Raft Protocol for Fault Tolerance and Self-Recovery in Federated Learning

Rustem Dautov
Erik Johannes Husom
rustem.dautov@sintef.no
erik.johannes.husom@sintef.no
SINTEF Digital
Oslo, Norway

Abstract

Federated Learning (FL) has emerged as a decentralised machine learning paradigm for distributed systems, particularly in edge and IoT environments. However, ensuring fault tolerance and self-recovery in such scenarios remains challenging, because of the centralised model aggregation which acts as a single point of failure. A possible solution to this challenge would rely on the continuous replication of the global FL state across participating nodes and the functional suitability of any node to replace the aggregator in case of failures. These functional requirements can be implemented using one of the existing distributed consensus algorithm, such as Raft. Our approach utilises Raft's leader election and log replication mechanisms to enable automatic stateful recovery after failures and thus to improve fault tolerance. The log replication process efficiently maintains consistency and coherence across distributed FL nodes, ensuring uninterrupted training process and model convergence. This enhances the robustness of the overall FL system, especially in dynamic and unreliable cyber-physical conditions. To demonstrate the viability of our approach, we present a proof-of-concept implementation based on the existing FL framework Flower. We conduct a series of experiments to measure the aggregator re-election time and traffic overheads associated with the state replication. Despite the expected traffic overheads growing with the number of FL nodes, the results demonstrate a resilient self-recovering system capable of withstanding node failures while maintaining model consistency.

CCS Concepts: • **Information systems → Data replication tools**; **Data federation tools**; • **Computer systems organization → Fault-tolerant network topologies**.

This work licensed under Creative Commons Attribution International 4.0 License.

SEAMS '24, April 15–16, 2024, Lisbon, AA, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0585-4/24/04
https://doi.org/10.1145/3643915.3644093

Keywords: Federated Learning, Fault Tolerance, Self-recovery, Flower, Raft, Consensus Algorithm

ACM Reference Format:
Rustem Dautov and Erik Johannes Husom. 2024. Raft Protocol for Fault Tolerance and Self-Recovery in Federated Learning. In *19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '24), April 15–16, 2024, Lisbon, AA, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643915.3644093

1 Introduction

Federated Learning (FL) has emerged as a promising paradigm for decentralised machine learning (ML) in distributed systems, particularly in scenarios involving edge devices and the Internet of Tings (IoT). While FL is designed to be decentralised in terms of data storage and privacy, the central aggregator plays a crucial role in coordinating the model updates across different devices or servers. If the central aggregator becomes unavailable or compromised, it can disrupt the entire FL process, making it a single point of failure for the whole system. To this end, ensuring fault tolerance and self-recovery in such environments remains a critical challenge.

A possible solution to this challenge would require two main functional elements to be implemented – namely, continuous replication of the global state of a FL cluster across participating nodes and functional suitability of any node to replace the aggregator in case of failures. To this end, this paper explores the integration of distributed consensus algorithms to implement a self-recovery mechanism in FL systems. Specifically, we delve into the significance of state replication and aggregator re-election as indispensable components in mitigating the impact of failures, ensuring the continued operation and reliability of FL across a network of heterogeneous and potentially unreliable devices. In this context, we see state replication as a promising instrument, which can enable every node in the federated system to maintain a consistent and up-to-date information about the training progress making it suitable to become the new aggregator and continue from the latest stored state in the event of failures or disruptions. In particular, the Raft consensus protocol is renowned for its simplicity and effectiveness, and

can be employed for replicating the state of the FL system across all participating nodes.

By employing Raft's leader election and log replication mechanisms to enhance fault tolerance and enable self-recovery, this paper investigates one of the possible ways of mitigating the single-point-of-failure problem in the next-generation FL architectures. Through this exploration, we aim to contribute to the ongoing discourse on enhancing the resilience and fault tolerance of FL in the face of real-world challenges. Accordingly, the contribution of this paper is threefold: *i)* the overall approach for self-recovery in FL using Raft, *ii)* a proof of concept demonstrating the viability of the proposed approach, and *iii)* an experimental evaluation of the key properties associated with the approach – namely, aggregator re-election time and network traffic overheads.

The rest of the paper is organised as follows. Section 2 briefs the reader on FL and highlights existing challenges associated with fault tolerance and self-recovery. Section 3 provides a summary of related work addressing the same issues. Section 4 then presents our own approach, followed by a proof-of-concept implementation in Section 5 and the experimental evaluation in Section 6. Section 7 concludes the paper and provides some directions for future work.

2 Research Context and Motivation: Federated Learning

FL is an ML approach that allows a model to be trained across multiple decentralised devices (such as smartphones, IoT devices, or edge servers) or even organisations (like hospitals or companies) without exchanging the underlying potentially sensitive data [35]. The central idea behind FL is to enable collaborative model training while keeping the data on the local devices or servers. The key advantages of FL lie in its ability to enhance privacy by keeping data localised and reducing the need for centralised data storage. This decentralised paradigm allows for model training on user devices, ensuring that sensitive information remains on the device, addressing privacy concerns associated with traditional centralised learning methods [20]. Another significant benefit of FL is its potential to improve efficiency and reduce communication costs. By training models locally on devices, FL minimises the need for extensive data transfer, making it suitable for scenarios with bandwidth limitations or high communication costs [9]. Moreover, FL enables the creation of more personalised and context-aware models, as training occurs directly on devices where data is generated, capturing local device-specific patterns [12]. Thanks to these benefits, FL is being increasingly adopted in various fields, including healthcare (for medical data analysis while protecting patient privacy), IoT (for smart devices with limited processing power and connectivity), and personalised recommendations (to improve user experiences without compromising user data) [23]. Some prominent FL frameworks include Tensorflow Federated,[1] FedML,[2], and Flower.[3] Here is how FL typically works:[4]

1. **Model initialisation**: A global ML model is initialised on a central server.
2. **Local training**: The local nodes perform model training on their respective datasets without sharing the data itself. The training process may involve multiple iterations of training and updating the model's parameters.
3. **Model update aggregation**: After local training, each device or server sends only the model updates (i.e., changes in model parameters) to the central server.
4. **Model aggregation**: The central server aggregates these model updates within the global model, following some pre-defined strategy (e.g., averaging the updates).
5. **Iteration**: Steps 2 to 4 are repeated for multiple rounds, improving the global model with each iteration.

