Peer-to-Peer Federated Learning Framework

Zhaoyang Li, Yunzhen Zhang, Sheng'en Li

1 Introduction

The machine learning landscape is shifting from centralized data lakes toward distributed edge intelligence. For the past decade, artificial intelligence systems have largely relied on a centralized setup: vast amounts of data from smartphones, IoT devices, medical equipment, and edge processors are collected and sent to large data centers, where high performance clusters train global models. This approach has led to major progress, but it is now facing technical and legal barriers. Privacy laws such as the EU General Data Protection Regulation (GDPR) restrict data storage and transfer. At the same time, the growing data volume at the edge exceeds available network bandwidth, making it increasingly expensive and slow to send raw data to a central server.

To address these challenges, Federated Learning (FL) has emerged as an effective alternative. First formalized in [2], FL allows multiple participants to collaboratively train a machine learning model without sharing their raw data [2]. In the standard centralized FL setup, a trusted server coordinates training rounds: clients download a global model, update it locally using private data, and upload only model parameters or gradients. The server then aggregates these updates, typically using algorithms such as Federated Averaging (FedAvg), to produce a new global model [3]. This architecture decouples learning from centralized data storage and has been widely adopted in both cross device and cross silo scenarios [1].

Despite its success, its standard client server architecture inherits fundamental limitations of centralization. The central aggregator becomes both a bottleneck and a point of vulnerability. Most critically, it represents a Single Point of Failure (SPOF): if the aggregator experiences downtime, hardware failure, or a denial of service attack, training halts immediately [4]. The model also relies on strong trust assumptions, since clients must depend on the central authority not only for coordination but for the integrity of aggregation. A compromised or malicious server can manipulate the global model or leak private information [6].

In contrast, gossip based decentralized federated learning (p2pfl) provides a more flexible way to collaborate. Model updates propagate gradually among nodes without a fixed aggregation point, making p2pfl suitable for dynamic network conditions and unstable node participation. Building on this insight, we extend p2pfl with improved communication and resilience mechanisms for practical deployment. First, we introduce a smarter gossip protocol that attaches version information to model updates, applying staleness weighted aggregation, maintaining neighbor model caches, and steering propagation based on interaction frequency. These designs allow decentralized training to better handle asynchronous participants and heterogeneous update quality. Second, we incorporate a recovery mechanism combining local and remote checkpoints, enabling failed nodes to rejoin training and restore model state. Together, these enhancements improve p2pfl reliability and training continuity.

2 Implementation

2.1 Communication Protocol

Our goal is to make the communication layer "smarter" while keeping the learning API unchanged. The original `p2pfl` framework defines a generic gossip-based communication protocol: in each round, nodes exchange model updates with their neighbors, and a FedAvg-style aggregator combines received models. However, the baseline design is largely oblivious to when a model was computed and how frequently a neighbor participates in the protocol. In particular, (i) all incoming models are treated equally in FedAvg,

regardless of their staleness, and (ii) each neighbor satisfying the round condition is selected almost uniformly as a gossip target.

We extend this communication layer along three dimensions: (1) attach a version to every model update and apply staleness-aware weighting in FedAvg; (2) maintain a per-neighbor cache of the most recently received model for potential similarity-based policies; and (3) track per-neighbor interaction degree and bias gossip towards high-degree neighbors, inducing a preferential-attachment-like communication pattern.

System model. We adopt the same system model as the original `p2pfl` framework: each node runs a local learner and a communication protocol, and the global training process is organized into logical rounds $r = 1, 2, \ldots$. In each round, a node (i) trains its local model on its private data, (ii) participates in model aggregation via FedAvg, and (iii) gossips the resulting aggregated model to its neighbors. We explicitly do not modify the learner interface nor the aggregator: our changes are confined to the communication commands, the gossip scheduling, and the node state.

Versioned model updates. We attach a version number to every model update in the communication layer. Concretely, each learner exposes its state via a `P2PFLModel` object with an `additional_info` dictionary. Before encoding the model parameters for gossip, the `GossipModelStage` sets

$$\texttt{model.additional\_info["version"]} = \texttt{state.round},$$

where `state.round` is the current logical round of the node. The model is then serialized and sent using the existing weights command. On the receiver side, our extended `PartialModelCommand` and `FullModelCommand` decode both the parameters and the `additional_info` field. If no version is present, we default it to the receiver's current round, so the protocol remains backward compatible.

