Leonardo Almeida

Framework Resiliente de Aprendizagem Federada

Resilient Federated Learning Framework

Universidade de Aveiro
2025

Leonardo Almeida

Framework Resiliente de Aprendizagem Federada

Resilient Federated Learning Framework

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor Mário Luís Pinto Antunes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro,
do Doutor Rui Luís Andrade Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Sérgio Guilherme Aleixo de Matos
Professor Associado C/ Agregação, Universidade de Aveiro

vogais / examiners committee

Doutor Angelo Corsaro
Ceo, Cto, Zettascale Technology

Prof. Doutor Mário Luís Pinto Antunes
Professor Auxiliar em Regime Laboral, Universidade de Aveiro

agradecimentos / acknowledgements

Agradeço a todos os que, com o seu apoio e acompanhamento, contribuíram para a concretização deste trabalho.

palavras-chave

Inteligência artificial, Aprendizagem Automática, Aprendizagem Federada, Resiliência, Modularidade, Protocolos de comunicação, Tolerância a falhas, Privacidade dos dados

resumo

Federated Learning (FL) oferece um paradigma promissor para a Machine Learning (ML) distribuído que preserva a privacidade, mas a sua aplicabilidade no mundo real é frequentemente dificultada pela necessidade de resiliência robusta e adaptabilidade em ambientes de rede dinâmicos, onde falhas de nós e conectividade mutável são comuns. Esta dissertação concebeu, implementou e avaliou o FlexFL, uma nova Framework Resiliente para FL, focada na modularidade e tolerância a falhas para abordar estes desafios. FlexFL apresenta uma arquitetura altamente modular, permitindo a integração flexível de backends de ML, algoritmos de FL e protocolos de comunicação, e incorpora mecanismos de resiliência como a gestão dinâmica do conjunto de trabalhadores e o reagendamento de tarefas. O desempenho e a robustez da estrutura foram rigorosamente avaliados em diversos cenários, incluindo falhas simuladas de trabalhadores em ambientes heterogéneos e em larga escala, utilizando os datasets UNSW-NB15 e ToN-IoT. De forma notável, num cenário exigente que envolveu 40 trabalhadores com uma taxa de falha probabilística de 1%, FlexFL demonstrou continuidade de treino bem-sucedida e convergência do modelo para ambos os datasets. Estes resultados validam a capacidade do FlexFL de gerir eficazmente a participação dinâmica dos trabalhadores e tolerar falhas substanciais nos nós, posicionando-o como uma solução robusta e escalável para implementações de FL no mundo real em ambientes de edge computing desafiadores.

keywords

Artificial Intelligence, Machine Learning, Federated Learning, Resilience, Modularity, Communication Protocols, Fault Tolerance, Data Privacy

abstract

Federated Learning (FL) offers a promising paradigm for privacy-preserving distributed Machine Learning (ML), yet its real-world applicability is often hindered by the need for robust resilience and adaptability in dynamic network environments, where node failures and changing connectivity are common. This dissertation designs, implements, and evaluates FlexFL, a novel Resilient Federated Learning Framework focusing on modularity and fault tolerance to address these challenges. FlexFL features a highly modular architecture, allowing flexible integration of ML backends, FL algorithms, and communication protocols, and incorporates resilience mechanisms such as dynamic worker pool management and task rescheduling. The framework's performance and robustness were rigorously evaluated across diverse scenarios, including simulated worker failures on large-scale heterogeneous environments using the UNSW-NB15 and ToN-IoT datasets. Notably, in a demanding scenario involving 40 workers with a 1% probabilistic failure rate, FlexFL demonstrated successful training continuity and model convergence for both datasets. These results validate FlexFL's capability to effectively manage dynamic worker participation and tolerate substantial node failures, positioning it as a robust and scalable solution for real-world FL deployments in challenging edge computing environments.

acknowledgement of use of AI tools

Recognition of the use of generative Artificial Intelligence technologies and tools, software and other support tools.

I acknowledge the use of Grammarly (https://app.grammarly.com/) and Gemini (https://aistudio.google.com/) to help and improve writing.

Contents

List of Figures

List of Tables

Glossary

ACCS	Australian Centre for Cyber Security
ACL	Access Control List
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
CK	Cohen's Kappa
CLI	Command Line Interface
CNN	Convolutional Neural Network
CoCoFL	Communication- and Computation-Aware Federated Learning
DDoS	Distributed Denial of Service
DL	Deep Learning
DoS	Denial of Service
FL	Federated Learning
FN	False Negative
FP	False Positive
GDPR	General Data Protection Regulation
HeteroFL	Heterogeneous Federated Learning
HPC	High-Performance Computing
IDS	Intrusion Detection System
IID	Independent and Identically Distributed
IoT	Internet of Things
JSONL	JavaScript Object Notation Lines
MAE	Mean Absolute Error
MCC	Matthews Correlation Coefficient
MITM	Man In The Middle
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MPI	Message Passing Interface
MQTT	Message Queuing Telemetry Transport
MSE	Mean Squared Error
mTLS	Mutual Transport Layer Security
NN	Neural Network
Non-IID	Non-Independent and Identically Distributed
NTP	Network Time Protocol
PI	Permutation Importance
PyPI	Python Package Index
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SAE	Sparse AutoEncoder
SMAPE	Symmetric Mean Absolute Percentage Error
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TN	True Negative
TP	True Positive
VM	Virtual Machine
XAI	Explainable Artificial Intelligence
XSS	Cross-Site Scripting

CHAPTER 1

Introduction

This chapter introduces the growing relevance of Artificial Intelligence and Federated Learning, highlighting the core challenges addressed in this research, specifically concerning resilience and modularity in dynamic, distributed environments.

1.1 Context

Artificial Intelligence (AI) is becoming increasingly used in our daily lives, revolutionizing industries ranging from healthcare and finance to transportation and education [1]. This widespread adoption highlights AI's potential to address complex problems, automate processes, and drive innovation. Yet, training ML models has significant challenges, especially when dealing with sensitive or distributed data [2].

Federated Learning (FL) offers a promising approach to resolving these challenges by leveraging the power of distributed systems while promoting collaboration, where entities do not share their data. This paradigm addresses data security and regulatory compliance concerns, such as the General Data Protection Regulation (GDPR) and AI Act [3], which is increasingly crucial in the era of big data and AI.

However, the journey towards leveraging FL benefits has technical challenges that involve ensuring the system's robustness and resilience in real-world scenarios. The concept of resilience in FL for this work refers to the ability of the system to maintain its training and performance, even in the presence of node failures, delays, and other adversities.

The following sections delve into the motivation for this research, the objectives it aims to achieve, contributions, and the overall structure of this dissertation.

1.2 Motivation

Despite its significant potential, the success of FL depends heavily on its ability to operate efficiently within dynamic and heterogeneous network environments, where devices may have connectivity issues and can join or leave the network at any time. This relies on the robustness

and resilience of the system that is given by the underlying communication infrastructure, such as 5G and 6G, where FL is a strong candidate for distributed learning in edge computing environments [4].

However, other technical challenges must be addressed, such as Non-Independent and Identically Distributed (Non-IID) data distribution [5], communication overhead, and achieving a reliable model aggregation [6]. This volatility can significantly impact the performance of the FL system, degrading the quality of the models, so it is crucial to address these challenges to make FL a practical solution for real-world applications.

While a substantial body of research explores the comparative performance, privacy benefits, and fundamental differences between FL and traditional centralized ML approaches [7], this dissertation explicitly addresses the technical challenges inherent within the FL training paradigm itself.

Existing FL frameworks mentioned in Chapter 2, while innovative, may address some of these challenges, but they often lack the flexibility to adapt to the dynamic nature of the network, and they lack modularity. This modularity is important to allow the integration of different components, such as communication protocols, model aggregation strategies, and optimization algorithms, to create a system that follows different requirements and constraints, but also to allow the integration of existing systems in real-world applications.

To this end, there is a compelling need for a resilient and modular FL framework designed to adapt dynamically to changing network conditions and to seamlessly handle the addition and removal of participating devices, ensuring uninterrupted training processes and improving the robustness and scalability of FL systems [8]. Furthermore, this framework would significantly expand the applicability of FL, making it suitable for diverse real-world environments ranging from stable enterprise networks to challenging edge computing contexts.

1.3 Objectives

The primary objective of this research is to design, implement, and evaluate a Resilient Federated Learning Framework with a focus on modularity. This framework aims to address key technical and practical challenges inherent in deploying FL in dynamic and unpredictable environments, particularly ensuring stable and uninterrupted operation in the face of node failures and delays.

Driven by the potential for such resilience to enable the reliable deployment of FL in real-world applications, this research is motivated by its potential contributions to environmental sustainability, by preventing wasted computational and communication energy associated with training disruptions and failures, and by allowing the use of existing devices in edge computing environments, which can reduce the need for additional infrastructure. Furthermore, it aims to contribute to the broader societal good by enabling reliable, privacy-preserving solutions in critical sectors.

The expected outcomes of this research are:

- **Open Source Framework:** Develop a modular, resilient, and open-source FL framework with a Massachusetts Institute of Technology (MIT) licence, that can be easily extended and integrated with existing systems.
- **Scientific Publication:** Write and publish a research paper in a conference or journal, presenting the framework and its evaluation, to contribute to the research community with valuable insights and knowledge about FL.
- **Dissertation Document:** This publicly available document will provide a comprehensive overview of the research, analyzing the state-of-the-art, stating the design choices, including the architecture and implementation details, and a comprehensive evaluation of the framework, by showcasing its performance in multiple scenarios.

1.4 Contributions

The primary contributions of this dissertation include both the development of a robust framework and the dissemination of new knowledge through academic publications and collaborative projects. This research has led to the design, implementation, and rigorous evaluation of *FlexFL*, a novel open-source framework specifically engineered for resilient FL.

FlexFL is distinguished by its highly modular architecture, which allows for flexible integration of various ML backends, FL algorithms, and communication protocols. It is designed to adapt dynamically to network conditions, effectively handle node failures, and ensure continuous training, thereby addressing critical challenges inherent in real-world FL deployments. The framework's codebase is publicly available on GitHub [1] and its package can be installed via PyPI [2].

This dissertation document itself serves as a comprehensive record of the research done, detailing the problem statement, systematically reviewing the state-of-the-art, elaborating on the proposed modular architecture, describing the implementation specifics (including resilience mechanisms and optimizations), and rigorously evaluating the framework's performance and robustness across various experimental scenarios.

Furthermore, my research engagement extends to participation in a significant project funded by a prestigious research grant. Awarded by the University of Aveiro and Instituto de Telecomunicações, this grant is part of the broader Agenda Mobilizadora para a Inovação Empresarial, in collaboration with NEXUS - Pacto de Inovação - Transição Verde e Digital para Transportes, Logística e Mobilidade. This work specifically focuses on the application of Machine Learning in SmartPort environments, particularly at the Port of Sines, encompassing the development of resilient communication layers for FL, thereby further contextualizing and extending the practical relevance of the topics explored.

Beyond individual research, a significant contribution involved the guidance and supervision of a student group's final year project. The project's objective was to develop a graphical interface to manage the department's current implementation of FL. It provides the means to control the training and visualize its progress. This is directly aligned with this dissertation.

[1] `https://github.com/leoalmPT/FlexFL`
[2] `https://pypi.org/project/flexfl/`

1.4.1 Scientific Publications

Furthermore, the insights and results derived from this research have been disseminated through several academic publications, thereby contributing to the broader scientific community. This includes five published papers, three accepted papers, and one additional papers currently under review as shown in Table 1.1.

While not all of these contributions focus exclusively on Federated Learning, they collectively demonstrate a comprehensive engagement with critical challenges and opportunities in AI and ML, particularly concerning distributed systems and resource-constrained environments, directly contributing to the comprehensive understanding and innovative design underpinning this dissertation.

Table 1.1: Overview of 9 scientific publications, categorized by their status: 5 published, 3 accepted, and 1 currently under review

Status	Title
Published	**1 -** Privacy-Preserving Defense: Intrusion Detection in IoT using Federated Learning [9]
	2 - Shallow vs. Deep Learning: Prioritizing Efficiency in Next Generation Networks [10]
	3 - Efficient training: Federated learning cost analysis [7]
	4 - AIDetx: A Compression-Based Method for Identification of Machine-Learning Generated Text [11]
	5 - Federated Learning for Dynamic Edge: A Modular and Resilient Approach [12]
Accepted	**6 -** From Black Box to Transparency: Consistency and Cost within XAI [13]
	7 - Resilient Federated Learning Framework for 6G [14]
	8 - Optimised Task Placement for MLOps [15]
Under Review	**9 -** Understanding What Federated Learning Models Learn: A Comparative Study with Traditional Models [16]

In [9] we address the critical challenge of securing Internet of Things (IoT) networks, particularly concerning data privacy for Intrusion Detection System (IDS), by exploring the efficacy of FL as a privacy-preserving approach for training robust IDS models. We evaluated the FL-based training of Neural Network models, comparing its performance against traditional centralized training methods across various numbers of workers. Our experimental results demonstrate that FL-trained IDS models achieve comparable detection performance to their centrally trained counterparts while exhibiting significantly faster convergence times.

[10] presents the computational challenges of deploying ML models in Next Generation Networks (5G/B5G), particularly for real-time applications such as network slicing attribution and IDS. Our study conducts a comprehensive comparative analysis between shallow ML models and state-of-the-art Deep Learning (DL) models across these key tasks. Our

experimental results demonstrate that shallow ML models achieve comparable performance to their DL counterparts while exhibiting significantly faster training and prediction times, often leading to over 90% acceleration.

The challenges of deploying AI and ML models in Next Generation Networks (6G), particularly concerning data privacy and resource constraints are presented in [7]. Where we investigate FL as a distributed and privacy-preserving solution for training these models. Our work specifically analyzes the performance and costs of various FL approaches, including training time, communication overhead, and energy consumption. Through experiments, we demonstrate that FL can significantly accelerate the training process and reduce data transfer, but its overall effectiveness is highly dependent on the chosen FL paradigm and underlying network conditions, underscoring that FL is not a one-size-fits-all solution.

[11] introduces AIDetx, a novel compression-based method for the identification of ML generated text, aiming to overcome the limitations of traditional deep learning classifiers such as high computational costs and interpretability issues. AIDetx leverages finite-context models (FCMs) to construct distinct compression models for human-written and AI-generated content. New text inputs are then classified based on which model yields a higher compression ratio, indicating a better statistical fit. This approach offers a highly interpretable and computationally efficient solution, significantly reducing training time and hardware requirements (e.g., eliminating the need for GPUs).

We propose a recent version of the FlexFL framework in [12], designed to address significant challenges, such as fault tolerance, elasticity, and communication efficiency, in dynamic, resource-constrained edge environments, including 5G/6G networks and IoT deployments. Our framework's architecture is built on decoupled modules that handle core FL functionalities, offering flexibility in integrating various algorithms, communication protocols, and resilience strategies. Through experimental evaluations, we demonstrate the framework's ability to seamlessly integrate three different communication protocols and three FL paradigms and show that protocol choice significantly impacts performance, with Zenoh emerging as the most efficient option due to its lower overhead compared to Kafka and Message Queuing Telemetry Transport (MQTT) in high-volume communication scenarios. Furthermore, the framework successfully maintains model training and achieves convergence even under simulated probabilistic worker failures.

In [13], we address the critical need for Explainable Artificial Intelligence (XAI) in 5G and 6G networks, driven by the increasing reliance on complex ML models for network operations and the regulatory demand for transparency. Our study investigates the consistency of feature importance explanations across three state-of-the-art XAI techniques (SHAP, LIME, and Permutation Importance (PI)) when applied to various ML models in five distinct 5G network scenarios, while also quantifing the temporal and energy costs associated with employing these XAI methods. Our findings demonstrate that while PI is the most cost-efficient XAI technique, there can be significant disagreements among XAI methods regarding feature relevance, even for highly accurate ML models.

[14] proposes an early version of the FlexFL framework designed to overcome key challenges

in dynamic and heterogeneous 6G and large-scale IoT environments. We address the critical limitations of traditional FL implementations, including communication costs, node failures, and scalability issues. Our framework leverages Zenoh as a communication protocol to enhance fault tolerance and efficiency. Through simulated experiments on the UNSW-NB15 dataset, we demonstrate that our proposed framework successfully handles simulated node failures, reduces communication overhead compared to Message Passing Interface (MPI), and maintains high model accuracy and convergence.

[15] presents a novel task placement system designed to optimize the execution of ML pipelines in heterogeneous computing environments. Our system employs a two-phase strategy: pipeline scheduling using Shortest Job First and task placement using a heuristic-based method that considers task type, data characteristics, ML model type, and node load. This system is integrated with Kubeflow for orchestration and manages the entire ML lifecycle from data preprocessing to model evaluation. Through experimental evaluations using various ML models and datasets, the proposed system significantly outperforms baseline methods and the Kubernetes default scheduler, achieving substantial reductions in total execution time (up to 68%) and average waiting time (over 80%).

Lastly, in [16], we investigate the impact of FL on the explainability of ML models. Addressing the need for model reliability and trust, we leverage XAI metrics to assess how FL affects the underlying patterns learned by ML models, by evaluating the correlation between XAI outputs from models trained using four distinct FL approaches and a traditional single-host method. Our analysis spans four publicly available datasets, employing two XAI metrics and two correlation metrics. The findings reveal a high correlation in XAI outputs for three of the four FL approaches, indicating that FL models generally learn similar patterns to their centralized counterparts. However, in some instances, even with comparable performance, FL models appear to learn different underlying patterns, underscoring the nuanced relationship between distributed training and model interpretability.

1.5 Document Outline

This dissertation document is structured as follows: Chapter 2 delves into the necessary background, covering fundamental concepts such as centralized, distributed and federated learning. Also explores relevant communication protocols and related work by presenting a systematic literature review alongside a comparative analysis of existing frameworks.

Chapter 3 details the proposed solution by discussing system requirements, presenting the architectural design, and showing a SWOT analysis. Chapter 4 describes the implementation details of the framework, including the software stack, the adaptation of FL algorithms for resilience, and additional features.

Chapter 5 presents the evaluation methodology and results, outlining the experimental setup, datasets, models, metrics, and the analysis of various scenarios demonstrating the framework's performance and resilience. Finally, Chapter 6 concludes the dissertation by summarizing the contributions, discussing the limitations of the work, and proposing directions for future research.

CHAPTER 2

Background and Related Work

Before diving into the work done in Resilient FL, it is essential to understand concepts related to ML, going from centralized to distributed ML, and understand how FL fits in the distributed ML landscape. Given the distributed nature of FL, it is also essential to analyze the communication protocols that can be used in this context.
This aims to show that FL is a compelling solution for training ML models in a collaborative manner, while preserving data privacy. However, its success relies on the robustness of the learning and communication layers, especially in heterogeneous environments, where devices have different capabilities and can enter or leave the network anytime.
To address these challenges, this chapter transitions to the state-of-the-art in FL research, focusing on resilient approaches and other FL frameworks that attempt to address the challenges of dynamic networks and unreliable devices, while highlighting their main strengths, limitations, and contributions. These insights are essential for developing the proposed solution, as they provide a solid foundation to identify the gaps and requirements.

2.1 Centralized Machine Learning

Centralized ML is the most widely used approach in the field of AI, where a single entity collects all the data, processes it and trains a ML model [17]. This data can be collected from multiple sources, including sensors, databases, and other devices.

The ML process can be divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning [18], each with its own characteristics and applications as shown in Figure 2.1.

Supervised learning is used when the data is labeled, and the goal is to predict the output based on the input data. This type of learning is used in classification problems, where the output is a category, and regression problems, where the output is a continuous value.

Unsupervised learning is used when the data is not labeled and the goal is to find patterns in it. This is used in clustering problems, where the goal is to group similar data points together, association rule discovery problems, where the goal is to find relationships between

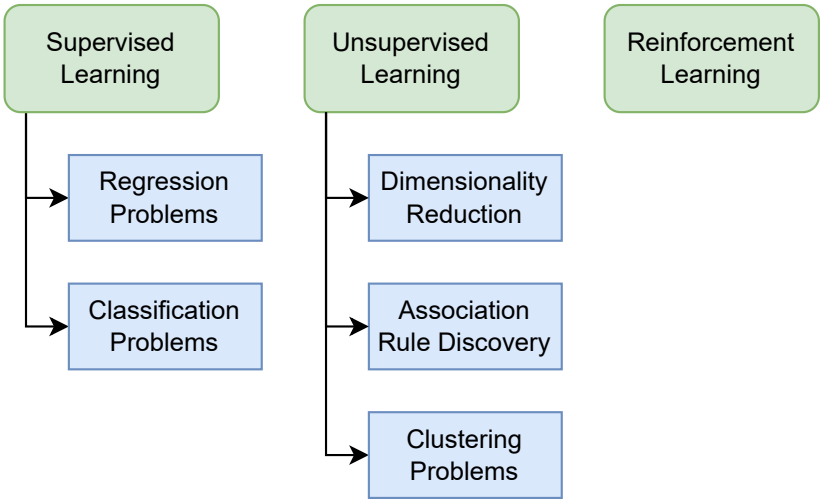


Figure 2.1: Comprehensive taxonomy of ML, detailing its three fundamental learning paradigms: Supervised Learning, Unsupervised Learning, and Reinforcement Learning, and their associated problem domains.

variables, and dimensionality reduction problems, where the goal is to reduce the number of variables in the data.

Finally, reinforcement learning is used when the agent learns to interact with the environment by taking actions and receiving rewards. The goal is to learn a policy that maximizes the cumulative reward.

Given that the data is centralized, the training process can be done on a single machine, without the need for communication between different entities. This makes the training process easier to implement, monitor, and resource-efficient. However, it also has some drawbacks, such as privacy concerns when the data is sensitive or confidential, a single point of failure, and scalability issues when the data or the model is too large to be computed by a single machine.

2.1.1 Neural Networks

Neural Network (NN) represent a fundamental and powerful class of models in ML, particularly central to the field of deep learning. While centralized ML can employ various model types, NNs are frequently used due to their ability to learn complex patterns.

A NN is structured in layers, typically including an input layer, one or more hidden layers, and an output layer. Each layer consists of interconnected nodes (or neurons), with connections having associated weights and biases. During training, data passes through the NN (forward pass), and calculations are performed at each neuron using activation functions, resulting in a prediction. The difference between the prediction and the actual target is quantified by a loss function.

To minimize this loss, the NN is trained using algorithms like backpropagation [19], which calculates the gradients of the loss function concerning the NN's weights and biases. These gradients indicate the direction and magnitude of the change needed for each weight and bias to reduce the loss. Various optimization methods (e.g., Gradient Descent, Adam [20])

then use these gradients to update the weights and biases, iteratively improving the model's performance.

A simple representation of an NN structure is shown in Figure 2.2. The input layer receives the data, the hidden layers process it, and the output layer produces the final prediction. Each connection between neurons has a weight that determines its influence on the output.

