



University of
Zurich^{UZH}

Deploying a Decentralized Federated Learning Platform in Resource-Constrained Devices

Nicolas Huber
Zurich, Switzerland
Student ID: 20-735-908

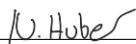
Supervisor: Chao Feng, Dr. Alberto Huertas Celdran, Prof. Dr.
Burkhard Stiller

Date of Submission: March 31, 2025

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 29.03.2025



Signature of student

Abstrakt

Federated Learning (FL) ist ein innovatives Konzept zum Trainieren von Machine-Learning-Modellen, das es ermöglicht, Modelle kollaborativ zu trainieren, ohne Trainingsdaten direkt auszutauschen. Dadurch bleiben private Daten auf den jeweiligen Geräten der Teilnehmer, wodurch die Privatsphäre gewahrt bleibt. Decentralized Federated Learning (DFL) ermöglicht das Trainieren gemeinsamer Modelle ohne einen zentralen Server, wie es beim Centralized Federated Learning (CFL) der Fall ist. NEBULA ist eine neuartige Plattform, die Nutzern erlaubt, Modelle in sowohl CFL- als auch DFL-Umgebungen zu trainieren. Allerdings basiert NEBULA hauptsächlich auf Virtualisierungstechnologien wie Docker-Containern, um Föderationen zu erstellen und physische Teilnehmer zu simulieren. Das Trainieren von DFL-Modellen in realen Umgebungen stellt jedoch besondere Herausforderungen dar, darunter begrenzte Rechenleistung, Speicherplatz und Energieverbrauch – Faktoren, die in einer virtuellen Umgebung nur unzureichend abgebildet werden. Um NEBULA unter realistischeren Bedingungen zu testen, wurde die Plattform in dieser Arbeit auf ressourcenbeschränkten Geräten, wie Raspberry Pis und NVIDIA Jetson Nano, implementiert. Die entwickelte Lösung wurde in verschiedenen Netzwerktopologien – darunter vollständig verbundene, Ring- und Stern-Topologie – getestet. Die Ergebnisse zeigen, dass NEBULA auch auf Edge-Geräten ein Modell von vergleichbarer Qualität trainieren kann wie in einer virtualisierten Umgebung auf einem modernen Laptop. Dabei blieb die Ressourcennutzung der teilnehmenden Geräte auf einem akzeptablen Niveau, was die Praxistauglichkeit von NEBULA für föderiertes Lernen in realen, ressourcenbeschränkten Umgebungen bestätigt.

Abstract

Federated Learning (FL) is an innovative approach to training machine learning models that enables collaborative training without the need to share training data directly. This method allows private data to remain on participants' devices, thus preserving their privacy. Decentralized Federated Learning (DFL) involves training shared models without relying on a central server, in contrast to Centralized Federated Learning (CFL). NEBULA is a new platform designed to facilitate model training in both CFL and DFL environments. It primarily utilizes virtualization technologies, such as Docker containers, to create federations and simulate physical participants. However, training DFL models in real-world environments presents specific challenges, including limited computing power, storage capacity, and energy consumption—factors that are not well represented in a virtual environment. To evaluate NEBULA under more realistic conditions, this thesis implemented the platform on resource-constrained devices, such as Raspberry Pis and NVIDIA Jetson Nano units. The developed solution was tested across various network topologies, including fully connected, ring, and star configurations. The results indicated that NEBULA can train models of comparable quality on edge devices as it does in a virtualized environment on modern laptops. Additionally, the resource utilization of the participating devices remained within acceptable limits, confirming NEBULA's practicality for federated learning in real-world, resource-constrained settings.

Acknowledgments

I hereby declare that I used generative AI, such as Grammarly and ChatGPT, to improve the language and structure of the written text of this thesis. The use of generative AI was restricted to the checking of spelling and improving the structure and form of the written text, but not to the key algorithms that represent the key contributions of this thesis.

Contents

Declaration of Independence	i
Abstract	iii
Acknowledgments	vii
1 Introduction	1
1.1 Motivation	2
1.2 Description of Work	2
1.3 Thesis Outline	3
2 Background	5
2.1 Federated Learning	5
2.1.1 Architecture of Federated Learning	6
2.1.2 Challenges in federated learning	9
2.2 Decentralized Federated Learning	10
2.2.1 Federation Architecture	11
2.2.2 Network Topologies	12
2.2.3 Communication Schemes	12
2.2.4 Challenges	13
2.3 NEBULA	14
2.4 Related Work	19

3	Architecture	21
3.1	Real-Time Metric Collection and Power Monitoring	22
4	Implementation	25
4.1	Deployment Environment	25
4.2	Controller and Node Deployment	27
4.2.1	Controller Deployment	27
4.2.2	Participant Node Deployment	28
4.3	Real-Time Metric Transmission in NEBULA	31
4.4	Reporting Energy Consumption	32
4.4.1	Power Measurement and Polling	32
4.4.2	Logging and Reporting Power Consumption	34
5	Evaluation	37
5.1	Participant Characteristics	38
5.2	Dataset and Federated Model	38
5.3	Network Topology and Federation Architecture	38
5.4	Results	39
5.4.1	Summary	48
6	Conclusion	53
	Bibliography	55
	Abbreviations	59
	List of Figures	59
	List of Tables	62

<i>CONTENTS</i>	xi
A Deployment	65
A.1 Deploying on the Controller Node	65
A.1.1 Detailed Code Changes	65
A.2 Deploying on Participant Nodes	67
A.2.1 Detailed Code Changes	68

Chapter 1

Introduction

Nowadays, edge devices generate vast amounts of data with the help of advanced sensors, enhanced processing capabilities, and widespread Internet connectivity, which enables seamless communication among devices [1]. A field that benefits significantly from this abundance of data is Machine Learning (ML). However, data originating from IoT devices is typically distributed across multiple sources, making traditional ML approaches—where data is collected and processed in a central location—impractical due to communication overhead, data privacy risks, and regulatory restrictions [2].

To address these challenges, Federated Learning (FL) was introduced by [3]. FL allows multiple devices to train a shared model collaboratively without exposing raw data. Instead of transmitting datasets to a central server, FL enables nodes to train models locally and exchange only model updates, ensuring enhanced privacy and security.

According to [2], FL can be categorized into Centralized Federated Learning and Decentralized Federated Learning:

- Centralized Federated Learning (CFL): A central server orchestrates training by distributing global model parameters to participant nodes and aggregating their updates into a shared model.
- Decentralized Federated Learning (DFL): No central server exists; nodes communicate directly with peers, self-organizing the training process and model aggregation.

While CFL has been widely adopted, DFL provides greater scalability and fault tolerance. Furthermore, it improves the limitations of having a single point of failure and trust dependencies [2].

NEBULA: A Powerful FL Platform

NEBULA is an advanced federated learning platform that facilitates both CFL and DFL architectures [4]. It streamlines the development, deployment, and management of federated applications across both physical and virtualized environments. NEBULA enables

users to create customized federations, allowing them to define the number and type of participating devices, establish the network topology for node communication, select federated learning algorithms for model training, and configure datasets and aggregation strategies to optimize performance. Additionally, NEBULA offers real-time monitoring of model performance, network efficiency, and system resources, making it a versatile tool for FL research and deployment [5].

1.1 Motivation

Training FL models on resource-constrained devices—such as smartphones, IoT devices, and edge computing nodes—introduces significant challenges. These devices often have limited computational power, memory, and energy resources, making it difficult to implement privacy-preserving techniques and sustain efficient model training [6].

Currently, NEBULA predominantly deploys FL scenarios in virtualized settings, such as Docker containers. This seems to be the state of things as many existing FL platforms focus on virtualized environments [5]. However, while virtualization may be convenient for development and testing, it does not fully reflect the real-world constraints and complexities of deploying FL on actual resource-limited devices.

1.2 Description of Work

To address these challenges, this Bachelor Thesis focuses on deploying NEBULA onto real resource-constrained devices to create a realistic testbed for DFL systems. Specifically, NEBULA will be deployed on Raspberry Pi 4 devices and Nvidia’s Jetson Nano developer toolkit.

The first stage of this work will explore the basic concepts, architectures, and working mechanisms of FL and DFL to foster the essential foundational knowledge needed for the design and implementation phases. In the second phase of the thesis, the primary focus will be on deploying the NEBULA platform on resource-constrained devices. Simultaneously, a mechanism will be implemented to track and record the energy consumption of each device throughout the platform’s operation. Additionally, technical challenges and issues encountered during development will be highlighted, along with proposed solutions.

The third stage will be dedicated to testing. The functionality and effectiveness of the physical DFL test bed will be assessed using diverse use cases. A series of experiments will be conducted to evaluate key performance metrics of the implemented system, including computational power, memory usage, network latency, and energy consumption. Furthermore, the challenges and limitations of implementing Federated Learning on resource-constrained devices will be examined and discussed.

1.3 Thesis Outline

This thesis is structured to provide a comprehensive understanding of FL and DFL while detailing the deployment of NEBULA on resource-constrained devices. The organization of this work is as follows:

- **Background (Section 2):** This section introduces the fundamental concepts of FL and DFL, highlighting their key differences and associated challenges. It covers three major FL architectures—Horizontal FL, Vertical FL, and Federated Transfer Learning—and explores DFL advantages, federation architectures, network topologies, and communication schemes. Additionally, this section provides an in-depth overview of NEBULA, describing its core features and modular architecture.
- **Architecture (Section 3):** This section presents the overall system architecture, outlining the modifications necessary to adapt NEBULA for deployment on resource-constrained devices such as Raspberry Pis and the NVIDIA Jetson Nano Developer Kit. It also details the enhancements made to the real-time monitoring module, particularly power and energy tracking integration.
- **Implementation (Section 4):** This section provides a detailed walkthrough of the deployment process, covering the software and hardware environments, necessary code modifications, encountered challenges, and the implementation of power and energy consumption tracking within NEBULA.
- **Evaluation (Section 5):** This section presents the experimental evaluation of NEBULA under both physical and virtualized environments. It discusses the performance results and analyzes the impact of deploying FL on resource-limited devices.
- **Conclusion and Future Work (Section 6):** The final section summarizes the findings of this research, reflecting on the feasibility and effectiveness of deploying NEBULA on edge devices. Additionally, it highlights areas for future research and development in the domain of decentralized federated learning.

Chapter 2

Background

In today's world, many people rely on mobile phones and tablets, which leads to the accumulation of a vast amount of data on these devices. Utilizing this data for training ML models can significantly improve their effectiveness. However, traditional centralized ML approaches require transferring raw training data to a central server, which raises significant privacy concerns for data owners, especially given the sensitive nature of the information stored on personal devices. To address these issues, federated learning (FL) was introduced in 2016 to separate model training from the need to access raw training data directly [3].

2.1 Federated Learning

FL can be divided into two main categories: Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL). In CFL, a central server coordinates the training process, managing a federation of participating devices (clients). The CFL process typically occurs in rounds and involves the following steps: 1. The central server initiates the CFL process by distributing the current global model to selected clients. 2. Each client trains the model locally using its dataset, computes an update, and sends it back to the central server. 3. The central server then aggregates the received updates to update the global model [7].

Although CFL requires a degree of trust in the coordinating server, it decouples model training from the need for direct access to raw training data. Thus, CFL enhances data privacy and reduces the attack surface from both the device and the cloud to only the device [7]. Another advantage of CFL is the potential to reduce bandwidth usage significantly. Instead of transmitting entire datasets to a central server, clients only send model updates, such as a single gradient. In exchange, clients must be willing to allocate computational resources for local model training [3].

CFL is particularly suitable for problems where real-world data from mobile devices provides a distinct advantage over proxy data, the client data is privacy sensitive or extensive (the data does not have to be sent to the data centre for the sole purpose of training the

model), and for supervised tasks where labels on the data can be inferred from user interaction [7].

However, CFL presents several optimization challenges, as highlighted by [7]. These are mainly:

1. Non-IID data: A client’s local dataset may not accurately represent the overall distribution.
2. Unbalanced Datasets: Some clients may have access to more substantial datasets due to more frequent device usage.
3. Massively distributed: The number of participating clients is anticipated to be much larger than each client’s average amount of data.
4. Limited communication: Mobile devices often encounter constraints such as slow or costly connections and the potential to go offline.

Additional challenges include clients deleting or adding data to their local datasets, inconsistencies in client availability, varying responsiveness among clients, and the risk of clients sending corrupted updates [7].

A widely used federated optimization algorithm in CFL is Federated Averaging (FedAvg), introduced by [7]. FedAvg demonstrates that CFL can be practical as it trains high-quality models with relatively few communication rounds. FedAvg is based on large-batch synchronous stochastic gradient descent (SDG). In each training round, a fraction of the participating clients are selected, and each selected client performs local model updates by iteratively computing gradient descent over its local dataset; the updated local models are sent back to the central server, and the server then aggregates the updates using a weighted average, producing the updated global model. One critical insight from [7] is that model initialization significantly affects convergence. The authors observed that their averaged model often performed poorly when training two models (each initialized with a different random seed) on two separate datasets. However, when models were initialized with the same random seed before independent training, their aggregated parameters resulted in significantly lower loss.