Staleness-aware aggregation. Given a decoded model update with version v that arrives at a node in round r, we define its staleness as

$$s = \max(0, r - v).$$

Let w be the nominal weight of this update in FedAvg (typically the number of local training samples used for this update). We apply a simple decaying function

$$\alpha(s) = \frac{1}{1 + s}$$

and compute an effective weight

$$w_{\text{eff}} = w \cdot \alpha(s).$$

In the implementation, w_{eff} is passed to the FedAvg aggregator in place of w as the "number of samples" for this update. Thus, model updates that are several rounds old still participate in aggregation but with a reduced influence. This design does not change the FedAvg interface and can be swapped out for more sophisticated decay functions without touching the learning layer.

Neighbor model cache. To enable future similarity-aware communication policies, we extend the node state with a per-neighbor model cache. Each node maintains

$$\texttt{neighbor\_models} : \texttt{addr} \mapsto \texttt{P2PFLModel},$$

where `addr` is a neighbor identifier. Whenever a model update is successfully decoded and wrapped as a `P2PFLModel`, the receiver stores it in `state.neighbor_models[source]`. In this work, we primarily use this cache for monitoring and for future extensions, but the abstraction is already integrated into the communication layer.

Degree tracking and preferential gossip. On top of the logical overlay defined by `p2pfl`, we introduce a lightweight notion of neighbor degree based on actual communication interactions. The `Neighbors` component maintains a statistics dictionary

$$\texttt{nei\_stats[addr]["degree"]},$$

which counts how many times the node has successfully communicated with neighbor `addr`. The `Gossiper` increments this degree via a `note_interaction(addr)` call whenever it sends a message or model to that neighbor.

We expose this information through the communication protocol as `get_neighbor_degree(addr)`, so that stages do not need to depend on the concrete `Neighbors` implementation. The `GossipModelStage` then implements a degree-based gossip scheduling policy. Let $\mathcal{N}$ be the set of direct neighbors of a node. In round r, we define the base candidate set as

$$\mathcal{C}_r = \{\, n \in \mathcal{N} \mid \texttt{nei\_status}[n] < r \,\},$$

i.e., neighbors that have not yet received the aggregated model for round r. The baseline `p2pfl` implementation essentially samples gossip targets uniformly from $\mathcal{C}_r$. Instead, we sort $\mathcal{C}_r$ in descending order of `get_neighbor_degree(n)` and use this ordering to drive gossip: neighbors with higher observed degree are contacted first. This implements a simple preferential-attachment flavor on top of the existing overlay: nodes that have historically been more central in communication are more likely to receive and propagate new models quickly.

For both configurations, we measure convergence in terms of test accuracy versus the number of global rounds, as well as communication cost such as the total number of messages exchanged and the distribution of interactions across nodes. This allows us to isolate the impact of the communication layer while keeping the learning layer identical.

2.2 Learning Process

We build the learning layer on top of the `p2pfl` library [5], which offers a modular and framework-agnostic architecture for federated learning. Rather than modifying the core learning logic, we leverage the existing learning abstractions and keep the interface unchanged. This approach lets us focus on communication-layer enhancements while ensuring that our improvements can be evaluated independently of the learning algorithms. It also makes our enhancements portable to other federated learning systems that use similar abstractions.

Learner abstraction and framework integration. The `p2pfl` framework uses a `Learner` abstract base class to standardize the interface for local training across different machine learning frameworks. Learners need to implement three core methods: `fit()` performs local training on the node's private dataset and returns an updated `P2PFLModel`, `evaluate()` evaluates the current model on a test set and returns metrics like accuracy and loss, and `get_framework()` returns the framework name. The framework supports multiple ML backends through concrete implementations: `LightningLearner` for PyTorch with PyTorch Lightning, `KerasLearner` for TensorFlow/Keras, and `FlaxLearner` for JAX/Flax. With this abstraction, our communication layer can treat all models uniformly whether they're implemented in PyTorch, TensorFlow, or JAX, enabling framework-agnostic gossip protocols that work across heterogeneous deployments.