2.1 Motivation: Need for Fault Tolerance and Self-Recovery

The term *federation* itself assumes there exists a central point of coordination. Thus, one of the most critical challenges FL is the vulnerability introduced by a potential single point of failure – that is, a centralised aggregator or coordinating node. The reliance on a single point for aggregating and coordinating model updates in FL systems makes them vulnerable to disruptions. Whether due to hardware malfunctions, network outages, or other unforeseen events, the failure of this central node can bring the entire learning process to a halt, jeopardising the integrity of the model and impeding the system's performance.

Addressing this single point of failure is imperative for the widespread adoption and success of FL. The implementation of fault tolerance and self-recovery mechanisms becomes a pressing need to ensure the continuous operation of the system. This motivation stems from the recognition that the robustness of FL hinges on its ability to adapt and recover seamlessly from failures, guaranteeing the reliability and scalability required for practical deployment in dynamic and diverse real-world environments. As we further highlight in the next section, the existing solutions are often too complex and heavy-weight to be applied in resource-constrained IoT scenarios, e.g., dealing with wearable devices coupled with smartphone gateways [11, 19]. This also poses a practical requirement that an envisioned solution should not introduce any additional overheads in terms of the required software stack, but rather be able to natively integrate with the existing FL frameworks, especially when deployed on resource-constrained cyber-physical platforms [10].

[1] https://www.tensorflow.org/federated
[2] https://www.fedml.ai/
[3] https://flower.dev/
[4] A more detailed FL sequence diagram is included in Fig. 2 as part of the description of the proposed approach.

3 Related Work

The use of a consensus algorithm for self-recovery and peer-to-peer communications in FL is reported by Behera et al. [1]. The authors share the same motivation of avoiding a single point of failure, and use the Raft protocol to enable the automatic re-election of FL aggregators. However, the approach does not use state replication, and the paper being just 5 pages long remains at a rather high conceptual level, lacking technical details describing the proof-of-concept implementation, as well as how the experiments were conducted. Albeit not peer-reviewed and not officially published, to the best of our knowledge, this is the only openly accessible research publication conceptually similar to our proposed approach. At the same time, the community has come up with a number of alternative approaches addressing the same challenge. We group them into two main categories and summarise in the next two subsections.

3.1 Decentralised Federated Learning

Decentralised FL is an extension of the traditional FL, which advocates for a fully decentralised architecture both for model training and aggregation across distributed nodes [18]. The research landscape in this area has been characterised by advancements in so-called *gossip learning* and *swarm learning*.

Gossip learning [15] involves decentralised communication between nodes in a network, where nodes exchange information about their local models. This behaviour is driven by gossip algorithms (whose name was inspired by how gossips are spread among people) [4, 5] – a class of distributed communication protocols wherein nodes or entities within a network disseminate information by exchanging messages with randomly selected peers, facilitating the spread of data and enabling decentralised systems to collectively converge on consistent states or make collective decisions through a process of iterative and probabilistic communication. In gossip learning, fault tolerance is achieved through redundancy and continuous communication. If a node fails or starts malfunctioning, other nodes can compensate by sharing their correct models. The decentralised nature of gossip learning inherently provides fault tolerance, as the failure of a single node does not compromise the entire learning process [16].

Swarm learning [31, 34] is another related concept that achieves fault tolerance through the diversity of nodes and the FL approach. Even if some nodes fail or provide inaccurate updates, the global model can still benefit from the contributions of other nodes. The FL paradigm ensures that no single point of failure exists, and the system can adapt to the varying reliability of nodes. A potential additional feature in swarm learning is the use of blockchain, which underpins peer-to-peer communications and state replication.

Both gossip and swarm learning, however, are still emerging technologies, and their adoption in real-world applications might not be as widespread as the traditional FL with centralised model aggregation or other more established ML techniques. In theory, the decentralised model aggregation would natively address the single point of failure, but in practice these approaches are still too complex to implement in real-life scenarios in a generic and re-usable manner.

3.2 Using Distributed Ledger Technology in Federated Learning

The integration of Distributed Ledger Technology (DLT) for state replication in the IoT represents another transformative approach that leverages the decentralised and secure features of DLT to ensure consistent and tamper-resistant states across a network of federated devices [21, 25, 28]. Noteworthy, Raft is also employed in some DLT implementations as the underlying consensus algorithm (e.g., Hyperledger Fabric[5]), as well as the more heavy-weight and DLT-oriented algorithms, such Proof-of-Work (PoW) and Proof-of-Stake (PoS). The combination of decentralised consensus, tamper-resistant records, smart contracts, and other features positions DLT as a powerful solution to address the challenges posed by state replication in IoT environments. DLT's distributed nature ensures resilience to individual node failures. In instances where one node goes offline or experiences issues, the remaining nodes continue to maintain and replicate the state. This resilience is crucial in cyber-physical environments where devices may be intermittently connected or face hardware failures.

The use of DLT for storing the FL progress is also often driven by the need to enable trustworthiness and data provenance among multiple potentially untrusted parties. Through a shared and public ledger, all participating nodes have visibility into the replicated state, which promotes transparency and accountability. This transparency is essential for auditing purposes, as the entire transaction history is recorded on the ledger, facilitating efficient tracking of state changes. This is often used in FL scenarios where there is a need for trustworthiness among participating nodes, such as financial or healthcare sectors [24, 32].

However, deploying DLT on resource-constrained devices in the IoT context poses challenges due to limited processing power, storage, and energy resources. Running complex DLT algorithms may strain these devices and compromise their primary functions. The energy-intensive consensus mechanisms of DLT can also reduce the efficiency of devices powered by batteries or energy harvesting. Also, scalability issues arise as maintaining a complete ledger on each device can overwhelm resources, especially in applications with a high volume of transactions [13]. Latency introduced by DLT

[5] https://www.hyperledger.org/projects/fabric

consensus processes may be impractical for real-time applications. The decision to deploy DLT on resource-constrained devices in FL should align with specific use case requirements, considering alternatives like lightweight consensus algorithms natively integrated at the code level for more practical and efficient solutions.

4 Proposed Approach: State Replication Using Raft Protocol

The surveyed related work suggests that fault tolerance in cyber-physical FL scenarios is not sufficiently addressed by the existing approaches, which are often too complex and heavy-weight to be implemented, especially on resource-constrained platforms. In this paper, we propose to natively integrate state replication into FL using a consensus algorithm to build a light-weight, yet efficient and reliable self-recovery solution.