2.1.1 Architecture of Federated Learning

According to [8], CFL can be categorized into three architectures based on how data is distributed among clients: Horizontal Federated Learning (HFL), Vertical Federated Learning (VFL), and Federated Transfer Learning (FTL).

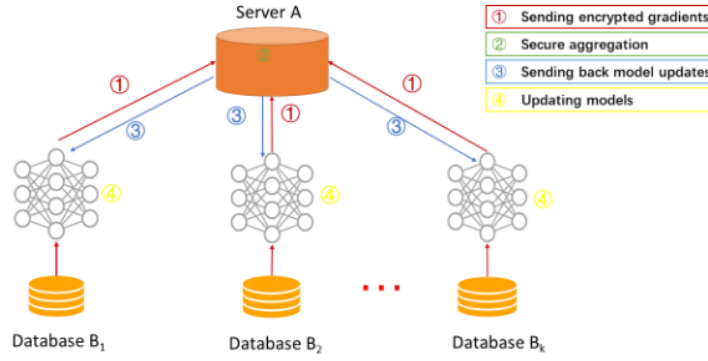


Figure 2.1: Architecture for a horizontal federated learning system [8]

Horizontal Federated Learning

In HFL, clients share the same feature space but have different data samples. A practical example is two regional banks with similar business models but distinct customer bases. Despite overlapping feature spaces (e.g., transaction history and credit scores), they do not share the same users.

A well-known example of an HFL algorithm is FedAvg [7]. In a standard HFL setup, it is typically assumed that clients are honest, while the server is honest but curious, meaning it does not alter updates but may attempt to infer information from them.

However, privacy risks arise in HFL when a central server receives model updates from multiple clients. The server could potentially infer sensitive information by analyzing the updates from individual clients [9]. To mitigate this risk, Secure Aggregation protocols have been introduced [9]. These protocols compute element-wise weighted averages of client update vectors while ensuring that the server cannot access individual client updates. As a result, the server still benefits from aggregated updates but cannot associate specific updates with particular clients.

Figure 2.1 shows a typical HFL architecture, which consists of k clients with a shared data structure and a centralized parameter server (cloud server). The training process in HFL can be described in the following five steps [8]:

1. Clients compute training gradients locally, mask selected gradients using secure masking techniques, and send masked gradients to the server.
2. The server aggregates the received gradients without accessing individual client updates.
3. The server updates the global model using the aggregated gradients.
4. The updated model is sent back to the clients who apply the decrypted gradients to update their local models.
5. Steps 1–4 are repeated until the loss function converges, marking the completion of the training process.

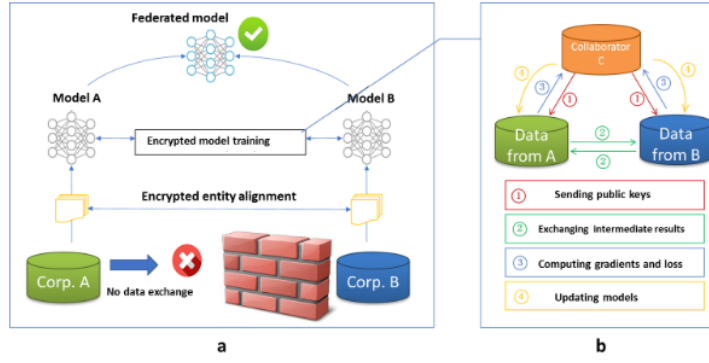


Figure 2.2: Architecture for a vertical federated learning system [8]

Vertical Federated Learning

In VFL, clients share the same data samples but have different feature spaces. For instance, a bank and an e-commerce company operating in the same city likely have overlapping users but collect different types of information about them. The e-commerce company tracks browsing behaviour and purchase history, whereas the bank records financial transactions and credit scores [8].

If both companies aim to develop a predictive model—for example, to recommend financial products based on user behaviour—they can leverage VFL to collaboratively train a model without sharing raw data. In this setup, the model aggregates complementary features from both organizations while ensuring data privacy [8].

Unlike Horizontal FL, which assumes honest clients and an honest-but-curious server, VFL assumes honest-but-curious participants. Additionally, a Semi-Honest Third Party (such as a government agency or a trusted collaborator) is often introduced to facilitate secure computations while ensuring confidentiality [8].

If a bank and an e-commerce company jointly train a model where the e-commerce company holds the label data but cannot share raw user information due to privacy regulations, a third-party collaborator C ensures secure computations during training [8]. The training process in VFL, as seen in Figure 2.2, consists of two key phases:

Part 1: Before training begins, the system must identify common users in both datasets without revealing non-overlapping users.

Part 2. Once common users are identified, the training process follows these four steps:

1. The trusted collaborator (C) generates encryption key pairs and distributes the public key to both companies.
2. Each company encrypts and exchanges intermediate training results required for gradient and loss calculations.

3. Both companies compute encrypted gradients locally. They apply random encrypted masks before sending gradients to C to prevent C from accessing sensitive information. Additionally, the e-commerce company also computes the encrypted loss and sends it to C.
4. C decrypts the gradients and loss values and returns them to the two companies. Each company removes the random masks and updates its model parameters accordingly.

Federated Transfer Learning

Transfer Learning enables a model to leverage knowledge from a related source domain to improve learning in a target domain with limited data or labels [10]. The effectiveness of transfer learning depends on the similarity between the two domains. In FTL, clients have different feature spaces and data samples, making traditional Horizontal and Vertical FL unsuitable for model training. A practical example, as described by [8], is the case of a bank in China and an e-commerce company in the United States. These organizations do not share the same user base or feature space. Since the overlapping sample space is minimal, Vertical FL is not applicable. However, according to [8], FTL allows these organizations to collaboratively train models by:

- Learning a common representation between their distinct feature spaces.
- Minimizing label prediction errors in the target domain (e.g., the bank) by leveraging labelled data from the source domain (e.g., the e-commerce company).

Although the overall FTL architecture is similar to Vertical FL (as shown in Figure 2.2), the gradient computation process differs [8].

FTL benefits from utilizing data from various sources while ensuring data privacy and maintaining accuracy. However, it also incurs additional computational and communication costs [11].

Other architectures for CFL include but are not limited to, the Multi-participant Multi-class Vertical Federated Learning (MMVFL) Framework [12]. MMVFL is an extension of VFL that supports more than two participants. Instead of training a single global model, MMVFL learns a personalized model for each participant, enhancing adaptability to diverse data distributions. Additionally, there is FEDF, a distributed deep learning framework designed for privacy-preserving and parallel training [13].

2.1.2 Challenges in federated learning

According to [14], four core challenges arise in FL:

- **Expensive Communication:** Federated networks can encompass a large number of clients, which can lead to significant communication overhead. Furthermore, the need to encrypt data for privacy reasons adds to the network load [11].
- **Systems Heterogeneity:** The client devices may vary in terms of storage, computational power, and communication capabilities. This variability can stem from differences in hardware (such as CPU and memory), network connectivity (including 3G, 4G, 5G, and Wi-Fi), and power (battery levels). Additionally, some devices may be unreliable or may drop out of the training process due to connectivity issues or energy constraints.
- **Statistical Heterogeneity:** The data collected by the clients' devices may be distributed non-uniformly across the network. The amount of data held by each device can vary significantly, and there may be hidden patterns in how data is distributed across devices. This variability can result in "stragglers," complicating modeling, analysis, and evaluation. Fairness can also become a concern if certain devices dominate the learned model due to having larger datasets or belonging to commonly occurring groups. Furthermore, when dealing with non-IID data, methods such as FedAvg can encounter divergence in practice [7].
- **Privacy Concerns:** Although FL keeps raw data on local devices, the updates shared during training can still reveal sensitive information. Protecting privacy is crucial, but stronger security measures often come at the cost of model accuracy or system efficiency.

Furthermore, the distributed nature of FL makes it vulnerable to threats such as poisoning attacks [15]. Poisoning attacks occur when malicious participants aim to compromise the global model by sending model updates generated from mislabeled data [15]. One specific type of poisoning attack is known as a Byzantine attack, which involves a scenario in FL where a malicious user controls multiple clients, referred to as Byzantine users [16]. These users can upload deceptive data, which may result from being controlled by a malicious entity, unreliable communication channels, or corrupted hardware. Such actions can manipulate the global model, leading to difficulties in its convergence [16]. The authors of [15] found that under the minimal assumption that each malicious FL participant can only alter the raw training data on their device, the following points emerge: 1) An attack can significantly reduce the model's utility, even when the proportion of malicious users is small. 2) By conducting targeted attacks, malicious participants can have a substantial negative impact on specific classes under attack, while causing little to no harm to other classes, thereby evading easy detection. 3) A malicious user can maximize the poisoning effect by participating in later training rounds and maintaining high availability.

2.2 Decentralized Federated Learning

DFL emerged in 2018 to address the limitations of CFL by eliminating reliance on a central server and promoting peer-to-peer model aggregation [2]. In a DFL framework,

participating clients independently handle local model training, Parameter exchange with neighbouring nodes, and Decentralized model aggregation.

According to [2], DFL offers several advantages over CFL:

- Nodes in a decentralized federation continuously update their understanding of the availability of other nodes, which increases the robustness of the network and reduces the risk of a single point of failure.
- DFL distributes trust among the federation nodes, which collectively determines the trustworthiness of the entire federation. This approach minimizes the risk of a single point of attack and improves overall trust.
- DFL enables a more even distribution of communication and workload among the nodes, which helps reduce the chances of network congestion or delays.
- DFL allows new nodes to join the network easily and participate in the training process.
- DFL distributes the computational power required for aggregating model parameters across all participating nodes, leading to more efficient use of resources.

2.2.1 Federation Architecture

DFL can be divided into two categories: cross-silo federation and cross-device federation. In cross-silo DFL, there are usually a relatively small number of nodes (fewer than 100) representing organizations or data centers that possess large amounts of data. These nodes have consistent, robust, and scalable computing capabilities, as well as high performance in the network. On the other hand, cross-device DFL involves a larger number of nodes (more than 100) that typically include edge devices or robots, such as UAVs (unmanned aerial vehicles). These nodes contain relatively small amounts of data, have limited computational power, experience weak communication between devices, and face power constraints. Because of these limitations, nodes in cross-device DFL cannot be expected to perform complex training tasks and may experience higher dropout rates, which can negatively impact DFL performance [2].

According to [2], nodes of DFL may have one or more of the following roles:

- **Trainer:** Trainer nodes are responsible for training a local model using their local dataset and then transmitting the updated parameters to neighboring nodes. They anticipate receiving and integrating the improved federated model back into their local model.
- **Aggregator:** Aggregator nodes are tasked with collecting parameters from neighboring nodes, integrating them into the global model, and then sending them back to these neighboring nodes.

- Proxy: In specific network topologies, proxy nodes serve as intermediaries, forwarding model parameters to aggregator nodes not directly reachable by other nodes. Additionally, proxy nodes can be utilized to connect different network topologies.
- Idle: An idle node is one that holds no specific roles and does not actively engage in the federation.

2.2.2 Network Topologies

According to [2], DFL networks are typically structured using the following topologies:

- Fully Connected Networks: Each node establishes a directed connection with every other node in the federation. While this topology offers high reliability and robustness against node failures, it incurs significant communication overhead and scalability challenges due to increased complexity when integrating new nodes.
- Partially Connected Networks - Star Structured: A single proxy node facilitates communication between all other nodes. New nodes can easily join by establishing a single connection to the proxy node. However, the proxy node represents a single point of failure and can become a communication bottleneck. The number of nodes that can participate depends on the computational capacity of the proxy node.
- Partially Connected Networks - Ring Structured: Nodes are connected circularly, either in a unidirectional or bidirectional manner. Bidirectional rings offer higher reliability and fault tolerance, whereas unidirectional rings are more susceptible to failures. As the number of nodes increases, transmission delays in model updates also increase.
- Partially Connected Networks - Random: Nodes establish connections dynamically based on heuristics such as proximity or computational capacity. This structure provides high flexibility and moderate fault tolerance but may result in higher communication costs and variable security risks due to the lack of a predefined topology.
- Node Clustering: The literature distinguishes between similarity-based and proxy-based clusters. In similarity-based clusters, nodes are grouped based on similar local model parameters, leading to homogeneous performance within clusters. However, this approach may reduce model generalization since models are trained on localized data distributions. In proxy-based clusters, proxy nodes interconnect different topologies, facilitating model exchange between clusters. However, proxy nodes introduce performance bottlenecks and require an initial connection to a coordinating node, which may not always be feasible.

2.2.3 Communication Schemes

DFL can be supported by synchronous, asynchronous, or semi-synchronous communication methods [2].