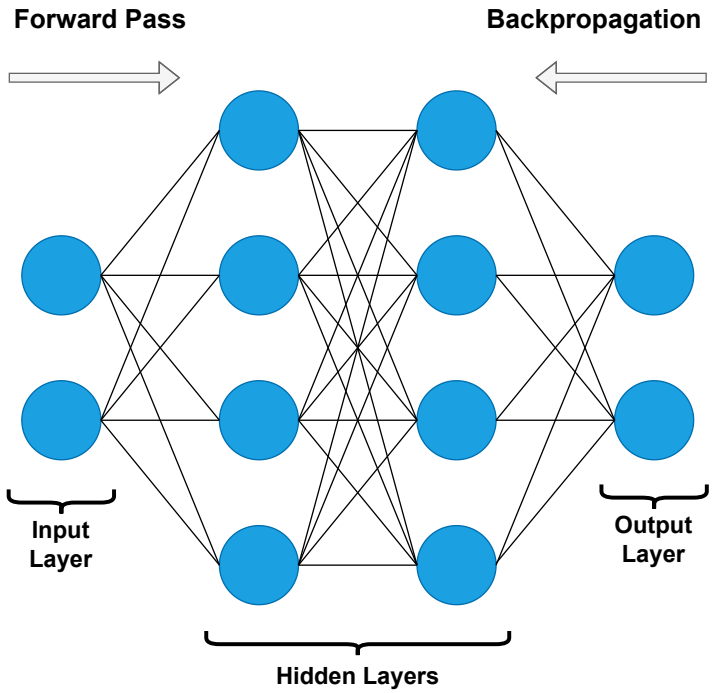


Figure 2.2: Structure of a typical Neural Network, detailing its Input, Hidden, and Output Layers. The diagram highlights the directional flow of computation during the Forward Pass and the gradient-based optimization process of Backpropagation.

In the context of distributed and FL, the model being trained is often a NN. It is crucial to distinguish this model network structure from the communication network of interconnected devices (workers and servers) participating in the distributed learning process. For the remainder of this dissertation, we use the term Model to refer specifically to a NN, and Network to denote a communication network, unless otherwise stated.

2.2 Distributed Machine Learning

Distributed ML is an approach where the data or model is distributed across multiple machines and the training process is done in a distributed manner that, under certain conditions, enables faster training and allows larger models and datasets that can't be processed by a single machine [9], [21].

The distributed training process can follow different architectures and techniques. In [22], the authors present a taxonomy that classifies distributed ML into three categories: Model vs Data Parallelism, Centralized vs Decentralized Optimization, and Synchronous vs Asynchronous Scheduling.

2.2.1 Model vs Data Parallelism

Model and Data Parallelism are two different strategies for distributing large workloads, one maps the model to multiple machines, and the other maps the data to multiple machines.

Model parallelism involves dividing a ML model into partitions to be processed by different machines, and it is particularly beneficial when the model is too large to fit in memory. This partitioning can be done in two ways: vertical partitioning, which splits the model between layers, or horizontal partitioning, which divides individual layers as shown in Figure 2.3.

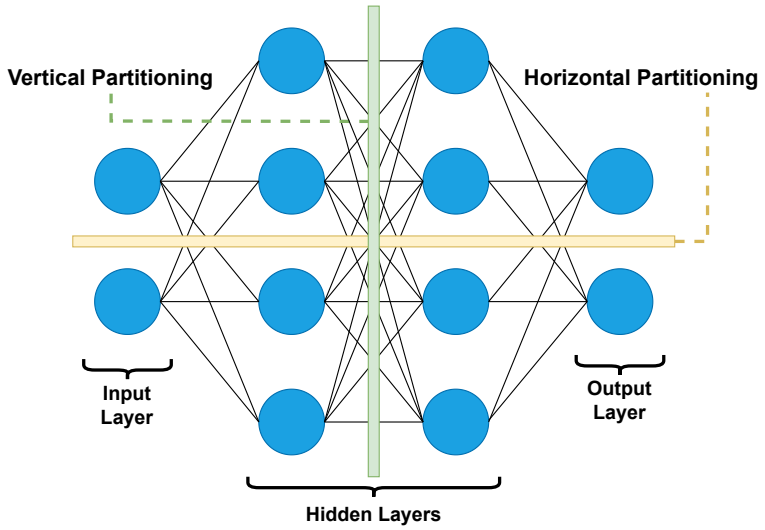


Figure 2.3: Illustration of Model Parallelism, distinguishing between Vertical Partitioning, where the model is split across layers, and Horizontal Partitioning, where individual layers are divided across multiple machines.

Vertical partitioning is simpler to implement since it requires transporting intermediate outputs (activations) between machines. On the other hand, horizontal partitioning, which splits the layers themselves, is more complex because it requires managing numerous inter-machine connections.

In data parallelism, multiple machines process different subsets of the data using the same model architecture. This approach aggregates gradients or weights from various machines to update the model parameters, facilitating parallelized computation. The communication overhead and scalability depend on the model size and the efficiency of the aggregation mechanism. Data parallelism is often easier to implement and scale than model parallelism, allows data privacy when combined with privacy-preserving techniques, and is widely adopted in distributed systems.

2.2.2 Centralized vs Decentralized Optimization

In centralized optimization, a central server (parameter server) aggregates gradients computed by distributed workers and updates the global model parameters. While this architecture simplifies the synchronization and coordination of workers, it can become a bottleneck in large clusters due to communication overhead and reliance on a single server

for updates, because the gradients are sent to the parameter server after each batch, and the updated weights are sent back to the workers. Advanced implementations mitigate these issues by distributing the parameter server role across multiple nodes. Figure 2.4 shows the centralized optimization process.

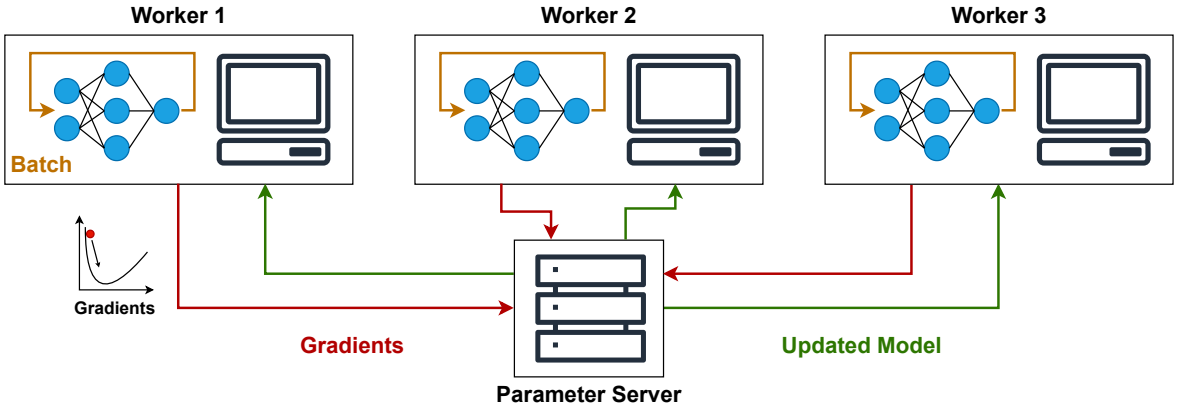


Figure 2.4: Diagram illustrating the Centralized Optimization process, where multiple Workers compute gradients and send them to a single Parameter Server, which then aggregates these gradients to update the global model and sends the Updated Model back to the workers.

Decentralized optimization involves each worker training its local model independently, periodically synchronizing with peers or a master node (parameter server) by sending the weights. This method can reduce dependency on a central server, enabling higher scalability and tolerance to communication delays. However, ensuring convergence and minimizing inconsistencies among workers requires careful tuning of synchronization strategies and hyperparameters. Figure 2.5 shows the decentralized optimization process.

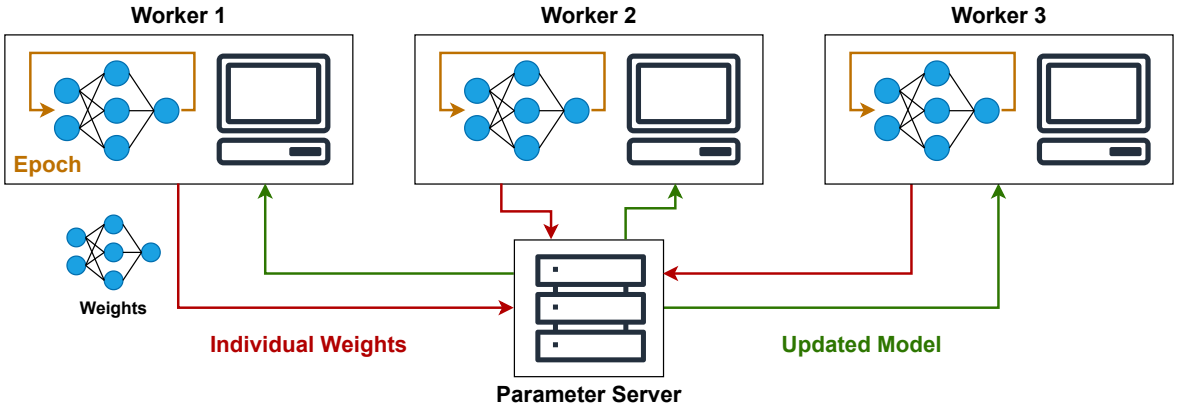


Figure 2.5: Diagram illustrating the decentralized optimization process, where workers train their local models independently and periodically synchronize with a parameter server by exchanging individual weights and receiving updated models.

These two approaches can then be used together or individually in a hierarchical manner [23], where the workers are divided into groups, each group has a master node, and the parameter server aggregates the gradients or weights from the master nodes. This approach

can reduce the communication overhead and improve the system's scalability.

2.2.3 Synchronous vs Asynchronous Scheduling

In synchronous systems, workers operate in lockstep, aggregating all updates before proceeding to the next training iteration. This approach minimizes inconsistencies among workers and is easier to implement and debug, but it can lead to resource underutilization due to waiting for slower nodes (stragglers). Model aggregation in synchronous systems typically uses weighted averaging, where each worker's update is scaled based on factors such as data size or importance before being combined into a global model.

The weighted averaging is defined as follows:

$$\theta_{t+1} = \frac{\sum_{i=1}^{N} w_i \cdot \theta_i}{\sum_{i=1}^{N} w_i} \tag{2.1}$$

where θ_{t+1} is the updated model parameters, N is the number of workers, w_i is the weight assigned to worker i, and θ_i is the model parameters from worker i. The weights are typically proportional to each worker's data size or reliability. The normalization ensures that the update is a true weighted average, maintaining scale consistency and promoting a balanced aggregation.

Asynchronous systems allow workers to proceed with their computations independently, updating the model parameters as they complete their tasks. While this maximizes resource utilization and can be more scalable, it introduces challenges such as stale updates, which can slow convergence or degrade model performance. To mitigate this, linear interpolation is often used for model aggregation, blending new updates with the current model state to maintain stability. These complexities make asynchronous systems harder to implement and debug, but potentially more efficient in heterogeneous environments.

The linear interpolation is defined as follows:

$$\theta_{t+1} = \theta_t + \alpha \cdot (\theta_i - \theta_t) \tag{2.2}$$

where θ_{t+1} is the updated model parameters, θ_t is the current model parameters, α is the interpolation factor ($0 < \alpha < 1$), and θ_i is the model parameters from worker i. This allows the system to blend new updates with the current model state, maintaining stability while incorporating fresh information.

2.3 Federated Learning

FL is a paradigm of distributed ML, which can differ from other distributed ML approaches in three ways [24]. First, the data cannot be shared between the devices, ensuring privacy, security, and ownership. This way, laws and regulations such as the GDPR are easier to comply with. Second, FL is designed to work in a distributed manner, while maximizing resource utilization, allowing organizations to collaborate without sharing data. Finally, FL can have security mechanisms such as encryption to further protect the data.

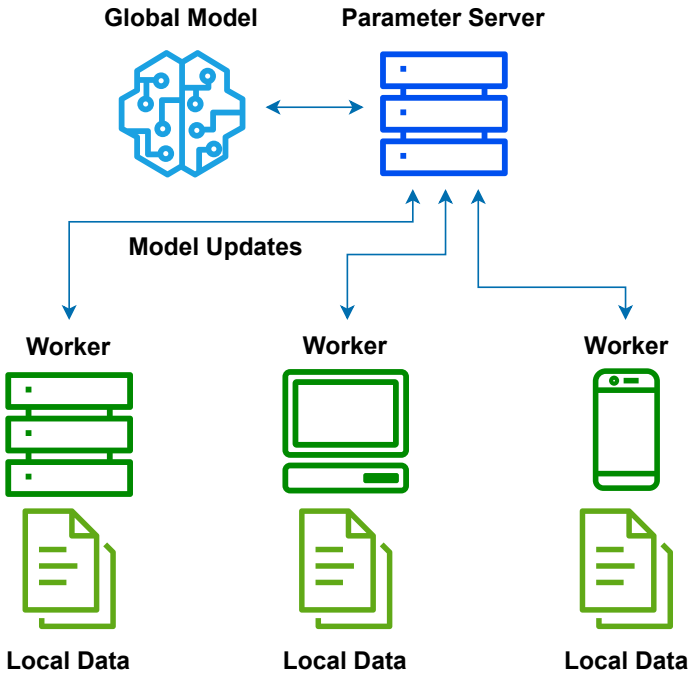


Figure 2.6: The core process of Federated Learning, illustrating how a central Parameter Server coordinates model training across multiple Workers, where each Data remains stored locally, ensuring privacy and security by only exchanging Model Updates for a Global Model.

The core process of FL, where training is coordinated across multiple devices while keeping data local, is illustrated in Figure 2.6.

FL can be divided into three main categories: horizontal FL, vertical FL and federated transfer learning [25]. Horizontal FL is applicable when the datasets of participating entities share the same feature space but have different samples [26]. Vertical FL is used when the datasets share the same samples but have different features [27]. Federated transfer learning is used when the datasets have different feature spaces and samples, and the goal is to transfer knowledge from one dataset to another [28].

Most of the FL use cases are related to mobile devices, industrial engineering, and healthcare, where the data is sensitive and cannot be shared, or the centralized approach is not feasible due to the large amount of data [29]. When these restrictions do not apply, the other approaches might be more suitable.

2.4 Communication Protocols

FL relies on efficient communication protocols to manage distributed training, ensuring privacy, scalability, and minimal overhead. The protocol choice depends on the system requirements, such as the number of devices, network conditions, and security constraints. This section evaluates the scalability, fault tolerance, security, and suitability for FL applications of four prominent communication protocols: MPI [30], MQTT [31], Kafka [32], and Zenoh [33], while highlighting their key features and limitations.

2.4.1 MPI

MPI is a widely used communication protocol in High-Performance Computing (HPC) environments, providing low-latency, high-throughput communication between nodes, because it is designed to handle the message-passing paradigm with explicit exchange of data by send and receive operations.

However, MPI relies on static configurations, requiring predefined communication endpoints and also lacks built-in support for fault tolerance, making it unsuitable for handling worker disconnections or failures, which are common in real-world FL systems.

While its synchronous communication model ensures consistency, it introduces constraints that hinder scalability in flexible, decentralized training scenarios. Additionally, MPI does not provide native support for encryption or authentication, relying instead on secure underlying transport layers like Secure Shell (SSH) or network-level security configurations.

Figure 2.7 illustrates the basic communication flow in an MPI system, where processes exchange messages explicitly.

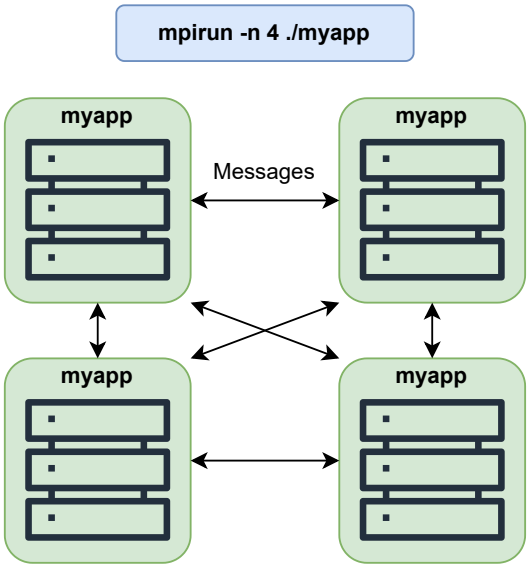


Figure 2.7: The communication flow within an MPI system, showing four interconnected application processes. This setup relies on a static configuration, where the number of workers is fixed and new participants cannot dynamically join, with all processes typically executing the same program.

2.4.2 MQTT

MQTT is a lightweight publish-subscribe protocol designed for resource-constrained devices and low-bandwidth networks. It employs a centralized broker to mediate client communication, allowing devices to dynamically join or leave the system without disrupting operations, making it appealing for FL use cases.

It also offers a native way to notify clients when other clients disconnect, which is helpful for FL systems where worker availability may fluctuate. Despite its advantages for edge deployments, this centralized architecture can become a bottleneck in large-scale deployments, particularly as the volume of exchanged model updates increases.

MQTT supports Transport Layer Security (TLS) and Mutual Transport Layer Security (mTLS) for secure communication on top of Transmission Control Protocol (TCP), with client authentication and role-based access control. However, the dependence on a broker limits scalability compared to fully decentralized protocols.

Figure 2.8 illustrates the communication flow in an MQTT system, where clients publish messages to topics and subscribe to receive updates.

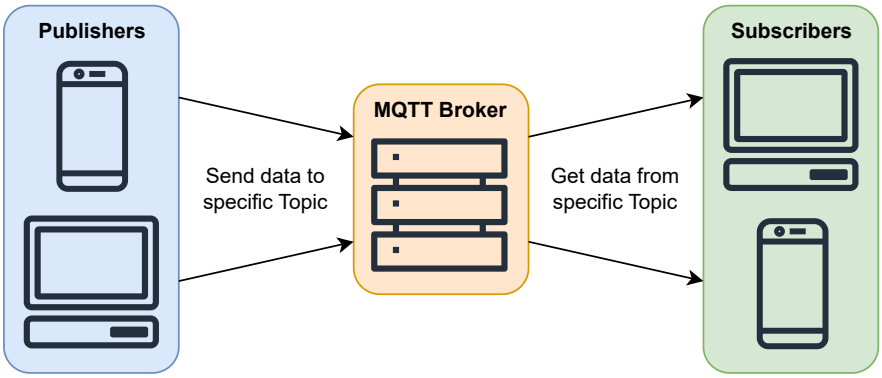


Figure 2.8: Illustration of the MQTT communication flow, showcasing the publish-subscribe model. Publishers send data to a central MQTT Broker, which then mediates the distribution of messages to relevant Subscribers based on specific topics.

2.4.3 Kafka

Kafka is a distributed streaming platform optimized for real-time data processing. Its publish-subscribe model supports dynamic and large-scale systems, offering persistent storage and robust fault tolerance through distributed broker clusters.

However, Kafka's suitability for FL is hindered by the latency introduced through its brokered communication model, where the additional communication hops required for routing messages through brokers can slow down the synchronization of model updates, which is critical in iterative training processes.

The only native disconnection notification mechanism is inspection of connected groups. To participate in a group, clients must subscribe with the same group ID, but this approach adds considerable overhead because when a client joins or leaves, partitions must be rebalanced, which can be time-consuming and resource-intensive.

Similarly to MQTT, Kafka supports TLS, mTLS, client authentication, and role-based access control for secure communication.

Figure 2.9 illustrates the communication flow in a Kafka system, where producers publish messages to topics and consumers subscribe to receive updates.

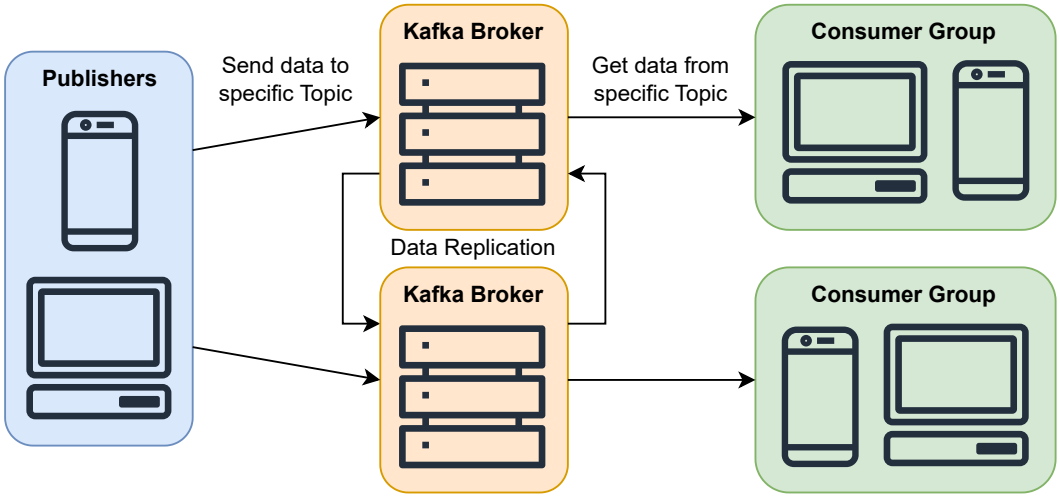


Figure 2.9: Illustration of the Kafka communication flow, detailing how Publishers send data to topics managed by a distributed Kafka Broker cluster. This system leverages data replication for persistence and fault tolerance, and messages are consumed by organized Consumer Groups from specific topics.

2.4.4 Zenoh

Zenoh is a decentralized communication middleware specifically designed for dynamic and resource-constrained environments. Unlike broker-based protocols, Zenoh adopts a fully decentralized architecture that eliminates single points of failure and enables seamless communication among devices, even in unreliable network conditions.

Similar to MQTT, it offers a native way that allows participants to be notified when other participants leave the network without needing a centralized broker.

It supports multiple communication methods, including TCP, TLS and mTLS, allowing devices to adapt to varying security and performance requirements. Zenoh also supports the use of Access Control Lists (ACLs), enabling fine-grained permission management to control which entities can publish or subscribe to specific data. Despite its advantages, Zenoh is a relatively new protocol with a less mature ecosystem compared to established solutions such as the ones mentioned above, which may limit its adoption in existing FL systems.

Figure 2.10 illustrates the communication flow in a Zenoh system, where devices communicate directly without a central broker and can join or leave the network dynamically.

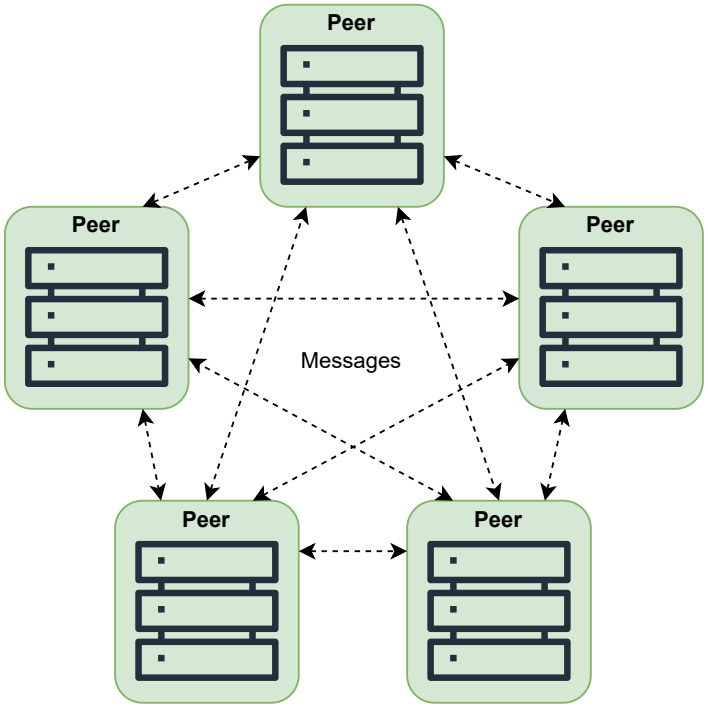


Figure 2.10: Illustration of the Zenoh communication flow, highlighting its fully decentralized peer-to-peer architecture. This design allows individual Peers to communicate directly and dynamically join or leave the network at any moment without requiring a central broker. Adapted from Zenoh Documentation.

2.4.5 Summary of Communication Protocols

Communication protocols are critical to enable efficient, secure, and scalable FL systems. Each protocol discussed offers unique strengths and limitations, making them suitable for different scenarios. The choice of protocol should be based on the specific requirements of the FL system, such as the number of devices, network conditions, and security constraints, to ensure optimal performance and reliability.