Unified model representation. The `P2PFLModel` class unifies model representation across different frameworks. It encapsulates the framework-specific model object (like a `LightningModule` for PyTorch), extracts model parameters as NumPy arrays, and stores metadata including the number of training samples (`num_samples`) and contributing nodes (`contributors`). Each `P2PFLModel` also has an `additional_info` dictionary for extensible metadata, which we use to attach version numbers and other communication-layer state. The interface provides methods for parameter serialization: `get_parameters()` and `set_parameters()` for accessing weights, and `encode_parameters()` and `decode_parameters()` for network transmission (with optional compression). This abstraction lets the communication layer exchange model updates without knowing the specific ML framework. We use it to attach version information: when serializing a model for gossip, we set `model.additional_info["version"]` to the current round number, and receivers extract this version to compute staleness during aggregation.

Local training process. During each training round, nodes in the training set train locally on their private datasets. The process starts when `TrainStage` calls `learner.fit()`, triggering the framework-

specific training loop. For PyTorch Lightning models, this means creating a `Trainer` with the specified epochs and running the training loop. The learner iterates through the data for the configured epochs, computing gradients and updating parameters using the framework's optimization algorithms (SGD, Adam, etc.). Callbacks registered with the aggregator may also run during training, which can modify the process for advanced algorithms like SCAFFOLD or FedProx. Once training finishes, the learner sets the model's contribution info: the node's address goes in the `contributors` list, and the number of local training samples goes in `num_samples`. The updated model is returned as a `P2PFLModel`, ready for aggregation and transmission.

Federated Averaging aggregation. We use a `FedAvg` aggregator [2] to combine model updates from multiple nodes. The process starts when nodes in the training set finish local training and produce a `P2PFLModel` with updated parameters and the number of local training samples. Nodes exchange models through the communication layer using gossip protocols, and the aggregator collects models until it has updates from all nodes in the training set. The aggregated parameters are computed as a weighted average:

$$\theta_{\text{agg}} = \frac{\sum_{i=1}^{n} w_i \cdot \theta_i}{\sum_{i=1}^{n} w_i},$$

where θ_i are parameters from node i, w_i is the effective weight (number of samples, potentially adjusted by staleness decay), and n is the number of contributing nodes. The aggregated model is distributed back to all nodes, which update their local models before the next round. In our enhanced communication layer, we compute the effective weight as $w_i = n_i \cdot \alpha(s_i)$, where n_i is the nominal number of samples and $\alpha(s_i)$ is the staleness decay function for model version v_i received in round r (i.e., $s_i = \max(0, r - v_i)$). The aggregator interface stays unchanged: it takes a list of `P2PFLModel` objects with associated weights and returns a single aggregated `P2PFLModel`. This lets us inject staleness-aware weighting in the communication layer by adjusting the weight passed to the aggregator, without changing the aggregator implementation or learning layer code.

Training workflow orchestration. A `LearningWorkflow` orchestrates the learning process, executing a sequence of stages in each training round. The workflow starts with **StartLearningStage**, which initializes the experiment, sets up the model, and gossips the initial model parameters to establish a common starting point. Then **VoteTrainSetStage** lets nodes vote to select a subset (the `train_set`) that will participate in local training for the current round. This partial participation mechanism means not all nodes need to train every round, which improves efficiency and scalability. Nodes in the training set enter **TrainStage**, where they evaluate the current model on their local test set, perform local training for a fixed number of epochs, add their updated model to the aggregator, and participate in gossip-based aggregation until all training-set models are collected. After aggregation completes, these nodes wait for the final aggregated model and update their local models. Nodes not in the training set enter **WaitAggregatedModelsStage**, where they wait to receive the aggregated model from neighbors without training locally. After aggregation, all nodes proceed to **GossipModelStage** to gossip the aggregated model to neighbors for network-wide synchronization. Finally, **RoundFinishedStage** increments the round counter and transitions to the next round or terminates if all rounds are complete. The workflow is identical in both baseline and enhanced configurations; our modifications are limited to communication commands and gossip scheduling in `TrainStage` and `GossipModelStage`, where we inject version tagging, staleness-aware weighting, and degree-based neighbor selection.