In synchronous communication, each node completes its local training before sending its model parameters to neighbouring nodes. In return, they receive the updated model that they need to integrate. This process is repeated for several federation rounds. However, this method may lead to slow convergence, since it depends on the completion times of slower or idle participants. In contrast, asynchronous communication allows participating nodes to send and receive model parameters independently of one another. This approach promotes faster convergence but can lead to higher communication costs and reduced generalization due to parameter updates being out of sync. Semi-synchronous communication strikes a balance between the two. Within this scheme, each node continues local training until it reaches a predefined synchronization point. The number of local training rounds each node performs thus depends on the amount of data it has and its available resources. Additionally, DFL can use either peer-to-peer or gossip communication methods for transmitting model parameters between federation nodes [2].

2.2.4 Challenges

While DFL offers several benefits, it also presents new challenges, as highlighted by [2]. Below is a summary of some of the challenges noted by [2].

- **Malware Attacks:** Since DFL lacks a centralized entity to coordinate the cooperation of its participants, a malicious user can easily create and deploy malware as a new network participant. If these malicious entities can propagate widely within the network, they may disrupt its functionality and alter the network's behaviour. Fully connected networks, star-structured networks, and random networks are particularly vulnerable to security threats due to their numerous connections to neighbouring nodes. Additionally, the significant volume of communication in DFL can amplify the impact of malware.
- **Adversarial Attacks:** DFL is susceptible to adversarial attacks, particularly poisoning attacks. Numerous nodes in DFL increase the risk of faulty behaviour from one or more of these nodes. Since local training data and processes are concealed from other nodes, verifying the authenticity of received updates becomes impossible. This lack of transparency can negatively impact secure aggregation protocols since malicious nodes may contribute misleading data, which makes it challenging to audit local updates when participants generate vastly different results. Moreover, without a central authority to verify node identities, DFL is vulnerable to Sybil attacks. Sybil attacks refer to a malicious entity creating multiple fake pseudonymous nodes to overwhelm DFL networks. This entity may also launch Distributed Denial of Service (DDoS) attacks, using these pseudonymous nodes to flood a targeted node with requests, preventing it from responding and disrupting communication.
- **Due to asynchronous communication in DFL,** model parameter exchanges between nodes might experience instability, leading to nodes processing and exchanging data at varying rates. This could lead to some nodes handling outdated information, leading to inconsistencies in the learning process.

- Heterogeneous aggregator devices can have different computational capabilities and communication protocols. This may lead to inconsistencies across various network segments, which ultimately negatively impacts the accuracy of models for a group of nodes.
- Asymmetric topologies can lead to uneven data distributions and processing capabilities among nodes, resulting in difficulties in achieving a uniformly distributed learning process.
- Participants may have heterogeneous datasets, including differing sizes, distributions, and data quality, which could affect model performance.
- DFL applications often involve frequent transmission of model parameters, including large amounts of data. This issue becomes more pronounced as the federation advances or as the model’s complexity increases. The communication-intensive nature of DFL can lead to significant network overhead, which may cause applications to struggle to meet their response time, quality, or resource utilization requirements.
- Participant mobility should not affect the stability and reliability of learning when participants join or leave the federation at any time. Continuous learning must be maintained despite the flow of participants.
- Federation models and participants must be trustworthy to ensure that the technology is used effectively for the task at hand. In a DFL setting, it is essential to assess the confidence of the participants and the federated models. This assessment helps to prevent dishonest behaviour and maintain the integrity of the federated learning process.
- Scalability: Federations must be resilient to participant dropouts and availability issues. As the number of participants in a federation increases, the need to dynamically select participants based on their availability, network connection, and trustworthiness becomes more important. Additionally, techniques are required to minimize communication and computation overhead, particularly in large-scale federations.

There are various security countermeasures in place to prevent malicious participation and protect data privacy, as outlined by the authors in [2]. These measures include but are not limited to, the following: Differential Privacy (DP) relaxations, the combination of DP mechanisms with secure aggregation and additive Homomorphic Encryption (HE), random neighbor selection, dynamic IP and port switching, Secure Multiparty Computation (SMC), and Trusted Execution Environments (TEE). Additionally, blockchains can be utilized to ensure the decentralization of the architecture, enhance inter-participant reliability, and guarantee the immutability of exchanged models.

2.3 NEBULA

NEBULA (previously Fedstellar) is a FL platform designed to facilitate the training of federated models within both centralized and decentralized architectures [4]. NEBULA

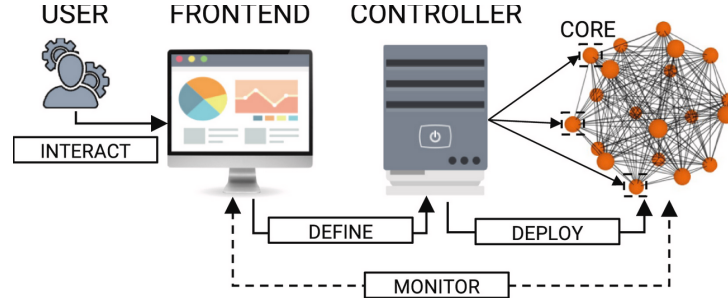


Figure 2.3: Overall architecture of NEBULA [5]

allows to create and manage federations. The user can specify the type of federation architecture and the network topology such as fully connected, ring, star, random or a custom topology.

Specifically, NEBULA allows to create and manage federations by offering the following features [5]:

- Customizing the number of participating devices.
- Generating and deploying network topologies to link within a federation, including fully connected, ring, star, random or custom topologies.
- Usage of different ML and Deep Learning (DL) models such as Deep Neural Networks and Convolutional Neural Networks are supported by well-established libraries such as PyTorch and Scikit-learn.
- Usage of standardized benchmark datasets, such as MNIST or CIFAR-10. Additionally, users can choose between Independent and Identically Distributed (IID) or non-IID data distributions.
- Usage of aggregation algorithms, such as FedAvg or more specialized alternatives such as Krum, or Trimmed Mean.
- Monitoring of KPIs, including resource KPIs, communication KPIs, federated model KPIs. Resource KPIs include computational load, memory usage, and local model accuracy. Communication KPIs focus on communication latency, data transmission speed and successful data transfer. Federated model KPIs assess the model accuracy, precision, recall, F_1 score, training time, and convergence speed of each local model.

As shown in 2.3, the architecture of NEBULA comprises the user, frontend, controller, and core.

The user is responsible for establishing, configuring, and monitoring federated scenarios.

The frontend is an interactive interface, designed to assist the user in managing the platform operation and providing ongoing KPIs performance tracking. Through a real-time monitoring functionality, the user can track the performance and progress of ongoing federated scenarios [5].

The controller serves as the central orchestrator of the platform, facilitating seamless interaction between the frontend and deploying core modules on devices. It interprets user commands from the frontend, manages the entire federated learning scenario, assigns appropriate learning algorithms and datasets, and configures network topologies. The controller handles the deployment of core modules primarily through Docker-based virtualization. By packaging applications and their dependencies into standardized units, Docker ensures a streamlined and replicable deployment process. Furthermore, Docker-level virtualization enables NEBULA to deploy virtualized devices across diverse operating systems, including Windows, Linux, and macOS. The controller also establishes a virtual internal network that mirrors the user-defined network topology, assigning distinct sockets to each device to ensure seamless collaboration [5].

Core is deployed on each device within the federation and is responsible for managing most of the computation and communication required for the successful operation of FL. It is subdivided into the monitoring module, the training module for handling the learning process, and the communications module for facilitating interactions between federated participants and with the frontend [5].

The monitoring module periodically collects data about the ongoing federated scenario. It gathers metrics categorized into three primary groups: Federated model, resource usage, and communications status. The federated model category includes metrics related to the model size, learning progress, model synchronization frequency, and the number of training rounds. The resources usage category captures, with the aid of auxiliary libraries such as psutil and pynvml, parameters such as CPU usage, GPU usage, memory utilization, network bytes, and component temperatures. The communications status category comprises metrics related to communication operations, such as status updates, latency, active connections, and data transmission rates [5].

The training module is responsible for data management and model training. It leverages Lightning with PyTorch as a training backend, simplifying complex ML logistics. It supports various aggregation algorithms for ML models and deep learning (DL) models.

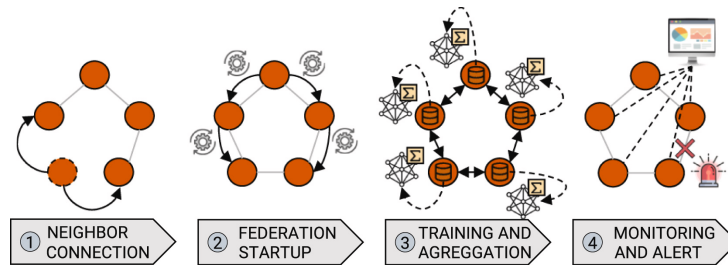


Figure 2.4: Communication process in NEBULA [5]

The communications module is primarily responsible for communicating and transmitting essential information within the network, including status updates to the frontend and asynchronous exchange of model parameters with neighbouring nodes. Figure 2.4 outlines the four steps of the communication process [5]:

- Connecting all nodes with their neighbours: Each device establishes communication links with its immediate neighbours, storing these connections through open sockets,

enabling efficient information exchange. Advanced encryption mechanisms, using both symmetric and asymmetric encryption from the pycryptodome library, ensure secure key management and message confidentiality.

- **Initiating the federation setup:** A designated node broadcasts the federation's configuration, including the number of training rounds. The nodes adjust their operations according to their assigned roles and responsibilities within the federation.
- **Executing the training process with decentralized aggregation:** Each node trains its local model with its respective data independently. A thread-based system manages asynchronous message exchange through incoming and outgoing message queues. When all neighbours have shared their parameters, or after a customizable timeout period, decentralized aggregation is triggered.
- **Monitoring and alerting:** Nodes continuously assess and report on any disruptions, such as link failures or devices leaving the topology. Affected neighbouring nodes are then removed from the internal queue of open sockets. Additionally, nodes maintain unidirectional connections with the frontend via secure WebSockets, enabling continuous transmission of metric data and ensuring constant federation monitoring.

Additionally, figure 2.5 provides a detailed explanation of each participant's internal cycle, aligning with the steps outlines above [5].

```

Require:  $R$ : local round,  $\alpha$ : learning rate,  $\lambda$ : regularization parameter,  $S_j$ :
           socket to neighbor  $j$ ,  $D$ : local dataset
1:  $D_{\text{Train}}, D_{\text{Test}} \leftarrow \text{split}(D)$ 
2: for  $r$  in  $R$  do
3:   Initialize Local Model with Parameters  $\theta$ 
4:   for each  $(x, y)$  in  $D_{\text{Train}}$  do
5:      $\theta \leftarrow \theta - \alpha(\nabla_{\theta} J(\theta, x, y) + \lambda\theta)$  ▷ Train
6:   end for
7:   for  $j$  in  $N$  do ▷ Send
8:     Model Parameters to  $j$  via  $S_j$ 
9:   end for
10:  while not Timeout do
11:    for  $j$  in  $N$  do ▷ Receive
12:       $RP_j \leftarrow$  Model Parameters from  $j$  via  $S_j$ 
13:    end for
14:    end while
15:     $\theta \leftarrow \frac{1}{|N|+1}(\theta + \sum_{j \in N} RP_j)$  ▷ Aggregate (FedAvg)
16:    Update Local Model with  $\theta$ 
17:  end for
18:  for each  $(x, y)$  in  $D_{\text{Test}}$  do
19:     $y_{\text{pred}} \leftarrow$  Predict with Local Model on  $x$  ▷ Test
20:     $L \leftarrow \frac{1}{|D_{\text{Test}}|} \sum_{i=1}^{|D_{\text{Test}}|} l(y, y_{\text{pred}})$  ▷ Compute Loss
21:  end for
22:  Send metrics to frontend ▷ Report Metrics

```

Figure 2.5: Federated participant cycle in NEBULA [5]

Figure 2.6 shows the sequence of interactions of a typical FL scenario run on NEBULA. The configuration of the scenario commences the deployment process. Once the controller receives the configuration from the frontend, it generates the network topology and establishes core instances for all virtualized devices. It then securely distributes the predefined configuration to each device, ensuring consistent and secure provisioning across

