over a network.

- **Epidemic Dissemination (Push Gossip):** Interaction is driven by a **Push-based Gossip Protocol**. The `GossipActor` periodically selects a random peer from its routing table and asynchronously "pushes" its serialized local model state (`GossipModelPush`).
- **Conflict Resolution via Model Averaging:** The system embraces *Eventual Consistency*. When a `ModelActor` receives a remote state, it applies a deterministic merge operation. Crucially, to prevent newly joined or recovering nodes from polluting the network with randomized or stale weights, this averaging is safely weighted based on the model's Maturity (number of completed training epochs). In the context of distributed systems, this Maturity indicator acts as a domain-specific Logical Clock (scalar clock). It provides a reliable metric to causally order the progression of the local training states, ensuring that older or stale states do not overwrite more advanced computations.

3.4.4 Observability: Periodic Request-Reply

Intra-node interaction for observability purposes follows a decoupled **Periodic Request-Reply** (or Polling) pattern. In this context, the `MonitorActor` acts as the boundary to the User Interface. To prevent the fast-paced `TrainerActor` from flooding the GUI with continuous updates, the `MonitorActor` periodically emits a query message (*Request*) directly to the `ModelActor`, which in turn responds with a snapshot of its current metrics (*Reply*). This strategic interaction completely decouples the high-throughput computational loop from the slower, blocking UI rendering thread, ensuring visual responsiveness without degrading training performance.

3.4.5 Failure Detection: Heartbeating

To guarantee Fault Tolerance (NFR1), the system must quickly adapt to network partitions or node crashes. This is achieved at the infrastructural layer through a continuous, background **Heartbeating** interaction pattern. The underlying Akka Cluster constantly monitors the health of the connected nodes by exchanging periodic heartbeat messages. If a peer stops responding to these probes within a configured threshold, the cluster's internal failure detection mechanism marks it as unreachable. This state change automatically triggers a domain event that updates the local routing tables, safely and dynamically excluding the crashed node from the Gossip dissemination pool.

3.4.6 Authentication: Request-Reply Proxying

Before any new node is allowed to participate in the Gossip overlay or request dataset partitions, it must be authenticated. The `AuthActor` utilizes a strict Request-Reply interaction to validate credentials and issue JWT tokens, effectively acting as a security gatekeeper at the infrastructure boundary.

3.4.7 Presentation Mechanism: Custom Marshalling

To fulfill the high-throughput requirements of the Gossip protocol without saturating network bandwidth, the system eschews verbose, human-readable presentation formats (like JSON or XML). Instead, it relies on a custom binary serialization adapter based on Scala Type Classes. This mechanism efficiently handles the Marshalling and Unmarshalling of pure linear algebra structures into compact byte arrays just before TCP transmission, achieving high communication efficiency (NFR3).