Table 2.1 summarizes the communication protocols' main features, highlighting their suitability for our scenario.

In our case, a centralized server (parameter server) coordinates the training process, and the system is designed to be resilient to worker disconnections. This means that a centralized communication broker such as MQTT is not a problem, and the system can benefit from its lightweight nature. However, MPI is unsuitable for our system because it is designed for static configurations.

Table 2.1: A comparative analysis of communication protocols, including MPI, MQTT, Kafka, and Zenoh, detailing their characteristics in terms of scalability, fault tolerance, security, and general suitability for FL deployments.

Protocol	Scalability	Fault Tolerance	Security	Suitability
MPI	Limited	No built-in support	Relies on SSH	Low, lacks fault tolerance and scalability
MQTT	Moderate	Moderate	TLS, mTLS role-based	High, suitable when using a central broker
Kafka	High	High	TLS, mTLS role-based	Moderate, latency can hinder synchronization
Zenoh	High	High	TLS, mTLS ACL	High, but limited ecosystem maturity

2.5 Systematic Literature Review

The state-of-the-art in FL is vast and diverse; however, studies focusing on the type of resilience for this work are harder to find. To address this challenge, a systematic literature review was conducted.

The first step in the review process was to define the research questions that will guide the search for relevant literature. The following questions were defined:

- **RQ1:** How can Federated Learning frameworks be designed to ensure robustness against node failures across dynamic network environments?
- **RQ2:** What communication layer architectures are most suitable for supporting fault-tolerant and resilient Federated Learning under dynamic network conditions?

The first research question aims to understand the strategies and mechanisms that can be used to improve the robustness of FL frameworks, while the second question focuses on the communication layer, which is a critical component for the success of FL in dynamic networks. These questions aim to identify keywords and concepts that can be used to search for relevant literature.

The next step was to define the queries that would be used to search for relevant literature in databases, specifically SCOPUS. To narrow the search and improve the quality of the results, only records from journals and conferences published in the last five years were considered.

Keywords such as ”Federated Learning”, ”Resilience”, ”Communication” and ”Fault Tolerance” emerge from the research questions, but some challenges remain to be addressed.

For instance, the term ”Resilience” in FL is primarily used in the context of attacks, data poisoning, and to ensure that the ML models converge. With the term ”Communication”, most of the results are related to papers focusing on communication costs, communication efficiency, or just an analysis of the communication layer, not including fault tolerance to node failures or network delays.

Therefore, the query with the most relevant results was: TITLE-ABS-KEY (”federated learning”) AND TITLE-ABS-KEY (”fault tole*”).

The review process, as shown in Figure 2.11, started with retrieving 121 records from SCOPUS using the previously defined query and filters. This step was conducted on October 31, 2024.

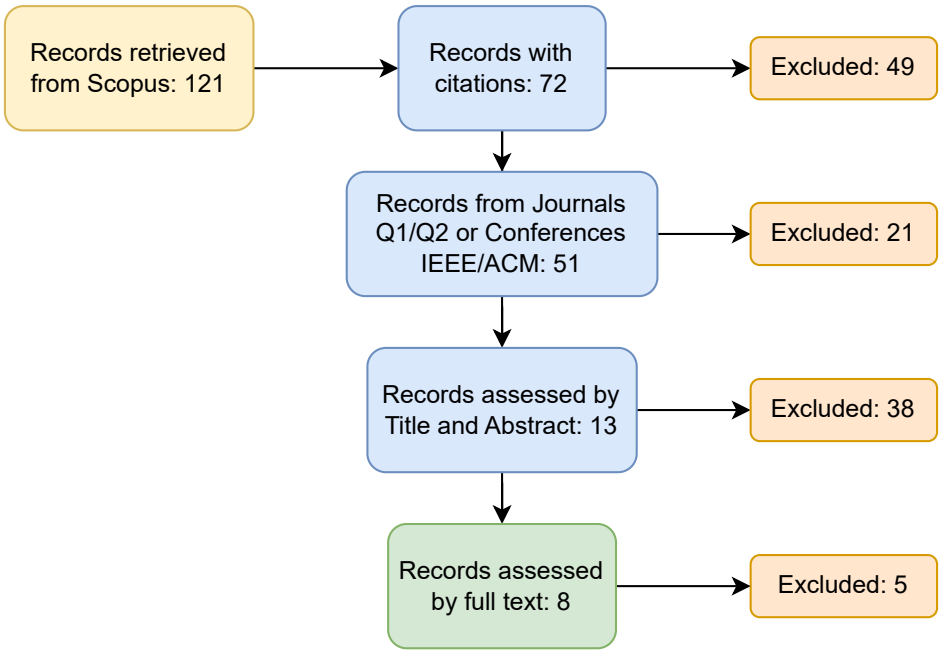


Figure 2.11: Detailed flow of the systematic literature review process, beginning with 121 retrieved records from SCOPUS. The diagram illustrates five sequential exclusion steps that systematically filtered irrelevant studies, culminating in a final set of 8 relevant records for in-depth analysis.

Studies without citations were excluded from these records to focus on work that has had an observable impact on the research community. This step reduced the pool to 72 records, excluding 49 records.

Given the proven quality and reputation of Journals Q1 and Q2 and Conferences IEEE and ACM, only records from these sources were considered, where 51 records were selected and 21 were excluded.

The next step was to screen the records by title and abstract, to identify studies relevant to the research questions, namely those that focus on fault tolerance of node failures and communication layer architectures. This step reduced the pool to 13 records, while 38 records were excluded. The main reasons for exclusion were: implementation in blockchain, focus on data privacy, and studies that the content overlaps with the problems mentioned before, with the ”Resilient” and ”Communication” queries.

While blockchain is a valid approach to ensure fault tolerance and data privacy, the implementation requires a different set of tools and knowledge, adding complexity and overhead to the solution, which is not the focus of this work.

Finally, a full-text screening was conducted on the remaining 13 records to evaluate their contributions in depth. From these, 5 records were excluded because their focus was security

and data encryption, did not specify the communication layer, or how they handle node failures. This resulted in a total of 8 records that were selected for the review.

Each paper has its own contributions and insights, so it is important to analyze and understand each one of them.

In [34], Bano propose a methodology and architecture to integrate FL with Apache Kafka. Although the authors did not implement the solution, they highlight the potential of using Kafka to improve scalability while ensuring fault tolerance. They also note that integrating this pub/sub system with FL is not trivial.

The authors of [35] introduce a novel communication protocol, SEPP-IoT, designed for secure, efficient, and fault-tolerant communication in FL systems. Their FL framework has five key components: IoT devices, Central Server, Communication Protocol, Model Management, and Trust Management. With this architecture, the system uses lightweight cryptography, data compression, a reputation-based trust model to identify malicious nodes, and a novel model aggregation mechanism. To ensure fault tolerance, the authors have error detection and correction that involves redundancy checks, and to handle node failures, tasks are reassigned to other nodes. This framework is evaluated with 100 IoT devices with the MNIST dataset and compared with two existing techniques, [36] and [37]. The results show that this framework outperforms the existing methods regarding accuracy, time, memory, and communication overhead.

In [38], Jayaram *et al.* present a new adaptive and scalable architecture for FL aggregation, called AdaFed. This design leverages serverless/cloud functions to aggregate the models, using a logical tree topology for hierarchical aggregation. This way, the system reduces communication overhead and scales up or down based on the number of devices, allowing tolerance to node failures. The authors evaluate the performance of AdaFed in a kubernetes cluster and compare to methods that use static tree-based aggregation, achieving up to 85% reduction in resources and cost savings.

The authors of [39], propose EPPDA, a privacy-preserving data aggregation scheme for FL. This work focuses on improving privacy and security by resisting reverse attacks, while maintaining low communication overhead and fault tolerance. The authors use homomorphic secret sharing and encryption, ensuring the server cannot reconstruct individual updates. When a user disconnects, the server can still aggregate the updates from the remaining users. Compared to [40], the proposed scheme offers lower computation cost, reduces error rate, and improves communication efficiency, but its evaluation is only a theoretical analysis.

In [41] the authors design a threshold-variant of the Joye-Libert secure aggregation scheme. Their secure and fault-tolerant FL framework uses this technique to protect clients' updates and aggregate them, while tolerating up to 1/3 of node failures. They compare their protocol with SecAgg and achieve up to 8x faster running times when using 1000 clients.

The authors of [42] introduce MCHFL, a FL framework that uses a distributed network of global aggregation centers located at the edge, replacing the traditional single central server architecture and improving fault tolerance. They compare MCHFL with three existing works, [43], [44] and [23] with three datasets: MNIST, FashionMNIST and CIFAR-10. Their results

show that MCHFL outperforms the existing works in terms of robustness to failures, accuracy, time, and communication costs.

In [45], the authors propose a novel algorithm for federated edge learning that adapts to asynchronous clients joining and leaving the computation. Their key features include dynamic self-adaptation to collaboration device variations, interoperability between web browsers and Python processes, and recovering from unexpected disconnections. To achieve this, the authors based their architecture on six actors: the initiator, the workers, the aggregator, the logical server, the distributed in-memory database, and the queue server. This algorithm is evaluated with the MNIST dataset and by leveraging a combination of adaptive aggregation strategies and efficient communication between its architectural components, the algorithm achieves high levels of accuracy and Cohen's Kappa (CK) score, even in highly volatile environments and Non-IID data distributions.

In [46], Dautov and Husom propose a novel approach to improve fault tolerance and self-recovery in FL systems by integrating the Raft consensus protocol. Their method allows nodes in an FL cluster to replicate the global state and dynamically elect a new aggregator upon failures, addressing the single-point-of-failure issue. The authors implement their approach as a proof of concept using the Flower FL framework, enhanced with PySyncObj, and test it on a Raspberry Pi cluster. Their evaluation on the CIFAR-10 dataset shows that the system achieves aggregator re-election within 4 seconds for clusters of up to 10 nodes and ensures training continuity from checkpoints. However, the Raft-based implementation has a network traffic overhead approximately five times higher than baseline systems due to state replication and leader election.

To better understand the contributions of these papers, we can categorize them into common topics found in the literature. These topics help to identify strengths and limitations, as well as to understand the gaps in the research field. The topics are:

- **Fault Tolerance:** Whether the solution can handle node failures and keep the system running or not.
- **Elasticity:** Ability to handle dynamic networks, where devices can join or leave the network at any time.
- **Scalability:** Number of workers used in the evaluation, to understand if the solution can handle a large number of devices.
- **Security:** Use of encryption or other security mechanisms to protect communication.
- **Evaluation:** Use of standard datasets and benchmarks to evaluate the performance of the solution.
- **Code Availability:** If the code is available for the community to use and reproduce the results.

The Compliance column represents the overall alignment of each related work with the desired characteristics for a robust and scalable FL framework, particularly those relevant to dynamic and resource-constrained edge environments. This percentage reflects how comprehensively each study addresses the various dimensions presented in the table, with each

dimension considered to have equal importance/weight in the context of our proposed system's objectives.

Table 2.2: Categorization of the 8 selected papers from the systematic literature review, classifying each study based on its demonstrated capabilities and contributions across key topics such as fault tolerance, elasticity, scalability, security, evaluation methods, and code availability

Ref	Fault Tolerance	Elasticity	Scalability	Security	Evaluation	Code	Compliance
[34]	✓	X	Not tested	✓	X	X	33.0%
[35]	✓	X	1000	✓	✓	X	66.0%
[38]	✓	✓	10000	X	✓	X	66.0%
[39]	✓	X	400	✓	X	X	50.0%
[41]	✓	X	1000	✓	X	✓	66.0%
[42]	✓	X	600	X	✓	X	50.0%
[45]	✓	✓	64	X	✓	✓	83.0%
[46]	✓	X	10	X	✓	✓	66.0%
Total:	100.0%	25.0%	88.0%	50.0%	63.0%	38.0%	

As shown in Table 2.2, all the papers have fault tolerance mechanisms, but only 2 out of 8 can handle dynamic networks. Regarding security, the encryption methods include a lightweight stream cipher algorithm, homomorphic encryption, and Shamir's secret sharing. Although the evaluation is done with standard datasets, such as MNIST, CIFAR, and FashionMNIST, these datasets are not representative of real-world scenarios, where FL is needed to protect sensitive data; datasets related to IoT or IDSs are more suitable. Finally, only 3 out of 8 papers have the code available for the community, which is essential to create new solutions or build upon existing ones.

2.6 Heterogenous Frameworks

In addition to the papers analyzed in the systematic review, other FL frameworks have been developed to address various challenges in FL, ranging from scalability and heterogeneity to fault tolerance and dynamic network conditions. These frameworks are: Heterogeneous Federated Learning (HeteroFL). Communication- and Computation-Aware Federated Learning (CoCoFL), Flower, and TensorFlow Federated.

These frameworks provide distinct solutions to the challenges of FL, having different strengths and limitations. These are summarized in Table 2.3 and are further discussed in Section 3.1.

Table 2.3: Key characteristics and drawbacks of prominent alternative FL frameworks such as HeteroFL, CoCoFL, Flower, and TensorFlow Federated, providing insight into their design philosophies and applicability.

Framework	Key Features	Limitations
HeteroFL	Supports heterogeneous client models Static batch normalization	Lack of customization Focus on heterogeneity
CoCoFL	Partial neural network freezing Quantization	Out of the box solution Lack of flexibility
Flower	Framework agnostic Customizable and scalable	Not a complete solution Requires additional components
TFF	Seamless integration with TensorFlow Supports dynamic client participation	Restricted to TensorFlow Fixed communication layer

2.6.1 HeteroFL

HeteroFL is a framework presented in [47] designed to allow clients with heterogeneous resources to participate in FL systems. The framework enables clients to use heterogeneous local models with different computational complexities to contribute to the global model, unlike traditional FL methods that require all clients to use the same model architecture.

This is achieved through a novel technique of adaptively distributing subnetworks to clients based on their capabilities, enabling efficient participation without compromising the overall system performance. HeteroFL also introduces key innovations such as static batch normalization to ensure privacy and stability during training, and a scaling mechanism to balance contributions from models of different sizes.

2.6.2 CoCoFL

CoCoFL is a novel framework designed to address the challenges of resource heterogeneity in FL systems and presented in [48]. Unlike traditional FL methods, which can exclude or limit the contributions of devices with constrained resources, CoCoFL ensures fairness and efficiency by introducing partial NN freezing and quantization.

These techniques allow devices to adapt their training process to available computation, communication, and memory resources. CoCoFL can significantly reduce resources by freezing specific NN layers and performing low-precision operations on frozen layers. This allows all devices to train on the whole model, ensuring that even less capable devices make meaningful contributions to the global model, preserving accuracy and fairness across participants.

The framework outperforms existing methods, such as HeteroFL, in environments with Non-IID data distributions, where weaker devices are critical for capturing diverse data patterns.

2.6.3 Flower

Flower is a versatile FL framework designed for research and deployment across various ML frameworks, such as TensorFlow, PyTorch, and JAX. It offers customization, scalability, and framework-agnostic capabilities, making it suitable for diverse FL use cases.

Flower emphasizes ease of use, allowing researchers to create and extend FL systems, integrate state-of-the-art strategies, and support synchronous and asynchronous approaches. By leveraging Flower's easy-to-use Application Programming Interfaces (APIs), users can quickly prototype FL systems without building everything from scratch, but it is just a tool, not a complete solution.

2.6.4 TensorFlow Federated

TensorFlow Federated is a framework designed to facilitate FL applications within the TensorFlow ecosystem. The framework supports both federated training and evaluation and provides tools for simulating FL experiments. It also handles heterogeneous devices and data, which allows dynamic client participation regardless of computational resources.

This customizable framework enables users to define their FL algorithms, models, and data sources. Still, it does not allow users to define their communication layer or use different ML frameworks.

CHAPTER 3

Proposed Solution

To address the challenges of FL in heterogeneous environments, it is essential to identify the core requirements that enable robust and scalable solutions. FL frameworks must accommodate dynamic network conditions, varying device capabilities, and constraints such as limited bandwidth and computational power. These requirements are important to develop a system architecture capable of handling these challenges.
This chapter explores the requirements necessary for designing a resilient FL architecture and introduces the proposed system's key components. The analysis begins with an evaluation and comparison of existing frameworks and technologies to establish baseline expectations. The proposed solution is then presented, detailing the system's architecture, components, and design choices. Finally, a SWOT analysis is conducted to assess the system's strengths, weaknesses, opportunities, and threats, providing a comprehensive overview of the proposed solution.

3.1 System Requirements and Comparison

To understand the current state of FL solutions, a comparative analysis of existing frameworks shown in Chapter 2 was conducted. The comparison includes the papers that evaluated their solution, the other frameworks, and is based on the following criteria:

- **Resilience:** The system should be able to maintain the model training even with node failures and allow devices to join or leave the network at any time.
- **Modularity:** The framework should be modular regarding the ML backend, FL algorithm, and communication layer.
- **Analysis:** The authors should provide a detailed analysis of the proposed solution, explaining their design choices and evaluating the system's performance.
- **Code and Documentation:** Availability of the source code and detailed documentation to help users understand and use the system.

These criteria are the minimum requirements for a FL framework to handle the challenges of heterogeneous environments and, equally important, to be extended and integrated with

other systems. Table 3.1 summarizes the results of the analysis, where each solution, including the proposed solution, is evaluated based on these criteria.

Table 3.1: A qualitative comparison of the proposed FL solution against existing frameworks and research papers, evaluated based on criteria including resilience, modularity, analysis depth, code and documentation availability, and overall compliance with design requirements.

Solution	Resilience	Modularity	Analysis	Code and Documentation	Compliance
[35]	X	X	✓	X	25.0%
[38]	✓	X	✓	X	50.0%
[42]	X	X	✓	X	25.0%
[45]	✓	X	✓	✓	75.0%
[46]	X	X	✓	✓	50.0%
HeteroFL	✓	X	✓	✓	75.0%
CoCoFL	✓	X	✓	✓	75.0%
Flower	X	✓	X	✓	50.0%
TTF	✓	X	X	✓	50.0%
Proposed	✓	✓	✓	✓	100.0%

While existing solutions and frameworks offer valuable contributions to the field of FL, none fully satisfy the specific combination of requirements central to this research: inherent resilience to dynamic network conditions alongside comprehensive modularity across the ML backend, FL algorithms, and crucially, the communication layer.

Frameworks like Flower and TensorFlow Federated provide tools and structures for FL, but they either lack the necessary built-in resilience mechanisms to handle arbitrary node failures and dynamic joins/leaves seamlessly without requiring significant additional components. Or they impose restrictions on key modules, particularly the communication protocol, which is vital for adapting to diverse network environments and ensuring fault tolerance.

The solutions from the systematic review demonstrate analysis and some fault tolerance, but generally lack the modularity and code availability needed for building a highly adaptable open-source platform.

Therefore, to fully address the identified gaps and achieve the objective of designing, implementing, and evaluating a FL framework that is both highly modular and robustly resilient to real-world network dynamics, developing a new framework tailored to these specific goals was deemed necessary.

3.2 Architecture

Our proposed framework is not a single, rigid FL implementation, but instead a versatile architecture comprising seven core modules that collaborate to support a range of FL

algorithms and configurations. This modular structure is fundamental to the framework's adaptability, enabling easy integration of new algorithms, communication methods, resilience approaches, and optimizations without requiring substantial modifications to the entire system. Figure 3.1 shows the architecture of our proposed framework.

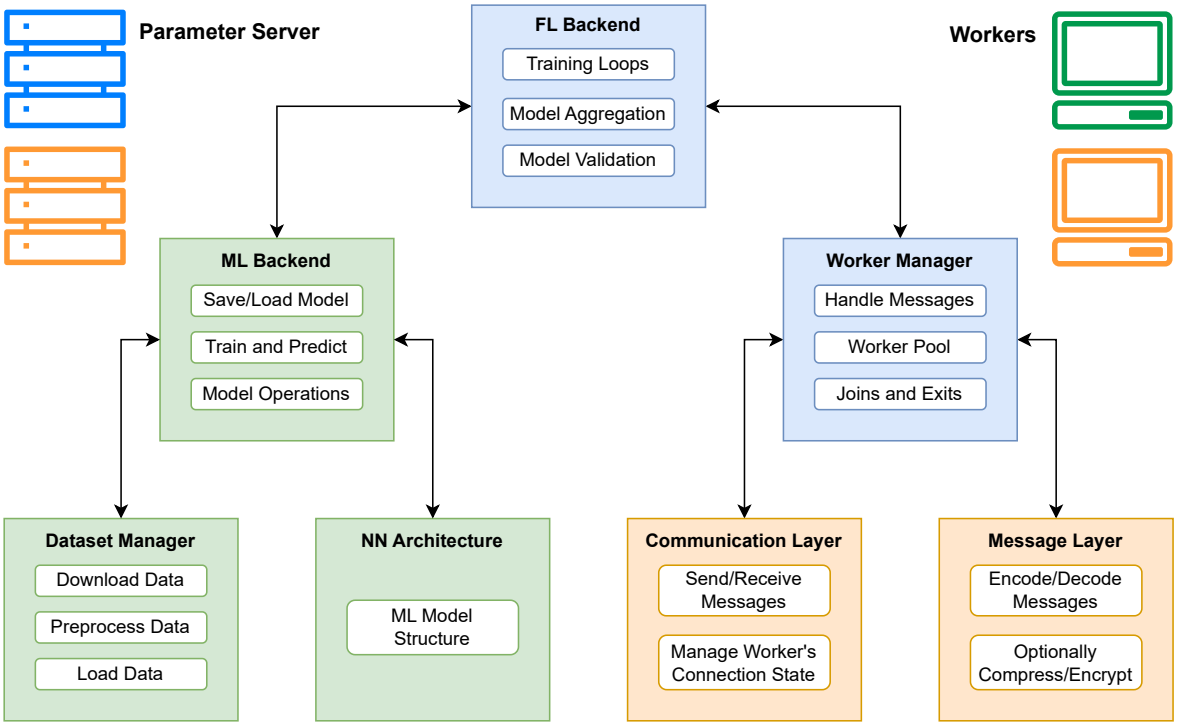


Figure 3.1: Conceptual design of the resilient FL system, showcasing its modular components and detailing the roles of the Parameter Server and Workers as key entities. The color-coding helps identify the primary functions of each module within the overall framework.

The architecture consists of the following core modules, each serving a specific purpose in the overall system:

FL Backend: This module serves as the central component for the Federated Learning process logic. On the Parameter Server, it manages the global training loop, orchestrates rounds, and determines worker scheduling, aggregates model updates from workers, updates the global model state, and validates it. On the worker side, it manages the local training loop and interaction with the server or peers, depending on the algorithm. It uses the *ML Backend* for model interactions and the *Worker Manager* to interact with workers.

ML Backend: This module provides an abstraction layer for the underlying ML framework, making the rest of the framework independent of the specific library used. It handles the ML model, offering interfaces for common operations such as saving/loading the model, training the model on local data, making predictions, and other general model operations like accessing weights or gradients. It employs the *Dataset Manager* to retrieve data and the *Neural Network Architecture* to define the model.

Dataset Manager: Responsible for all aspects related to handling data on the client devices. It oversees the data extraction process from local storage, potentially managing tasks like downloading (if data needs to be fetched), preprocessing the data, and loading split data

batches for training, validation, and testing. It supplies processed data samples to the *ML Backend*.

Neural Network Architecture: This module defines the structure of the ML model being trained. It contains the specifications for the layers and connections of the NN and is used by the *ML Backend* to create the model instance before training starts. Separating this allows for straightforward swapping of model architectures for the same dataset or FL scenario.