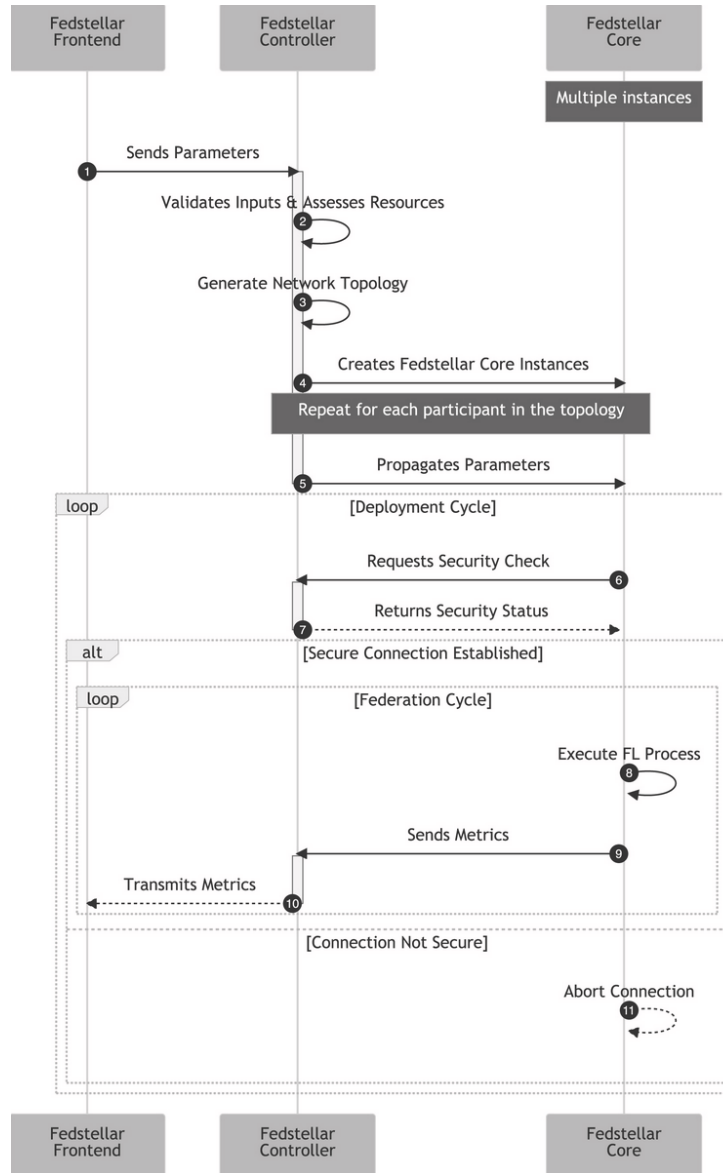


Figure 2.6: Sequence diagram showing the interaction of NEBULAs (previously Fedstellar) components [5]

all devices. The process concludes with the execution of the FL operation, accompanied by the monitoring and processing of key metrics that provide insights into the scenario’s performance and efficiency.

2.4 Related Work

Several studies have explored the feasibility of deploying FL on resource-constrained devices in order to demonstrate that FL can effectively operate in such environments. This section highlights some notable practical implementations in recent years.

The authors of [17] implemented a CFL framework to train ML algorithms involving heterogeneous FL clients. Their setup included Raspberry Pi devices and virtual machines as client nodes, with a Windows PC serving as the central server. The framework utilized FedAvg as the aggregation algorithm and successfully trained an image classification model (CIFAR-10) across Raspberry Pi clients.

To enhance FL performance on edge devices, [18] proposed two improved aggregation algorithms: FedAM and FedAALR. These algorithms are modifications of FedAvg, designed to enhance scalability and model convergence in resource-limited environments. The authors deployed and evaluated their approach on a testbed of 40 Raspberry Pi devices, showcasing the effectiveness of their optimization techniques.

Another practical implementation, FLoES (Federated Learning on Edge Systems), was introduced in [19]. FLoES is a lightweight and flexible FL software library specifically designed for single-board computers. It has been deployed and tested on Raspberry Pi devices and NVIDIA Jetson Nanos.

Additionally, [20] investigated real-world federated learning scenarios by deploying FL on 5, 10, and 20 Raspberry Pi devices. Their work utilized the FedAvg algorithm for model aggregation. Furthermore, they proposed two extensions to standard FL, enabling decentralized communication among participating devices in a peer-to-peer fashion [20].

Chapter 3

Architecture

The architecture needs some adjustments for NEBULA to run on physical devices, primarily concerning deploying the core module on edge devices and the monitoring module for real-time data collection. When NEBULA runs in a virtualized environment using Docker containers, the frontend, controller, and virtualized devices all operate on the same machine. This allows the virtualized devices to access configuration data and share real-time metrics via a shared directory. However, direct file-sharing mechanisms are unavailable in a real-world deployment, and a more explicit communication protocol between the controller and participant nodes is needed.

The primary modifications of the architecture include:

- Establishing a local server on each node to receive configuration files from the controller via HTTP POST requests.
- Enabling nodes to send collected monitoring metrics back to the controller through periodic network transmissions.
- Extending the monitoring module to track energy consumption on participating devices during the FL process.

Figure 3.1 shows the high-level overview of the adjusted architecture. In general, the core structure remains largely unchanged. To initiate the FL process, nodes must first receive configuration data defined by the user. Each node hosts a local server, which listens for incoming POST requests from the controller. Specifically, the server:

1. Receives the FL configuration from the controller.
2. Saves the configuration locally.
3. Terminates the local server. This is necessary to free resources and free the port occupied by the server since it will later be needed to establish a TCP connection with a neighbouring node

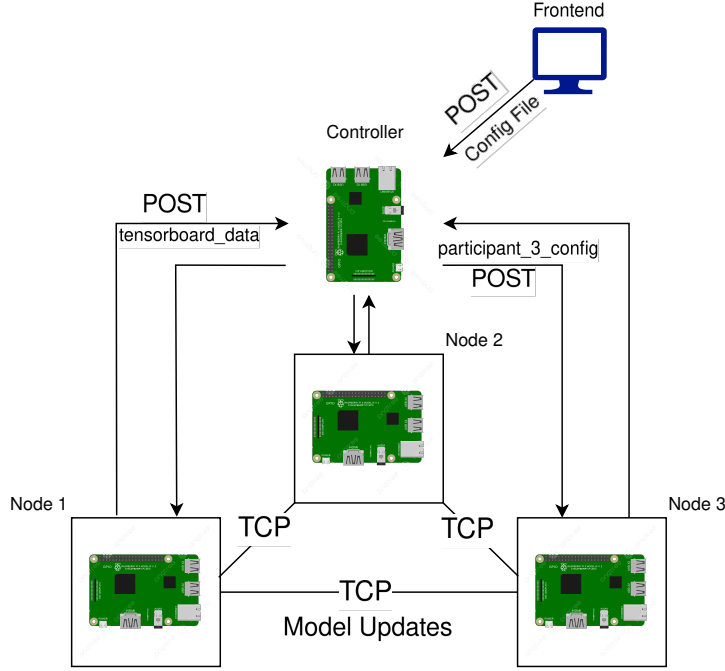


Figure 3.1: Overall architecture of NEBULA on Raspberry Pis

Once the controller receives the configuration from the frontend, it will transfer it to each device via POST requests. Once the configurations have been saved and the server terminated, the nodes proceed with the setup process according to the received configurations. After the setup process, the nodes transition into the communication phase, where they establish TCP connections with neighbouring nodes based on the predefined topology (explained in Figure 2.4).

3.1 Real-Time Metric Collection and Power Monitoring

In a virtualized environment, each device logs performance metrics to a shared directory, making them readily available to the controller. To replicate this behaviour in a real-world scenario, devices must transmit these metrics via network requests. Specifically:

1. Each node periodically sends collected metrics to the controller via HTTP POST requests after every training round.
2. The controller listens for these incoming metric files and stores them in a designated directory for real-time visualization.

Additionally, the monitoring module tracks power consumption when energy measurements are enabled. Specifically, the monitoring module reports power measurements in two ways:

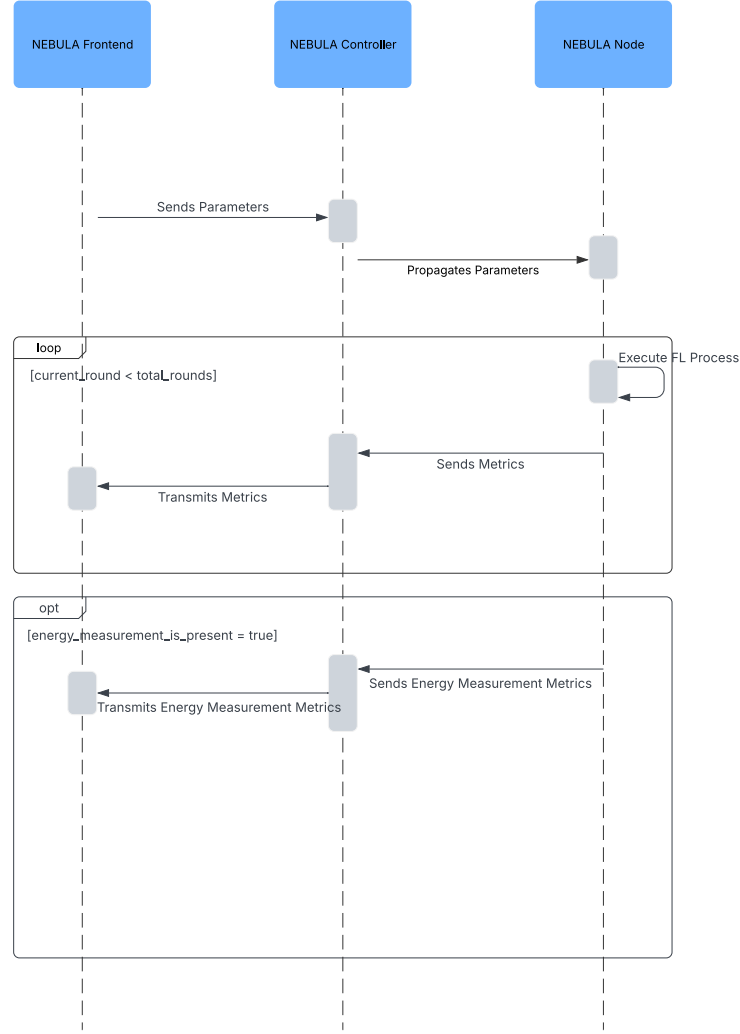


Figure 3.2: Sequence diagram showing the simplified interaction between NEBULAs frontend, controller, and nodes.

- Periodic metric transmission: Power measurements are sent together with the performance metrics (e.g., CPU usage, memory consumption) every few seconds. This enables real-time visualization of the power consumption.
- Batch reporting: To ensure accuracy, power consumption is measured every second and stored locally throughout the FL process. At the end of the FL scenario, they are aggregated and sent to the controller. This minimizes unnecessary network overhead and reduces the risk of interference with FL operations.

Once the FL process concludes, each node verifies whether energy consumption measurements were enabled. If so, the recorded power metrics are transmitted to the controller for storage and display in the real-time monitoring view.

Figure 3.2 presents a simplified version of the adjusted interaction sequence described above, illustrating the key interactions between NEBULA's frontend and the controller and between the controller and the devices.

This diagram highlights the sequential communication flow:

1. The frontend sends the FL configuration to the controller.
2. The controller distributes configurations to all participant nodes via HTTP POST requests.
3. Nodes acknowledge receipt and apply the configuration.
4. Nodes initiate FL training, exchanging model updates and monitoring metrics.
5. Nodes transmit collected metrics to the controller at periodic intervals.
6. At the end of the FL process, energy consumption data is transmitted (if enabled).

Chapter 4

Implementation

This section outlines the key implementation steps required to deploy NEBULA on resource-constrained devices, such as the Raspberry Pi 4 and NVIDIA Jetson Nano. The provided code snippets highlight essential steps, with some parts omitted for clarity. A more complete implementation can be found in the appendix.

4.1 Deployment Environment

For this deployment, four Raspberry Pi 4 Model B devices and an NVIDIA Jetson Nano Developer Kit were utilized. One Raspberry Pi serves as the controller node, handling communication with the frontend and coordinating the FL process. The remaining three Raspberry Pis act as participant nodes. The Jetson Nano acts as an additional participant node in the federated learning process. Additionally, JT-TC66C USB 3.0 volt- and ammeters were used to monitor energy consumption across participating devices during the federated learning process. All devices communicated over a local Ethernet network.

Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B is a single-board computer, equipped with a 64-bit quad-core ARM Cortex-A72 processor. Two of the Raspberry Pis have 4 GB of RAM and one has 2 GB of RAM. It utilizes a 32 GB microSD card as its primary storage medium and runs on Debian GNU/Linux 12 (Bookworm). The device supports both wired (Gigabit Ethernet) and wireless (Wi-Fi, 2.4/5 GHz) networking. The board requires a 5V, 3A USB-C power supply for stable operation. It offers performance comparable to entry-level x86 PC systems [21].

NVIDIA Jetson Nano

The NVIDIA Jetson Nano is a compact, energy-efficient computing platform designed for AI and embedded applications. According to [22], it features:

- A 128-core NVIDIA Maxwell GPU and a quad-core ARM Cortex-A57 CPU with a maximum frequency of 1.43GHz.
- 4GB LPDDR4 RAM with a memory bandwidth of 25.6GB/s.
- High-speed connectivity including USB 3.0 (5 Gbps), USB 2.0, and PCIe Gen2 (1×4 lane).
- Gigabit Ethernet for stable network communication.
- Adjustable power consumption between 5W and 10W.