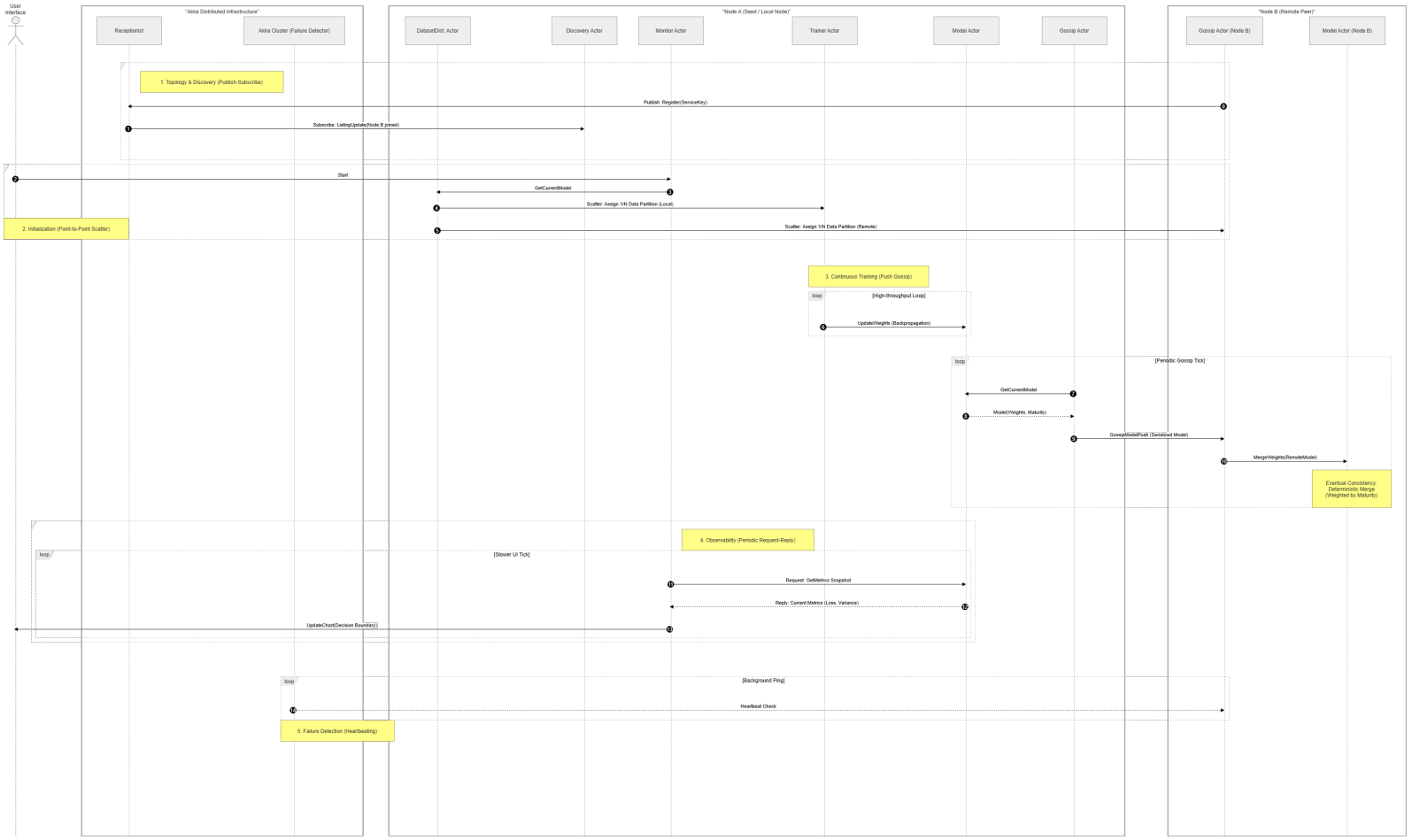

Figure 3: Sequence diagram of temporal interactions and communication patterns (Publish-Subscribe, Scatter, Gossip, Polling, and Heartbeating) within the DNTS decentralized infrastructure.

3.5 Behaviour

The behavioral design of DNTS is strictly reactive and message-driven. Since the system is heavily concurrent, understanding how components react to events and how the application state mutates over time is crucial for guaranteeing correctness and avoiding race conditions.

In accordance with the Actor Model and Functional Programming principles, the system separates stateless computations from stateful behavioral transitions.

3.5.1 Stateless Components: The Pure Functional Core

The components responsible for the heaviest computational tasks—namely, the linear algebra primitives, the Neural Network forward/backward passes, and the model-averaging logic—are entirely **stateless**. When a component like the `TrainerActor` needs to compute a gradient, it invokes pure functions from the Domain Layer. These functions evaluate the input data and return a completely new, immutable instance of the network's matrices. Because these components have no internal state and produce no side-effects, they are inherently thread-safe and behave deterministically regardless of the system's execution timing.

3.5.2 State Ownership and Mutation: The ModelActor

The `ModelActor` is exclusively in charge of updating and holding the primary state of the system: the Neural Network's current weights, biases, and maturity. It acts as the local *Single Source of Truth*.

- **When:** State mutation occurs only when the `ModelActor` processes specific messages from its mailbox: an `UpdateModel` message containing local gradients (from the `TrainerActor`) or a `MergeWeights` message containing a remote model (from the `GossipActor`).
- **How:** Utilizing `Akka Typed`, the `ModelActor` does not mutate internal variables (no `var` assignments). Instead, upon receiving a valid message, it computes the new state and explicitly returns a new `Behavior` encompassing the updated immutable `NNModel`. This guarantees atomic, strictly sequential state updates without blocking operations.

3.5.3 Stateful Behaviors and Finite State Machines (FSM)

While the `ModelActor` acts as an accumulator, other components exhibit complex, lifecycle-dependent behaviors implemented as **Finite State Machines (FSM)**.

The `TrainerActor` is the primary example of this stateful behavior. Its response to incoming messages radically changes based on its current operational state:

- **Idle State:** The actor awaits configuration. It ignores training triggers but listens for a `StartTraining` command.

- **Training State:** The actor actively processes data. Upon receiving a `NextBatch` self-message, it delegates the batch to the pure core, sends the result to the `ModelActor`, and immediately schedules the next batch.
- **Paused State:** Triggered by the user (via GUI) or by a transient network freeze. In this state, the actor suspends the `NextBatch` loop. Any residual training events in the mailbox are either dropped or stashed until a `ResumeTraining` command is received, dynamically swapping the behavior back to the Training State.