Worker Manager: Primarily located on the Parameter Server, this module manages the pool of participating client devices (workers). It tracks connected workers, handles joins/disconnects from the network, and manages the worker subpool used for each training round (particularly in client-selection scenarios), making it a key component in ensuring the system's elasticity and fault tolerance. It interacts with the workers using the *Communication Layer* and the *Message Layer*, allowing it to encode a message once and send it to multiple workers.

Communication Layer: This vital module handles all network communication between any nodes in the system, including the server and workers. It provides an abstract send and receive interface, concealing the complexities of the specific communication protocol employed. Importantly, this layer is also responsible for providing fundamental resilience to the system by managing connection status, addressing message delivery issues, and incorporating protocol-specific fault tolerance features.

Message Layer: Conceptually positioned below the Communication Layer, this module is responsible for defining the format and processing the content of exchanged messages. It includes functionalities to encode and decode messages and can optionally perform data compression and encryption of model updates or other exchanged information to enhance efficiency and security.

Having detailed the individual roles of the core modules, it is important to explicitly highlight how their collaborative design fulfills the system requirements outlined in Section 3.1. Resilience is a key outcome, achieved through the Worker Manager's dynamic worker pool management and task rescheduling capabilities, ensuring training continuity despite node failures or disconnections.

The Communication Layer (including the Message Layer for secure and efficient data transfer) further contributes by providing robust fault detection and handling mechanisms.

Modularity is inherently built into the framework: the ML Backend abstracts diverse machine learning libraries, the FL Backend supports various federated learning algorithms, and the Communication Layer (with its various protocol implementations) allows for flexible integration of communication paradigms. This architectural flexibility, combined with the structured logging by modules like the Worker Manager and FL Backend, facilitates comprehensive Analysis of system performance and training progress.

The framework's commitment to Code and Documentation is demonstrated by its open-source availability on GitHub and PyPI, alongside its intuitive Command Line Interface (CLI) tools, making it accessible for researchers and practitioners to understand, use, and extend.

Further technical details regarding the implementation of these design choices, including software stack and specific mechanisms, are provided in Chapter 4.

3.3 Communication Strategies

To further illustrate the proposed architecture's dynamic interactions and resilience mechanisms, particularly in worker participation and task management, we detail the communication sequences during key worker lifecycle events: initial join, network disconnection, and training round participation.

Coordinated by the core modules, these sequences demonstrate the framework's robustness in dynamic environments with fluctuating worker availability.

3.3.1 Worker Join Sequence

When a new worker connects to the system, it undergoes a registration process with the Parameter Server. This handshake establishes its identity and prepares it for participation in the training rounds. This sequence involves the worker initiating contact, the Parameter Server assigning it a unique identifier and a run identifier, and the worker confirming its registration by sending initial metadata before becoming available for tasks.

The underlying communication protocol primarily handles the initial steps, while the subsequent metadata exchange and readiness signaling involve the Message Layer and Worker Manager. The detailed steps are illustrated in the sequence diagram in Figure 3.2.

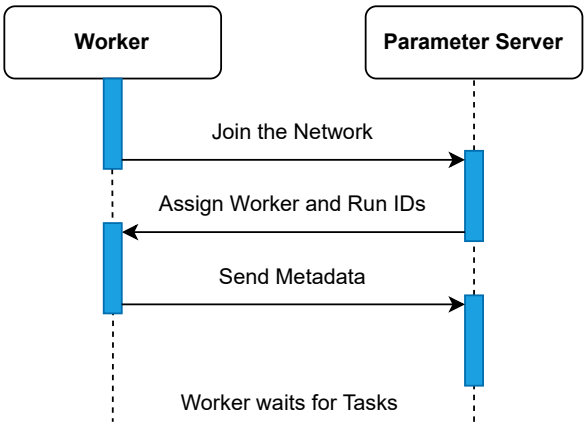


Figure 3.2: Communication sequence illustrating how a worker joins the FL system by initiating contact, the Parameter Server assigning unique identifiers, the worker sending initial metadata, and finally, the worker becoming ready to receive tasks.

3.3.2 Worker Leaves Sequence

Workers may leave the FL system either gracefully (sending an explicit disconnect message) or abruptly (due to unexpected disconnections or failures). The framework's resilience relies on the Parameter Server's ability to detect both scenarios and update the active worker pool managed by the Worker Manager accordingly. If a worker sends a leave message, the server receives and processes it. If a worker disconnects unexpectedly, the protocol-specific

mechanisms in the Communication Layer detect the loss of connection. Figure 3.3 illustrates the communication flow for a worker leaving the system.

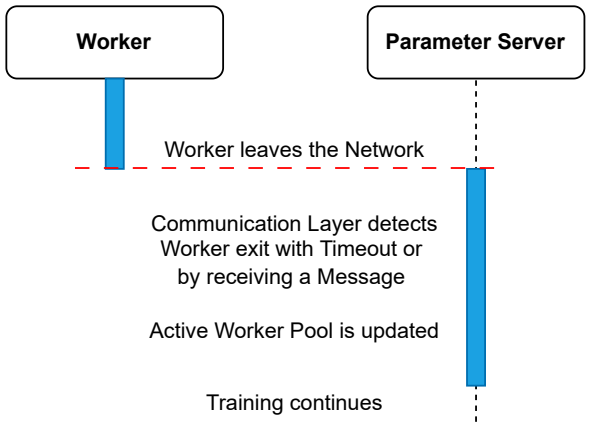


Figure 3.3: Communication sequence illustrating how a worker leaves the FL system. The Parameter Server's communication layer detects the worker's exit, either through an explicit message or a timeout, which triggers an update to the active worker pool and allows the training process to continue uninterrupted.

3.3.3 Work Cycle Sequence

This sequence depicts the typical communication flow when a worker receives and processes a task during a training round. It starts with the Parameter Server sending the necessary data or model parameters for the task. The worker then performs its local computation (training on its data). Upon completion, the worker returns its results or model updates to the Parameter Server and waits for future messages. The sequence diagram in Figure 3.4 illustrates this successful cycle.

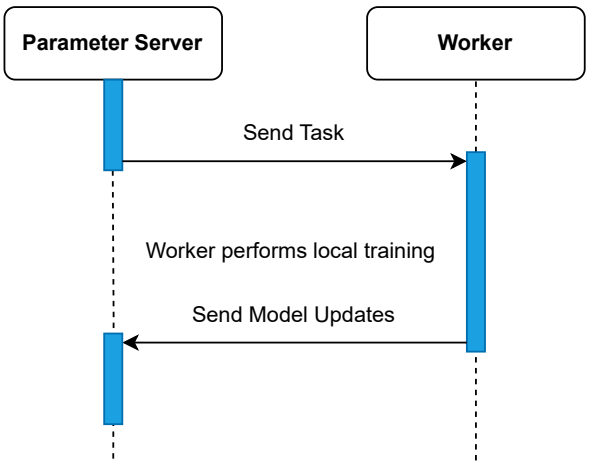


Figure 3.4: Communication sequence illustrating a worker successfully receiving and completing a task within the FL system. The process shows the Parameter Server sending a task, the worker performing local training, and then sending model updates back to the server.

A critical aspect of resilience is handling failures during the work cycle. The Parameter Server must detect if a worker fails after receiving a task but before successfully submitting its results. Depending on the FL algorithm (synchronous vs. asynchronous) and the resilience

mechanisms implemented (e.g., completion thresholds, task rescheduling by the Worker Manager), the framework adapts to continue training. The sequence diagram in Figure 3.5 shows an example of this failure scenario and server detection.

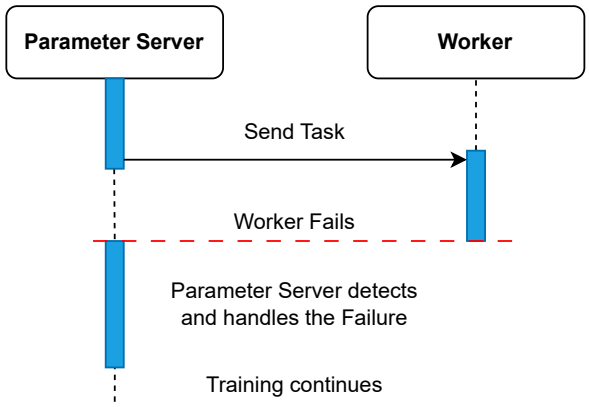


Figure 3.5: Communication sequence illustrating a worker failure during task execution and the Parameter Server's subsequent detection and handling of this event. This ensures that the FL training process can continue despite the disruption.

3.4 SWOT Analysis

To provide a comprehensive understanding of the proposed architecture, a SWOT analysis was conducted to evaluate its internal strengths and weaknesses alongside external opportunities and threats. The SWOT analysis is summarized in Table 3.2 and it serves as a guide for future development and evaluation, highlighting areas for improvement and potential risks that must be addressed.

Table 3.2: A comprehensive SWOT analysis of the proposed FL framework, detailing its internal strengths and weaknesses, alongside external opportunities and threats. This provides a balanced view of its potential and limitations for future development.

	Helpful	**Harmful**
Internal	**Strengths** Resilient, highly modular and easy to use	**Weaknesses** Validation is limited to the simulation environment
External	**Opportunities** Easy to extend and integrate with other systems	**Threats** Scalability may be limited

The main strengths are: resilience, modularity, and ease of use. The system is designed to handle dynamic network conditions, ensuring seamless operation even when nodes join or leave the network. The architecture features a highly modular design, working as a puzzle where each piece can be replaced or extended without affecting the other modules.

Despite its strengths, the system has certain internal weaknesses that may affect its practical utility. The proposed solution will be evaluated in a simulated environment using Virtual Machines (VMs). While this provides valuable initial insights, it does not fully replicate real-world conditions. Challenges such as hardware diversity, network instability, and scalability in large-scale deployments are not fully captured in the simulation, limiting the system's validation.

The external opportunities for the proposed solution are vast, as the system is designed to be easily extended and integrated with other systems. This flexibility allows for the integration of new FL algorithms, communication protocols, and ML models, providing a versatile platform for research and development. The system's modularity also enables the integration of new features and functionalities, allowing for continuous improvement and adaptation to changing requirements.

Finally, the system faces certain external threats that may impact its long-term viability. Despite the system being designed to be scalable, environments with a larger number of workers than the simulation environment may pose challenges. The system's performance may degrade as the number of workers increases, affecting the overall efficiency and effectiveness of the FL process.

CHAPTER 4

Implementation

Building upon the conceptual architecture proposed in Chapter 3, this chapter delves into the practical implementation of the Resilient Federated Learning Framework. It details the technical choices made and the components developed to realize a modular and resilient system capable of operating in dynamic network environments. The following sections describe the selected software stack, the implementation details of the FL algorithms, including specific resilience mechanisms and optimizations, additional features built to support the framework, and key technical considerations such as communication protocol specifics, worker scheduling, and framework extensibility.

4.1 Software Stack

Figure 4.1 provides an overview of the software stack used in implementing the Resilient Federated Learning Framework. The diagram illustrates the various components and their interactions, highlighting the system's modularity and flexibility.

The implementation of the proposed framework is built primarily using Python. This choice was motivated by Python's widespread adoption and robust ecosystem within the data science and ML communities, offering a balance of ease of use and performance through highly optimized libraries, as highlighted by Raschka *et al.* in their survey of the field [49].

For effective management of project dependencies and ensuring reproducible development environments, crucial for any substantial software project, we used UV [1]. UV is an extremely fast Python package and project manager, written in Rust. It is known for its significant speed advantages over traditional tools like pip and aims to consolidate the functionality of multiple tools (such as pip, pip-tools, and virtualenv) into a single, efficient utility. UV provides comprehensive features including dependency resolution, package installation, virtual environment creation, and even Python version management, making it a powerful choice for streamlining the development workflow.

[1] `https://docs.astral.sh/uv/`

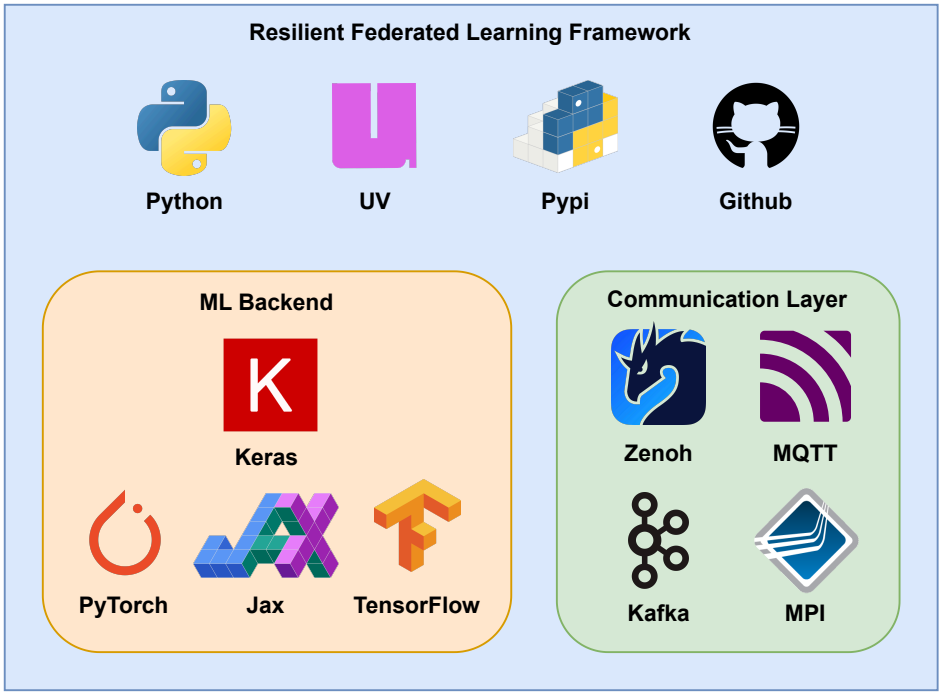


Figure 4.1: Diagram presenting the key software components comprising the Resilient FL Framework, including core development tools, interchangeable ML backends, and diverse communication protocols, emphasizing its modular design.

Version control is indispensable for tracking changes, collaborating, and maintaining a software project's history. Git was employed for this purpose, with the framework's codebase hosted on GitHub [2]. The choice of GitHub aligns with industry standards and best practices for software development, ensuring that the project is accessible to contributors and users alike. The publicly available repository allows for transparency and community engagement in the development process.

The implemented framework is available as a standard Python package on the Python Package Index (PyPI) to facilitate easy access and distribution for other researchers and practitioners. This availability allows users to effortlessly install the framework and its dependencies using simple commands (e.g., `pip install flexfl`), providing both a Python library for programmatic use and a CLI tool for direct execution and experimentation, contributing to the goal of making the framework an open-source resource.

For the core ML operations, the framework needed to integrate with established ML libraries. While several powerful libraries exist, TensorFlow, PyTorch, and JAX are currently among the most widely used in the Python ecosystem for deep learning and numerical computation. These libraries, however, differ significantly in their APIs, design philosophies, and typical use cases. As mentioned in Chapter 3, the ML Backend module of our framework is designed specifically to abstract away these differences, providing a consistent interface for the rest of the system regardless of the underlying ML library.

In this implementation, we have built support primarily around Keras, a high-level API

[2] `https://github.com/leoalmPT/FlexFL`

that acts as a powerful abstraction layer. The Keras philosophy emphasizes that it is a deep learning API designed with a focus on debugging speed, code elegance and conciseness, maintainability, and deployability. Crucially, Keras achieves this while offering a multi-backend approach, giving users the freedom to work with JAX, TensorFlow, and PyTorch, and allowing them to build models that can move seamlessly across these frameworks and leverage the strengths of each ecosystem. Additionally, we have also integrated support for TensorFlow and PyTorch directly, allowing users to choose the backend that best fits their needs. This flexibility is essential for researchers and practitioners who may have specific requirements or preferences for their ML workflows.

Moving to the communication infrastructure, a critical component for coordinating clients and the server in a distributed setting, we implemented support for all four protocols discussed in Chapter 2: MPI (via Open MPI), MQTT, Kafka, and Zenoh. While MPI may not be ideally suited for the dynamic network environments inherent in FL due to its reliance on static configurations and lack of built-in fault tolerance for node failures, its inclusion serves as a valuable benchmark for performance comparison, given its prevalence and optimization for speed in HPC environments.

The implementation of MQTT, Kafka, and Zenoh, on the other hand, allows the framework to leverage protocols better designed for dynamic and potentially unreliable networks, offering flexibility and enabling empirical evaluation of their performance and the framework's resilience mechanisms under conditions with and without worker failures or disconnections.

4.2 Federated Learning Algorithms

Building upon the architectural principles established in Chapter 3 and the algorithmic taxonomy discussed in Chapter 2, our framework implements the four primary types of FL algorithms to demonstrate versatility and evaluate performance across different collaboration and scheduling paradigms. These are Centralized Synchronous, Centralized Asynchronous, Decentralized Synchronous, and Decentralized Asynchronous algorithms. Each algorithm is designed to operate within the framework's modular architecture, allowing for easy integration and experimentation with various components.

These standard algorithmic implementations incorporate several key adaptations and features beyond their basic theoretical descriptions to address the objectives of resilience and modularity in dynamic network environments. These are discussed in detail in this section.

4.2.1 Dynamic Worker Pool and Epoch Definition

A core adaptation, facilitated by the Worker Manager module, is managing a dynamic pool of potential workers. Instead of assuming a fixed set of participating devices, the framework dynamically selects a `subpool` of available and suitable workers at the start of each training round or epoch. This allows devices to join or leave the network seamlessly without disrupting the training process, contributing directly to elasticity and resilience.

Within our framework, a training `epoch` or `round` represents a complete cycle of global model coordination and update involving the selected worker subpool. A key parameter

governing this process is the concept of `iterations` per epoch for Synchronous approaches. This defines the number of times that the parameter server sends tasks to the workers and collects their updates. The definition of epochs and iterations depends on the algorithm type:

- **Centralized Approaches:** The primary task for each worker in a round is to receive the current global model, compute gradients on a local mini-batch, and send these gradients to the Parameter Server. The total number of effective computations (tasks) in a round is thus related to the sum of the number of batches processed by all selected workers. The number of iterations is defined by the total number of tasks divided by the number of workers in the subpool.
- **Decentralized Approaches:** Each worker's task in a round involves performing a specified number of local epochs and then exchanging their resulting local model or parameters with the parameter server. One global synchronization round corresponds to each selected worker completing their local training phase and submitting their updates. Only 1 iteration is performed per round.

4.2.2 Resilience Mechanisms

Regardless of the algorithm type, the framework implements resilience mechanisms to handle worker failures and disconnections. These mechanisms are designed to ensure that the training process can continue despite unexpected disruptions, aligning with the framework's goal of resilience in dynamic network environments.

In Asynchronous algorithms, where the system can tolerate some delays, the framework monitors the progress of tasks assigned to workers in the worker subpool. If a worker fails, disconnects, or becomes unresponsive while executing its task (either gradient computation in centralized async or local training in decentralized async), the Worker Manager detects it.

The task associated with the failed worker can then be automatically rescheduled and assigned to another available worker from the dynamic pool, according to the scheduling policy, ensuring that the necessary contribution for the round is eventually completed. If the number of workers drops below a certain threshold, the Parameter Server will pause the training round by stopping the distribution of new tasks until the Worker Manager detects that enough workers are available again.

Synchronous algorithms inherently require a higher degree of coordination. To prevent single stragglers or failures from halting the entire training process, a configurable completion threshold (e.g., 50%) is introduced for each iteration. If the number of workers in the selected worker subpool that successfully complete and submit their tasks meets or exceeds this threshold, the Parameter Server proceeds with the aggregation step using the available updates. This allows the training to advance even if a subset of workers fails or is significantly delayed.

4.2.3 Optimizations

Several optimizations have been incorporated into the framework to enhance its efficiency and scalability, particularly in handling large models and coordinating multiple workers.

A key optimization concerns aggregating model weights or gradients received from workers on the Parameter Server. To conserve memory and avoid the need to store the whole model or gradient updates from every worker simultaneously, the aggregation process is performed on the fly. As updates arrive from individual workers, they are incrementally incorporated into a running weighted sum or average. This allows the memory associated with a worker's update to be released immediately after it has been processed.

For instance, if M represents the size of a single model update and there are N workers, this approach reduces the server's peak memory footprint from potentially $N \times M$ bytes (if all updates were buffered simultaneously) to approximately $2 \times M$ bytes (for the current global model and one incoming update). This significantly enables support for a larger number of workers or larger model sizes than would otherwise be possible.

Furthermore, to minimize idle time and increase overall throughput, the framework optimizes the timing of task distribution to workers. Instead of waiting for the Parameter Server to complete the full validation process on the newly updated global model, tasks for the next round can be initiated as soon as the global model state is mathematically finalized by the aggregation process. This helps to pipeline the training process, reducing the latency between rounds.

For example, if validation for each epoch takes T seconds and the training runs for E epochs, this optimization can reduce the total training time by up to $E \times T$ seconds by overlapping validation with the start of the subsequent round, thereby improving the utilization of both the server and worker resources, especially when the Parameter Server is not a bottleneck.

Additionally, a crucial change for asynchronous approaches was made. With dynamic worker participation, we must ensure that workers train on models that do not excessively lag behind the global model. In the original decentralized asynchronous algorithms, model differences were used for updates, but this is not suitable when workers might skip rounds due to the dynamic pool selection. To address this, at the start of each round, the Parameter Server sends the current global model to all workers in the current subpool, and each worker then performs their tasks using this model. This ensures that, despite the asynchronous nature and potential for variable participation, workers start their local computations from a recent common reference point, which helps convergence stability.

4.3 Additional Features

Beyond the core modular architecture and implemented FL algorithms, the framework provides a suite of additional features, primarily exposed through CLI tools. These tools are designed to streamline the entire Federated Learning workflow, from data preparation and setup to experimentation, simulation, and analysis, making the framework more accessible and practical for researchers and practitioners.

The primary CLI commands available are:

- `flexfl`: This is the main command for executing FL experiments. It provides extensive configuration options through command-line arguments to define the setup of all seven

core modules of the framework, enabling flexible experimentation with different algorithm types, communication protocols, ML backends, datasets, and more. Details on the argument parser and configuration options will be provided later in Section 4.4.

- `flexfl-preprocess`: Used to manage the dataset preparation process. This command facilitates the downloading and initial preprocessing of datasets, allowing users to prepare data for specific experiments or batch-process multiple datasets supported by the framework's Dataset Manager module.
- `flexfl-division`: Handles the division of datasets for distributed training. This command splits processed datasets into training, validation, and testing sets. Crucially for FL, it also partitions the training data among a specified number of workers, supporting both Independent and Identically Distributed (IID) and Non-IID data distributions to simulate various real-world scenarios encountered in FL.
- `flexfl-benchmark`: Provides tools for benchmarking communication performance and resilience of the network layer. Designed to be run typically on two nodes (one acting as a server, one as a worker), this command allows for testing specific communication protocols implemented in the Communication Layer, simulating basic worker failures, and measuring the time required to send and receive payloads of varying sizes multiple times between nodes.
- `flexfl-res`: A utility specifically for simulating worker resilience under failure conditions. This command launches a specified process (e.g., a worker instance) and introduces controlled disruptions by randomly killing and restarting the process based on configurable probabilities and time intervals. This tool is helpful for empirically testing the fault tolerance mechanisms integrated into the framework under realistic failure scenarios.
- `flexfl-plot`: Dedicated to visualizing experiment results and system performance, this command processes the structured logs generated by framework runs and automatically generates various data visualizations, including plots and tables, for analyzing key metrics such as communication times, worker computation times, worker timelines, size of exchanged payloads, and overall run duration. This aids in understanding performance characteristics and the impact of different configurations.

These CLI tools collectively provide a comprehensive environment for setting up, running, simulating, and analyzing FL experiments with the proposed framework, facilitating both research and practical application. But while these tools provide the interface for configuring and running FL experiments on individual nodes, conducting realistic distributed and FL studies requires managing a cluster of interconnected machines.