4.1 Consensus Algorithms for Replicating the Training Progress

Consensus algorithms are commonly used in distributed databases, distributed file systems, and blockchain technology to ensure that multiple nodes or processes in a network reach an agreement or a consistent state. They play a crucial role in maintaining the reliability and fault tolerance of distributed systems. They help ensure that data remains consistent and that decisions are made across all nodes in a distributed network, despite failures or network issues.

Consensus algorithms often involve a leader, which is a specific node responsible for proposing decisions and driving the consensus process. Leader election is used to select this node. Leader election mechanisms ensure that the leader is chosen in a way that is fair and robust, even in the presence of node failures or network partitions. Consensus algorithms and leader election are often used in conjunction to achieve coordination and fault tolerance in a group of distributed nodes or processes. Common leader election algorithms include, for example, the Bully algorithm [26] and the Ring algorithm [14].

Consensus algorithms also enable state replication – a technique used to ensure that the state (data) of a distributed system remains consistent across multiple nodes. In a distributed system, multiple nodes may store and modify data independently, and state replication helps maintain a coherent state across all nodes. Some state replication techniques include *primary-backup replication* [6], *multi-version concurrency control* [27], and *log-based replication* [2]. The latter involves recording all state-changing operations in a log. All nodes receive and apply the same sequence of operations to their local state, ensuring that they stay consistent.

Some common consensus algorithms include *Paxos*, which is a family of algorithms designed to handle network and node failures [33], and *Practical Byzantine Fault Tolerance (PBFT)*, which was specifically designed for systems with Byzantine failures [7]. *Raft* is another prominent consensus algorithm, conceived by Ongaro and Ousterhout [29], which provides a foundational framework for achieving distributed consensus in fault-tolerant systems. Operating on a leader-follower model, Raft orchestrates a cluster of nodes with the primary objective of maintaining a consistent, replicated log across all participants. The process initiates with an election phase, where nodes engage in distributed voting to select a leader. Heartbeat messages are exchanged to monitor if the leader is still responsive, triggering a new election if the leader becomes unreachable.

Raft organises time into terms, each characterised by a unique numeric identifier, with the leader managing log replication within a term. The log itself comprises commands representing state machine transitions. To ensure consensus, the leader dispatches messages to followers, containing log entries for replication. Entries are committed only when a majority of nodes acknowledge the receipt, preventing inconsistencies in the event of node failures or network interruptions. Incorporating a randomised election timeout mitigates the risk of split votes, enhancing the algorithm's resilience. Furthermore, Raft enables nodes to voluntarily step down if they discover a more up-to-date leader during the election process, contributing to operational efficiency.[6]

Taken together, Raft provides a clear and intuitive approach to distributed consensus, offering fault tolerance, consistency, and leader stability in diverse distributed computing scenarios. Its well-defined mechanisms for leader election, log replication, and commitment make it widely adopted for ensuring consensus in distributed systems. Finally, Raft is more understandable and user-friendly than Paxos [17], thanks to which the community has come up with multiple software implementations in many mainstream languages, making Raft a convenient option for natively integrating state replication at the code level, rather than using external tools or libraries.

4.2 Enhancing Federated Learning with Raft

Now, equipped with the general knowledge about FL and consensus algorithms, we explore how the two can be effectively combined to facilitate self-recovery. We first propose and discuss 4 possible levels, at which self-recovery can be integrated into FL. Table 1 summarises these levels, where we introduce the term *checkpoint* – a point, when the aggregator or worker nodes write their current state into the global replicated state of the whole FL cluster.

Determining the most practical and efficient way to implement checkpoints in FL using the Raft protocol involves considering several factors, including system architecture, communication overheads, fault tolerance requirements, and

[6] Explaining the internals of the Raft protocol goes beyond the scope of this paper. Interested readers are referred to the original paper [29].

Level	State replication	Description
Level 0: no self-recovery	N/a	This level represents the current state of practice, where no FL state is stored/replicated. With the aggregator being a single point of failure, the FL progress is lost and cannot be automatically recovered in case of failures.
Level 1: self-recovery with the same aggregator.	Aggregated model parameters stored locally without replication.	The aggregator continuously stores intermediate training results locally and is able to restart the FL process from the latest checkpoint after a crash. This, however, strictly assumes that the same node will remain the aggregator. This kind of functionality is offered by most ML frameworks, which allow storing serialised training results on disk.
Level 2: stateless self-recovery.	N/a	The FL cluster is able to re-elect a new aggregator among the remaining worker nodes following the Raft protocol, but the re-elected aggregator will start the FL from scratch, since no intermediate training progress was stored/replicated.
Level 3: stateful self-recovery.	Aggregated model parameters at the server side.	This is a combination of the previous two levels, where the aggregator not only stores the intermediate FL progress, but also replicates it across all FL nodes. Any node is suitable to become a new aggregator and re-start the training from the latest checkpoint.
Level 4: stateful self-recovery of the aggregator and worker nodes.	Aggregated model parameters at the server side. Individual parameters at the client side.	All possible information is stored and replicated across all nodes. This means that any worker node can re-start its training routine by fetching its previous state from peer nodes. To a great extent, this might not be very practical due to the increased amounts of redundant information being exchanged and the fact that a failure of a worker node (as opposed to the aggregator) can often be tolerated/neglected. Still, this kind of fully stateful self-recovery can be possibly applied in some critical scenarios, where worker nodes are valuable in terms of their unique training data.

Table 1. Possible levels of self-recovery in FL using Raft.

the specific characteristics of the FL scenario. Another degree of freedom here is the frequency of checkpoints, especially on the worker nodes which may be configured to run several training iterations (epochs) within a single training round. Since on resource-constrained devices, even a single epoch might a considerable amount of time, storing the intermediate progress might be beneficial. The increased frequency of checkpoints may enhance fault tolerance but will inevitably increase the overall system load and communication overheads.

With the current state of the FL technology, we consider Level 3 as the most straightforward and practical way of implementing state replication and self-recovery, as we further explain it in Section 5. At this level, the centralised storage at the aggregator simplifies the checkpoint management process, as all updates come from a single point and are then replicated to the rest of the nodes. Furthermore, the primary challenge to be addressed is the aggregator being the single point of failure, whereas the failures of worker nodes can often be tolerated, while the communication costs of replicating their state is unproportionally high.

5 Proof of Concept

We now proceed with an explanation of a proof-of-concept implementation, starting with the description of the two main underpinning technologies – namely, Flower framework and PySyncObj library.