Other actors exhibit timer-driven stateful behaviors. The `GossipActor`, for instance, maintains a volatile routing table of reachable peers (updated via Discovery events). Its behavior is governed by periodic timer events (`GossipTick`): at every tick, it reads its routing table state, selects a peer, requests the latest snapshot from the `ModelActor`, and dispatches the payload over the network.

3.6 Data and Consistency Issues

The system intentionally avoids centralized databases, prioritizing High Availability and Partition Tolerance (AP in the CAP Theorem) over strong distributed consistency.

3.6.1 Data Storage

While the system does not rely on global databases, specific data must be persisted locally on each node to guarantee Fault Tolerance (Crash Recovery):

- **Local Dataset Partition (`local_dataset.bin`):** A static, immutable slice of the global training data assigned to the node during the bootstrap phase.
- **Model Snapshot (`local_model_snapshot.bin`):** The dynamic, mutable state of the Neural Network (Weights, Biases, and *Maturity* indicator).

Storing this data on the local filesystem allows a node that crashes to recover its state independently. Upon reboot, the node reads its local files, bypasses the initial data-scattering phase, and immediately rejoins the Gossip pool, resuming training from its last saved *Maturity* state.

Persistent data is **not** stored using relational, document, or graph databases. Instead, data is stored as **Flat Binary Files** via a Custom Binary Serialization protocol.

The core domain data consists exclusively of dense linear algebra structures (high-dimensional matrices and vectors). Traditional databases would introduce unnecessary mapping overhead and latency. Direct binary serialization (implemented via Scala *Type Classes*) provides the most compact memory footprint and the fastest serialization/deserialization times, adhering to the performance requirements.

3.6.2 Data Access, Queries, and Concurrency

Because the system uses local flat files instead of a database engine, there is no traditional Query Language nor complex query execution plans. Access patterns are defined by the system's lifecycle:

- **Reads:** Performed exclusively at node startup (Bootstrap/Recovery) by the infrastructural components to load the dataset and initialize the `ModelActor`.
- **Writes (Checkpointing):** To avoid disk I/O bottlenecks during the high-throughput training loop, writes are asynchronous and periodic. Every K epochs, the state is serialized and flushed to disk.
- **Concurrency Management:** There are zero concurrent write or read issues.
    1. *Inter-node:* Nodes are completely physically isolated; they only write to their own local filesystems.
    2. *Intra-node:* The Actor Model naturally acts as a concurrency guard. Only a designated component (e.g., a `PersistenceManager` or the `ModelActor` itself) handles the file I/O sequentially, completely eliminating race conditions without relying on file-system locks or database-level transactions.

3.7 Fault-Tolerance

The dependability and resilience of DNTS are engineered around the assumption that physical nodes operate in an asynchronous, unreliable network environment where unannounced crashes, message drops, and transient network partitions are normal runtime events.

3.7.1 Heart-beating, Timeouts, and Retry Mechanisms

To dynamically adapt to changes in the network topology and prevent bandwidth waste, the system implements continuous background health monitoring, operating at two different levels of abstraction:

- **Infrastructural Heart-beating:** The underlying `Akka Cluster` infrastructure constantly monitors the liveness of all connected physical endpoints by exchanging periodic heartbeat messages over the established mutual TLS transport layer.
- **Failure Detection and Adaptive Timeouts:** If a peer fails to reply to these heartbeats within a strict, pre-configured timeout threshold, the internal failure detector flags the node as *Suspected* and subsequently as *Unreachable*. This mechanism is crucial because in an asynchronous distributed system, it is theoretically impossible to distinguish between a node that has crashed and one that is experiencing severe network latency.
- **Routing Table Pruning:** Once Akka Cluster emits an unreachability domain event, the `DiscoveryActor`—which is subscribed to the cluster membership topic via the `Akka Receptionist`—asynchronously intercepts the topology change. It immediately updates the local routing table, safely purging the failed node reference. This prevents the `GossipActor` from attempting to send matrix payloads to dead endpoints, eliminating unnecessary network retries and connection timeouts at the application layer.

- **Deterministic Membership Governance:** The design delegates the stability of the cluster to a formal governance policy that reacts differently based on the system's operational phase (e.g., Bootstrap vs. Running). A core design requirement is that unreachability should not lead to immediate exclusion; instead, a grace period is enforced via timers. Crucially, the policy also defines a self-healing path: if a node restores its reachability before the grace period expires, the system automatically cancels the scheduled eviction and ensures the node's immediate re-acceptance into the active membership. If a node fails to recover within this duration, a deterministic downing rule is applied. To maintain global consistency, the removal of a node from the official membership is coordinated—typically requiring the agreement of a leader—ensuring that the system converges toward a stable and unified topology even in the presence of permanent faults.