The Jetson Nano runs Ubuntu 18.04.6 LTS, a Debian-based Linux distribution. Its primary storage medium is a 30 GB microSD card.

JT-TC66C Energy Measurement Device

The JT-TC66C is a compact, multifunctional USB-C multimeter with a refresh rate of 2 Hz. It supports various Quick-Charge standards such as QC2.0, QC3.0, Huawei FCP/SCP, Samsung AFC, and PD. Additionally, it features a Micro-USB port for PC connectivity, enabling data logging and analysis [23]. According to [23], it supports:

- Voltage measurement range: 0.005V to 30V with a resolution of 0.1mV.
- Current measurement range: up to 5A with an accuracy of $\pm 1\%$.

Communication and Orchestration

To enable communication between participant nodes and the controller, as well as inter-node communication, the following tools are utilized:

- FastAPI: Provides asynchronous REST endpoints, enabling communication between participant nodes and the controller.
- Uvicorn: An ASGI-compatible web server that handles web connections from API clients, allowing FastAPI to process and serve requests efficiently [24].
- Requests Library: A HTTP library for Python, used to transmit FL scenario metrics from the participant nodes to the controller.
- TCP Sockets: Inter-node communication is managed through existing TCP connections, established using Python's asyncio library. This functionality is already integrated into NEBULA and does not require further modifications.
- Python Virtual Environment: A virtual environment is used to isolate dependencies, ensuring that installed libraries remain confined to the application without affecting the broader system.

4.2 Controller and Node Deployment

To enable NEBULA to run on resource-constrained devices, various adjustments to NEBULA's code must be made, and each device must be configured with a suitable environment. This section outlines the required implementation steps to deploy NEBULA on Raspberry Pis and the NVIDIA Jetson Nano Kit.

4.2.1 Controller Deployment

As described earlier, one of the Raspberry Pis functions as the controller in the federated learning setup. The primary modifications required for the controller include:

1. Communicating the FL configurations to each participant device.
2. Receiving real-time FL metrics to enable visualization on the frontend.

Distributing FL Configurations

To distribute FL configurations, the `start_physical_nodes` method is introduced in the controller's implementation. This method assigns the IP of the start node to each participant's configuration file and transmits the configuration using HTTP POST requests via Python's `requests` library:

```
def start_physical_nodes(self):
    for idx, node in enumerate(self.config.participants):
        node["network_args"]["ip_start"] = self._start_node_ip
        with open(f"{self.config_dir}/participant_{node['device_args']}['idx']}.json", "w") as f:
            json.dump(node, f, indent=4)

    for idx, node in enumerate(self.config.participants):
        ip = node["network_args"]["ip"]
        port = node["network_args"]["port"]

        with open(f"{self.config_dir}/participant_{idx}.json", "r") as f:
            node_json_config = json.load(f)

        url = f"http://{ip}:{port}/start"

        response = requests.post(url, json=node_json_config)
```

Receiving Real-Time Metrics

To receive real-time FL metrics, a REST API endpoint is implemented. This endpoint listens for incoming metric files from participant nodes, saves them in the designated location, and ensures proper directory creation before storing the files:

```

@app.post("/nebula/controller/upload-event-files")
async def upload_event_files(files: list[UploadFile]):
    for file in files:
        path = file.filename
        filename = path.split("/")[-1]
        abs_location = f"''.join(path.split('/')[:-1])/"
        rel_location = f"../../../../{abs_location}"
        os.makedirs(rel_location, exist_ok=True)
        with open(f"{rel_location}{filename}", "wb") as f:
            content = await file.read()
            f.write(content)
    return Response(content="Success", status_code=200)

```

4.2.2 Participant Node Deployment

In order for NEBULA to work on the participant nodes, each device requires a local server that listens for incoming configuration files. Once the configuration file arrives, the server must be shut down to free up system resources and release the occupied port for subsequent use. This is achieved by running a custom Uvicorn server on each node which is shown in the following code snippet:

```

class Server(uvicorn.Server):
    def install_signal_handler(self):
        pass

    @contextlib.contextmanager
    def run_in_thread(self):
        thread = threading.Thread(target=self.run)
        thread.start()
        try:
            while not self.started:
                time.sleep(1e-3)
            yield
        finally:
            self.should_exit = True
            thread.join()

```

The Server class extends `uvicorn.Server` and uses Python's context manager to run the server in a separate thread. This allows the server to listen for incoming POST requests while keeping the main thread available for other tasks. The context manager also simplifies lifecycle management by ensuring the server shuts down gracefully when no longer needed, releasing the occupied port.

To start the server the following code is used:

```

def main():
    config = uvicorn.Config("start:app", host="0.0.0.0")
    server = Server(config=config)

    with server.run_in_thread():
        event.wait()

```

The `main` function initializes the Uvicorn server and runs it in a background thread. The `event.wait()` function ensures the server remains active until the configuration file arrives, at which point the `event` is set, causing the `with` block to exit and the server to shut down.

To receive the configuration file, the server listens for POST requests. The following code snippet demonstrates the REST endpoint used by the controller to send configuration data:

```
@app.post("/start")
async def start_node(request: Request, background_tasks: BackgroundTasks
):
    # remaining implementation

    save_config_file(path, file_name, data)

    background_tasks.add_task(run_scenario, path, file_name)
    return
```

Once the configuration file is received, the nodes save it as a JSON file. The `run_scenario` function is then added to the background tasks to initiate the federated learning setup process.

The `run_scenario` function starts the FL scenario by executing the `node.py` script with the provided configuration, sets the threading event to signal the server to shut down, and runs the node as a subprocess:

```
def run_scenario(path_config_dir, file_name):
    # remaining implementation
    command = ["python3", node_path, config_path]

    event.set()
    try:
        subprocess.run(command, check=True)
    # remaining implementation
```

After deployment, nodes initiate a TCP connection with their neighbors and establish the federation. The deployment process follows these steps:

1. If a node is not the start node, it notifies its neighbors by sending a **FEDERATION_READY** message.
2. If a node is the start node, it waits until the federation is ready and then sends a **FEDERATION_START** message.
3. Neighboring nodes forward the **FEDERATION_START** message until all nodes are notified.

However, during testing, this implementation encountered complications, particularly in ring-structured topologies, where the federation did not initialize as expected. To address these issues, minor adjustments were made, which are described in the following subsection.

Fixing Deployment Issues in a Ring-Structured Network

During testing, the federation failed to deploy in a ring topology due to two key issues:

1. `FEDERATION_READY` messages were not forwarded, preventing the start node from receiving readiness confirmation from indirect neighbors.
2. Each node sent the `FEDERATION_READY` message only once, meaning if the start node was slow to initialize, it could miss messages from other nodes.

To address these problems, the following adjustments were implemented:

- Nodes that receive a `FEDERATION_READY` message for the first time must forward it to their neighbors.
- A node must wait until a direct or indirect connection to the start node is established before sending its `FEDERATION_READY` message.

The following code snippets demonstrate these adjustments:

```
async def deploy_federation(self):
    await self.federation_ready_lock.acquire_async()
    if self.config.participant["device_args"]["start"]:
        # remaining implementation

    else:
        ip_start_node = self.config.participant["network_args"]["ip_start"]
        while not ip_start_node in self.cm.ready_connections:
            await asyncio.sleep(1)
        message = self.cm.mm.generate_federation_message(nebula_pb2.FederationMessage.Action.FEDERATION_READY)
        await self.cm.send_message_to_neighbors(message)
```

```
async def handle_incoming_message(self, data, addr_from):
    try:
        # remaining implementation

    elif message_wrapper.HasField("federation_message"):
        if await self.include_received_message_hash(hashlib.md5(data).hexdigest()):
            if self.config.participant["device_args"]["proxy"]
            or message_wrapper.federation_message.action ==
            nebula_pb2.FederationMessage.Action.Value("FEDERATION_START") \
            or message_wrapper.federation_message.action
            == nebula_pb2.FederationMessage.Action.Value("FEDERATION_READY"):
                await self.forwarder.forward(data, addr_from=addr_from)
                await self.handle_federation_message(source, message_wrapper.federation_message)
        elif message_wrapper.HasField("model_message"):
```

With this improved design the configuration files are properly propagated, nodes establish connections correctly, and the FL process starts reliably in various network topologies. Additionally, it ensures that once the FL process begins and the server is no longer needed, the resources are freed for training and communication.

4.3 Real-Time Metric Transmission in NEBULA

NEBULA uses TensorBoard to monitor and collect metrics from each participating device throughout federated learning process. The gathered metrics are displayed on the frontend in real time, enabling continuous monitoring of the FL process.

Each device transmits its collected metrics at the end of each training round, as shown in the following implementation:

```
async def _learning_cycle(self):
    last_file = None
    while self.round is not None and self.round < self.total_rounds:
        # remaining implementation

        self.trainer.on_round_end()
        last_file = await self._send_tensorboard_files(last_file)
        self.round = self.round + 1

    # remaining implementation
```

The `self._send_tensorboard_files` determines which newly gathered metrics have not yet been sent to the controller and sends them using Python's requests library.

```
async def _send_tensorboard_files(self, last_file):
    content = os.listdir(self._log_dir)
    content.sort()
    if len(content) > 0:
        files_to_send = self._get_files_to_send(content, last_file)
        last_file = self._get_new_last_file(content)

    files_to_upload = self._get_files_to_upload(files_to_send)

    response = requests.post(
        f"http://{self.controller}/nebula/controller/upload-event-files",
        files=files_to_upload
    )
    # remaining implementation
    return last_file
```

This ensures that the gathered metrics are available on the controller after each training round, supporting NEBULA's real-time monitoring capability.

4.4 Reporting Energy Consumption

Each node records its energy consumption throughout the FL process to enhance NEBULA's real-time monitoring capabilities and assess its feasibility on resource-constrained devices. The JT-TC66C USB-C multimeter is used to measure power usage, real-time voltage and current readings.

For a node to measure energy consumption, the JT-TC66C device must be connected via USB-C, with the measuring device linked to the node via a Micro-USB port. When properly connected, the TC66C appears as an "ACM" device in Linux under `/dev/ttyACMx` (typically `x=0`).

Each node uses a modified Python utility based on [25] to establish serial communication with the device and retrieve AES-encrypted power consumption data.

4.4.1 Power Measurement and Polling

The TC66C class initializes the serial connection and configures AES decryption for processing power consumption readings:

```
class TC66C:
    # Remaining implementation

    def __init__(self, port_dev=None):
        self._cumulative_energy = 0
        self._current_power = 0
        self._detailed_power_consumption = []
        STATIC_KEY = [0x58, 0x21, ...] #AES key

        if port_dev == None:
            port_dev = DEFPORT
        try:
            self._SIF = serial.Serial(
                port=port_dev,
                baudrate=115200,
                bytesize=8,
                parity='N',
                stopbits=2,
                xonxoff=0,
                rtscts=0,
                dsrdtr=1,
                timeout=5)
            sleep(1.0)
        except Exception as e:
            print(f"failed to open {port_dev}: {e}")

        self._AES = AES.new(bytes(STATIC_KEY), AES.MODE_ECB)
```

Each node retrieves power consumption data from the JT-TC66C device by sending the `getva` command over a serial connection. This command prompts the device to return three encrypted 64-byte data blocks containing power-related measurements. The polling

process operates asynchronously to avoid blocking the federated learning process. Data retrieval occurs at a fixed interval of 1 second:

```

async def start(self):
    start = monotonic()
    now = monotonic()-start
    try:
        while True:
            now = monotonic()-start
            pd = self.Poll()
            self._current_power = pd.Power
            self._detailed_power_consumption.append(
                self._current_power)
            self._cumulative_energy += self._current_power

            elapsed = (monotonic()-start) - now
            if elapsed < 1:
                await asyncio.sleep(1 - elapsed)
    except KeyboardInterrupt:
        print("Finished")

```

The Poll method performs the following steps:

1. Sends the `getva` command to the JT-TC66C device via serial communication.
2. Reads and decrypts the AES-encrypted 192-byte response.
3. Extracts key metrics such as voltage (V), current (A), and power consumption (W).
4. Saves the data for real-time monitoring and analysis.