5.1 Flower: Baseline for Federated Learning

Flower is an open-source Python framework designed to facilitate the implementation of FL systems [3]. Flower aims to simplify the development of FL applications by providing a high-level interface and abstractions for communication and coordination among devices in a distributed system. One of the notable features of Flower is its use of strategies to define the communication and collaboration patterns between the server (aggregator) and the clients (worker nodes). Flower's strategy interface is designed to be user-friendly, enabling developers to implement and experiment with FL without delving into the internals of distributed systems and communication protocols. More specifically, users can benefit from Flower strategies in the following ways:

- **Strategy definition**: Flower allows users to define custom strategies that dictate how the server interacts with the clients during the FL process. Strategies are responsible for managing communication, aggregating model updates, and handling synchronisation.
- **Communication abstraction**: strategies in Flower hide away the details of communication between the server and clients. This includes specifying how model parameters are exchanged, how updates are aggregated, and how synchronisation is managed across the distributed FL cluster.

- **Integration with ML frameworks**: Flower integrates seamlessly with multiple ML tools, including the mainstream frameworks TensorFlow, PyTorch, XGBoost and MXNet, allowing users to define how models are trained and updated across multiple nodes.
- **Customisation**: Flower allows developers to define strategies tailored to their specific FL use cases. This adaptability is crucial in cyber-physical scenarios where different communication patterns or synchronisation methods are needed to accommodate slow execution and/or device failures. Strategies can also be designed to accommodate different data distribution patterns and privacy considerations. This latter feature was used extensively in this work to implement the proposed state replication.

To use Flower effectively, developers typically define their FL strategies, integrating them with their ML models and leveraging Flower's communication abstractions. This allows for the seamless orchestration of FL processes across a distributed network of devices constituting a FL cluster.

5.2 PySyncObj: a Raft Protocol Implementation

PySyncObj[7] is an open-source Python library for building fault-tolerant distributed systems with the ability to replicate and synchronise objects across multiple nodes. It implements the Raft consensus algorithm and is designed to provide a simple and flexible solution for building distributed applications. PySyncObj is designed to be user-friendly and easy to integrate in Python applications. It abstracts the complexities of distributed systems, making it accessible for developers who may not be experts in distributed computing. Key features of PySyncObj include:

- **Consensus algorithm**: the underlying Raft consensus algorithm helps achieve distributed consensus among nodes in a network. As already discussed, Raft is known for its simplicity and ease of understanding compared to more complex and heavy-weight alternatives.
- **Data replication**: PySyncObj enables the replication of objects across multiple nodes, ensuring that changes to the state of an object are synchronised across the distributed system.
- **Leader election**: The Raft consensus algorithm involves the election of a leader among nodes, and PySyncObj implements this leader election process to coordinate and manage the distributed state.

5.3 Prototype Implementation

The proof of concept was built on top of the Flower framework enhanced with PySyncObj used for aggregator election and state replication.[8] Normally, Flower requires the developers to implement at least two separate Python classes – i.e., one for the server, and the other one for the clients. However, as we argued throughout the paper, in order to replace a faulty aggregator, a worker node must not only be able to recover the state, but also be equipped with the required aggregator execution logic. In other words, in our implementation all FL nodes execute exactly the same code base, and, depending on the election outcome, proceed with either the aggregator or the worker role. The corresponding algorithm is depicted in the activity diagram in Fig. 1.

The process starts with the aggregator election, based on which each node proceeds with the corresponding execution logic. The newly elected aggregator, no matter whether it is the initial FL launch or recovery, first checks whether there is any previously stored state to continue from. If it is a recovery, the state will include the number of completed and remaining training rounds, as well as latest aggregated model parameters. If not, the state will be empty. The elected aggregator then starts coordinating the FL process by first distributing input parameters and then collection results from several rounds of training and evaluation. The algorithm includes three checkpoints, when the aggregator stores aggregated parameters, which are then replicated to worker nodes. Upon completion of each training or evaluation round, worker nodes check whether the aggregator is still alive. If not, they transition back to the initial leader election phase. The expected final outcome of this process is a converged and consistent ML model.

A slightly different view on the same dynamics within a FL cluster is depicted in a simplified sequence diagram in Fig. 2, which includes two main stages of the resulting design – namely, the initial phase of leader election among several equally suitable nodes and the following phase of actual FL with an elected aggregator and several worker nodes. In case of aggregator failures, all worker nodes switch back to the initial phase and participate in the re-election, which results in a new aggregator and new worker nodes (provided there is still sufficient nodes for federation). The new aggregator will check the progress since the last checkpoint and use the existing model parameters as the new initial parameters for the newly selected worker nodes.

As discussed in Section 4, implementing state replication only at the aggregator side (Level 3 in Table 1) seems the most practical and realistic approach for most FL scenarios, which do not need tracking of each FL client's progress. These checkpoints are included in the extended strategy definition, and are executed by the aggregator. At these points, the aggregator writes most recent model parameters to a replicated state object (essentially – a key-value dictionary), which is then asynchronously copied to the rest of the nodes in a separate thread, parallel to the main FL execution logic. This way, all worker nodes are synchronised on the training progress and equipped with the required execution logic in order to step up as the new elected aggregator, if needed.

[7] https://github.com/bakwc/PySyncObj/
[8] Source code with instructions for running the experiments: https://github.com/SINTEF-9012/raft_flower/

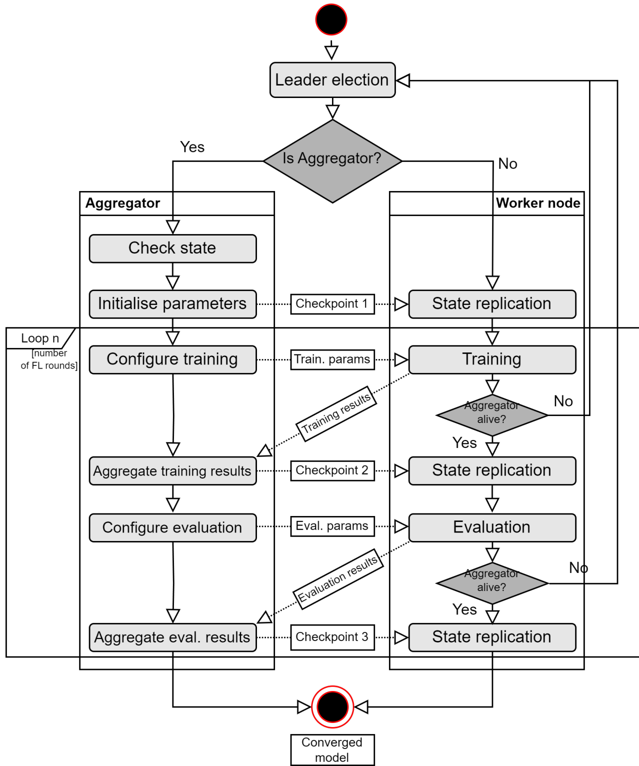


Figure 1. Algorithm of a Raft-enhanced FL node.