3.7.2 Error Handling and Node Failure Lifecycle

The lifecycle of a component failure is governed by Akka's supervision strategies and a decoupled crash-recovery design based on asynchronous local checkpointing:

- **Inter-node Resilience (Crash Isolation):** When a physical node crashes abruptly, the execution of the surviving peers remains completely unaffected. Because the model-averaging algorithm is fully decentralized and asynchronous, surviving nodes continue to cycle between local gradient descent via the `TrainerActor` and epidemic parameter exchanges with the remaining active peers.
- **Crash-Recovery Lifecycle (Bypassing Cold-Start):** To prevent disk I/O bottlenecks during the high-throughput training loops, the `ModelActor` asynchronously flushes its mutable state (current weights, biases, and metadata) to a local flat file (`local_model_snapshot.bin`) every K training epochs. When a crashed node reboots:
  1. The initialization logic inside the main entry point detects the pre-existing binary file on the filesystem.
  2. It bypasses the cold-start phase, avoiding re-requesting the dataset from the seed node, and directly deserializes both the static dataset and the training state.
  3. The recovered state is safely injected during the sequential spawn of the `TrainerActor` and the `ModelActor`, allowing the node to immediately resume local training from the exact epoch it completed before the failure.
- **Passive Gossiping and Logical Ordering:** Upon re-entering the P2P overlay, the recovered node might hold a significantly outdated model compared to the rest of the network. To guarantee that stale weights do not pollute the global consensus, the system leverages the model's *Maturity* attribute (the number of completed epochs) as a domain-specific Logical Clock. When a remote node receives a model

from the recovered peer, the deterministic functional merge operator heavily discounts the received parameters using a weighted average based on the maturity delta:

$$W_{new} = \frac{M_{local} \cdot W_{local} + M_{remote} \cdot W_{remote}}{M_{local} + M_{remote}} \tag{1}$$

This causal ordering constraint acts as a synchronization barrier in space and time, smoothly reintegrating the recovering node without degrading the accuracy of the converged cluster.

3.8 Availability

As a decentralized Peer-to-Peer system, DNTS inherently prioritizes Availability and Partition Tolerance. The architecture ensures that the deep learning simulation remains active and responsive even under severe network degradation or component isolation.

3.8.1 Local State Snapshotting and Persistence

DNTS employs a Local State Snapshotting mechanism to ensure the continuous availability and recovery of the computation layer. State preservation occurs by persisting the necessary data directly to the local physical storage of each node. Specifically, the system manages the serialization of the latest replica of the Neural Network weights (handled by the ModelActor) and the local Dataset partition (handled by the TrainerActor) onto local files.

3.8.2 Load Balancing

DNTS avoids centralized load balancers, as they would reintroduce a Single Point of Failure and contradict the P2P philosophy. Instead, the system relies on **Intrinsic Distributed Load Sharing**:

- **Computational Load:** Every node receives exactly $1/N$ of the total training data. Consequently, the heavy computational load of the mathematical domain is perfectly balanced horizontally across all active physical machines.
- **Network Load:** Network traffic is balanced through the randomized nature of the **Gossip Protocol**. The `GossipActor` does not broadcast its matrix payload to all peers simultaneously, instead its epidemic dissemination naturally and uniformly distributes the communication overhead across the entire cluster.

3.9 Security

Operating over a decentralized Peer-to-Peer architecture introduces inherent security challenges, notably the risk of unauthorized access and model poisoning. To mitigate these risks, the system implements a foundational security layer.

3.9.1 Authentication

To guarantee the integrity of the distributed learning overlay, the system requires a basic user authentication before granting a new node access to the cluster, consisting of a single-step username and password verification.

Acting as a gatekeeper at the infrastructure boundary, this process is exclusively orchestrated by a specialized `AuthActor`. Architecturally, the `AuthActor` is instantiated only on the Seed node. Incoming Client nodes simply obtain a remote reference to the Seed's `AuthActor` to submit their credentials.

Once successfully authenticated, the Client is allowed to fully join the symmetric peer-to-peer topology. This baseline security measure is fundamental in a purely decentralized Gossip network, where any node can theoretically push its model weights to any peer.

3.9.2 Authorization and Access Control

Within the domain, different architectural roles are logically separated to maintain clearly defined operational boundaries.