The relevant implementation is shown in the following code snippet:

```

def Poll(self):

    if not self._SIF.isOpen():
        self._SIF.open()
    self.SendCmd('getva')

    buf= self._SIF.read(192)
    try:
        data = self._AES.decrypt(buf)
    except:
        print('decrypt_error')

    # remaining implementation

    PAC1_VOLT = 12 # volts in 100uV
    PAC1_AMPS = 13 # current in 10uA
    PAC1_PWR = 14 # power in 100uW

    # remaining implementation

```

```

pac1 = struct.unpack('<4s4s4s13I', data[0:64])
pac2 = struct.unpack('<4s15I', data[64:128])
pac3 = struct.unpack('<4s15I', data[128:192])

# remaining implementation

pd = self.PollData(
    ...
    Volt          = float(pac1[PAC1_VOLT])*1E-4,
    Current = float(pac1[PAC1_AMPS])*1E-5,
    Power         = float(pac1[PAC1_PWR])*1E-4,
    ...
)

return pd

```

4.4.2 Logging and Reporting Power Consumption

To enable real-time power monitoring, the TC66C class is integrated into NEBULA's `Reporter` module, and the recorded values are reported using TensorBoard. The `start` method of `Reporter` attempts to initialize TC66C. If successful, power and energy consumption measurements begin asynchronously, ensuring that data collection does not interfere with the FL process:

```

async def start(self):
    await asyncio.sleep(self.grace_time)
    try:
        self.tc66c = TC66C(self.trainer.logger, "/dev/ttyACM0")
        asyncio.create_task(self.tc66c.start())
    except Exception as e:
        print(f"Failed to initialize TC66C: {e}")
        asyncio.create_task(self.run_reporter())

```

The `Reporter` module extends the `_report_resources` method to log energy metrics alongside CPU, memory, and network usage. The power consumption is logged using the following method:

```

async def _report_resources(self):
    # Remaining implementation

    power = self.tc66c.current_power()
    energy = self.tc66c.cumulative_energy()

    current_connections = await self.cm.get_addrs_current_connections(
        only_direct=True)

    resources = {
        "CPU/CPU_global(%)": cpu_percent,
        ...
        "Power/Power_Consumption(W)": power,
        "Power/Total_Energy(J)": energy,
    }

```

```
}  
  
self.trainer.logger.log_data(resources)  
  
# Remaining implementation
```

With the above procedure, the power usage data is consistently logged and available for real-time monitoring on NEBULA's frontend.

Chapter 5

Evaluation

To assess the performance and feasibility of deploying NEBULA on resource-constrained devices, a series of experiments were conducted. These experiments evaluated the effectiveness of the platform under two scenarios:

- Physical Scenario: Running NEBULA on real-world edge devices, considering constraints such as limited computational power, memory, and energy consumption.
- Running NEBULA in a Docker-based virtual environment, serving as a baseline for comparison.

Table 5.1 presents a comparative overview of these two deployment settings.

Table 5.1: Comparison of validation scenarios in NEBULA using physical and virtualized environments.

Characteristic	Physical Scenario	Virtualized Scenario
Participant Characteristics	4 (3 Raspberry Pi 4/1 NVIDIA Jetson Nano)	4 (Docker containers)
Dataset	MNIST [26] FashionMNIST [27]	MNIST FashionMNIST
Federated Model	MLP: Multi-layer Perceptron	MLP
Network	Fully connected, Star, Ring, Random	Fully Connected
Federation Architecture	DFL	DFL

5.1 Participant Characteristics

The physical scenario was deployed as described in Section 3, consisting of four edge devices: three Raspberry Pi 4 Model B boards and one NVIDIA Jetson Nano Developer Kit. The Jetson Nano introduces heterogeneity to the system by featuring a weaker CPU (Quad-core Cortex-A57 at 1.43 GHz) than the Raspberry Pi 4 (Quad-core Cortex-A72 at 1.5 GHz), but compensating with superior GPU capabilities (128-core Maxwell GPU).

The virtualized scenario replicated the same number of participant devices using Docker containers on a host machine with an AMD Ryzen 7 5825U CPU (2.0-4.5 GHz), 16 GB RAM, and 512 GB storage.

5.2 Dataset and Federated Model

Both physical and virtualized scenarios were evaluated using the MNIST and FashionMNIST datasets. MNIST is a well-established dataset containing handwritten digit images, frequently used for benchmarking ML models [26]. FashionMNIST, designed as a direct drop-in replacement for MNIST, contains grayscale images of Zalando’s fashion articles and is commonly used for benchmarking machine learning algorithms [27].

In both scenarios, NEBULA employed a Multi-Layer Perceptron (MLP) model, a widely used class of artificial neural networks known for its strong learning potential, robustness to noise, nonlinearity, parallelism, fault tolerance, and generalization capabilities [28]. The federated models were aggregated using the FedAvg algorithm. To simulate real-world conditions, where data distribution across devices is often uneven, both physical and virtualized scenarios used non-IID data [5].

During the experiments, participant devices underwent ten federation rounds, with local models training for one epoch per round. These ten rounds encompassed the entire FL cycle, including parameter transmission and aggregation.

5.3 Network Topology and Federation Architecture

In the physical scenario, four different network topologies were implemented: fully connected, star, ring, and random. These variations were used to evaluate the impact of different communication structures on the performance of DFL. In contrast, the virtualized scenario only utilized the fully connected topology.

Each topology presents unique advantages and limitations, as discussed in Section 2.2.2. All topologies were tested using the MNIST and Fashion-MNIST datasets to assess their effectiveness.

Figure 5.1 illustrates the network topologies used in the evaluation. Each node within these topologies functions as an aggregator node (refer to Section 2.2.1 for details on aggregation mechanisms).

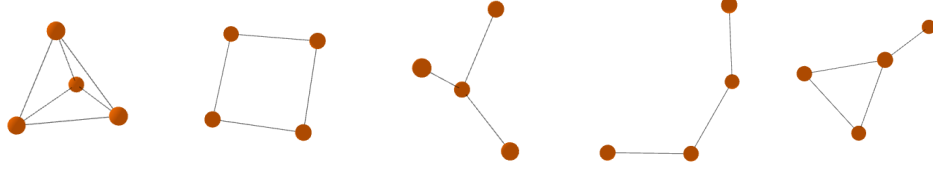


Figure 5.1: Network topologies evaluated in NEBULA. From left to right: Fully connected, ring, star, random (evaluated with MNIST), and random (evaluated with Fashion-MNIST).

A DFL architecture was selected for both the physical and virtualized scenarios, enhancing the system’s robustness and resilience [2].

5.4 Results

The evaluation of both physical and virtualized scenarios focuses on key performance metrics, including the F_1 -score of the federated models, global CPU usage (%), RAM utilization, network traffic (MB), power consumption, and total energy consumption throughout the FL scenario. It is important to note that power consumption measurements were conducted on only one device per experiment. Additionally, as observed in the results, one of the Raspberry Pis consistently exhibited twice the RAM usage compared to the other two. This discrepancy arises because one Raspberry Pi has only 2 GB of available RAM, while the other two devices are equipped with 4 GB of RAM (see Section 4).

Fully Connected Topology with MNIST Dataset

Figure 5.2 presents the performance of NEBULA in a physical deployment utilizing a fully connected topology with the MNIST dataset. The results indicate that this topology leads to a rapid improvement in model performance, reaching an F_1 -score of 80% after just five training rounds and terminating with a final F_1 -score of 82%.

The CPU usage exhibited high fluctuations across all participants, with a maximum CPU usage of 42.8% and an average CPU usage of 25.7%. These fluctuations are likely attributed to the computational load associated with local model training and synchronization. Similarly, power consumption followed a fluctuating pattern, mirroring the CPU load. The maximum power consumption reached 3.9 watts, while the average power consumption remained at 3.3 watts. Notably, power consumption and CPU usage followed a cyclical pattern with 10 peaks and troughs, aligning with the 10 training rounds of the FL process. The total energy consumption of the scenario was 1404 joules.

RAM usage across participants gradually increased, likely due to the growing model size over training rounds.

- Participant 0 (Nvidia Jetson Nano) had an average RAM usage of 26.8% and a maximum of 28.6%.

- Participant 1 had the highest memory consumption, with an average of 48.7% and a maximum of 51%.
- Participants 2 and 3 had an average RAM usage of 24.4% and a maximum usage of 26.3%.

Network usage showed a steady increase across all participants, with a total data transmission (MB sent and received) of approximately 30 MB per participant.

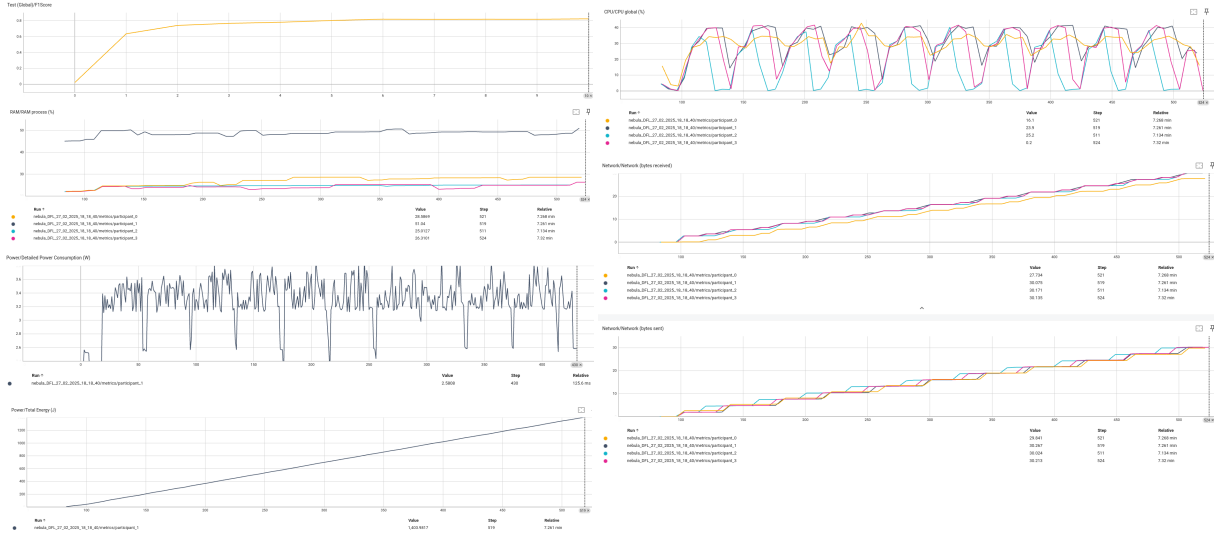


Figure 5.2: Results of the physical scenario with a fully connected topology evaluated with the MNIST dataset.

Fully Connected Topology with Fashion MNIST Dataset

Figure 5.2 illustrates the performance of NEBULA in a physical deployment utilizing a fully connected topology with the Fashion MNIST dataset. Compared to the MNIST experiment, the model exhibited slightly slower improvements, reaching an F_1 -score of 72% after five training rounds and ultimately achieving a final F_1 -score of 81%.

The results indicate a slightly higher average CPU usage compared to the MNIST experiment, with an average of 28.7% and a maximum of 41.9%. As observed in the previous experiment, power consumption followed the cyclical pattern of CPU utilization, a pattern which remains consistent across all experiments. The average power consumption was 3.5 watts, with a peak consumption of 4.5 watts. The total energy consumption for this scenario amounted to 1350 joules.

The RAM usage across participants remained similar to that observed in the MNIST experiment:

- Participant 0 (Nvidia Jetson Nano) had an average RAM usage of 25.8% and a maximum of 27.2%.

- Participant 1 had an average of 49.5% and a maximum of 54.8%.
- Participants 2 and 3 maintained average RAM usages of 24.6%, with maximum values reaching 26.5%.

Network traffic remained consistent across both datasets.

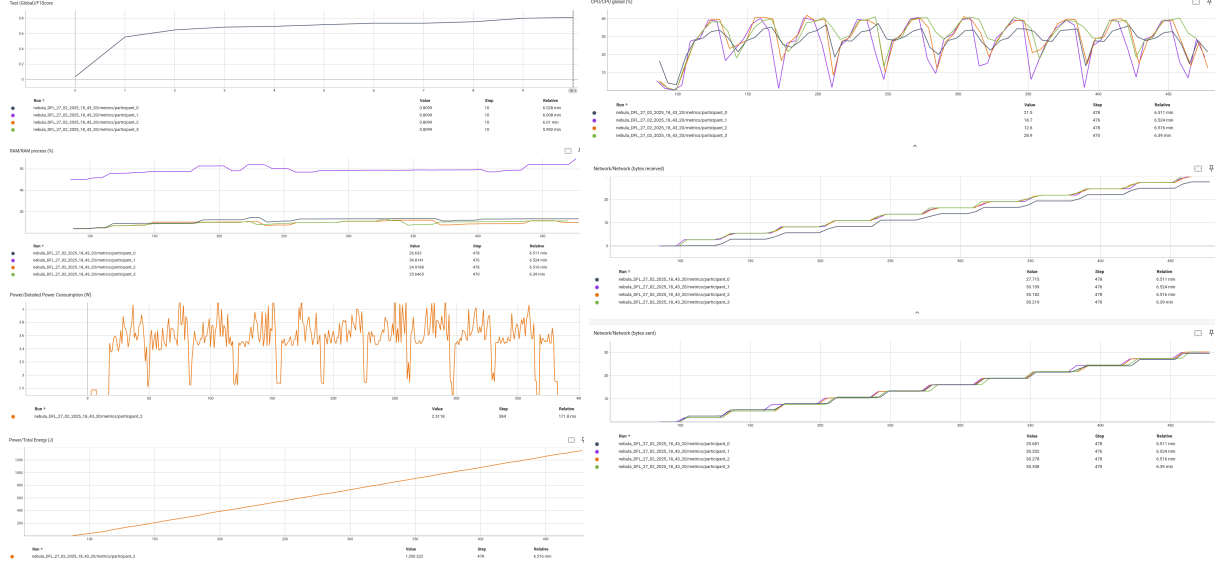


Figure 5.3: Results of the physical scenario with a fully connected topology evaluated with the Fashion-MNIST dataset.