6 Experimental Evaluation

To demonstrate the viability of the proposed approach, we now proceed with the experimental evaluation of the implemented proof of concept. The experiments involve training a convolutional neural network (CNN) on the CIFAR-10 dataset [22] in a federated setup. The full dataset consists of 60,000 colour images 32x32 pixels each, divided into 10 classes with 6,000 images per class. CIFAR-10 is widely used in the ML community for benchmarking image classification algorithms, owed to its diverse content and moderate size, making it a practical choice for experimenting with new models and techniques. CNNs are one of the most used methods in deep learning, particularly suited for tasks like image classification due to their proficiency in processing data with grid-like structures, commonly present in imagery data. This compatibility makes them a natural choice for analysing the CIFAR-10 dataset. As already explained, Flower's compatibility with various ML frameworks ensured seamless integration with PyTorch[9] [30] – a widely-used open-source ML library developed by Facebook's AI Research lab. Our proof of concept extends the quick-start tutorial[10] on how to use Flower together with PyTorch. The setup consists of a central server coordinating a pre-defined set of client nodes. Each client independently processes a subset of the CIFAR-10 dataset. This distributed approach not only simulates the decentralised nature of FL, but also adheres to privacy-preserving principles, as each client's data remains local. Clients generate individual model updates (approx. 250 kB in size when serialised) based on their local datasets, which are then aggregated by the server to enhance the model. These updates constitute one FL round. This baseline experiment reflects a real-life FL scenario where, for example, smartphones could train a model using their local photos, such that each device (client) would process its data locally, updating and improving a shared model while maintaining data privacy.

6.1 Experiment Testbed: a Raspberry Pi Cluster

In our testbed, we established a network comprising 11 Raspberry Pi 3 boards interconnected through a local wireless network, forming the foundation for a FL cluster (1 aggregator node + up to 10 worker nodes). Each Raspberry Pi board is assigned with an IP address and is SSH-accessible. All boards run a 64-bit Raspbian Bullseye OS and are equipped with all necessary software (Python 3.9 + all required packages). Such FL setup offered an easily controllable hands-on environment to investigate the dynamics of decentralised learning, perform the measurements and evaluate our proposed approach against the baseline version of the Flower framework. Using physical Raspberry Pi boards is also intended to demonstrate the viability of the proposed approach in real-world edge and IoT scenarios, offering insights into resource constraints, scalability, model convergence, and communication efficiency for FL applications.

6.2 Experiments and Results

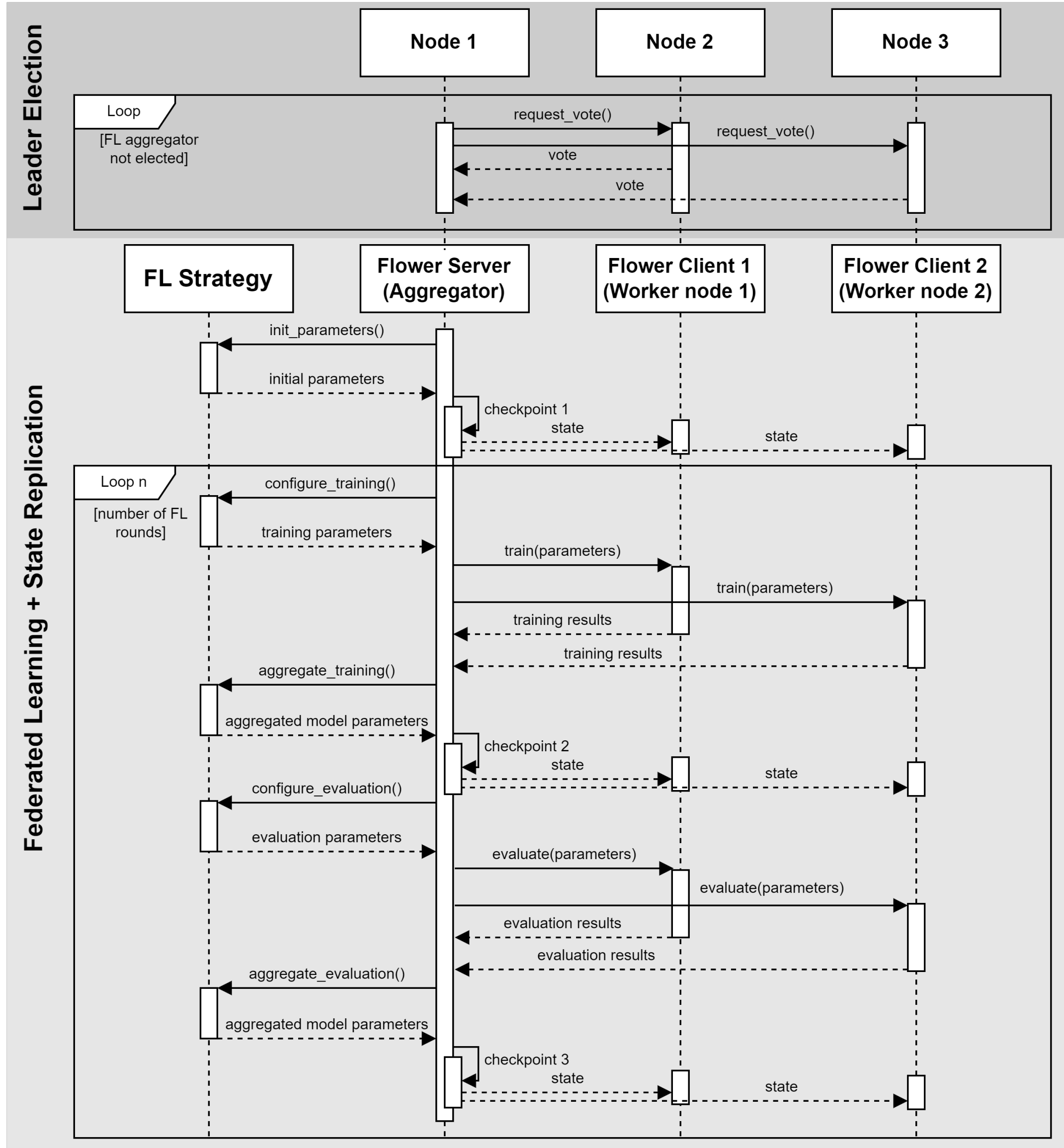
In our experiments we gradually increased the number of worker nodes from 1 (which is the minimum possible number to start and run a FL process) to 10. This way, we were able to observe how the two main measured metrics – namely, aggregator re-election time and network throughput – react to the increasing number of FL nodes. In both experiments we compared our Raft-enhanced Flower implementation against the baseline version of Flower's PyTorch tutorial.[11]