- *Seed Nodes (Administrators):* Act as administrators with elevated privileges. They are authorized to define the global Neural Network topology, trigger the synthetic dataset generation, distribute the data partitions to other peers via the `DatasetDistributionActor`, and issue global simulation directives (such as Start, Pause, and Stop) directly via the GUI.
- *Client Nodes (Workers):* Operate as standard workers. Their privileges are restricted to requesting their assigned dataset partition, performing local Gradient Descent, and participating in the bidirectional P2P Gossip protocol. To maintain the structural consistency of the simulation, client nodes are prohibited from altering the global hyperparameters once the network has been formed.

3.9.3 Cryptographic Schemas

To secure data in transit and verify identities without relying on a centralized session database, DNTS leverages standard cryptographic mechanisms at both the application and transport layers:

- **Token Verification (JWT):** The system embraces a stateless authentication mechanism. Upon successful credential verification, the `AuthActor` utilizes a custom `JwtCodec` to generate and issue a JSON Web Token (JWT). Client nodes must embed this cryptographic token in subsequent high-privilege distributed requests (e.g., when formally requesting the initial dataset partition from the Seed). Receiving nodes independently verify the JWT signature to ensure the token is genuine, untampered, and has not expired.
- **Transport Encryption (mTLS):** At the infrastructural level, all inter-node TCP communication is secured using Akka Artery Remoting configured with mutual

Transport Layer Security (mTLS). This ensures that all transmitted data are fully encrypted, preventing eavesdropping and Man-In-The-Middle attacks.

- **Credential Hashing:** Any shared secrets or passwords used during the bootstrap registration phase are protected using standard cryptographic one-way hashing functions (via the pure `Crypto` domain module). This ensures that plain-text credentials are never exposed in memory or within the `application.conf` files.

4 Implementation

This section details the technology-dependent choices made to translate the theoretical Peer-to-Peer, Actor-based design into a concrete, executable software artifact. The implementation relies heavily on the Scala 3 ecosystem and the Akka framework to fulfill the stringent requirements of concurrency, high-throughput networking, and pure functional programming.

4.0.1 Network Protocols

The system relies on TCP (Transmission Control Protocol) as its foundational transport layer, which is abstracted and encapsulated within the Akka Artery Remoting framework. Unlike real-time applications where UDP is often acceptable despite packet loss, distributed machine learning requires absolute data integrity. TCP guarantees the reliable and ordered message delivery that is essential for the P2P Gossip protocol to function correctly. Furthermore, Akka Artery provides a high-performance, asynchronous message-passing transport specifically optimized for Actor systems.

4.0.2 In-Transit Data Representation

Standard human-readable formats are strictly avoided across the entire system. A neural network model consists of high-dimensional dense matrices; serializing millions of floating-point numbers into JSON strings would result in massive payloads, saturating the network bandwidth and introducing parsing latency. Consequently, all in-transit data—from the heavy model weights exchanged during gossip to the internal control commands—is represented exclusively through Custom Binary Serialization. This mechanism is natively implemented using Scala Type Classes, which serialize the domain entities directly into highly optimized, minimal-footprint byte arrays before transmission.

4.0.3 Database Querying and Storage

DNTS avoids deploying any centralized or local database engine (whether SQL or NoSQL). Introducing a database would create an unnecessary infrastructure dependency and a potential computational bottleneck. Instead, persistence is managed entirely through a custom `FileNodeRepository` module that interacts directly with the local filesystem. Heavy domain entities, such as the static dataset partitions and the sequential model snapshots, are written and read as flat binary files. Crucially, the storage of registered

users and their authentication credentials follows this same file-based repository pattern. Data access does not involve SQL queries or complex indexing; rather, it consists of direct, sequential file deserialization into memory during node bootstrapping or user login, guaranteeing maximum I/O performance.

4.0.4 Component Authentication and Authorization

While the architectural necessity of access control was discussed in Section 3.9, its implementation is kept stateless and integrated into the Scala type system. Component authentication does not query a session database; instead, the `AuthActor` utilizes a custom `JwtCodec` to generate, cryptographically sign, and verify JSON Web Tokens (JWT). For authorization, the system avoids heavy external RBAC/ABAC middleware. It leverages Scala 3's Algebraic Data Types (ADTs) by defining the `NodeRole` as a strict `enum` (e.g., *Seed*, *Client*). Akka's pattern matching is then used directly within the Actor's `Behavior` to inspect the embedded role in the incoming messages, natively accepting or dropping administrative commands directly at the mailbox level based on the sender's type.