Star Topology with MNIST Dataset

Figure 5.4 presents the performance of NEBULA in a physical deployment using a star topology with the MNIST dataset. Unlike previous experiments, the F1-score reveals notable differences among participants. Participant 0, acting as the central node, achieved the highest model performance, reaching a final F_1 -score of 82%. Participants 1, 2, and 3 achieved final F_1 -scores of 77%, 75%, and 70%, respectively, resulting in an average F_1 -score of 76%.

CPU usage remained consistent with previous experiments, showing no significant deviations. The average CPU usage was 25.8%, with a peak usage of 41.9%. As in previous experiments, power consumption followed the cyclical nature of FL training rounds with an average power consumption of 3 watts, a maximum power consumption of 4.1 watts and a total energy consumption of 1251 joules.

Memory usage across participants remained similar to previous experiments:

- Participant 0 (Nvidia Jetson Nano): Average RAM usage of 26.1%, peaking at 28.5%.
- Participant 1: Average of 48.4%, with a maximum of 50.5%.

- Participants 2 and 3: Average RAM usage of 23.8%, peaking at 24.6%.

Network traffic aligns with expectations for a star topology: the central node (Participant 0) handled the highest communication load, sending and receiving approximately 30 MB - comparable to a node in a fully connected topology. The remaining participants each sent and received around 10 MB.

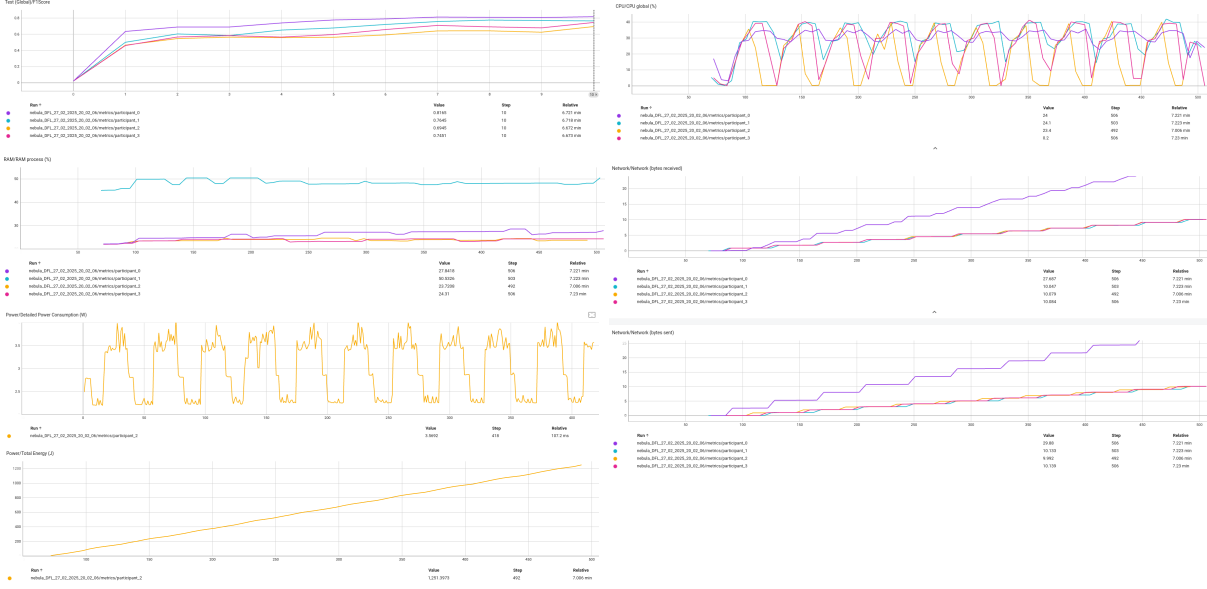


Figure 5.4: Results of the physical scenario with a star topology evaluated with the MNIST dataset.

Star Topology with Fashion MNIST Dataset

Figure 5.5 illustrates the performance of NEBULA in a physical deployment using a star topology with the Fashion-MNIST dataset. Similarly to the previous experiment, Participant 0, acting as the central node, achieved the highest F_1 score, reaching 83%. The models of Participants 1 and 2 performed similarly to those in the MNIST experiment. However, Participant 3 exhibited a lower F_1 -score of 64%, resulting in an overall average F_1 -score of 75%.

The computational load was slightly higher compared to the MNIST experiment, likely due to the increased data complexity of Fashion-MNIST:

- Average CPU usage: 29.1%
- Maximum CPU usage: 41.5%
- Average power consumption: 3.5 watts
- Maximum power consumption: 4.2 watts
- Total energy consumption: 1324 joules

As seen in previous experiments, CPU usage and power consumption followed a cyclical pattern, aligning with the training rounds of the federated learning process.

RAM usage remained largely consistent with the MNIST experiment:

- Participant 0: Average RAM usage of 26.2%, peaking at 27.8%.
- Participant 1: Average of 48.7%, with a maximum of 51.7%.
- Participants 2 and 3: Average RAM usage of 23.9%, peaking at 25%.

Network traffic remains nearly identical between the MNIST and Fashion-MNIST experiments.

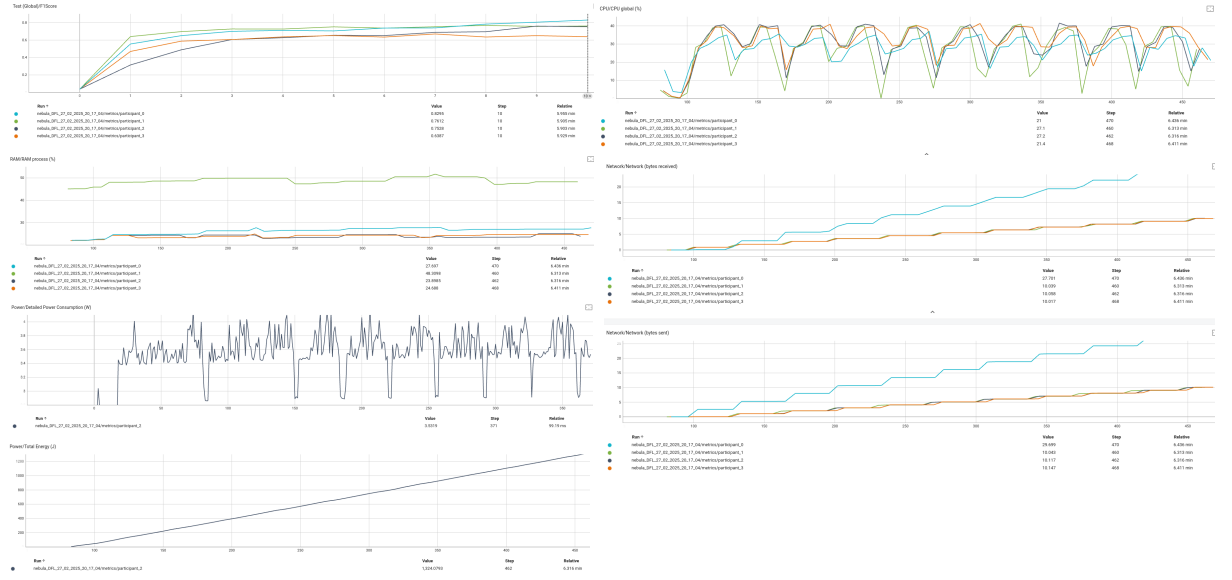


Figure 5.5: Results of the physical scenario with a star topology evaluated with the Fashion MNIST dataset.

Ring Topology with MNIST Dataset

Figure 5.6 presents the performance of NEBULA in a physical deployment using a ring topology with the MNIST dataset. Participant 2 achieved the fastest convergence, finishing with an F_1 -score of 82%, while the remaining participants performed slightly lower, with F_1 -scores ranging between 76% and 78%. The overall average F_1 -score across all participants was 78.5%.

Resource Utilization

- CPU Usage: The average CPU utilization across participants was 26%, with a maximum usage of 42.1%.

- **Power Consumption:** The average power draw was 3 watts, reaching a maximum of 4 watts.
- **Total Energy Consumption:** The cumulative energy usage reached 1249 joules.

RAM Usage

RAM usage across the participants was as follows:

- **Participant 0:** Average RAM usage of 25.3%, peaking at 27.1%.
- **Participant 1:** Average of 49.3%, with a maximum of 52%.
- **Participants 2 and 3:** Average RAM usage of 24.2%, peaking at 25.7%.

In terms of network performance, each device sent and received approximately 20 MB of data, requiring nearly twice the bandwidth compared to the star topology.

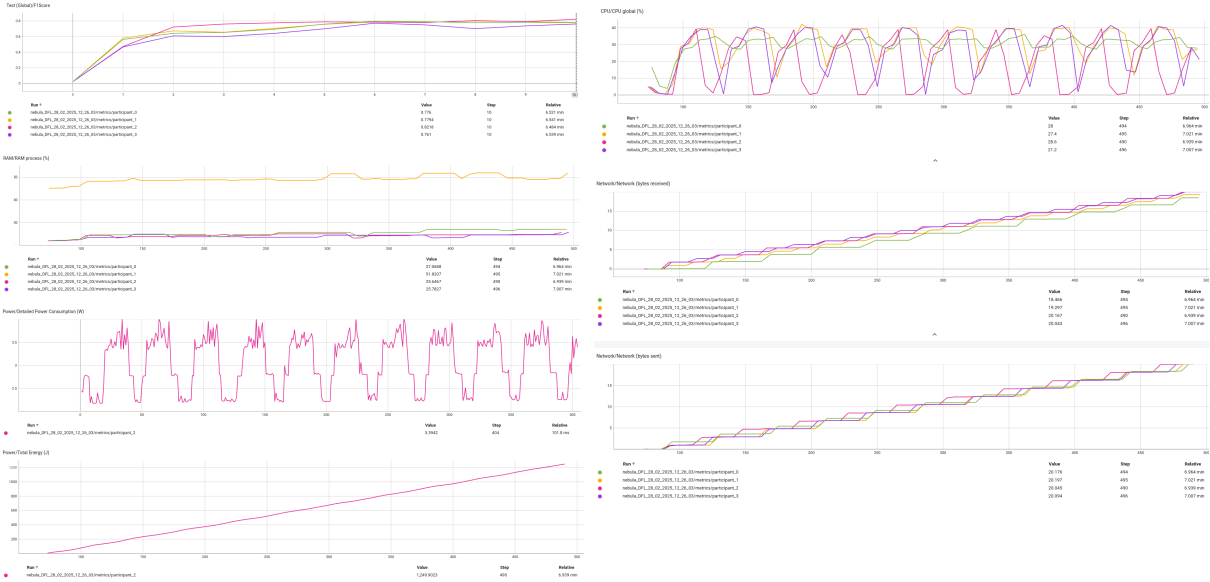


Figure 5.6: Results of the physical scenario with a ring topology evaluated with the MNIST dataset.

Ring Topology with Fashion MNIST Dataset

Figure 5.7 presents the performance of NEBULA in a physical deployment using a ring topology with the Fashion-MNIST dataset. Unlike the MNIST experiment, Participant 1 exhibited the best performance, achieving an F_1 -score of 82%. In contrast, Participants 2 and 3 underperformed, both reaching only an F_1 -score of 59%. As a result, the ring topology demonstrated lower overall performance when using Fashion-MNIST compared to MNIST, with an average F_1 -score of 68.8%

Resource Utilization

- **CPU Usage:** The average CPU utilization across participants was 28.3%, with a maximum usage of 41.3%.
- **Power Consumption:** The average power draw was 3.4 watts, reaching a maximum of 4.2 watts.
- **Total Energy Consumption:** The cumulative energy usage reached 1404 joules.

RAM Usage

RAM usage across the participants was as follows:

- **Participant 0:** Average RAM usage of 26.4%, peaking at 28.5%.
- **Participant 1:** Average of 48.8%, with a maximum of 51%.
- **Participants 2 and 3:** Average RAM usage of 24.2%, peaking at 25.7%.

Network bandwidth usage was comparable to the MNIST experiment (Figure 5.6), with each device sending and receiving similar amounts of data.