To address the complexities of creating, configuring, and managing a fleet of VMs for experimentation, a separate package named `pxm-tools` and also available on PyPI was created. This tool leverages the Proxmox Virtual Environment API, a widely used open-source platform for virtualization management. `pxm-tools` provides a command-line interface designed for ease of use when performing bulk operations on VMs.

Its core functionalities include easily creating, editing, removing, starting, and stopping multiple VMs simultaneously. It also offers flexible configuration by allowing any argument available through the Proxmox API to be set during VM creation or editing. This is crucial for simulating diverse and heterogeneous environments by configuring VM resources like the number of CPU cores, RAM size, storage, and critically, network interface bandwidth limits. The ability to control network conditions is essential for evaluating the framework's resilience under varying real-world constraints.

Furthermore, `pxm-tools` automatically retrieves necessary VM information, such as their unique IDs and assigned IP addresses, which are vital for subsequent interaction and logging. By abstracting the direct interaction with the Proxmox API into simple CLI commands, `pxm-tools` significantly simplifies the process of setting up and tearing down experimental clusters, enabling rapid iteration and evaluation cycles.

Once the experimental VMs are provisioned and configured using `pxm-tools`, an efficient method is needed to interact with all or a subset of these machines collectively. Custom scripts were developed to enable parallel execution of commands across the chosen VMs, leveraging SSH for secure communication.

These scripts are designed to streamline the experimental workflow by allowing users to execute arbitrary shell commands concurrently on all selected VMs, drastically reducing the time required for setup tasks like installing dependencies or configuring the environment. They also efficiently distribute the preprocessed and partitioned datasets (prepared using `flexfl-division`) to the corresponding worker VMs, ensuring each node receives its designated data split for the experiment, and gather experimental outputs, such as logs, performance metrics, and final models, from all participating VMs in parallel after a run is complete.

Finally, these scripts can launch the main `flexfl` command on the designated server VM and all selected worker VMs simultaneously to initiate a distributed training run. These parallel interaction scripts, combined with `pxm-tools`, provide a complete toolchain for managing the experimental environment and executing Federated Learning experiments at scale on a cluster of VMs, crucial for thoroughly evaluating the framework's performance and resilience.

4.4 Technical Details

This section delves into the technical intricacies of the framework's implementation, focusing on key aspects such as the communication layer protocols, the worker scheduling policy, the functionality of the Worker Manager, early stopping mechanisms, the configuration interface via the argument parser and optional libraries, how experimental results are saved, and guidelines for extending the framework. These details highlight the practical realization of the modular and resilient design principles discussed in Chapter 3.

4.4.1 Communication Protocols

As discussed in Chapter 2, the framework implements support for multiple communication protocols (Zenoh, MQTT, Kafka, and MPI) to offer flexibility and adaptability to diverse network environments. While the Communication Layer module provides a consistent `send` and `recv` interface to the rest of the framework, there are protocol-specific nuances and implementation details, particularly regarding asynchronous operation and fault detection.

All supported protocols require an initial connection phase where workers register with the Parameter Server to obtain a unique identifier and establish communication channels. The implementations for Zenoh and MQTT leverage their native support for asynchronous message handling, typically through threading or event loops, and include built-in capabilities to detect client disconnections (e.g., via callbacks or subscription mechanisms). This native support simplifies the implementation of resilience features dependent on knowing worker connection status.

Kafka, in contrast, is optimized for high-throughput, persistent message streaming and lacks native real-time disconnection notifications without incurring significant partition rebalancing overhead, which is undesirable in a dynamic FL setting. To overcome this, the Kafka implementation required a manual heartbeat mechanism, where workers periodically send health check messages to the server.

MPI, being primarily designed for synchronous, blocking communication in high-performance computing environments, inherently operates in a request-response manner. But, as previously mentioned, it lacks built-in capabilities to detect client disconnections directly at the protocol level without explicit application-level checks, which imposes constraints on handling dynamic worker pools compared to the other protocols.

4.4.2 Worker Scheduling Policy

The framework incorporates a worker scheduling policy that is responsible for selecting the subset of available workers participating in each training round or epoch. The currently implemented policy is Round Robin, a well-established [50], simple, and fair scheduling approach that cycles through the pool of available workers. In our dynamic setting, the Round Robin scheduler selects workers sequentially from the currently connected pool at the start of each round.

A key advantage of the framework's modular design is that the scheduling policy is encapsulated within the FL algorithm. This allows users to easily replace the default Round Robin implementation with alternative policies (e.g., based on worker capabilities, data distribution, historical performance, or network conditions) to suit specific requirements, without needing to modify other core components of the framework. The scheduler's integration with the dynamic worker pool management enables seamless adaptation to workers joining or leaving the training process.

4.4.3 Worker Manager

The Worker Manager is a pivotal module for achieving resilience and abstracting the complexities of dynamic worker participation from the core FL algorithms. It acts as the central coordinator on the Parameter Server and also runs a client-side component on each worker. Its core functionalities include managing the list of currently connected workers and the selection of the worker subpool for each round, effectively hiding the complexities of workers joining or leaving from the FL algorithm logic.

Users can configure custom callbacks for key events, such as when a new worker joins the system (executable on both the Parameter Server and the joining worker) or when a worker disconnects or fails (primarily handled on the Parameter Server). These callbacks allow for custom logic to be executed during these events, enhancing flexibility.

Communication within the framework is structured using message types (e.g., work, work_done). that the user specifies when sending messages. The Worker Manager facilitates flexible message reception by allowing users to register callbacks to be executed automatically when a message of a specific type is received, which enables asynchronous processing of incoming messages.

Users can also perform blocking receives for messages of a specific type, waiting until a message of that type arrives. Furthermore, the framework allows for the combination of blocking receives with callbacks. During a blocking wait for a specific message type, the Worker Manager can process other incoming messages in the background using their registered callbacks, such as join messages.

Integrated with the message handling, the Worker Manager monitors blocking receive operations for worker failure detection and rescheduling. If a worker fails or disconnects while a blocking receive is pending for a task assigned to it, the Worker Manager can trigger a user-defined callback that allows for the implementation of task rescheduling logic, enabling the framework to assign the failed task to another worker.

Messages received in the background that do not have an immediate callback registered are stored in an internal queue, so blocking receive operations first check this queue before waiting for new messages from the communication layer, ensuring that already received messages are processed promptly. The manager can also store relevant metadata for each connected worker, regarding the FL training or potentially performance metrics, which can be used by scheduling policies or custom callbacks.

Finally, the Worker Manager provides a mechanism for conditional waiting, allowing the process to wait for a specific condition (defined by a callback function) to become true while the manager continues to process incoming messages in the background. This is useful for coordinating actions that depend on the state of multiple workers or received messages.

Figure 4.2 presents a detailed flow diagram that illustrates how incoming messages are processed, how callbacks enable asynchronous handling and how the internal queue manages background messages, underscoring the robust communication pipeline designed to support the framework's operations under challenging conditions.

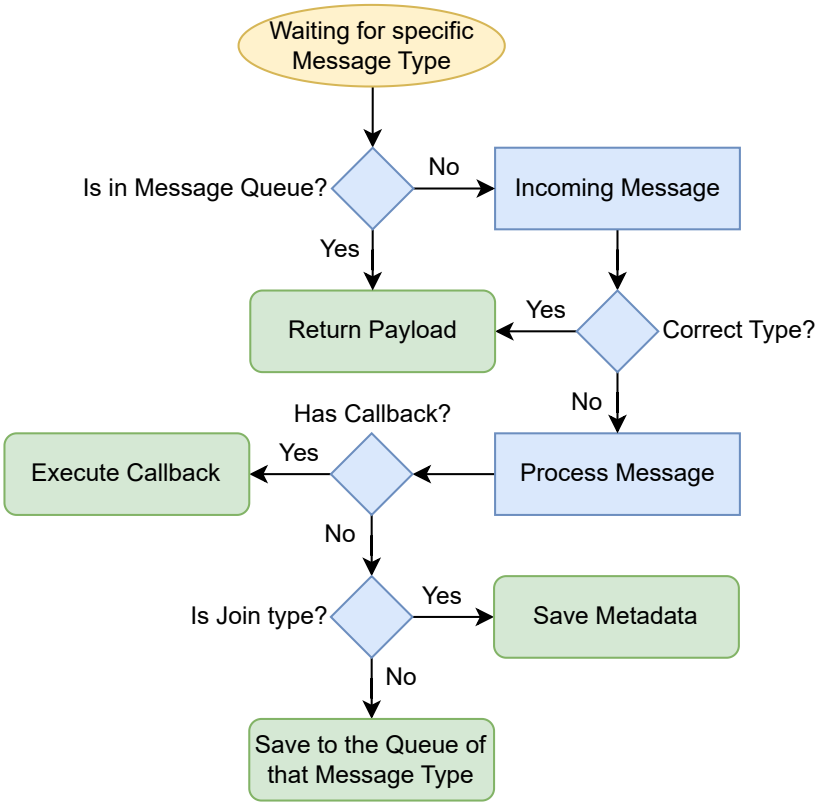


Figure 4.2: Flow diagram illustrating the Worker Manager's message processing, callback handling, and internal queue management.

4.4.4 Early Stopping

To prevent overfitting and optimize training time, the framework implements an early stopping mechanism controlled on the Parameter Server. This feature allows users to define criteria for halting the training process based on the model's performance on a validation set. Configurable parameters include `patience`, `delta`, `target_score`, and the `main_metric` used for evaluation.

The validation is performed at the end of each epoch or round using the current global model. If the monitored metric's value meets or exceeds the target score (e.g., achieving a certain accuracy level) or if the metric has not improved by at least delta for patience consecutive validation checks, the Parameter Server signals all participating workers to stop training, and the experiment concludes. The interpretation of "improvement" depends on the metric: in general, for classification problems, higher metric values (closer to 1) are generally better, while for regression problems, lower values (closer to 0) indicate better performance.

In each epoch, the model state corresponding to the best-achieved metric value seen so far is automatically saved. Upon termination, regardless of whether early stopping was triggered or the maximum number of epochs was reached, the framework automatically loads and provides the best model for further analysis or deployment.

4.4.5 Argparser and Optional Libraries

The framework's flexibility and modularity are heavily supported by a sophisticated configuration system, primarily accessed through the main `flexfl` CLI command. This system uses a powerful argument parser that allows users to specify the desired configuration for every aspect of the FL experiment, including selecting specific implementations for each of the seven core modules (ML Backend, FL Backend, Communication Layer, etc.) and providing their module-specific arguments. Configuration can be provided directly via command-line arguments or through a configuration file, enabling reproducible setups.

Addressing the challenge of managing a wide array of potential module options and avoiding library conflicts (e.g., between TensorFlow and PyTorch), the argument parser incorporates logic that inspects each module's available Python code implementations dynamically. By analyzing the Python syntax tree of the installed module classes, it automatically identifies configurable constructor arguments and their expected data types.

This information is used to configure the argument parser dynamically at runtime, ensuring that only valid options are presented and correctly parsed. This dynamic configuration enables lazy loading of module dependencies, so that the required libraries for a chosen module implementation are only imported when that module is actually instantiated, significantly improving startup performance and reducing potential conflicts in environments with multiple large ML libraries. A key benefit is that when new module implementations are added by extending the base classes, their constructor arguments are automatically recognized and incorporated by the argparser without requiring manual updates to the configuration system.

4.4.6 How Results Are Saved

Experimental results generated by the framework are systematically organized and saved to facilitate analysis and reproducibility. When an experiment is executed via the `flexfl` command, a dedicated output folder is created. The name of this folder is determined either by a user-provided name through the argument parser or defaults to a timestamp representing the start time of the Parameter Server process, ensuring unique identification for each run.

Several subdirectories and files are stored within this root experiment folder. For perfect reproducibility, a file containing the exact configuration arguments used for the run is saved. The best model checkpoint (as determined by the early stopping criteria or the final model if early stopping is disabled) is saved in a dedicated location. Furthermore, for each node (Parameter Server and each Worker) involved in the experiment, a subfolder named after the node's identifier is created.

Inside each node's folder, a comprehensive log file is generated in JavaScript Object Notation Lines (JSONL) format. This log file captures a rich stream of events and data throughout the experiment, including timestamps for send and receive operations, details about exchanged payloads, epoch-level training and validation metrics (loss, accuracy, etc.), worker computation times, and other relevant system events. This structured JSONL format allows subsequent analysis using external tools, as supported by the `flexfl-plot` utility.

4.4.7 How to Extend the Framework

Extending the framework with new implementations for any of the seven core modules is designed to be straightforward, leveraging the modular architecture and Python's object-oriented capabilities. The framework provides a set of abstract built-in base classes, with one abstract class defined for each core module type.

To add a new implementation, a user simply needs to create a new Python class that inherits from the corresponding abstract base class, which defines a set of required methods (interfaces) that the new implementation must provide. The user then writes the code for their specific algorithm, protocol, or component within these methods, adhering to the defined interfaces.

Once the new class is implemented, it becomes discoverable by the argument parser due to its dynamic inspection capabilities. This design ensures that adding new capabilities does not require modifying the core framework logic, promoting clean separation of concerns and fostering community contributions and customization.

CHAPTER 5

Evaluation

A series of experiments was designed and conducted in a controlled environment to rigorously assess the performance, resilience, and modularity of the proposed Resilient Federated Learning Framework. This chapter details the experimental setup, the datasets and models used, the metrics measured, the various experimental scenarios investigated, and the results and analysis for each scenario.

5.1 Experimental Setup

The experimental setup was designed to emulate scenarios involving resource-constrained client devices participating in a federated training process coordinated by a more powerful central server. This mirrors typical edge computing FL use cases where client devices (like mobile phones or IoT devices) have limited computational resources and unstable network connectivity compared to a central server or cloud instance.

The experiments were conducted on a cluster of VMs running on underlying ARM-based hardware, specifically the HUAWEI Kunpeng 920 7260 processor[1], with Ubuntu 24.04 LTS as the guest operating system. This cluster was hosted on a Proxmox server, which provided a flexible and efficient virtualization environment.

The cluster configuration for the experiments included a single VM designated as the Parameter Server, configured with 8 CPU cores and 16 GB of RAM, providing significantly more resources than the worker VMs, and with a network interface configured with a maximum bandwidth of approximately 5 GBps, simulating a high-capacity central node connection. This configuration ensured that the Parameter Server could handle the computational load of aggregation and avoid becoming a network bottleneck, allowing us to focus on communication challenges between the server and the workers.

To represent different client populations, two sets of worker VMs were used. The first, a uniform worker set, consisted of 10 VMs, each configured identically with 2 CPU cores

[1] `https://www.hisilicon.com/en/products/kunpeng/huawei-kunpeng/huawei-kunpeng-920`

and 4 GB of RAM, and a network interface bandwidth cap of 50 Mbps, simulating typical broadband or limited wireless connections for client devices.

The second, a heterogeneous worker set, was configured with varying specifications for CPU cores, RAM, and network bandwidth caps across the 30 VMs to simulate a diverse population of client devices with heterogeneous capabilities. This worker set consists of 6 groups of 5 VMs with varying CPU cores (2, 4, and 6), RAM (4, 6, and 8 GB), and network bandwidth (50, 75, and 100 Mbps).

Finally, accurately measuring communication times and synchronization events is vital for assessing the framework's performance and resilience. To ensure this, precise clock synchronization was essential across all participating nodes. The parameter server VM was configured to act as an Network Time Protocol (NTP) server, with all worker nodes synchronized to it. This setup minimized time drift between machines, allowing for the reliable calculation of communication latencies and event timestamps throughout the experiments.

Figure 5.1 illustrates the described experimental setup, showing the configuration of the Parameter Server and the two worker sets.

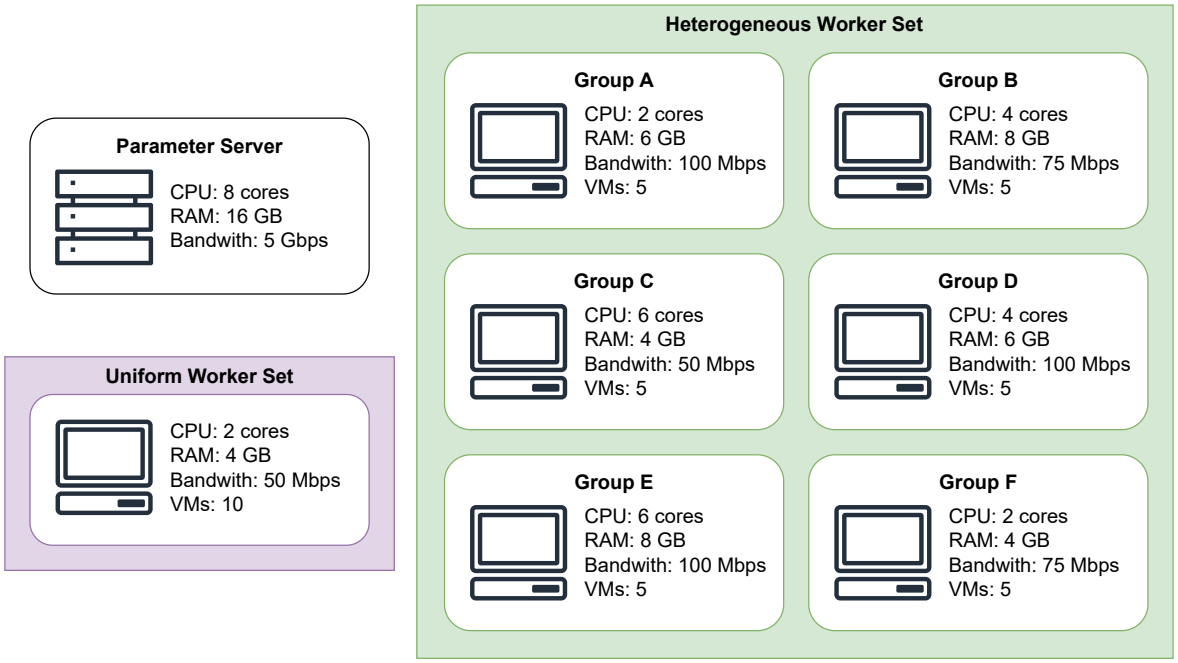


Figure 5.1: Illustration of the experimental setup, detailing the central Parameter Server and the two distinct sets of worker VMs: a uniform worker set with 10 VMs, and a heterogeneous worker set with 30 VMs comprising multiple groups with varying resource configurations.

5.2 Datasets and Models

Effective evaluation of a FL framework requires careful selection of datasets and models relevant to its intended real-world applications. IDS and IoT environments stand out as highly practical and pertinent domains for FL implementation. This is primarily due to the critical need for data privacy and security, alongside the inherent distributed nature of data sources

in these settings. Training robust IDS models often necessitates access to sensitive data, like network traffic or device information, which is challenging or impossible to centralize [9].

FL presents a viable solution by enabling collaborative model training among participants without requiring the sharing of raw data. Evaluating our framework within the context of IDS and IoT enables us to demonstrate its capabilities in scenarios that directly reflect these practical constraints and opportunities.

While well-established datasets like KDD98 [51], KDDCUP99 [52], or NSLKDD [53] exist for training IDS models, they are widely regarded as outdated and not fully representative of today's network traffic, as detailed in [54]. This significantly restricts their applicability to contemporary network security research. Considering this limitation, we decided to use two recent and widely-recognized state-of-the-art datasets, UNSW-NB15 [55] and ToN-IoT [56], which better capture current network threats and reflect IoT environments.

5.2.1 UNSW-NB15

The UNSW-NB15 dataset is synthetically generated from the Australian Centre for Cyber Security (ACCS). It combines real modern regular network traffic with synthesized contemporary attack activities. The dataset comprises over 2 million records, each with 49 features, and is suitable for both binary and multi-class classification tasks. Malicious activities are labeled with one of nine specific attack categories: Denial of Service (DoS), Analysis, Backdoor, Exploits, Fuzzers, Generic, Reconnaissance, Shellcode, and Worms.

To avoid complex preprocessing procedures, in this study, we used the NF-UNSW-NB15[2] preprocessed dataset published in [57] by the University of Queensland. This dataset version incorporates the same data as the original but standardizes features using NetFlow. NetFlow is a widely adopted Cisco protocol for collecting IP traffic information and monitoring network flow (e.g., total bytes transferred, number of packets, duration). The creators of this version state that their motivation was the lack of a standard feature set commonly found in general IDS datasets.

Following prior work on this dataset, specifically [58], which investigated various NNs including Artificial Neural Network (ANN), Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN) and found them to achieve similar results, we selected the ANN model. The ANN strikes a good balance between performance and computational efficiency, making it a suitable choice for potentially resource-limited FL workers. However, this ANN model is rather small, in [59] the authors used a larger model that we considered a better fit for our experiments.

The ANN model used in this study consists of three hidden layers (300, 100, and 50 neurons) using the Rectified Linear Unit (ReLU) activation function and a softmax for the output layer as shown in Figure 5.2.

[2] `https://espace.library.uq.edu.au/view/UQ:ffbb0c1`

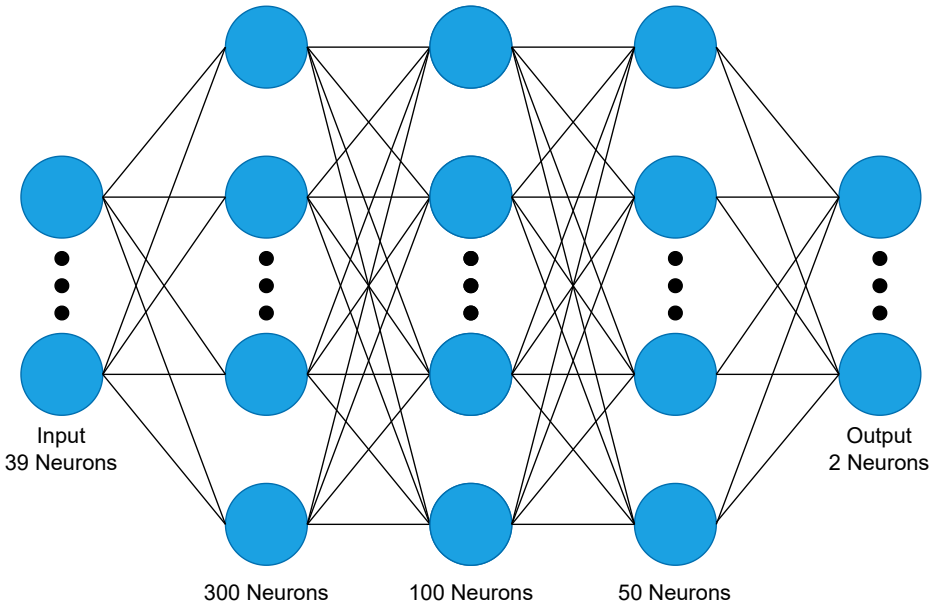


Figure 5.2: Architecture of the ANN model used for the UNSW-NB15 dataset, consisting of three hidden layers with 300, 100, and 50 neurons, respectively.

5.2.2 ToN-IoT

The ToN-IoT is a heterogeneous dataset created by the ACCS and released in 2019, specifically designed to include telemetry data from an IoT network. Generated on a realistic testbed, it provides various traces of IoT services, network traffic, and operating system logs. The dataset contains over 13 million samples and numerous attack scenarios, including Backdoor, DoS, Distributed Denial of Service (DDoS), Injection, Man In The Middle (MITM), Password, Ransomware, Scanning, and Cross-Site Scripting (XSS). Like UNSW-NB15, ToN-IoT supports both multi-class attack classification and binary normal/malicious classification.

For this study, we used the NetFlow version of the ToN-IoT dataset, NF-ToN-IoT[3], published by the same authors as the NF-UNSW-NB15 dataset.