4.1 Technological Details

The DNTS technology stack was selected to enforce the Actor Model and pure functional programming within a unified ecosystem.

- **Scala 3:** Used to build the Pure Functional Core. Mathematical entities (vectors, matrices) are modeled as immutable structures to eliminate side-effects during Gradient Descent. Algebraic Data Types (ADTs) and pattern matching safely define domain models and strictly-typed message protocols.
- **Akka Typed & Akka Cluster:** *Akka Typed* enforces the Actor Model, avoiding shared-memory pitfalls (deadlocks, race conditions) by encapsulating mutable state within asynchronous, strictly-typed actors. *Akka Cluster* provides the leaderless P2P infrastructure, managing node discovery (via Receptionist) and failure detection.
- **Akka Artery Remoting:** Handles high-throughput, TCP-based inter-node communication. It provides a low-latency transport layer that is completely transparent to the application-level actors and natively secured via mutual TLS.
- **Java Swing:** Forms the boundary observation layer (GUI). To prevent UI rendering from bottlenecking the concurrent training, Swing is decoupled from the Actor System. The `MonitorActor` periodically pushes metric updates to the Swing *Event Dispatch Thread (EDT)*, enabling real-time plotting without degrading distributed computation performance.

5 Validation

The validation of the DNTS system was conducted with the aim of ensuring the mathematical correctness of the model, the robustness of the distributed protocol, and the resilience of the system. The separate architecture between pure logic and concurrent state management greatly facilitated test development, allowing components to be isolated and tested in a targeted manner.

5.1 Automatic Testing

The automated testing suite is completely implemented using **ScalaTest** and the **Akka Typed TestKit**. The strategy isolates the pure mathematical domain from the asynchronous distributed behaviors to prevent flaky tests and ensure maximum coverage.

Unit Testing of Individual Components (Pure Functional Core): The components inside the domain layer (Linear Algebra primitives, Data Generation, Network Initialization, and Backpropagation algorithms) are entirely stateless and pure. Consequently, they are exhaustively unit-tested using standard ScalaTest suites (e.g., `LinearAlgebraTest`, `BackpropagationTest`).

- *Rationale:* Ensure that the underlying mathematics of the Neural Network computes gradients and applies model averaging correctly before deploying it to a distributed environment.
- *Requirements Tested:* FR4 (Pure Functional Backpropagation), FR2 (Model Averaging logic).

Interaction and Communication Testing (Actor System): Testing distributed communication requires verifying asynchronous message passing and state-machine transitions. This was achieved using the **Akka ActorTestKit**.

- *How it works:* Instead of spinning up a real TCP cluster for every test, the TestKit provides `TestProbe` objects. These probes act as mock remote peers or user interfaces.
- *Requirements Tested:* FR1 (P2P Membership Management), FR2 (Asynchronous Gossip Learning).

Automation and Execution: All automated tests are fully integrated into the SBT (Scala Build Tool) lifecycle. They can be executed uniformly via the command line utilizing the `sbt test` command. In a production-like environment, this suite is designed to be seamlessly integrated into a Continuous Integration (CI) pipeline (e.g., GitHub Actions), automatically running upon every commit to prevent regressions.

5.2 Acceptance Testing

While the mathematical core and actor behaviors are rigorously covered by automated scripts, certain aspects of the system—specifically visual rendering and extreme infrastructural chaos—require manual acceptance testing.

1. **Visual Convergence and GUI Responsiveness:** Launching the full cluster on a local machine and visually monitoring the Java Swing Dashboard. The acceptance criteria required observing the Decision Boundary plot dynamically adapting to separate the XOR/Spiral dataset clusters over time, alongside a smooth, non-blocking rendering of the Loss and Consensus metrics.

2. **Fault Injection Testing:** Deploying the nodes across separate physical machines connected via a LAN. During an active training session, a node process was forcefully killed via the OS Task Manager (simulating an abrupt hardware crash), and later the Wi-Fi connection of another node was severed (simulating a network partition). The criteria required observing the surviving nodes dynamically updating their routing tables and continuing the training process without crashing.

Automating the GUI tests (e.g., visual regression testing on dynamic plotting charts) is notoriously brittle, computationally expensive, and falls outside the scope of testing distributed consensus. Furthermore, simulating true physical network partitions and hard OS-level process kills across multiple actual machines requires a complex test environment. Given the project's constraints, manual validation provided immediate, reliable proof of the system's compliance with the Availability and Fault-Tolerance requirements under real-world hostile conditions.