Dataset abstraction and partitioning. The `p2pfl` framework offers a `P2PFLDataset` abstraction that supports loading data from various sources: CSV files, JSON files, Parquet files, Pandas DataFrames, Hugging Face datasets, and SQL databases. It also supports data partitioning strategies for both IID and non-IID distributions, which lets researchers simulate realistic federated learning scenarios with uneven data distribution. The dataset abstraction can export to framework-specific formats, converting the unified representation into what each framework expects (e.g., PyTorch `DataLoader` for PyTorch Lightning, TensorFlow `tf.data.Dataset` for Keras). In our experiments, we use the MNIST dataset partitioned across nodes, with each node holding a subset of the training data. The abstraction ensures that each node's learner gets data

in the format its framework expects, while the learning interface stays framework-agnostic. This lets us run experiments with different ML frameworks without changing the communication or aggregation logic, and enables fair comparisons between baseline and enhanced communication layers by keeping the learning workload the same.

2.3 Checkpoints

Local checkpoints. We implement a local checkpoint mechanism to enable recovery from node failures and support fault tolerance. The system saves each node's state at key points during training, and checkpoints are saved automatically as part of the training workflow without manual intervention.

Each node maintains its own checkpoint directory organized by experiment name and node address. The directory structure follows `checkpoints/{experiment_name}/{node_addr}/`, and checkpoint files are named `checkpoint_round_{round}_{type}.pkl`. We save three types of checkpoints per training round: a `local` checkpoint after local training completes, an `aggregated` checkpoint after model aggregation finishes, and a `round_finished` checkpoint at the end of each round before moving to the next.

The checkpoint data structure is implemented as a `LocalCheckpoint` class containing everything needed to restore a node's state. This includes encoded model parameters (which may be compressed), experiment metadata like model architecture and aggregator type, the list of contributing nodes, the number of training samples used, and any additional model information from callbacks. The checkpoint also stores its own metadata: node ID, experiment name, round number, timestamp, and framework name.

When saving a checkpoint, the system collects the current model from the learner, encodes the parameters using the same serialization mechanism as network transmission, and packages all metadata into a `LocalCheckpoint` object. This object is serialized with Python's `pickle` module and written to disk. Checkpoint saving happens automatically at the right stages: after local training in `TrainStage`, after aggregation in `TrainStage`, and at round completion in `RoundFinishedStage`. If saving fails, the system logs a warning but continues training to avoid interrupting the learning process.

The checkpoint mechanism offers a simple but effective way to recover from failures. If a node crashes or disconnects, it can reload its most recent checkpoint and resume from the last saved state. Checkpoint files are stored locally on each node, so recovery doesn't depend on network connectivity or remote storage. This keeps the implementation simple and avoids the complexity of distributed checkpoint coordination, while still providing basic fault tolerance for individual nodes in the peer-to-peer network.

Remote checkpoints. To improve system fault tolerance, we introduce a remote checkpoint mechanism that complements local checkpointing. Local checkpoints are sufficient to handle **process-level crashes**, where the training process terminates but the underlying storage persists, allowing recovery by reloading the most recent local snapshot. However, they fail under **node-level failures**, where the execution environment is lost entirely, such as when a Docker container is deleted or a virtual machine instance is destroyed. In such cases, its local state becomes unavailable, so a remote checkpointing strategy is necessary.

The system periodically replicates compressed model snapshots to topologically distant peers, providing backup beyond each node's own storage. Remote checkpoints contain both model content and associated metadata required for versioning, traceability, and recovery, including the storage node ID, the original creator node ID, the checkpoint round and timestamp, and the compression method. This design enables the system to verify consistency and correctly locate and interpret checkpoints during restoration.

Remote checkpoints are generated every k rounds and transmitted to n other nodes, both of which are configurable. To reduce communication overhead, only model deltas are propagated after the initial backup. The system computes the difference between the current model and the most recent checkpoint, serializes the delta, and compresses it using `gzip` prior to transmission. This delta-based compression design substantially lowers bandwidth and storage cost in the P2P network while retaining full recovery capability.

To further improve resilience, replicas are placed on peers that are far away in the network topology. We approximate shortest path distances using breadth-first search and select maximally distant reachable nodes

as backup targets. This placement strategy increases the probability that replicas survive localized failures, enabling more robust recovery under stronger fault scenarios.

2.4 Recovery

To support rejoining of failed nodes, we adopt a three tier recovery strategy that minimizes restoration cost while maximizing the use of available information.

Priority 1: Local checkpoint recovery. When a node attempts to rejoin, it first searches for locally stored checkpoints, including local, aggregated, and round finished snapshots, since these correspond to valid model states captured at different stages of execution. If a matching checkpoint is found, it is loaded, decoded, and directly applied to the model parameters. Once restored, the node can immediately resume training without additional network communication.