Based on the model evaluations for the ToN-IoT dataset presented in [60], which compared a Sparse AutoEncoder (SAE) and an ANN model, we selected the ANN model. The ANN architecture used for this dataset features two hidden layers with 32 and 16 neurons, respectively, also employing the ReLU activation function and a softmax for the output layer. Dropout layers with a rate of 0.2 were included after each hidden layer. The model was trained using the Adam optimizer, mirroring the approach in the cited paper.

This model is smaller than the one used for UNSW-NB15, allowing it to exhibit distinctive characteristics and ensuring that the experiments are not biased by the model size. The architecture of the ANN model used for the ToN-IoT dataset is shown in Figure 5.3.

[3] `https://espace.library.uq.edu.au/view/UQ:38a2d07`

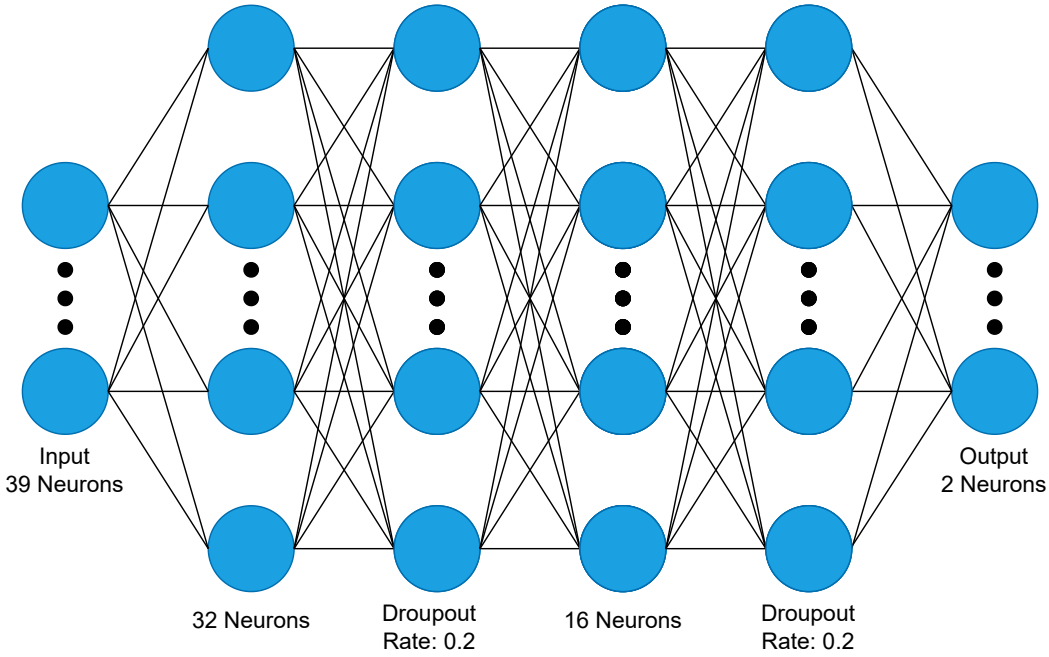


Figure 5.3: Architecture of the ANN model used for the ToN-IoT dataset, consisting of two hidden layers with 32 and 16 neurons, respectively, with dropout layers in between.

5.2.3 Data Division

Our experiments focused on the binary classification task using the UNSW-NB15 and ToN-IoT datasets, aiming to train models that can distinguish between normal and malicious network traffic. To prepare the data for the distributed training process, we first preprocessed the datasets, without data normalization, and split them into three primary partitions: a training set (70%), a validation set (15%), and a testing set (15%) for the UNSW-NB15 dataset, and a training set (90%), a validation set (5%), and a testing set (5%) for the ToN-IoT dataset, because this dataset is significantly larger than UNSW-NB15 (over 13 million samples compared to UNSW-NB15's 2 million). This division ensures that the model can be trained effectively while also being evaluated on unseen data.

The training data were then partitioned among the workers participating in each experiment's subpool according to specific distribution strategies, primarily IID, achieved by globally shuffling the dataset and distributing equal-sized partitions to each worker. A critical aspect of this data handling, consistent with the principles of FL and preserving data privacy, is that each worker node performed data normalization (e.g., scaling features) using only the statistics (such as mean and standard deviation) derived from its own local data split, without sharing raw data or statistics with other nodes or the Parameter Server.

In this FL setup, the validation set was used by the parameter server for periodic evaluation of the global model's performance throughout the training process. It is important to note that, for our experiments, the validation data was not used to influence the aggregation mechanism or global model updates, thereby ensuring the reported metrics effectively represent the model's generalization performance on unseen data, akin to a test set. The remaining testing

sets were reserved for future evaluation of the final trained model's performance on unseen data.

Furthermore, the resilience mechanisms integrated into the system, such as handling client dropouts, do not alter the aggregation logic or influence the model update process. This ensures consistency and integrity in the training dynamics. Nonetheless, the modular design of the FL Backend does allow for future integration of validation-aware aggregation strategies.

5.3 Metrics

Rigorous evaluation of a ML framework, particularly one designed for distributed and potentially unreliable environments like FL, needs a comprehensive set of metrics. These metrics fall into two main categories: standard ML performance metrics and system-level metrics derived from detailed logging.

Regarding the ML performance, the specific metrics evaluated depend on the type of task being performed (classification or regression) and the loss function used. For the binary classification tasks utilizing the UNSW-NB15 and ToN-IoT datasets in this study, the following standard classification metrics are evaluated: Matthews Correlation Coefficient (MCC), Accuracy, and F1-score [61], [62]. They serve as primary indicators of the model's learning progress, convergence, and final quality.

Accuracy provides the overall proportion of correct predictions, while the F1-score offers a balanced view of precision and recall, which is particularly important for tasks like intrusion detection, where both minimizing false positives and false negatives is crucial. MCC is a single-value metric that measures the quality of binary classifications, particularly useful for imbalanced datasets. It takes into account all four values in the confusion matrix: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). It is calculated as:

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}} \tag{5.1}$$

For completeness and to demonstrate the framework's versatility beyond classification, standard regression metrics such Symmetric Mean Absolute Percentage Error (SMAPE), Mean Squared Error (MSE), and Mean Absolute Error (MAE) [63] are also considered as potential evaluation criteria depending on the model and dataset used. MSE is the average of the squared differences between predicted and actual values, and MAE is the average of the absolute differences. SMAPE is a measure of prediction accuracy, expressed as a percentage, calculated as:

$$\text{SMAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|} \times 100 \tag{5.2}$$

Where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of data points.

For this evaluation, we primarily focused on global model validation metrics to assess the framework's overall convergence and generalization capabilities on unseen data, rather than

tracking training metrics from individual clients. This choice was made to avoid additional overhead, and more importantly, because local training metrics in a federated setting, especially under Non-IID data distributions, can be misleading and do not reliably reflect the global model's learning progress. In such scenarios, client-specific metrics often exhibit high variance and are not indicative of the model's ability to generalize, making global validation performance a more meaningful and consistent measure.

In addition to model performance, evaluating the framework's resilience, scalability, and efficiency requires collecting detailed system-level data. As discussed in Section 4.4.6, the framework generates comprehensive JSONL log files for each node (Parameter Server and Workers).

These logs capture a rich stream of events and data points throughout the experiment, including timestamps for send and receive operations of messages, details about exchanged payloads including size and type, epoch-level training and validation metrics (loss, accuracy, etc.), worker computation times for local training tasks, encode and decode times for messages, events indicating when workers join or leave the network, events indicating when workers fail, and markers for the start and end of the entire training run.

This detailed logging allows for the measurement and analysis of various system performance aspects, including the total run duration of the experiment, the evolution of loss and other ML metrics over epochs, the number of messages sent and received and the total payload size exchanged, the total communication time across the network, and the total worker computation time and idle/wait time for both the Parameter Server and workers.

It also allows tracking and quantifying the types and timing of worker failures and their corresponding states during a round. This is crucial because it will enable us to differentiate between failures that halt progress (critical failures) and those that are less disruptive (non-critical failures). This provides empirical evidence for the framework's resilience claim by demonstrating how well it continues to train despite various failure scenarios. A worker's status during a round can be categorized into one of five states:

i) **Idle without fail:** The worker was available and did not experience a failure event while idle waiting for a task or the next round.
ii) **Worked successfully without failing:** The worker was assigned a local training task, completed it, and successfully communicated its update to the server without experiencing a failure event during this epoch.
iii) **Failed while idle:** A failure event occurred while the worker was available but not actively performing a local training task or communicating. This is generally less disruptive as no computation result is lost. This is considered a **non-critical failure** as the worker can rejoin the training process without any loss of contribution.
iv) **Failed while working:** A failure occurred while the worker actively performed a local training task or communicated their update. This is considered a **critical failure** as the ongoing computation or communication is interrupted, potentially requiring task reassignment in an asynchronous setting.

v) **Failed after working successfully:** The worker completed its local training task and successfully communicated its update, but experienced a failure event afterward, before the start of the next round. This is considered a **non-critical failure** as the contribution for the current round was successfully made.

These metrics provide insights into convergence speed and stability under different conditions, the impact of worker failures on training progress, communication overhead, and the overall efficiency of the framework. By analyzing these metrics, we can assess the framework's performance in terms of resilience, scalability, and efficiency, as well as its ability to adapt to varying worker capabilities and network conditions.

5.4 Experimental Scenarios

A series of experimental scenarios was designed and executed to comprehensively assess the performance, resilience, modularity, and scalability of the proposed Resilient Federated Learning Framework. These scenarios evaluate the framework's capabilities under different conditions, focusing on key aspects relevant to real-world FL deployments, such as communication efficiency, fault tolerance, and scaling with a dynamic and heterogeneous worker pool. Three primary scenarios were investigated in this evaluation.

Scenario 1: Impact of Communication Protocols and FL Algorithms - This scenario is designed to measure and compare the communication overhead associated with different FL algorithms implemented within the framework using the UNSW-NB15 dataset and aims to understand inherent communication costs. This is particularly important for identifying the most efficient approaches in bandwidth-limited environments, which are typical of edge computing scenarios where FL is often applied.

This scenario uses the first set of workers, consisting of 10 VMs with uniform specifications, connected to the Parameter Server. This setup allows for a controlled environment and better visualization of results.

Scenario 2: Resilience Evaluation under Worker Failures - This scenario focuses on evaluating the framework's resilience mechanisms and their effectiveness in handling worker failures or disconnections across different communication protocols. By simulating worker failures while employing various communication protocols and training on the UNSW-NB15 dataset, we aim to assess how the framework maintains training continuity and performance. This scenario also uses the uniform worker set (10 VMs).

Scenario 3: Scalability and Convergence with Large-Scale Failures - This scenario evaluates the framework's scalability and overall robustness when operating with a larger and more heterogeneous worker pool while introducing simulated worker failures. By using both the uniform and heterogeneous worker sets (a total of 40 VMs with diverse specifications and network conditions) and conducting experiments with both the UNSW-NB15 and ToN-IoT datasets, this scenario provides a more realistic representation of a large-scale, dynamic FL environment. The goal is to demonstrate that the framework can scale effectively and that

the training process can still converge and achieve acceptable model performance despite numerous worker failures and disconnections.

In scenarios where failures were introduced, these were implemented as probabilistic worker failures occurring at different configured rates. Workers perform a graceful exit upon failure, attempting to rejoin the training process shortly after the failure event, mimicking realistic client behavior. This ensures that logs are saved and no results are lost.

Across these scenarios, a set of common hyperparameters and configurations was maintained to ensure consistency and comparability, while specific parameters were adjusted as needed for the targeted evaluation of each scenario. Table 5.1 summarizes the common hyperparameters, which are fully tunable, used throughout the evaluation. These parameters are aligned with commonly adopted settings in state-of-the-art ML training, ensuring the relevance and generalizability of our experimental results.

Table 5.1: Core hyperparameters applied across the FL experiments, representing a typical configuration used throughout the evaluation scenarios.

Hyperparameter	**Value**
ML Backend	Keras with TensorFlow
Optimizer	Adam
Loss function	Categorical Crossentropy
Batch size	1024
Learning rate	0.0001
Number of Epochs	10
Local Epochs per worker	3
Worker threshold	50%
Workers per epoch	70%

These scenarios, coupled with the detailed metric collection and analysis, provide a robust framework for evaluating the Resilient Federated Learning Framework's capabilities and validating its design principles against the challenges of real-world distributed environments.

All experimental results are publicly available on the framework's GitHub [4] repository for full transparency and reproducibility.

5.5 Scenario 1: Impact of Communication Protocols and FL Algorithms

This scenario was designed to comprehensively evaluate the performance implications of selecting different communication protocols and FL algorithms within the proposed framework. The primary goal is to quantify the communication overhead and execution times associated with various combinations, particularly highlighting how these factors interact in a controlled environment, such as the uniform worker setting, where network conditions are consistent across participants.

[4] `https://github.com/leoalmPT/FlexFL/tree/dissertation`

This analysis is crucial for understanding the inherent performance characteristics of the framework's core components and informing appropriate protocol and algorithm choices for specific deployment environments, especially those with bandwidth or latency constraints typical of edge computing.

To achieve this, we tested all four implemented communication protocols: Zenoh, MQTT, Kafka, and MPI. These protocols were evaluated in conjunction with the four primary FL algorithm paradigms supported by the framework: Centralized Synchronous, Centralized Asynchronous, Decentralized Synchronous, and Decentralized Asynchronous.

It's important to recall the fundamental difference in communication patterns between Centralized and Decentralized FL. Centralized approaches typically involve workers sending gradients to a central Parameter Server frequently. In a Centralized Synchronous setting, this aggregation occurs after a fixed number of worker batches have been processed, while in Centralized Asynchronous, the server processes updates from individual batches as they arrive.

Decentralized approaches, conversely, involve workers exchanging their full local models or weights less frequently, just once per training round or epoch. This distinction significantly impacts the volume and frequency of messages exchanged, potentially making Centralized approaches more susceptible to communication bottlenecks at the Parameter Server.

Furthermore, due to the design characteristics of the MPI protocol, particularly its reliance on synchronous, blocking communication and lack of native support for a callback-based system, its evaluation was limited to the Decentralized Synchronous setting, to avoid incorrect time measurements and to ensure a fair comparison with the other protocols. Asynchronous communication is possible with MPI, but it requires constant checking for incoming messages, adding considerable overhead. While this restricts a complete comparison across all algorithm types, it still serves as a valuable benchmark for evaluating the performance of other protocols, given its popularity in HPC.

The performance was assessed using the metrics defined in Section 5.3: average communication time per worker, average working time per worker (local computation time), and average total run time for the entire experiment. The results presented here represent the average of three independent runs for each protocol-algorithm combination after a warmup run. These results are shown in Table 5.2 and Table 5.3.

Table 5.2: Average communication volume metrics, including the number of messages and total payload size per worker, for various communication protocols and FL approaches evaluated in Scenario 1.

FL Approach	Comm Protocol	Average Number Of Messages	Average Total Payload Size (MB)
Decentralized Synchronous	MPI	17.4 ± 0.932	2.71 ± 0.164
	Zenoh	17.4 ± 0.932	2.71 ± 0.164
	MQTT	17.4 ± 0.932	2.71 ± 0.164
	Kafka	17.4 ± 0.932	2.71 ± 0.164
Decentralized Asynchronous	Zenoh	17.4 ± 0.932	2.71 ± 0.164
	MQTT	17.4 ± 0.932	2.71 ± 0.164
	Kafka	17.4 ± 1.302	2.71 ± 0.229
Centralized Synchronous	Zenoh	2285.4 ± 2.044	403.0 ± 0.360
	MQTT	2285.4 ± 1.935	403.0 ± 0.309
	Kafka	2285.4 ± 1.069	403.0 ± 0.188
Centralized Asynchronous	Zenoh	2285.4 ± 1.302	403.0 ± 0.229
	MQTT	2285.4 ± 1.975	403.0 ± 0.348
	Kafka	2285.4 ± 1.302	403.0 ± 0.229

Table 5.3: Average time metrics per worker, including communication time, working time, and total run time, for different communication protocols and FL approaches evaluated in Scenario 1.

FL Approach	Comm Protocol	Average Comm Time (s)	Average Work Time (s)	Average Total Run Time (s)
Decentralized Synchronous	MPI	0.422 ± 0.094	83.92 ± 13.54	139.6 ± 17.88
	Zenoh	0.097 ± 0.021	82.69 ± 7.823	136.7 ± 5.091
	MQTT	0.115 ± 0.022	82.51 ± 7.927	138.1 ± 8.274
	Kafka	1.705 ± 0.213	83.43 ± 7.800	139.9 ± 4.286
Decentralized Asynchronous	Zenoh	0.079 ± 0.014	82.45 ± 8.541	114.2 ± 1.436
	MQTT	0.098 ± 0.016	82.71 ± 7.866	114.8 ± 1.519
	Kafka	1.690 ± 0.182	83.20 ± 7.322	117.6 ± 0.276
Centralized Synchronous	Zenoh	23.35 ± 5.524	45.35 ± 2.023	269.2 ± 1.609
	MQTT	25.39 ± 3.492	48.39 ± 3.820	250.9 ± 26.09
	Kafka	360.3 ± 2.206	47.36 ± 2.408	1288.2 ± 6.104
Centralized Asynchronous	Zenoh	9.737 ± 0.248	47.19 ± 2.287	728.6 ± 17.57
	MQTT	12.77 ± 0.195	46.58 ± 2.796	744.2 ± 15.76
	Kafka	253.8 ± 3.879	48.37 ± 2.251	1865.0 ± 19.69

5.5.1 Federated Learning Algorithms Comparison

As expected based on the principles mentioned earlier, the Decentralized Asynchronous approach consistently achieves a lower average total run time compared to the Decentralized Synchronous approach across all evaluated protocols (e.g., with Zenoh, 114 vs. 136 seconds, respectively). To illustrate this, Figure 5.4 and Figure 5.5 show the timeline of the Decentralized Synchronous and Asynchronous approaches, respectively.

We can observe the in the first case the parameter server waits for all workers to finish their local training before aggregating the updates, while in the second case, the workers can send their updates to the parameter server as soon as they finish their local training, allowing for a more efficient use of resources, avoiding the straggler problem and reducing the overall training time.

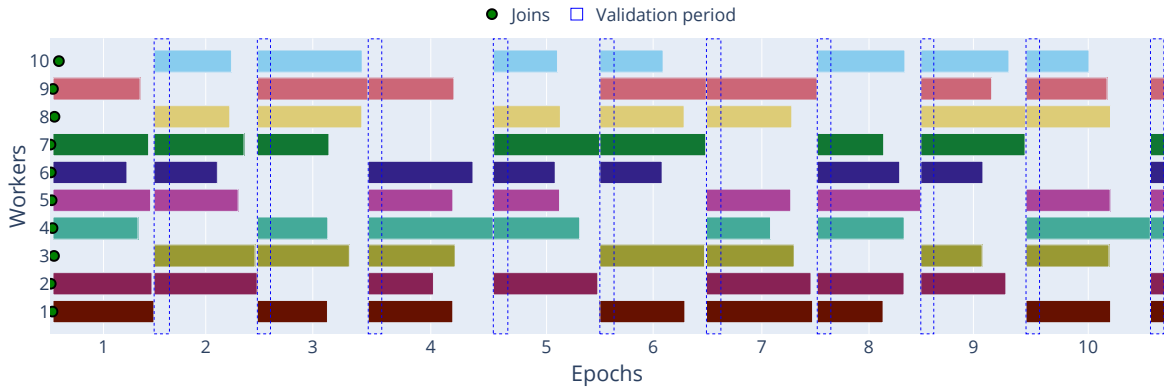


Figure 5.4: Timeline illustrating worker activity and idle periods across training epochs in the Decentralized Synchronous FL approach with Zenoh, showing waiting for synchronization.

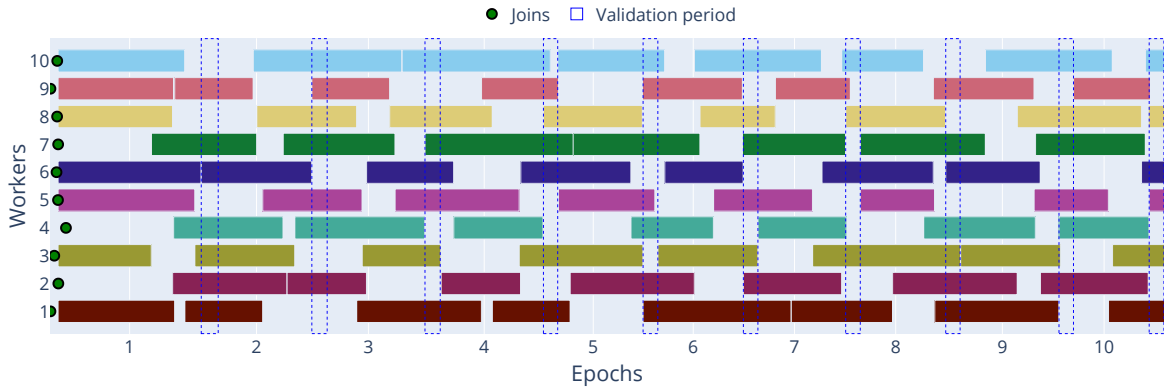


Figure 5.5: Timeline illustrating worker activity and idle periods across training epochs in the Decentralized Asynchronous FL approach with Zenoh, demonstrating independent task completion and update submission.

This inherent advantage of asynchronous FL is particularly critical in dynamic edge environments characterized by significant device heterogeneity, where worker speeds can vary drastically. When a worker joins, metadata is shared with the Parameter Server, this metadata can include information regarding their computational capabilities (e.g., CPU, memory, or GPU availability) and current network conditions (e.g., bandwidth and latency) and can

be used in our modular Framework to implement more sophisticated scheduling policies to improve performance further.

As anticipated, Centralized methods involve a significantly higher frequency and volume of communication compared to Decentralized methods. For instance, across the Centralized experiments, the Parameter Server and workers exchanged a substantially larger number of messages (2285) and total payload size (403 MBs) compared to the Decentralized experiments, with 17 messages and a 2.7 MBs payload size.

We can also observe that workers in the Centralized approaches spend significantly less time on local computation compared to Decentralized workers. This discrepancy indicates that the server acts as a bottleneck in Centralized training, with workers spending considerable time idle or waiting for the server to aggregate updates and distribute the new global model, decreasing the local computation time from ≈ 83 seconds in Decentralized approaches to ≈ 47 seconds in Centralized approaches across all communication protocols.

This bottleneck effect was particularly noticeable in the Centralized Asynchronous setting, where the server must process every batch update from workers individually and immediately, potentially leading to increased server load and worker idle time, increasing the Total Run Time from 114 seconds in Decentralized Asynchronous with MQTT to 744 seconds in the Centralized Asynchronous approach with the same communication protocol.

In asynchronous approaches, the number of messages exchanged and the total payload size vary from run to run, as reflected in standard deviations, due to the system dynamically adapting to worker availability. If a worker takes longer to complete their local training, other workers may complete more tasks, leading to a higher number of messages and a larger payload size. Conversely, in synchronous approaches, where workers proceed in lockstep, the number of messages and total payload size are expected to be the same across runs.

This is the case for the Decentralized Synchronous approach, but not for the Centralized Synchronous approach. This is due to the number of iterations per epoch being determined by the total number of batches processed by the selected workers, if the dynamic worker pool selection results in different sets of workers (and thus potentially different total numbers of batches), it can introduce variability in the total number of iterations per round and consequently lead to varying numbers of messages and payload sizes.

5.5.2 Communication Protocols Comparison

Observing the results in Table 5.3, distinct patterns emerge regarding the impact of the chosen communication protocol. Zenoh and MQTT consistently exhibit lower average communication times across the tested FL algorithm types compared to Kafka. Zenoh generally showed slightly lower communication times than MQTT, although they were often closely comparable.