5.3 Degree of Coverage

All packages	42.67%
actors.authentication	0.00%
actors.cluster	64.39%
actors.cluster.adapter	18.18%
actors.cluster.effect	72.55%
actors.cluster.membership	75.00%
actors.cluster.timer	0.00%
actors.discovery	77.78%
actors.gossip	87.68%
actors.gossip.configuration	83.13%
actors.gossip.consensus	66.90%
actors.gossip.dataset_distribution	83.72%
actors.model	61.86%
actors.monitor	36.02%
actors.root	0.23%
actors.trainer	67.38%
app	0.00%
cli	67.44%
config	59.74%

actors.root	0.23%
actors.trainer	67.38%
app	0.00%
cli	67.44%
config	59.74%
domain.authentication	7.32%
domain.common	0.00%
domain.data	86.17%
domain.data.dataset	66.67%
domain.data.pattern	90.70%
domain.data.sampling	88.00%
domain.data.util	94.87%
domain.model	100.00%
domain.network	67.54%
domain.serialization	41.02%
domain.training	85.84%
domain.training.consensus	74.55%
view	10.47%
view.components	0.00%

Figure 4: Package coverage report generated via sbt-scoverage.

As illustrated in the report generated by sbt-scoverage, the test suite ensures reasonable coverage across all key system packages. Central modules, such as the mathematical core (domain), synchronization and membership logic (cluster, gossip), and validation mechanisms (config), feature high coverage. This ensures the complete reliability of the business logic, distributed computing, and protocol resilience.

In areas where the percentage is naturally lower, the decline is due to a specific and natural engineering choice. Modules such as the graphical boundary layer (view) or the very low-level network error cases associated with the Akka framework are intrinsically difficult to test deterministically or offer little added value compared to the robustness demonstrated in the rest of the architecture.

6 Deployment

The project uses sbt (Scala Build Tool). To generate the multi-platform executable (Fat JAR) containing the application and all its dependencies, run the following from the project root:

```
sbt assembly
```

Note: for convenience in the subsequent commands, we will rename the generated file to ‘dnts.jar‘

7 User Guide

The application is cross-platform (Windows, Linux, macOS) and only requires the installation of Java (JRE/JDK 11 or higher). The entire system can be started via the generated JAR, without external dependencies.

Simulation Configuration The neural network topology, the dataset to be generated, and the training hyperparameters must be defined in a `.conf` configuration file. Ensure that the file is present in the same folder where you are launching the Seed node's JAR.

Starting the Seed Node The first node to launch is the Seed. It is responsible for reading the configuration file, starting the cluster, and acting as an entry point for other peers.

Open a terminal and run:

```
java -jar dnts.jar --role seed --cluster MyCluster \
--config simulation.conf --port 2500
```

Starting the Client Nodes Once the Seed node is started, you can launch as many client nodes as you wish. Each node will join the cluster, receive its portion of data, and participate in the Gossip Learning cycle. Once a simulation instance is started, it will no longer be possible to add nodes to the created cluster. Clients must provide authentication options and link to the Seed node's address.

Open a terminal and run:

```
# Starting and registering the first client
java -jar dnts.jar --role client --cluster MyCluster \
--seedAddress 127.0.0.1:2500 --port 2501 \
--action register --username alice \
--password secret --fullName "Alice Smith"

# Starting and logging in a second existing client
java -jar dnts.jar --role client --cluster MyCluster \
--seedAddress 127.0.0.1:2500 --port 2502 \
--action login --username bob --password secret
```

(You can add additional clients simply by specifying a different `--port` for each.)

7.1 CLI Arguments

The executable accepts the following parameters.

Common Parameters These parameters are valid or required regardless of the node's role.

Parameter	Description	Requirement
`--role <role>`	Defines the node's role in the cluster: `seed` or `client`.	**Mandatory**
`--cluster <name>`	The identifier name of the cluster.	**Mandatory**
`--port <port>`	Listening binding port for the node (e.g., `2500`).	**Mandatory**
`--help`	Shows the CLI help menu.	Optional

Table 1: Common CLI parameters

Seed Node (`--role seed`) The Seed node is the entry point of the cluster and manages the initial setup of the simulation.

Parameter	Description	Requirement
`--config <path>`	Path to the configuration file (e.g., `simulation.conf`) containing topology, dataset, and hyperparameters.	**Mandatory**

Table 2: Seed node parameters

Client Node (`--role client`) Client nodes join the existing cluster, receive their portion of data, and participate in distributed training (Gossip Learning). They require additional parameters for connection and authentication.