Priority 2: Remote checkpoint recovery. If no usable local checkpoint exists, the system falls back to remote recovery, where the node queries neighbors for replicas of its state. The recovery logic iterates over neighboring nodes and scans checkpoint histories within a bounded round window. For each candidate checkpoint, the system verifies whether it was originally generated for the recovering node to ensure namespace correctness. If the remote checkpoint contains only delta information, the system loads the most recent full backup and reconstructs parameters layer by layer using the stored delta. This mechanism enables recovery even when a node's entire local state is lost.

Priority 3: Model reinitialization. If neither local nor remote checkpoints can be used, the node initializes a fresh model and continues training. While this approach discards the node's historical progress, it prevents the overall learning process from stalling. With a reasonable checkpoint schedule, this fallback is rarely triggered, but it provides a necessary safety net in extreme failure scenarios.

3 Experimental Evaluation on MNIST

The MNIST handwritten digit dataset is one of the most widely used datasets in machine learning, containing 60,000 training images and 10,000 test images across ten digit classes. Because of its simplicity and well defined evaluation criteria, MNIST is commonly adopted as a validation platform for distributed learning algorithms. In our study, we deploy MNIST under a P2PFL setting to assess the practical effects of our design innovations. Specifically, we evaluate whether our communication protocol improvements accelerate convergence and whether our failure recovery mechanisms preserve model performance in the presence of node crashes and state loss.

3.1 Convergence Improvement

Baseline setting. In the baseline setting, nodes communicate over a fixed topology and exchange model updates using a standard gossip protocol, where neighbors are randomly selected. We fix the training configuration to 10 nodes, 20 rounds, and 1 local epoch per round. Compared with the recovery experiment, we use more nodes and rounds because, at larger scales, the advantages of degree-aware gossip over random gossip become more visible. Increasing the communication layer's pressure and complexity better reveals the benefits of optimizing communication strategies. This configuration is also closer to real-world distributed learning systems.

Experimental setting. To assess whether our proposed communication protocol improves convergence efficiency, we design a controlled experiment where the communication mechanism is the only difference. In the experimental setting, the gossip policy is replaced with our degree aware version, which records how often nodes interact and preferentially communicates with highly connected peers.

Outcome analysis.

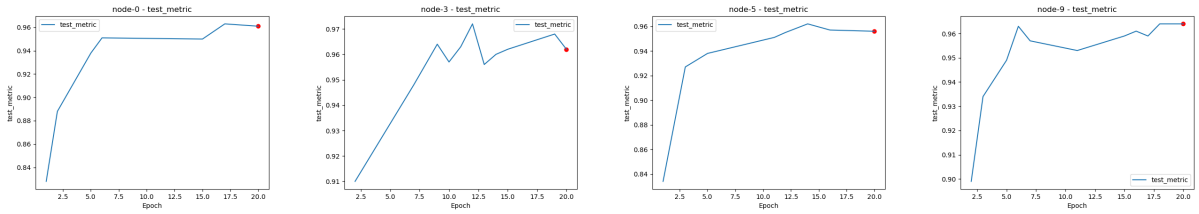
To analyze the performance of the improved protocol, we selected 4 nodes from the 10 nodes that showed obvious differences when using the two protocols. Their test accuracy curves are shown in Figure 1 and Figure 2.

For node 0, the standard gossip protocol showed a clear plateau between epochs 5 and 17, where the convergence curve nearly stagnated. The improved protocol reduced this stagnation and maintained continuous performance improvement during the mid training phase.

For node 3, the standard gossip protocol experienced several severe fluctuations, producing a sawtooth shaped curve. Although the improved protocol still showed some fluctuations, the oscillation amplitude was significantly reduced, resulting in a much smoother and more stable upward trend overall.