MPI, evaluated only in the Decentralized Synchronous context, showed higher communication times than Zenoh and MQTT in that specific setting, and Kafka demonstrated the highest communication times across all paradigms where it was tested.

This difference in communication performance is particularly evident when comparing

the Centralized and Decentralized FL approaches. Given the higher frequency and volume of communication, protocols with higher inherent latency, like Kafka, have a more pronounced negative impact on the overall communication time, achieving an average communication time of 360 seconds in the Centralized Synchronous compared to 23 and 25 seconds for Zenoh and MQTT, respectively.

In terms of average working time per worker, the communication protocol generally had a relatively similar impact across all protocols within the same FL approach. This suggests that the time workers spend on local computation tasks is mainly independent of the communication method chosen, assuming the network does not become a severe bottleneck, completely halting progress.

Further analysis of Zenoh and MQTT revealed that the average communication time was slightly higher in the Synchronous setting compared to the Asynchronous one. This observation suggests that the simultaneous arrival of synchronous updates from multiple workers at the central parameter server or broker might induce additional processing or queuing delays, which are then reflected in the increased communication time.

Finally, analyzing the standard deviations across multiple runs provides valuable insights into the consistency and reliability of the framework's performance under different configurations. Low standard deviation values generally indicate that the experimental runs for a given configuration produced consistent results with no significant outliers, suggesting a stable and predictable behavior.

5.6 Scenario 2: Resilience Evaluation under Worker Failures

Building upon the foundational analysis of communication protocols and algorithms in Scenario 1, this scenario focuses specifically on evaluating the resilience mechanisms of the proposed Resilient Federated Learning Framework under simulated worker failures. The primary objective is to demonstrate the framework's ability to maintain training continuity and achieve model convergence even when individual worker nodes experience unexpected disconnections or failures.

The evaluation focused on the communication protocols best suited for dynamic and potentially unreliable network environments: Zenoh, MQTT, and Kafka with the Decentralized Asynchronous FL approach. This allows the training to proceed as long as there are active workers, processing updates as they arrive and naturally handling stragglers or failed nodes without waiting, by reassigning tasks and dynamically adapting to the available worker pool, which is crucial for resilience in dynamic environments.

To simulate worker failures, we introduced probabilistic failures into the system. Each worker was configured to have a certain probability of experiencing a failure event per second. We tested three different failure rates: 0.5%, 1%, and 3% probability of failure per worker per second.

When a failure occurred, the worker performed a graceful exit, simulating a planned shutdown or disengagement from the network, to ensure that logs are saved. Importantly, the framework was configured to allow failed workers to attempt to rejoin the training process

shortly after the failure event, mimicking the behavior of real-world devices that might temporarily lose connectivity but eventually return.

To quantify the impact and handling of failures, we used the detailed system-level metrics, particularly focusing on the types of worker states described in Section 5.3:

- **Non-critical failures:** These include *iii: Failed while idle* and *v: Failed after working successfully*. These failures are generally less disruptive as they do not interrupt an ongoing task that was assigned to the worker for the current round. The worker's contribution for the round, if any, was already submitted, or no task was missed.
- **Critical failures:** This refers to the *iv: Failed while working* state, where a worker fails while actively performing its assigned task (local computation or communication). This type of failure is more disruptive as the partial or completed work for that task is lost, potentially requiring rescheduling or impacting the aggregation process, depending on the algorithm and framework's handling mechanisms.

Additionally, the other two states, *i: Idle without fail* and *ii: Worked successfully without failing*, are not directly related to failures but provide context for the worker's activity during the training process.

Table 5.4 presents a summary of the observed failure counts across the evaluated communication protocols and failure rates in the Decentralized Asynchronous setting.

Table 5.4: Summary of observed worker failures during training with the Decentralized Asynchronous approach, detailing counts of non-critical, critical, and total failures, alongside the total run duration for each communication protocol and failure rate.

Failure Rate	Comm Protocol	Non-Critical Failures	Critical Failures	Total Failures	Total Run Duration (s)
0.5% every Second	Zenoh	3	5	8	123.3
	MQTT	2	6	8	125.0
	Kafka	1	7	8	121.6
1% every Second	Zenoh	3	10	13	123.8
	MQTT	2	12	14	129.5
	Kafka	5	8	13	124.1
3% every Second	Zenoh	8	30	38	149.4
	MQTT	2	38	40	155.7
	Kafka	7	37	44	170.6

The results from this scenario provide strong empirical evidence for the framework's resilience. A key observation is that all experimental runs, regardless of the communication protocol or failure rate, completed successfully. This demonstrates that the framework's built-in mechanisms, such as the dynamic worker pool, task rescheduling for asynchronous approaches, and the Parameter Server's ability to handle sporadic updates, are effective in allowing the training process to continue and converge despite the presence of worker failures.

As expected, the total number of observed failures increased significantly as the probabilistic failure rate was increased from 0.5% to 3%. While the runs for each configuration were initiated with the same random seed to promote consistency, slight variations in the exact number of failures occurred across different runs.

This variability is inherent in the dynamic nature of the simulation, where factors such as the precise timing of failure events, the varying working times of individual workers, and the dynamic selection of workers for the subpool in each round can influence which workers are active and thus susceptible to failure at any given moment.

It is worth noting that a 1% and 3% failure rate per second represents an extremely chaotic and likely unrealistic scenario for most real-world deployments, serving as a stress test for the framework's robustness.

As anticipated, introducing failures led to a modest increase in the average total run time compared to the no-failure baseline. For the 0.5% and 1% failure rates, the increase was relatively small, varying with the training progress that needs to be rescheduled due to Critical Failures.

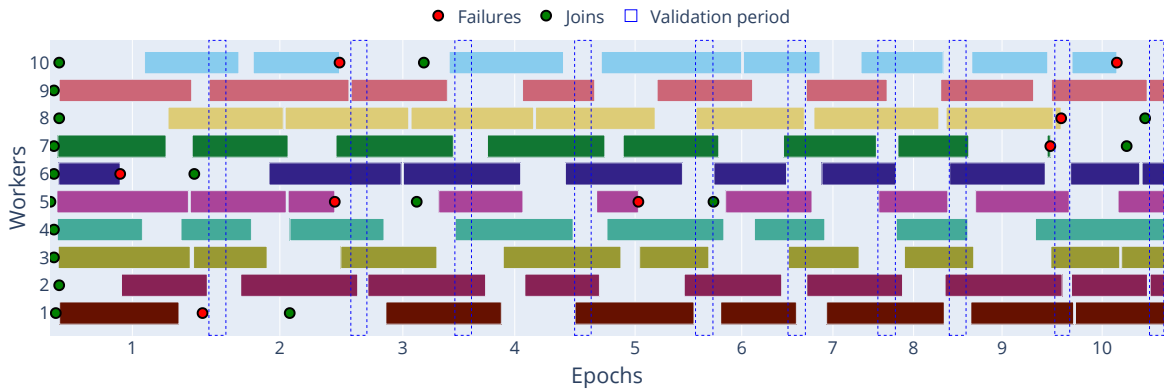
However, at the 3% failure rate, the increase in run time was more pronounced, this greater increase can be attributed not only to the higher volume of critical failures and subsequent task rescheduling but also to periods where the training process might have paused. The framework is designed to ensure a minimum number of active workers are available in each round to maintain training, if the dynamic worker pool drops below this threshold (7 workers or 70% of the 10 workers), the parameter server pauses the training, by not sending new tasks and waits for enough workers to become available to proceed.

To provide a deeper understanding of how failures were handled and their impact on the training process, we visualize the results through timelines, worker state matrices, and validation loss curves. Figure 5.6, Figure 5.7, and Figure 5.8 illustrate the results for the Kafka, MQTT, and Zenoh protocols, for the 0.5%, 1%, and 3% failure rates, respectively. These visualizations offer a comprehensive overview of the behavior of various communication protocols under different failure scenarios.

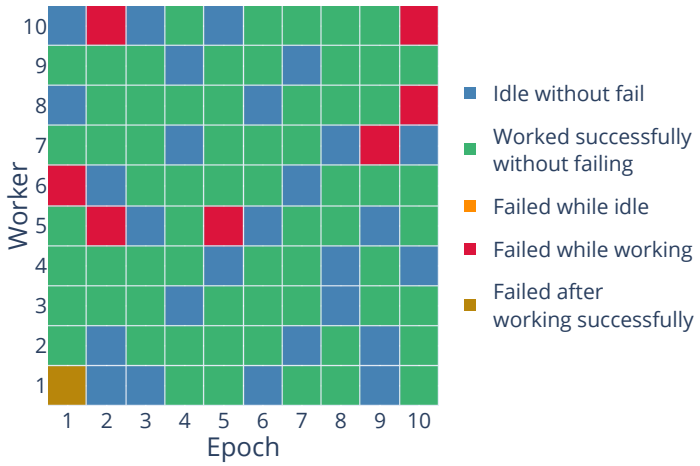
We can observe that the timelines show the activity (working, idle) of each worker over time, clearly marking when failure events occurred and when workers rejoined the network. They also illustrate how the Parameter Server reacted to critical failures, such as rescheduling tasks in the Decentralized Asynchronous setting. Specifically, when a worker fails while working or finishes their task, the Parameter Server sends a new task to another worker as soon as possible, allowing the training to continue without significant delays.

In cases where the number of active workers dropped below the threshold (7 workers), the Parameter Server paused the training until enough workers were available to proceed. This is evident in Figure 5.8a at epoch 7, as soon as the number of workers reached the threshold (when worker 5 joined), the parameter server resumed the training process by sending new tasks to 5 workers $\{1, 4, 5, 7, 8\}$, while 2 were already working on their tasks $\{2, 3\}$ and the remaining 3 workers were offline due to failures $\{6, 9, 10\}$.

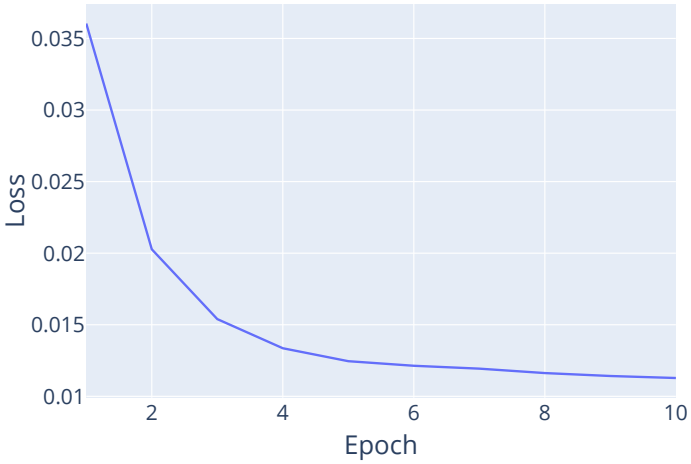
The worker state matrices offer a clear, epoch-by-epoch view of each worker's state (Idle



(a) Timeline of worker activity and failure events.



(b) Worker state matrix across training epochs.

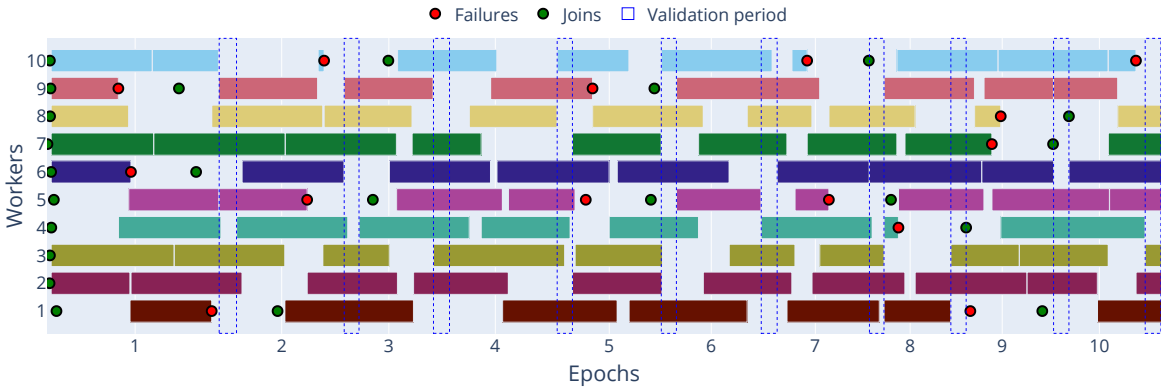


(c) Validation loss curve.

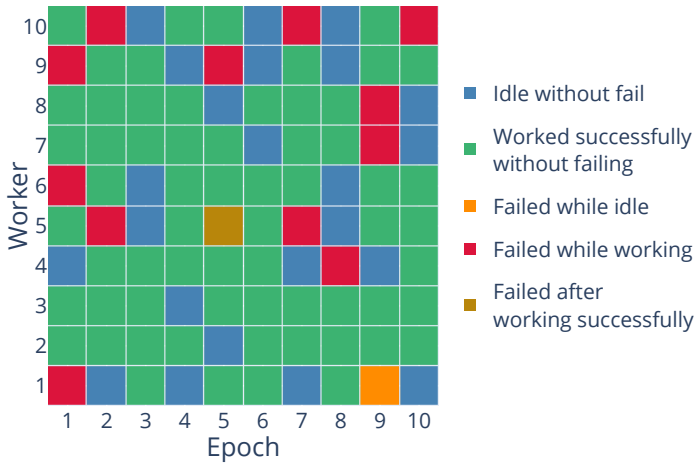
Figure 5.6: Training results with the Kafka protocol (0.5% failure rate).

without fail, Worked successfully, Failed while idle, Failed while working, Failed after working successfully), making it easier to evaluate the frequency and type of failures experienced by individual workers throughout the experiment.

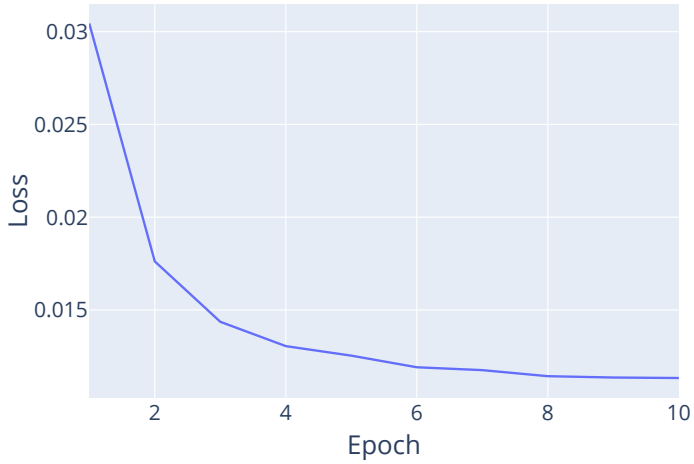
Finally, plots showing the validation loss curve over epochs for representative runs (Kafka with 0.5% failure rate, MQTT with 1% failure rate, and Zenoh with 3% failure rate) visually



(a) Timeline of worker activity and failure events.



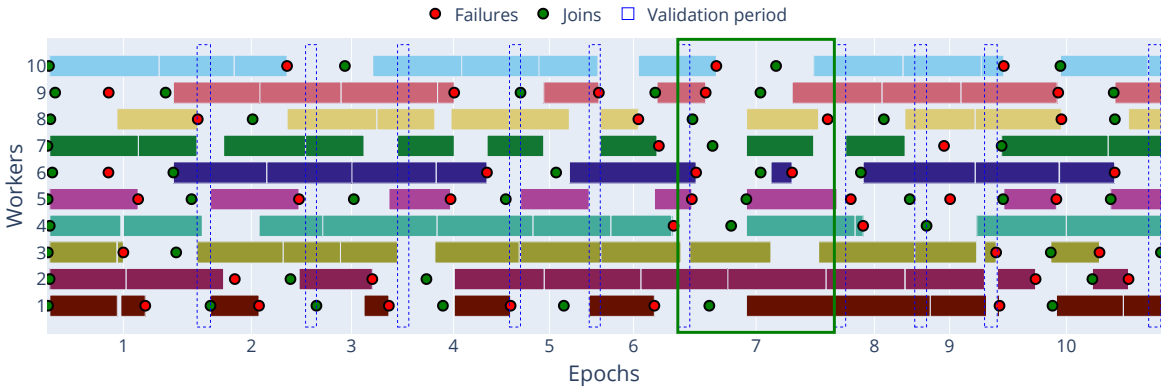
(b) Worker state matrix across training epochs.



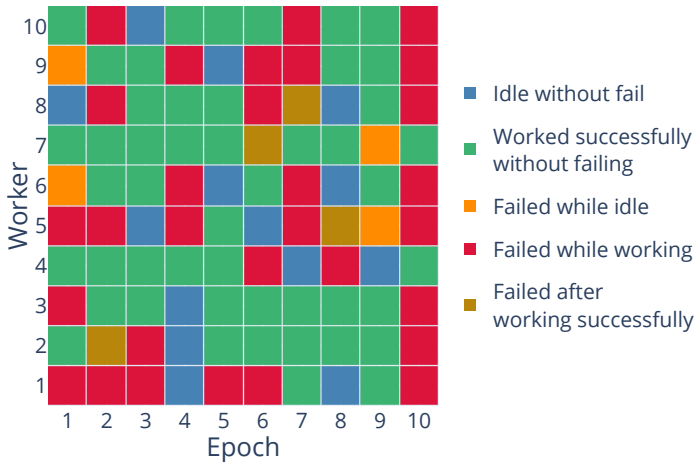
(c) Validation loss curve.

Figure 5.7: Training results with the MQTT protocol (1% failure rate).

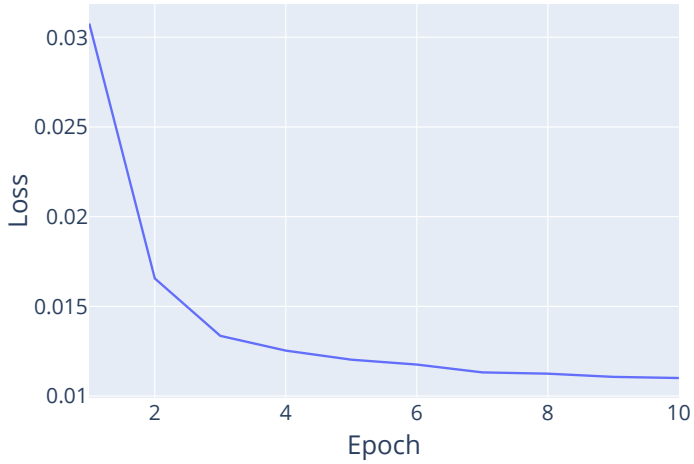
confirm that the model consistently converged, even under the highest simulated failure rate, demonstrating that the framework's resilience mechanisms effectively preserved the integrity of the training process and allowed the model to learn despite the disruptions.



(a) Timeline of worker activity and failure events.



(b) Worker state matrix across training epochs.



(c) Validation loss curve.

Figure 5.8: Training results with the Zenoh protocol (3% failure rate).

5.7 Scenario 3: Scalability and Convergence with Large-Scale Failures

This final scenario is designed to push the boundaries of the Resilient Federated Learning Framework by evaluating its scalability and overall robustness when operating with a significantly larger worker pool and simultaneously introducing substantial simulated failures. The

objective is to demonstrate that the framework can effectively scale the training process and achieve model convergence even in a large-scale, dynamic FL environment where numerous worker nodes are experiencing disruptions.

For this evaluation, we used the combined worker set, comprising both the uniform set (10 VMs) and the heterogeneous set (30 VMs), totaling 40 worker VMs. This configuration provides a more realistic representation of real-world FL deployments, involving a diverse group of devices with varying computational capabilities and network conditions.

Given the significantly larger worker pool (40 VMs compared to 10 in previous scenarios), the total dataset was partitioned across a greater number of devices. Consequently, each worker had access to a smaller local data subset. To ensure robust model convergence and to adequately compensate for the reduced training data available per client in each round, the number of training epochs was increased from 10 to 20. This adjustment enables more global aggregation steps, which is crucial for achieving comparable model quality and convergence in environments with distributed and sparse data partitions.

Based on the findings from Scenario 1 and Scenario 2, we selected the Decentralized Asynchronous algorithm due to its inherent suitability for dynamic environments and resilience to failures, and the Zenoh communication protocol, which consistently exhibited lower communication overhead and native support for dynamic participant management. This combination is expected to be the most robust for large-scale, failure-prone scenarios.

The experiments in this scenario were conducted using both the UNSW-NB15 and ToN-IoT datasets. Evaluating the framework with two distinct datasets demonstrates its versatility and ability to handle different data characteristics and problem domains at scale. Both datasets were configured for the binary classification task, as described in Section 5.2, with data partitioned using the IID strategy.

Simulated worker failures were introduced at a 1% probabilistic failure rate per worker per second. This rate represents a significant level of churn and disruption across the 40 worker nodes, testing the framework's ability to maintain continuity under considerable stress.

Table 5.5 presents the average counts of non-critical, critical, and total failures observed across runs for each dataset at the 1% failure rate.

Table 5.5: Failure counts observed during training with the Decentralized Asynchronous approach and a 1% failure rate with 40 workers, detailing non-critical, critical, and total failures, alongside the total run duration for both the UNSW-NB15 and ToN-IoT datasets.

Dataset	Non-Critical Failures	Critical Failures	Total Failures	Total Run Duration (s)
UNSW-NB15	9	8	17	62.6
ToN-IoT	20	27	47	122.5

Similar to Scenario 2, all experimental runs in this scenario, for both datasets and with the 40-worker setup and 1% failure rate, completed successfully. This further validates the framework's resilience mechanisms, which effectively managed the dynamic worker pool and

allowed for continuous training despite the presence of numerous worker failures. This is a critical finding, demonstrating that the framework scales effectively and maintains its resilience properties even under increased load and disruption.

Comparing the total run durations, for the UNSW-NB15 dataset, the total run duration of 62 seconds compared to 123 seconds in Scenario 2 with 10 workers and the same failure rate, this counter-intuitive result, despite the increased number of epochs (20 vs. 10), is primarily due to the data partitioning strategy. With 40 workers instead of 10, each worker processed approximately one-quarter of the total training data per local epoch. While the total number of global epochs doubled, the reduced local training time per worker per epoch, combined with the asynchronous nature of the algorithm, allowed for continuous progress without waiting for all stragglers, enabling a faster overall completion due to increased parallelization.

Conversely, the ToN-IoT dataset resulted in a significantly longer average total run duration of 122.5 seconds compared to UNSW-NB15 (62.6 seconds) in this scenario. As noted in Section 5.2, the ToN-IoT dataset is considerably larger than UNSW-NB15 (over 13 million samples versus 2 million), and despite the ToN-IoT model having a smaller architecture, the sheer volume of data in ToN-IoT requires more extensive local computation per worker, leading to a longer overall training duration.

To visually illustrate the training process at scale, Figures 5.9 and 5.10 show the validation loss and other key epoch metrics over training epochs for a representative run for the UNSW-NB15 and ToN-IoT datasets, respectively.