6.2.1 Measuring the FL aggregator re-election time. The main functional property of the described approach is self-recovery – the ability of a running FL cluster to continue the distributed training process even if the aggregator fails, by re-electing a new aggregator and continuously replicating the training progress across all FL nodes. By launching from 2 to 10 worker nodes, and intentionally killing the current aggregator, we were able to observe how the nodes soon

[9] https://pytorch.org/

[10] https://flower.dev/docs/framework/tutorial-quickstart-pytorch.html

[11] Please note that in this work, we are not reporting on the training results of the FL process and the accuracy of the resulting ML model. In fact, our implementation is based on an existing PyTorch CNN and does not affect the internal ML training logic or data, which means the model accuracy and loss remained the same as in the original baseline implementation.



* The inter-node communication in the leader election phase is simplified for clarity purposes. In reality, the Raft protocol relies on several rounds of bi-lateral message exchange with random time-outs across all nodes.

** During the state replication at the checkpoints, the message exchange is more complicated and includes acknowledgement receipts from all nodes. Also, there is continuous heartbeat monitoring of the aggregator node. For clarity purposes, we also omit these interactions in the diagram.

Figure 2. Raft-enhanced FL sequence diagram.

detect the absence of the aggregator and proceed with re-electing a new one following the Raft protocol.

In this first batch of experiments, the remaining nodes were able to successfully re-elect a new aggregator and complete the training. The time required to complete the re-election grows slightly with respect to the number of FL nodes. As demonstrated in Fig. 3, with a FL cluster of up to 10 nodes the re-election after a single failure can be accomplished within 4 seconds. In case of consecutive failures of re-elected aggregators, the total re-election time overheads will be proportional to the number of failures. Some additional time should also be considered for recovering the replicated training progress, which will depend on a specific ML training algorithm and the size of training parameters. In any case, we consider the re-election time overhead to be relatively low compared to the usual duration of ML training, especially on resource-constrained devices. Noteworthy, in the current implementation, the serialised training parameters are not persisted on disk, but stored in memory. This implies that as long as there is a single functioning node remaining from the original FL cluster, the state can be again replicated and recovered on newly-added nodes.

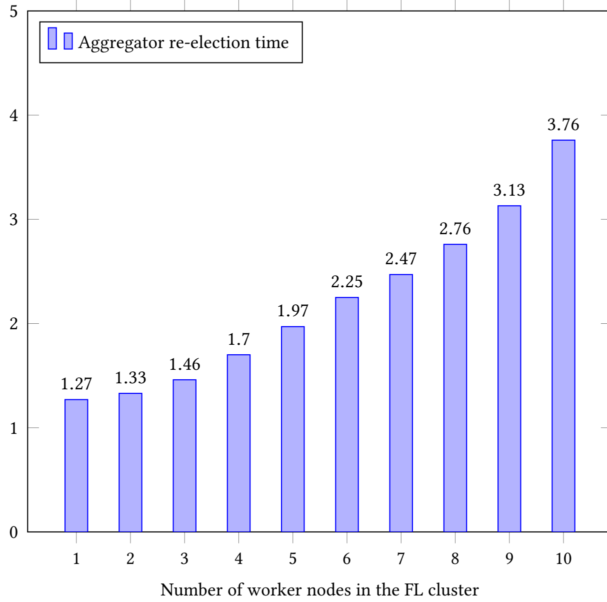


Figure 3. Aggregator re-election time (in seconds).

6.2.2 Measuring the FL cluster throughput.

Admittedly, state replication using Raft introduces additional traffic overheads on top of the baseline implementation, and the goal of this experimental evaluation was to quantify these overheads. To achieve this, we relied on the established **iftop**[12] utility that provides a convenient way of measuring TCP/UDP network traffic per network port. In the proof-of-concept implementation, we separated the network communication on the port 8080, where all native FL-related traffic is exchanged, and the port 5000, which we have assigned to the Raft-based leader election and state replication activities. This way, by distinguishing between the TCP traffic on ports 8080 and 5000, we were able to measure the network throughput separately for both cases.

In this experiment, another controllable parameter in addition to the number of nodes could be the number of FL training rounds. The initial experiments quickly showed that the network throughput proportionally increases with the number of FL rounds (i.e., doubles with 2 rounds, triples with 3 rounds, and so on). For simplicity, in our experiments we limited the FL operation to only 1 round.

In the second batch of experiments summarised in Fig. 4, two clear trends can be observed. On the one hand, it is evident that with the baseline Flower strategy implementation (i.e., blue bars), the amount of network traffic is in a linear dependency to the number of FL node. This is quite understandable, given the star topology of the baseline FL setup. On the other hand, the introduction of the Raft protocol and state replication introduces considerable traffic overheads (i.e., red bars), growing sharply with respect to the number of worker nodes. The trend, however, is still somewhat linear, and the additional traffic is approx. 5 times more than the baseline traffic. This increase can be explained by the internal design and implementation of the Raft protocol which relies on a mesh topology to perform leader election, followed by a star topology for state replication and continuous heartbeat message interchange. While this might be an acceptable price for having stateful self-recovery, there is still room for further traffic optimisations, as we discuss in Section 7.