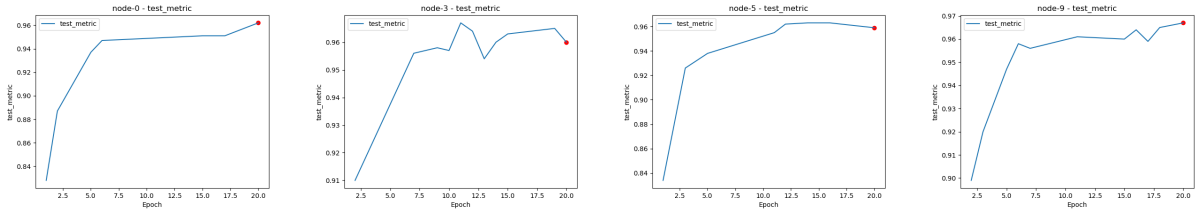
For node 5, the standard gossip protocol reached a peak of 0.962 at epochs 13 through 15, then declined to 0.955, showing approximately a 0.7% performance drop. The improved protocol remained stable after reaching its peak at epoch 13, effectively reducing late stage training fluctuations.

For node 9, the standard gossip protocol showed a noticeable spike at epochs 6 and 7 (approximately 0.963). This early peak could be misleading for the training process, as performance subsequently declined. During the mid training phase, the standard protocol dropped from its peak of 0.963 at epoch 7 to about 0.952 at epoch 12, a drop of about 1.1%. In comparison, the improved protocol steadily increased from 0.958 at epoch 6 to 0.961, avoiding performance regression and ultimately achieving about 0.3% improvement (0.967 vs. 0.964).



(a) Node 0 (b) Node 3 (c) Node 5 (d) Node 9

Figure 1: Baseline experiment: test accuracy curves for 10 nodes over 20 rounds.



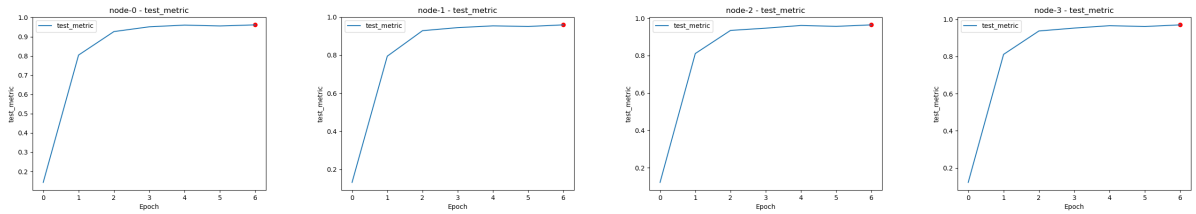
(a) Node 0 (b) Node 3 (c) Node 5 (d) Node 3

Figure 2: Convergence experiment: test accuracy curves for 10 nodes over 20 rounds.

3.2 Effectiveness of Recovery

Baseline setting. We fix the training configuration to 4 nodes, 6 rounds, and 1 local epoch per round. A round is functionally equivalent to a local training epoch. We use a small number of nodes and rounds for two reasons. First, it makes it easier to observe and track each node's state changes. Second, with fewer nodes, a single node failure represents a larger proportion of the total, which better reflects the impact of large-scale node failures. The baseline experiment represents a standard gossip-based training process without failures, state loss, or rejoining events, meaning that all nodes remain active and retain their local models throughout execution. Figure 3 presents the test accuracy curves observed across the four nodes under this baseline configuration.

Experimental setting. To evaluate the effectiveness of our checkpointing and multi tier recovery mechanisms, we design an experiment that emulates node failure and rejoining in real-world settings. At the

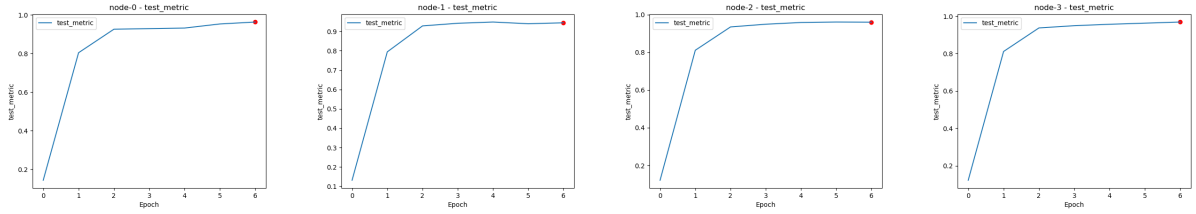


(a) Node 0 (b) Node 1 (c) Node 2 (d) Node 3

Figure 3: Baseline experiment: test accuracy curves for 4 nodes over 6 rounds.

beginning of training, all nodes participate normally and are connected according to the predefined topology. After the completion of round 2, we remove node 0 to simulate device dropout or network failure, disconnecting it from all other peers while the remaining nodes continue training. When round 4 begins, the failed nodes are recreated, reconnected to the network, and attempt to recover their model state through the checkpoint restoration process.