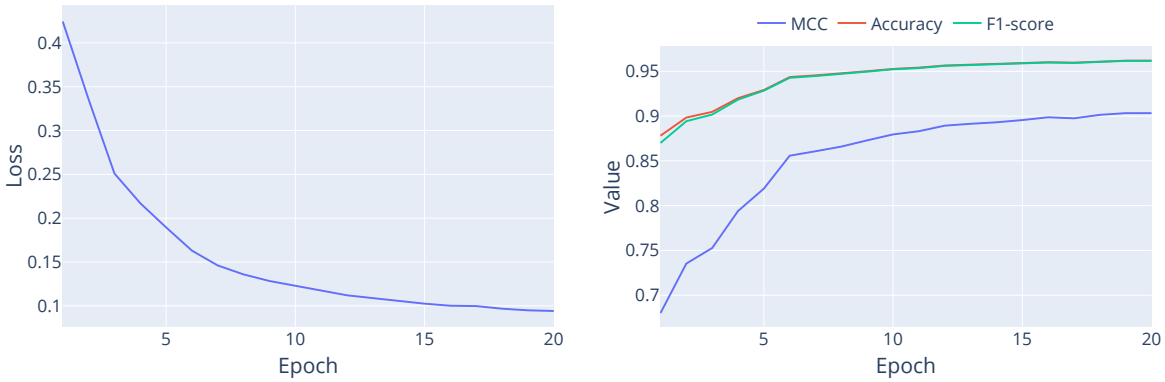
(a) Validation loss curve for UNSW-NB15 dataset.

(b) Epoch metrics for UNSW-NB15 dataset.

Figure 5.9: Training results with the UNSW-NB15 dataset using Decentralized Asynchronous approach and Zenoh protocol with 40 workers and 1% failure rate.

These plots provide visual confirmation of the framework's ability to maintain stable training and achieve model convergence even under the demanding conditions of Scenario 3, characterized by a large and heterogeneous worker pool and frequent simulated failures.

For both datasets, the validation loss curves consistently decrease over epochs, indicating that the models are effectively learning and improving. Simultaneously, the MCC, Accuracy, and F1-score metrics exhibit a clear upward trend, converging towards high-performance values, which demonstrates the framework's robustness in ensuring model quality despite disruptions.



(a) Validation loss curve for ToN-IoT dataset.

(b) Epoch metrics for ToN-IoT dataset.

Figure 5.10: Training results with the ToN-IoT dataset using Decentralized Asynchronous approach and Zenoh protocol with 40 workers and 1% failure rate.

The successful completion of all runs in this challenging scenario, coupled with clear evidence of model convergence and high final performance metrics, validates the core design principles of the Resilient Federated Learning Framework. It proves its capability to effectively manage dynamic worker participation, tolerate significant node failures, and adapt to varying data and network conditions.

For the UNSW-NB15 dataset, the final MCC achieved was 0.944, and for the ToN-IoT dataset, it was 0.903, further underlining the effectiveness of the training process even under adverse conditions. This comprehensive evaluation demonstrates that the framework is a robust and scalable solution for real-world Federated Learning deployments, particularly in edge computing environments where resilience and efficiency are paramount.

CHAPTER 6

Conclusion

This chapter summarizes the key contributions of the Resilient Federated Learning Framework, discusses its limitations, and proposes directions for future research, reinforcing its role in advancing robust and scalable FL deployments.

6.1 Findings and Accomplishments

The growing prevalence of AI applications in dynamic, resource-constrained edge environments, significantly enabled by technologies like 5G/6G and the proliferation of IoT devices, underscores the critical need for robust and scalable distributed ML paradigms such as FL.

However, deploying FL effectively in these real-world settings presents considerable technical challenges, particularly concerning fault tolerance, elasticity to dynamic participation, and communication efficiency under variable network conditions. This dissertation addressed these key issues by proposing, implementing, and evaluating a novel FL framework explicitly designed for resilience and modularity. Guided by the challenges, this research specifically aimed to answer:

- **RQ1:** How can Federated Learning frameworks be designed to ensure robustness against node failures across dynamic network environments?
- **RQ2:** What communication layer architectures are most suitable for supporting fault-tolerant and resilient Federated Learning under dynamic network conditions?

The core of our proposed framework is its highly modular architecture, comprising seven decoupled components responsible for distinct aspects of the FL process, including communication layer, FL algorithms, and worker management. This design shows remarkable flexibility, allowing for straightforward integration and experimentation with various ML libraries, communication protocols, and FL strategies without requiring extensive modifications to the entire system.

This modularity directly contributes to the framework's adaptability, enabling it to be tailored to diverse requirements and constraints encountered in different edge computing scenarios, and crucially, facilitates the integration of features that enhance resilience.

In response to the first research question regarding how to design FL frameworks for robustness against node failures in dynamic environments, our work focused on implementing key mechanisms within the framework's architecture. These include a dynamic worker pool managed by the Worker Manager module to handle device joins and leaves seamlessly, configurable completion thresholds for synchronous FL approaches to tolerate stragglers without halting progress, and task rescheduling mechanisms for asynchronous FL to ensure that tasks from failed workers are reassigned and completed.

The empirical evaluation in Scenario 2 and Scenario 3 (Sections 5.6 and 5.7), simulating significant probabilistic worker failures across various configurations, demonstrated that the framework successfully maintained training continuity and model convergence, evidenced by the high final MCC values of 0.944 for the UNSW-NB15 dataset and 0.903 for the ToN-IoT dataset with 40 workers and 1% failure rate every second, providing strong evidence that these specific design choices effectively contribute to resilience under dynamic and adverse conditions.

Addressing the second research question concerning what communication layer architectures are most suitable for fault-tolerant and resilient FL under dynamic network conditions, the evaluation conducted in Scenario 1 (Section 5.5) provided critical comparative insights into communication efficiency and overhead. By testing Zenoh, MQTT, and Kafka, we observed that protocol characteristics identified in this scenario significantly influence overall system performance and, crucially, robustness in the failure scenarios investigated in other experiments.

Protocols with native support for dynamic participant management and efficient handling of sporadic messages, such as Zenoh and MQTT, generally exhibited better suitability for dynamic, failure-prone environments. Their built-in disconnection detection mechanisms proved advantageous compared to protocols like Kafka, which required additional application-level heartbeats in our implementation.

Zenoh's decentralized nature further enhances its potential resilience by avoiding a single point of failure associated with a centralized broker. This comparative analysis highlights that the suitability of communication protocols is indeed variable, with Zenoh and MQTT demonstrating stronger alignment with the requirements for resilient FL in dynamic edge settings based on our empirical findings.

In summary, this research contributes with a modular and Resilient Federated Learning Framework, validated through experimental evaluation, that effectively addresses critical challenges in dynamic edge environments. The framework's design and implementation, coupled with the insights gained from the experimental scenarios regarding both the mechanisms for ensuring robustness and the comparative suitability of communication protocols, provide a strong foundation for developing more robust and adaptable FL systems for real-world applications where data privacy, security, and continuous operation are paramount.

6.2 Limitations

While the proposed framework demonstrates promising capabilities, it is important to acknowledge certain limitations inherent in the current study and experimental setup. The

evaluation was primarily conducted in a controlled, simulated environment using VMs, although this setup allowed for systematic testing under configurable conditions, it does not fully replicate the complexity and variability of real-world deployments.

Specifically, the worker nodes were implemented as VMs with relatively similar hardware specifications (CPU, memory). Real-world edge devices exhibit significant heterogeneity in computational capabilities, which can significantly impact FL performance and convergence.

Furthermore, the evaluation primarily focused on approximating IID data distribution across workers. Real-world FL scenarios frequently involve highly divergent, Non-IID data distributions, which pose additional challenges for model convergence and fairness that were not exhaustively explored in this evaluation.

The simulated worker failures predominantly involved graceful exits followed by potential re-joins, which may not fully capture the unpredictable nature and impact of abrupt crashes, sudden power losses, or prolonged network disconnections often encountered in practice.

Finally, while the framework is designed for scalability, the experimental validation involved a relatively small number of worker VMs (10 in the uniform set, 40 in the heterogeneous set), which provides valuable insights but does not definitively validate its performance and scalability to massive FL deployments involving potentially hundreds of devices.

6.3 Future Work

Building upon the foundation established by this research, several promising avenues for future work can be pursued to enhance the framework and broaden its evaluation. A primary direction is to expand the experimental validation to more closely approximate challenging real-world scenarios.

Crucially, future work should involve rigorous testing under Non-IID data distributions to assess the framework's performance and explore strategies for mitigating the associated convergence and fairness challenges. The framework's modular design, particularly the FL Backend, offers a key advantage here by enabling integration of advanced aggregation strategies and intelligent client selection mechanisms that prioritize data diversity or balance contributions from heterogeneous data distributions [5].

Furthermore, the Dataset Manager module could incorporate local data augmentation, re-sampling techniques, or privacy-preserving methods to expose local data statistics that the FL Backend could use for more informed aggregation and an improved scheduling policy. Evaluating the framework with other types of datasets beyond IDS/IoT, such as those from healthcare or finance, would also demonstrate its versatility.

Investigating the trade-offs associated with integrating various message compression and model quantization techniques within the Message Layer module is also crucial. Additionally, assessing the performance overhead of the Message Layer's optional encryption capabilities for practical deployments is important, noting that many modern edge devices are increasingly equipped with hardware-accelerated cryptographic modules or dedicated processor instructions that can significantly mitigate this burden, making the performance impact for basic encryption minimal or negligible in many practical scenarios [64], [65].

Expanding the resilience evaluation to include worker failures without graceful exits, such as abrupt crashes or prolonged network disruptions, is also necessary to test the robustness of the framework's recovery mechanisms under more realistic conditions.

From an algorithmic perspective, future work could involve implementing and evaluating more advanced FL algorithms that are specifically designed for robustness and efficiency in dynamic and heterogeneous environments. Additionally, exploring and implementing other worker scheduling policies beyond the default Round Robin, particularly those considering worker hardware capabilities (e.g., CPU, memory, or GPU availability), network conditions (e.g., bandwidth and latency), could potentially improve overall training efficiency and fairness.

Finally, a critical direction is the comprehensive evaluation of the framework's scalability to a much larger quantity of edge devices, ranging from hundreds to thousands. This expansion highlights potential bottlenecks, predominantly the central parameter server and the communication broker's capacity. To address this, integrating advanced distributed aggregation strategies such as hierarchical aggregation [24], where multiple intermediate aggregators handle subsets of devices before a global server consolidates their updates, would be beneficial. Integrating the framework with real-world edge computing platforms beyond simulated VMs would provide a more definitive validation of its practical applicability and performance.

References

[1] A. B. Rashid and A. K. Kausik, «Ai revolutionizing industries worldwide: A comprehensive overview of its diverse applications», *Hybrid Advances*, p. 100 277, 2024.

[2] R. Xu, N. Baracaldo, and J. Joshi, «Privacy-preserving machine learning: Methods, challenges and directions», *arXiv preprint arXiv:2108.04417*, 2021.

[3] M. Durovic and T. Corno, «The privacy of emotions: From the gdpr to the ai act, an overview of emotional ai regulation and the protection of privacy and personal data», *Privacy, Data Protection and Data-driven Technologies*, pp. 368–404, 2024.

[4] B. Luo, P. Han, P. Sun, X. Ouyang, J. Huang, and N. Ding, «Optimization design for federated learning in heterogeneous 6g networks», *IEEE Network*, vol. 37, no. 2, pp. 38–43, 2023.

[5] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, «Federated learning with non-iid data», *arXiv preprint arXiv:1806.00582*, 2018.

[6] B. S. Guendouzi, S. Ouchani, H. E. Assaad, and M. E. Zaher, «A systematic review of federated learning: Challenges, aggregation methods, and development tools», *Journal of Network and Computer Applications*, vol. 220, p. 103 714, 2023.

[7] R. Teixeira, L. Almeida, M. Antunes, D. Gomes, and R. L. Aguiar, «Efficient training: Federated learning cost analysis», *Big Data Research*, vol. 40, pp. 100 510–100 510, 2025.

[8] B. Chen, N. Ivanov, G. Wang, and Q. Yan, «Dynamicfl: Balancing communication dynamics and client manipulation for federated learning», in *2023 20th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, IEEE, 2023, pp. 312–320.

[9] L. Almeida, P. Rodrigues, R. Teixeira, M. Antunes, and R. L. Aguiar, «Privacy-preserving defense: Intrusion detection in iot using federated learning», in *2024 IEEE 22nd Mediterranean Electrotechnical Conference (MELECON)*, 2024, pp. 908–913. DOI: 10.1109/MELECON56669.2024.10608461.

[10] R. Teixeira, L. Almeida, P. Rodrigues, M. Antunes, D. Gomes, and R. L. Aguiar, «Shallow vs. deep learning: Prioritizing efficiency in next generation networks», in *2024 11th International Conference on Future Internet of Things and Cloud (FiCloud)*, IEEE, 2024, pp. 308–315.

[11] L. Almeida, P. Rodrigues, D. Magalhães, A. J. Pinho, and D. Pratas, «Aidetx: A compression-based method for identification of machine-learning generated text», in *2025 Data Compression Conference (DCC)*, IEEE, Mar. 2025, pp. 358–358. DOI: 10.1109/dcc62719.2025.00046.

[12] L. Almeida, R. Teixeira, G. Baldoni, M. Antunes, and R. L. Aguiar, «Federated learning for a dynamic edge: A modular and resilient approach», *Sensors*, vol. 25, no. 12, p. 3812, 2025.

[13] J. Corona, P. Rodrigues, L. Almeida, R. Teixeira, M. Antunes, and R. L. Aguiar, «From black box to transparency: Consistency and cost within xai», in *IEEE Global Communications Conference 2024*, IEEE, Dec. 2024.

[14] L. Almeida, P. Rodrigues, M. Antunes, and R. L. Aguiar, «Resilient federated learning framework for 6g», in *2025 IEEE International Conference on Consumer Technology (ICCT-Europe)*, IEEE, Apr. 2025.

[15] P. Rodrigues, L. Almeida, J. Corona, M. Antunes, and R. L. Aguiar, «Optimised task placement for mlops», in *2025 IEEE International Conference on Consumer Technology (ICCT-Europe)*, IEEE, Apr. 2025.

[16] R. Teixeira, L. Almeida, P. Rodrigues, J. Corona, M. Antunes, and R. L. Aguiar, «Understanding what federated learning models learn: A comparative study with traditional models», in *2025 33rd International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, Oct. 2025.

[17] S. AbdulRahman, H. Tout, H. Ould-Slimane, A. Mourad, C. Talhi, and M. Guizani, «A survey on federated learning: The journey from centralized to distributed on-site learning and beyond», *IEEE Internet of Things Journal*, vol. 8, no. 7, pp. 5476–5497, 2020.

[18] J. SAHOO, A. K. NAIR, and R. SHARMA, «The evolution of machine learning: From centralized to distributed», *Federated Learning: Principles, Paradigms, and Applications*, p. 1, 2024.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, «Learning representations by back-propagating errors», *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[20] D. P. Kingma, «Adam: A method for stochastic optimization», *arXiv preprint arXiv:1412.6980*, 2014.

[21] R. Teixeira, M. Antunes, D. Gomes, and R. L. Aguiar, «The learning costs of federated learning in constrained scenarios», in *2023 10th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2023, pp. 18–25. DOI: 10.1109/FiCloud58648.2023.00011.

[22] M. Langer, Z. He, W. Rahayu, and Y. Xue, «Distributed training of deep learning models: A taxonomic perspective», *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2802–2818, 2020. DOI: 10.1109/TPDS.2020.3003307.

[23] L. Liu, J. Zhang, S. Song, and K. B. Letaief, «Client-edge-cloud hierarchical federated learning», in *ICC 2020-2020 IEEE international conference on communications (ICC)*, IEEE, 2020, pp. 1–6.

[24] J. Liu, J. Huang, Y. Zhou, *et al.*, «From distributed machine learning to federated learning: A survey», *Knowledge and Information Systems*, vol. 64, no. 4, pp. 885–917, 2022.

[25] Q. Yang, Y. Liu, T. Chen, and Y. Tong, «Federated machine learning: Concept and applications», *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[26] Q. Yang, Y. Liu, Y. Cheng, Y. Kang, T. Chen, and H. Yu, «Horizontal federated learning», in *Federated learning*, Springer, 2020, pp. 49–67.

[27] Y. Liu, Y. Kang, T. Zou, *et al.*, «Vertical federated learning: Concepts, advances, and challenges», *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3615–3634, 2024.

[28] S. Saha and T. Ahmad, «Federated transfer learning: Concept and applications», *Intelligenza Artificiale*, vol. 15, no. 1, pp. 35–44, 2021.

[29] L. Li, Y. Fan, M. Tse, and K.-Y. Lin, «A review of applications in federated learning», *Computers & Industrial Engineering*, vol. 149, p. 106 854, 2020.

[30] E. Gabriel, G. E. Fagg, G. Bosilca, *et al.*, «Open mpi: Goals, concept, and design of a next generation mpi implementation», in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, Springer, 2004, pp. 97–104.

[31] R. A. Light, «Mosquitto: Server and client implementation of the mqtt protocol», *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[32] J. Kreps, N. Narkhede, J. Rao, *et al.*, «Kafka: A distributed messaging system for log processing», in *Proceedings of the NetDB*, Athens, Greece, vol. 11, 2011, pp. 1–7.

[33] A. Corsaro, L. Cominardi, O. Hecart, *et al.*, «Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller», in *2023 26th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2023, pp. 422–428.

[34] S. Bano, «Phd forum abstract: Efficient computing and communication paradigms for federated learning data streams», 2021, pp. 410–411. DOI: 10.1109/SMARTCOMP52413.2021.00086.

[35] K. A. Awan, I. U. Din, A. Almogren, and J. J. P. C. Rodrigues, «Privacy-preserving big data security for iot with federated learning and cryptography», *IEEE Access*, vol. 11, pp. 120 918–120 934, 2023. DOI: 10.1109/ACCESS.2023.3328310.

[36] C. L. Stergiou, E. Bompoli, and K. E. Psannis, «Security and privacy issues in iot-based big data cloud systems in a digital twin scenario», *Applied Sciences*, vol. 13, no. 2, p. 758, 2023.

[37] S. Venu, J. Kotti, A. Pankajam, *et al.*, «Secure big data processing in multihoming networks with ai-enabled iot», *Wireless Communications & Mobile Computing (Online)*, vol. 2022, 2022.

[38] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell, «Adaptive aggregation for federated learning», 2022, pp. 180–185. DOI: 10.1109/BigData55660.2022.10021119.

[39] J. Song, W. Wang, T. R. Gadekallu, J. Cao, and Y. Liu, «Eppda: An efficient privacy-preserving data aggregation federated learning scheme», *IEEE Transactions on Network Science and Engineering*, vol. 10, no. 5, pp. 3047–3057, 2023. DOI: 10.1109/TNSE.2022.3153519.

[40] K. Bonawitz, V. Ivanov, B. Kreuter, *et al.*, «Practical secure aggregation for privacy-preserving machine learning», in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[41] M. Mansouri, M. Önen, and W. B. Jaballah, «Learning from failures: Secure and fault-tolerant aggregation for federated learning», 2022, pp. 146–158. DOI: 10.1145/3564625.3568135.

[42] X. Chen, G. Xu, X. Xu, H. Jiang, Z. Tian, and T. Ma, «Multicenter hierarchical federated learning with fault-tolerance mechanisms for resilient edge computing networks», *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2024. DOI: 10.1109/TNNLS.2024.3362974.

[43] J. Konečnỳ, «Federated learning: Strategies for improving communication efficiency», *arXiv preprint arXiv:1610.05492*, 2016.

[44] F. Sattler, K.-R. Müller, and W. Samek, «Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints», *IEEE transactions on neural networks and learning systems*, vol. 32, no. 8, pp. 3710–3722, 2020.

[45] J. Á. Morell and E. Alba, «Dynamic and adaptive fault-tolerant asynchronous federated learning using volunteer edge devices», *Future Generation Computer Systems*, vol. 133, pp. 53–67, 2022. DOI: 10.1016/j.future.2022.02.024.

[46] R. Dautov and E. J. Husom, «Raft protocol for fault tolerance and self-recovery in federated learning», 2024, pp. 110–121. DOI: 10.1145/3643915.3644093.

[47] E. Diao, J. Ding, and V. Tarokh, «Heterofl: Computation and communication efficient federated learning for heterogeneous clients», *arXiv preprint arXiv:2010.01264*, 2020.

[48] K. Pfeiffer, M. Rapp, R. Khalili, and J. Henkel, «Cocofl: Communication-and computation-aware federated learning via partial nn freezing and quantization», *arXiv preprint arXiv:2203.05468*, 2022.

[49] S. Raschka, J. Patterson, and C. Nolet, «Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence», *Information*, vol. 11, no. 4, p. 193, 2020.

[50] R. V. Rasmussen and M. A. Trick, «Round robin scheduling–a survey», *European Journal of Operational Research*, vol. 188, no. 3, pp. 617–636, 2008.

[51] I. Parsa, *KDD Cup 1998 Data*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5401H, 1998.

[52] S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan, *KDD Cup 1999 Data*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C51C7N, 1999.

[53] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, «A detailed analysis of the kdd cup 99 data set», in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, 2009, pp. 1–6. DOI: 10.1109/CISDA.2009.5356528.

[54] K. Kostas, M. Just, and M. A. Lones, «Iotgem: Generalizable models for behaviour-based iot attack detection», *arXiv preprint arXiv:2401.01343*, 2023.

[55] N. Moustafa and J. Slay, «Unsw-nb15: A comprehensive data set for network intrusion detection systems (unsw-nb15 network data set)», in *2015 Military Communications and Information Systems Conference (MilCIS)*, 2015, pp. 1–6. DOI: 10.1109/MilCIS.2015.7348942.

[56] T. M. Booij, I. Chiscop, E. Meeuwissen, N. Moustafa, and F. T. H. d. Hartog, «Ton_iot: The role of heterogeneity and the need for standardization of features and attack types in iot network intrusion data sets», *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 485–496, 2022. DOI: 10.1109/JIOT.2021.3085194.

[57] M. Sarhan, S. Layeghy, and M. Portmann, «Towards a standard feature set for network intrusion detection system datasets», *Mobile Networks and Applications*, vol. 27, no. 1, pp. 357–370, Nov. 2021, ISSN: 1572-8153. DOI: 10.1007/s11036-021-01843-0. [Online]. Available: http://dx.doi.org/10.1007/s11036-021-01843-0.

[58] R. Abou Khamis and A. Matrawy, «Evaluation of adversarial training on different types of neural networks in deep learning-based idss», in *2020 international symposium on networks, computers and communications (ISNCC)*, IEEE, 2020, pp. 1–6.

[59] R. Abou Khamis, M. O. Shafiq, and A. Matrawy, «Investigating resistance of deep learning-based ids against adversaries using min-max optimization», in *ICC 2020-2020 IEEE international conference on communications (ICC)*, IEEE, 2020, pp. 1–7.

[60] P. Kumar, R. Tripathi, and G. P. Gupta, «P2idf: A privacy-preserving based intrusion detection framework for software defined internet of things-fog (sdiot-fog)», in *Adjunct Proceedings of the 2021 International Conference on Distributed Computing and Networking*, 2021, pp. 37–42.

[61] D. Chicco and G. Jurman, «The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation», *BMC genomics*, vol. 21, pp. 1–13, 2020.

[62] R. Diallo, C. Edalo, and O. O. Awe, «Machine learning evaluation of imbalanced health data: A comparative analysis of balanced accuracy, mcc, and f1 score», in *Practical Statistical Learning and Data Science Methods: Case Studies from LISA 2020 Global Network, USA*, Springer, 2024, pp. 283–312.

[63] D. Chicco, M. J. Warrens, and G. Jurman, «The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation», *Peerj computer science*, vol. 7, e623, 2021.

[64] S. S. Dhanda, B. Singh, and P. Jindal, «Lightweight cryptography: A solution to secure iot», *Wireless Personal Communications*, vol. 112, no. 3, pp. 1947–1980, 2020.

[65] C. Silva, V. A. Cunha, J. P. Barraca, and R. L. Aguiar, «Analysis of the cryptographic algorithms in iot communications», *Information Systems Frontiers*, vol. 26, no. 4, pp. 1243–1260, May 2023, ISSN: 1572-9419. DOI: 10.1007/s10796-023-10383-9.