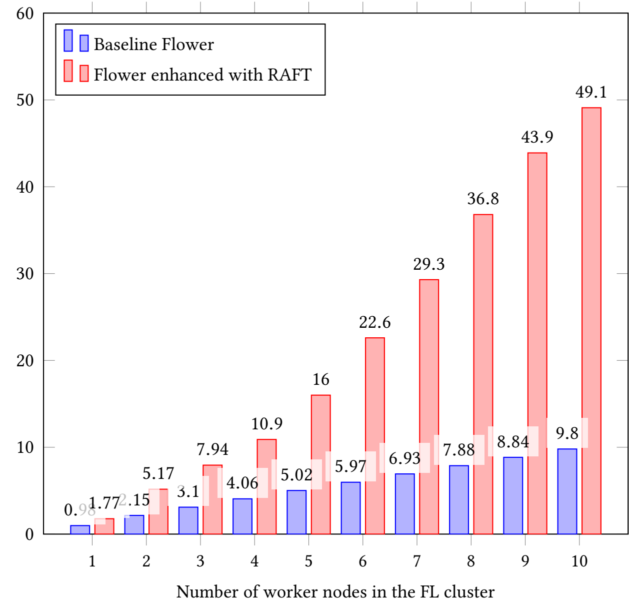


Figure 4. Network throughput (in MBs).

6.3 Assumptions and Threats to Validity

Model initialisation and code redundancy: The proof-of-concept implementation assumes that each node might be re-elected as a new aggregator, and thus initially possesses the ML model. This might not align with real-world scenarios where nodes might join the network dynamically with different initial states. Same goes for the code base, which is the same for all nodes, even though only one of them will execute the aggregator logic.

Idealised experiment conditions: The conducted experiments on network throughput were 'clean runs', i.e., did not include any node crashes and re-elections. This does not reflect real-world conditions where network instability, node failures, and dynamic changes are common. In such cases of disruption, the throughput is expected to be somewhat higher, depending on the size of the FL cluster. Furthermore, same considerations apply to the re-election time, which will depend on the congestions and latency of real-world networks.

Data handling and external traffic: In the experiments, training and evaluation datasets were pre-downloaded and

[12] https://pdw.ex-parrot.com/iftop/

not part of the network traffic. In practice, data might be generated or collected on the nodes, which means additional considerations for data traffic and bandwidth usage are needed. The assumption of pre-downloaded data simplifies the experiment, but might not accurately reflect the complexities of data management in a real-world distributed setting.
Increased network traffic and workload: One of the inherent benefits of FL in resource-constrained environments is the minimal network traffic, primarily involving model parameter exchanges. Our approach, with Raft's mechanisms for log replication and state synchronisation, introduces substantial network traffic and (de)serialisation overheads. This could challenge the suitability of the proposed system in environments with limited bandwidth or computational resources, deviating from one of the key advantages of the traditional FL. On the other hand, this footprint is still considerably lower than, for example, using DLT or a more heavy-weight and complex consensus algorithm like Paxos.

In our CIFAR-10 example, the average size of serialised parameters is approximately 250 kB. In real-life applications, particularly those involving more complex ML models or larger datasets, the size of these serialised parameters can be significantly larger, reaching the order of megabytes or even more. Multiplied by an increased number of federated nodes (i.e., tens or hundreds), and a higher number of training rounds, the network traffic burden escalates considerably. Such scenarios need to be considered, particularly in environments where bandwidth is a limiting factor. This potential increase in network load underscores the need for further optimisations to maintain the efficiency in more complex FL scenarios.

7 Conclusion and Future Work

In conclusion, this paper has presented a novel and promising approach to enhance the robustness and reliability of FL systems through the integration of self-recovery mechanisms using the Raft consensus protocol. By leveraging Raft's distributed consensus algorithm, we have addressed the inherent challenges of FL, such as node failures and network inconsistencies, leading to a more resilient and fault-tolerant system. Our experimental results demonstrate the viability of the proposed self-recovery mechanism, albeit at a cost of increased network traffic. The Raft protocol's ability to dynamically elect leaders and maintain consistency among distributed nodes has proven instrumental in reducing downtime and mitigating data loss.

In more practical terms, the benefits of our proposed approach extend beyond the theoretical realm, offering a real-world solution to enhance the reliability of FL systems in distributed and dynamic cyber-physical environments. As the demand for FL continues to grow across various domains, incorporating light-weight self-recovery mechanisms without strict dependency on external heavy-weight tools represents a significant step forward in ensuring the scalability, fault tolerance, and long-term sustainability of these systems. The findings presented in this paper contribute valuable insights to the ongoing research and development efforts aimed at advancing the field of FL.

7.1 Future work

Evaluation against DLT: DLT and Raft-based log replication offer distinct approaches to state replication in distributed systems. DLT, commonly associated with blockchain, employs decentralised consensus mechanisms, providing tamper resistance and transparency. However, it may face scalability challenges and can be computationally intensive. In contrast, Raft offers a more efficient consensus approach, particularly suitable for scenarios prioritising simplicity, quick consensus, and scalability. The choice between the two depends on specific application requirements, with DLT excelling in trust-sensitive applications like cryptocurrencies, while Raft being more suitable for distributed databases and systems such as FL clusters where rapid, decentralised consensus is critical. A more formal evaluation of the two approaches against a common benchmark would be a promising next step for further work.
Selective state replication: As demonstrated by the experiments, there is a considerable traffic overhead growing sharply with respect to the number of FL nodes. In order to reduce this, it might me convenient to replicate the state only to a limited sub-set of nodes, which are most suitable to become the newly elected aggregator. The suitability can be decided based on, for example, the physical resources available to some devices, i.e., powerful computing resources, network bandwidth, or fixed power supply [8].
Gossip algorithms for more traffic-efficient state replication: Another possible solution to the same challenge is applying a gossip algorithm, where each node communicates with a small subset of randomly chosen neighbours, exchanging information about the system state. By disseminating information through a series of local interactions, gossip algorithms efficiently propagate updates throughout the network without relying on global coordination. This selective and random communication strategy is expected to significantly reduce the overall volume of messages exchanged compared to our own approach.

8 Acknowledgments

This work has received funding from: the European Union's Horizon Europe research and innovation programme under grant agreements No. 101135576 (INTEND), No. 101095634 (ENTRUST), and No. 101120657 (ENFIELD), the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101020416 (ERATOSTHENES), the Research Council of Norway's BIA-IPN programme under grant agreement No. 309700 (FLEET).