Parameter	Description	Requirement
`--seedAddress <address>`	IP address and port of the Seed node to connect to (e.g., `127.0.0.1:2500`).	**Mandatory**
`--port <port>`	Local binding port for the client node.	**Mandatory**
`--action <action>`	Action to perform towards the Seed: `register` (new user) or `login` (existing user).	**Mandatory**
`--username <name>`	Client's username for authentication.	**Mandatory**
`--password <pwd>`	Client's password.	**Mandatory**
`--fullName <name>`	User's full name (mainly useful during the `register` phase).	Optional

Table 3: Client node parameters

8 Self-evaluation

As required by the project specifications, this section provides an individual self-evaluation for each team member, detailing their specific role, contributions, and an objective assessment of the strengths and weaknesses of the implemented components.

8.1 Giorgio Fantilli

Role within the project: My primary contributions centered around the Pure Functional Core, data persistence, and the presentation layer. I developed the mathematical engine (Backpropagation, SGD) and the `TrainerActor`. I also engineered the serialization mechanisms and data export features, including the local `Snapshotting` mechanism for crash recovery. Furthermore, I managed the application bootstrap, network configuration, and implemented the Graphical User Interface alongside the `MonitorActor`.

Strengths of the product:

- *Domain Purity and Testability:* By strictly adhering to functional programming, the core mathematical components are entirely free of side-effects, making them highly predictable and exhaustively unit-testable.
- *Decoupled Observability:* The implementation of the `MonitorActor` effectively isolates the GUI rendering from the distributed computation, ensuring that visual updates do not block the intense training loops.

Weaknesses of the product:

- *Local I/O Overhead:* While the snapshotting mechanism guarantees resilience, persisting the state via local file dumping can introduce a minor I/O overhead if configured to trigger too frequently during particularly heavy training epochs.

8.2 Lorenzo Colletta

Role within the project: I focused on the distributed infrastructure and the Actor System topology. My core responsibilities included the design and implementation of the `ClusterManager` and the `DiscoveryActor`. I managed the Akka Cluster configuration, the node lifecycle, and the dynamic peer discovery mechanism via the Receptionist pattern. Additionally, I contributed to the data engineering phase, specifically handling dataset generation, splitting, and batching logic.

Strengths of the product:

- *Resilient Topology:* The integration of Akka's discovery and failure detection mechanisms provides a highly resilient infrastructure, allowing the system to dynamically adapt to network changes and gracefully handle peer disconnections.
- *Concurrency Safety:* The strict use of Akka Typed for the supervision hierarchy completely eliminates shared-memory concurrency issues across the distributed simulation.

Weaknesses of the product:

- *Bootstrapping Dependency:* Although the continuous training phase is fully leaderless and decentralized, the data splitting and cluster initialization inherently depend on the Seed node, creating a transient structural bottleneck during startup.

8.3 Domenico Francesco Giacobbi

Role within the project: My work focused on the distributed communication protocols, state management, and system security. I implemented the `ModelActor`, the `GossipActor`, and the `ConsensusActor`, managing the core P2P model-averaging logic and network metrics. I also developed the actors responsible for configuration and dataset distribution across the network. Furthermore, I engineered the security layer, managing user passwords, cryptography, and authentication to secure the distributed overlay.

Strengths of the product:

- *Efficient Synchronization:* The Gossip and Consensus implementations successfully distribute the learning process with high efficiency, enabling robust model synchronization without a centralized parameter server.
- *Protection against Data Poisoning:* The cryptographic security and authentication layers effectively prevent unauthorized nodes from joining the cluster, safeguarding the global consensus from malicious gradient injections.

Weaknesses of the product:

- *Static Role Assignment:* The current access control and security policies are evaluated primarily at initialization. Introducing dynamic privilege modifications during an active simulation session would require further infrastructural enhancements.

9 Future Works

To evolve the DNTS framework from an academic simulation into a more versatile distributed middleware, several functional improvements could be explored in future works. First, transitioning from a static dataset bootstrap to dynamic data ingestion—such as streaming new local samples during the training phase—would allow the analysis of how decentralized Gossip Learning reacts to concept drift in real-time. Additionally, the underlying network topology could be optimized by replacing the purely random peer selection with a latency-aware policy, limiting the maximum number of concurrent connections to evaluate the trade-offs between partial-mesh message overhead and global consensus convergence speed. On the observability front, the telemetry system could be extended to persistently export detailed logs of individual node losses and consensus metrics, facilitating advanced offline comparative plotting across different network sizes. Finally, the currently static Role-Based Access Control could be upgraded to a dynamic authorization model, enabling new client nodes to securely authenticate and join the P2P overlay even after the simulation has already commenced, thereby increasing the overall elasticity of the cluster.