Outcome analysis. The experiment primarily examines system behavior, with model accuracy serving as a secondary evaluation criterion. We observed that training continued after node failures, and the failed node successfully rejoined the process. Its model state was correctly restored through checkpointing, and the network topology and interactions were not disrupted by disconnection or reentry. Furthermore, Figure 4 shows the test accuracy curves of 4 participating nodes. By comparing these results against the failure free baseline, we found that performance degradation caused by node failures is within an acceptable range. Throughout the process, system logs recorded events such as node shutdown, restart, reconnection, state restoration, and reintegration, supporting detailed analysis and validation.



(a) Node 0 (b) Node 1 (c) Node 2 (d) Node 3

Figure 4: Recovery experiment: test accuracy curves for 4 nodes over 6 rounds.

4 Conclusion & Reflection

Based on comparative experiments on the MNIST dataset, we validated that both our failure recovery mechanism and our communication protocol improvements achieved their intended goals. For failure recovery, equipping the system with both local and remote checkpointing enables decentralized federated learning to operate reliably under node failures. This design addresses a critical challenge in real world distributed learning deployments, where connectivity loss, device crashes, and storage failures are inevitable. For communication optimization, the improved protocol effectively enhanced training stability and reduced severe fluctuations during convergence. It also prevented performance degradation in the later stages of training, leading to more consistent model behavior. We expect that in more adversarial production environments, the benefits of our failure recovery and communication enhancements will become even more noticeable.

During development, our main challenges arose from adding new functionality on top of a highly abstracted framework. Since the framework offered limited entry points for inserting custom logic, implementing checkpoint handling and node recovery required careful state management and execution control. Moreover, a

restarted node must be reintegrated into the system, which involves updating neighbors, rescheduling training tasks, and managing timing and synchronization, which significantly increases implementation complexity.

Naturally, our system also has limitations. The most notable one is that it cannot fully remove scalability bottlenecks. Because each round still requires agreeing on an aggregator, federated learning continues to suffer from bandwidth concentration and coordination delays. In future work, we plan to investigate truly decentralized aggregation methods at larger scales and test the protocol under more demanding workloads. We expect that such settings will make the benefits of our communication mechanism more visible and give deeper insight into how fault tolerant decentralized systems can be designed and improved.

References

[1] Daniel Mauricio Jimenez Gutierrez, Yelizaveta Falkouskaya, José Luis Hernández-Ramos, Aris Anagnostopoulos, Ioannis Chatzigiannakis, and Andrea Vitaletti. On the security and privacy of federated learning: A survey with attacks, defenses, frameworks, applications, and future directions. *ArXiv*, abs/2508.13730, 2025.

[2] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[3] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[4] Dinh C. Nguyen, Viet Quoc Pham, Pubudu N. Pathirana, Ming Ding, Aruna Prasad Seneviratne, Zihuai Lin, Octavia A. Dobre, and Won Joo Hwang. Federated learning for smart healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55:1 – 37, 2021.

[5] p2pfl contributors. p2pfl: Peer-to-peer federated learning framework. https://github.com/p2pfl/p2pfl, 2024. GitHub repository.

[6] Ye Yuan, Jun Liu, Dou Jin, Zuogong Yue, Ruijuan Chen, Maolin Wang, Chuan Sun, Lei Xu, Feng Hua, Xincheng He, Xinlei Yi, Tao Yang, Hai-Tao Zhang, Shaochun Sui, and Han Ding. Decefl: A principled decentralized federated learning framework. *National Science Open*, 2021.

5 Appendix

The full implementation of our system is publicly available at:

https://github.com/sophia-mathcs/p2pfl_evolved.git

The `communication` branch contains our communication protocol enhancements, and the corresponding convergence experiment can be run using the following command:

```
DISABLE_RAY=1 python -m p2pfl.examples.mnist.mnist \
  --nodes 10 \
  --rounds 20 \
  --epochs 1 \
  --protocol memory \
  --topology full \
  2>&1 | tail -50
```

The `fault_tolerance` branch provides the implementation of checkpointing and recovery mechanisms, along with stored experiment checkpoints. The recovery experiment can be executed with:

```
bash run_mnist_with_failure.sh
```