References

- [1] Monik Raj Behera, Suresh Shetty, Robert Otter, et al. 2021. Federated learning using peer-to-peer network for decentralized orchestration of model weights. (2021).
- [2] Kevin Beineke, Stefan Nothaas, and Michael Schöttner. 2016. High throughput log-based replication for many small in-memory objects. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 535–544.
- [3] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, et al. 2020. Flower: A friendly federated learning research framework. (2020). https://doi.org/10.48550/arXiv.2007.14390
- [4] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. 2005. Gossip algorithms: Design, analysis and applications. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 3. IEEE, 1653–1664.
- [5] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. 2006. Randomized gossip algorithms. *IEEE transactions on information theory* 52, 6 (2006), 2508–2530.
- [6] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. 1993. The primary-backup approach. *Distributed systems* 2 (1993), 199–216.
- [7] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.
- [8] Rustem Dautov and Salvatore Distefano. 2019. Automating IoT data-intensive application allocation in clustered edge computing. *IEEE Transactions on Knowledge and Data Engineering* 33, 1 (2019), 55–69.
- [9] Rustem Dautov and Salvatore Distefano. 2020. Stream processing on clustered edge devices. *IEEE Transactions on Cloud Computing* 10, 2 (2020), 885–898.
- [10] Rustem Dautov, Salvatore Distefano, Dario Bruneo, Francesco Longo, Giovanni Merlino, and Antonio Puliafito. 2018. Data processing in cyber-physical-social systems through edge computing. *IEEE Access* 6 (2018), 29822–29835.
- [11] Rustem Dautov, Erik Johannes Husom, Fotis Gonidis, Spyridon Papatzelos, and Nikolaos Malamas. 2022. Bridging the Gap Between Java and Python in Mobile Software Development to Enable MLOps. In *2022 18th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 363–368.
- [12] Rustem Dautov, Erik Johannes Husom, Sagar Sen, and Hui Song. 2023. Towards Community-Driven Generative AI. *Position Papers of the 18th Conference on Computer Science and Intelligence Systems, Annals of Computer Science and Information Systems* 36 (2023), 43–50.
- [13] Atis Elsts, Efstathios Mitskas, and George Oikonomou. 2018. Distributed ledger technology and the internet of things: a feasibility study. In *Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems*. Association for Computing Machinery, 7–12.
- [14] Greg N Frederickson and Nancy A Lynch. 1987. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)* 34, 1 (1987), 98–115.
- [15] István Hegedűs, Gábor Danner, and Márk Jelasity. 2019. Gossip learning as a decentralized alternative to federated learning. In *Proceedings of the 19th IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS 2019*. Springer, 74–90.
- [16] István Hegedűs, Gábor Danner, and Márk Jelasity. 2021. Decentralized learning works: An empirical comparison of gossip learning and federated learning. *J. Parallel and Distrib. Comput.* 148 (2021), 109–124.
- [17] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 12–21.
- [18] Chenghao Hu, Jingyan Jiang, and Zhi Wang. 2019. Decentralized federated learning: A segmented gossip approach. *arXiv preprint arXiv:1908.07782* (2019). https://doi.org/10.48550/arXiv.1908.07782
- [19] Erik Johannes Husom, Rustem Dautov, Adela Nedisan Videsjorden, Fotis Gonidis, Spyridon Papatzelos, and Nikolaos Malamas. 2022. Machine Learning for Fatigue Detection using Fitbit Fitness Trackers. In *Proceedings of the 10th International Conference on Sport Sciences Research and Technology Support (icSPORTS 2022)*. SciTePress, 41–52.
- [20] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning* 14, 1–2 (2021), 1–210.
- [21] Hyesung Kim, Jihong Park, Mehdi Bennis, and Seong-Lyun Kim. 2019. Blockchained on-device federated learning. *IEEE Communications Letters* 24, 6 (2019), 1279–1283.
- [22] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Master's Thesis. University of Toronto, ON, Canada.
- [23] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. 2020. A review of applications in federated learning. *Computers & Industrial Engineering* 149 (2020), 106854.
- [24] Yuan Liu, Zhengpeng Ai, Shuai Sun, Shuangfeng Zhang, Zelei Liu, and Han Yu. 2020. Fedcoin: A peer-to-peer payment system for federated learning. In *Federated learning: privacy and incentive*. Springer, 125–138.
- [25] Chuan Ma, Jun Li, Long Shi, Ming Ding, Taotao Wang, Zhu Han, and H Vincent Poor. 2022. When federated learning meets blockchain: A new distributed learning paradigm. *IEEE Computational Intelligence Magazine* 17, 3 (2022), 26–33.
- [26] Quazi Ehsanul Kabir Mamun, Salahuddin Mohammad Masum, and Mohammad Abdur Rahim Mustafa. 2004. Modified bully algorithm for electing coordinator in distributed systems. *WSEAS Transactions on Computers* 3, 4 (2004), 948–953.
- [27] Shojiro Muro, Tiko Kameda, and Toshimi Minoura. 1984. Multi-version concurrency control scheme for a database system. *J. Comput. System Sci.* 29, 2 (1984), 207–224.
- [28] Dinh C Nguyen, Ming Ding, Quoc-Viet Pham, Pubudu N Pathirana, Long Bao Le, Aruna Seneviratne, Jun Li, Dusit Niyato, and H Vincent Poor. 2021. Federated learning meets blockchain in edge computing: Opportunities and challenges. *IEEE Internet of Things Journal* 8, 16 (2021), 12806–12825.
- [29] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. USENIX Association, 305–319.
- [30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.
- [31] Oliver Lester Saldanha, Philip Quirke, Nicholas P West, Jacqueline A James, Maurice B Loughrey, Heike I Grabsch, Manuel Salto-Tellez, Elizabeth Alwers, Didem Cifci, Narmin Ghaffari Laleh, et al. 2022. Swarm learning for decentralized artificial intelligence in cancer histopathology. *Nature Medicine* 28, 6 (2022), 1232–1239.
- [32] Muhammad Shayan, Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. 2020. Biscotti: A blockchain system for private and secure federated learning. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1513–1525.
- [33] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–36.
- [34] Stefanie Warnat-Herresthal, Hartmut Schultze, Krishnaprasad Lingadahalli Shastry, Sathyanarayanan Manamohan, Saikat Mukherjee,

Vishesh Garg, Ravi Sarveswara, Kristian Händler, Peter Pickkers, N Ahmad Aziz, et al. 2021. Swarm learning for decentralized and confidential clinical machine learning. *Nature* 594, 7862 (2021), 265–270.
[35] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 1–19.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009