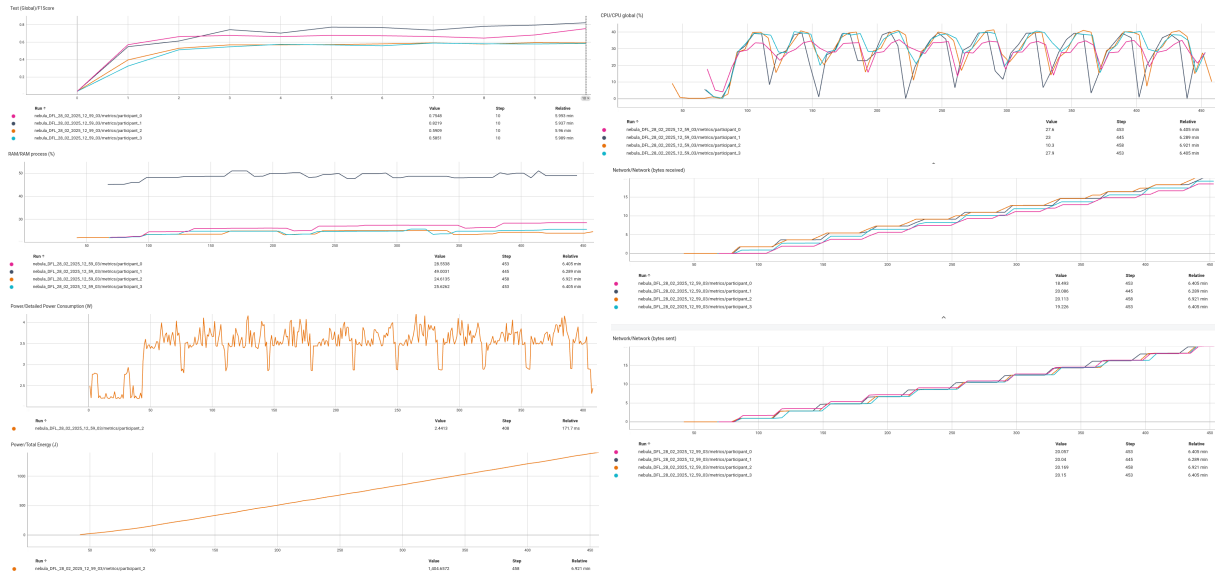


Figure 5.7: Results of the physical scenario with a ring topology evaluated with the Fashion MNIST dataset.

Random Topology with MNIST Dataset

Figure 5.8 presents the performance of NEBULA in a physical deployment using a random topology with the MNIST dataset (see the second topology from the right in Figure 2.2.2). Participants 1 and 2 achieved significantly higher F_1 -scores of 83% and 81%, respectively, while Participants 0 and 3 performed worse, with scores of 64% and 72%. This discrepancy may be attributed to Participants 0 and 3 functioning as edge nodes, receiving fewer model

updates and consequently experiencing slower or limited learning progression. As a result, the random topology exhibited uneven model performance across participants, leading to an overall average F_1 -score of 74.7%

Resource Utilization

- **CPU Usage:** The average CPU utilization across participants was 26.1%, with a maximum usage of 42.3%.
- **Power Consumption:** The average power draw was 3.1 watts, reaching a maximum of 4.2 watts.
- **Total Energy Consumption:** The cumulative energy usage reached 1305 joules.

RAM Usage

RAM usage across the participants was as follows:

- **Participant 0:** Average RAM usage of 25.1%, peaking at 26.1%.
- **Participant 1:** Average of 51%, with a maximum of 55.8%.
- **Participants 2 and 3:** Average RAM usage of 24.2%, peaking at 25.2%.

In terms of bandwidth consumption, Participants 2 and 3 exhibited nearly twice the network usage compared to Participants 0 and 1, with around 20MB of data sent and received. This is likely due to their central positioning within the topology, requiring them to communicate with two neighbouring nodes instead of one.

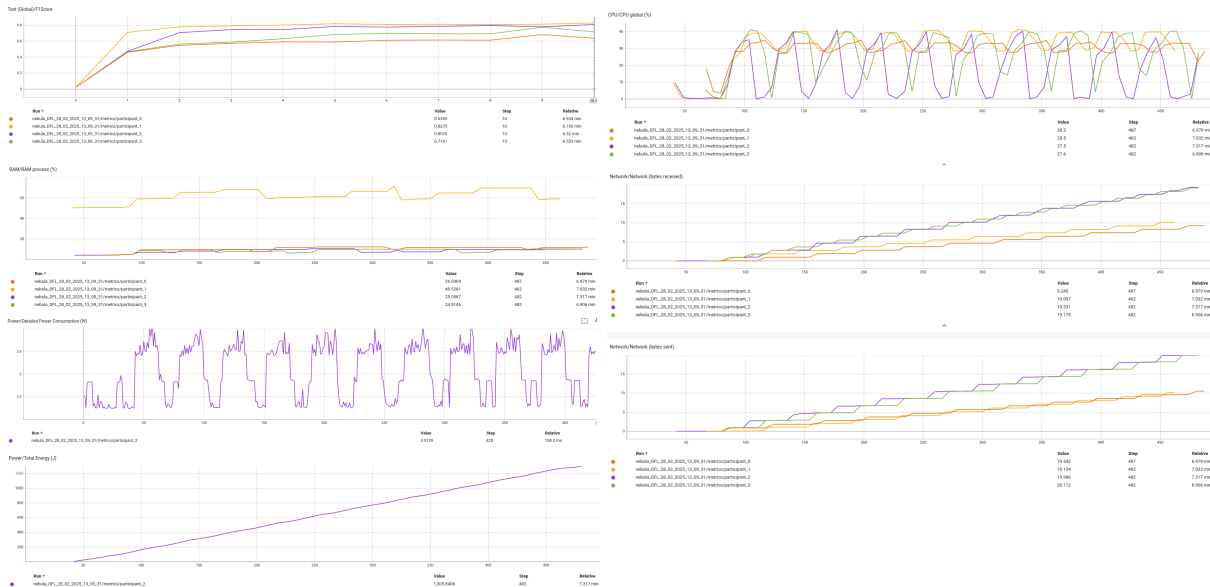


Figure 5.8: Results of the physical scenario with a random topology evaluated with the MNIST dataset.

Random Topology with Fashion MNIST Dataset

Figure 5.9 illustrates the performance of NEBULA in a physical deployment using a random topology with the Fashion-MNIST dataset (see the most right topology in Figure 2.2.2). Interestingly, Participant 0 achieved the highest F_1 -score of 78%, despite being positioned as an edge node. In contrast, Participants 1, 2, and 3 exhibited significantly lower performance, with respective F_1 -scores of 63%, 60%, and 60%. The overall average F_1 -score for this experiment was 64.9%.

Resource Utilization

- **CPU Usage:** The average CPU utilization across participants was 28.7%, with a maximum usage of 41.3%.
- **Power Consumption:** The average power draw was 3.5 watts, reaching a maximum of 4.3 watts.
- **Total Energy Consumption:** The cumulative energy usage reached 1368 joules.

RAM Usage RAM usage across the participants was as follows:

- Participant 0: Average RAM usage of 26%, peaking at 27.3%.
- Participant 1: Average of 49.1%, with a maximum of 52.6%.
- Participants 2 and 3: Average RAM usage of 23.9%, peaking at 26.3%.

Network communication followed expected patterns, with nodes exhibiting higher connectivity consuming more bandwidth. Participant 1, acting as the central node, sent and received around 30MB of data, while Participants 2 and 3 exchanged approximately 20MB, and Participant 0 handled only about 10MB.

Interestingly, this random topology resembles a star topology, yet the central node did not achieve the best model performance. Additionally, overall model performance was significantly worse than in the ring topology.

Virtualized Scenarios

To compare the results of the physical scenario, two additional experiments were conducted in a virtualized environment using Docker containers as nodes, as shown in Figures 5.10 and 5.11. The findings indicate that the F_1 -scores were comparable between the two environments when using a fully connected topology.

However, a key difference was observed in the total training time required to complete 10 training rounds. In the physical scenario, the training duration ranged between six and seven minutes, whereas in the virtualized scenario, the same process was completed in approximately 2 to 2.5 minutes. This significant reduction in execution time is primarily

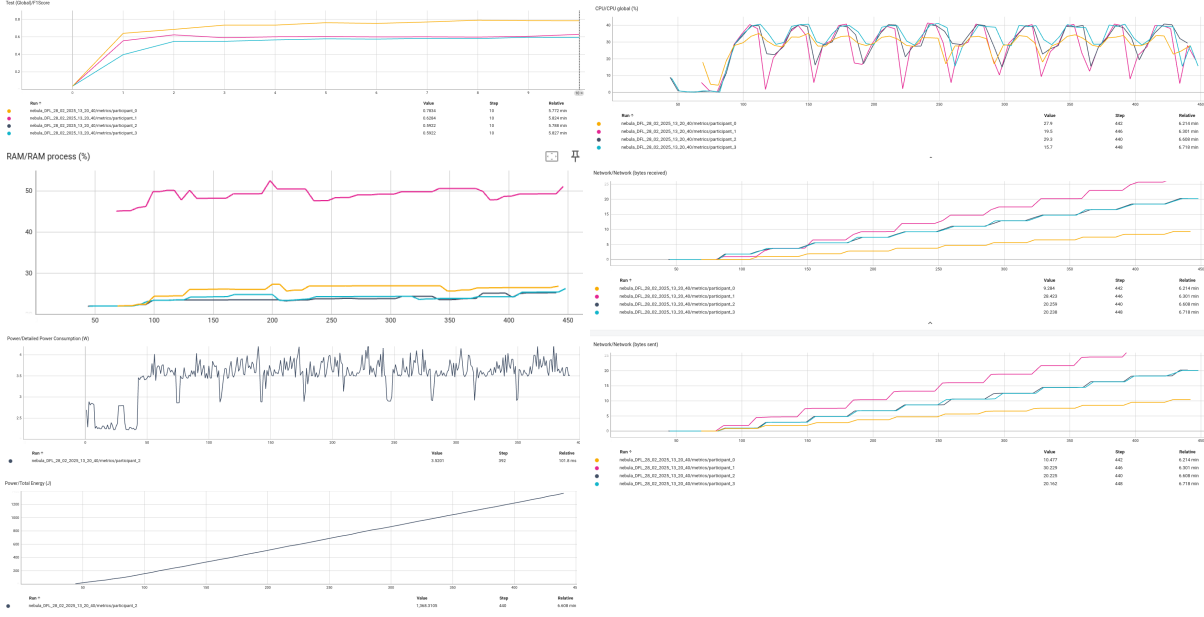


Figure 5.9: Results of the physical scenario with a random topology evaluated with the Fashion MNIST dataset.

attributed to the vast difference in computational resources between the distributed edge devices and the modern machine used for the containerized setup.

The average CPU usage during the MNIST experiment was 26.8%, with a peak usage of 100%. In contrast, the Fashion-MNIST experiment exhibited a higher average CPU usage of 34.4%, also reaching 100% at its peak. In both cases, RAM usage remained relatively consistent, averaging 7.4%, with a maximum utilization of approximately 8.7%. Notably, the CPU usage in the virtualized environment did not exhibit a distinct cyclical pattern of ten peaks and troughs, which was observed in the physical setup. This may be due to the training process being more stable and less impacted by resource constraints in a modern machine.

In contrast, network traffic remained almost identical between both scenarios. This is expected since the number of messages exchanged between nodes in a fully-connected topology remains constant

5.4.1 Summary

Table 5.2: Summary of Experimental Results for Physical and Virtualized Scenarios

Scenario	Avg. F ₁ -Score	CPU Usage (%)	Avg. RAM Usage (%)	Network Traffic (MB)	Power (W)	Energy (J)
Fully Connected MNIST (Physical)	82.0	25.7 (max: 42.8)	33.3	30	3.3 (max: 3.9)	1404
Fully Connected Fashion MNIST (Physical)	81.0	28.7 (max: 41.9)	33.3	30	3.5 (max: 4.5)	1350
Star MNIST (Physical)	76.0	25.8 (max: 41.9)	32.76	P0: 30, Others: 10	3.0 (max: 4.1)	1251
Star Fashion MNIST (Physical)	75.0	29.1 (max: 41.5)	33.9	P0: 30, Others: 10	3.5 (max: 4.2)	1324
Ring MNIST (Physical)	78.5	26.0 (max: 42.1)	32.9	20	3.05 (max: 4.06)	1249
Ring Fashion MNIST (Physical)	68.8	28.3 (max: 41.3)	33.1	20	3.45 (max: 4.21)	1404
Random MNIST (Physical)	74.7	26.1 (max: 42.3)	33.4	P2,3: 20, P0,1: 10	3.1 (max: 4.2)	1305
Random Fashion MNIST (Physical)	64.9	28.7 (max: 41.3)	33	P1: 30, P2,3: 20, P0: 10	3.5 (max: 4.3)	1368
Fully Connected MNIST (Virtual)	82.0	26.8 (max: 100)	7.4 (max: 8.7)	30	N/A	N/A
Fully Connected Fashion MNIST (Virtual)	81.0	34.4 (max: 100)	7.4 (max: 8.7)	30	N/A	N/A

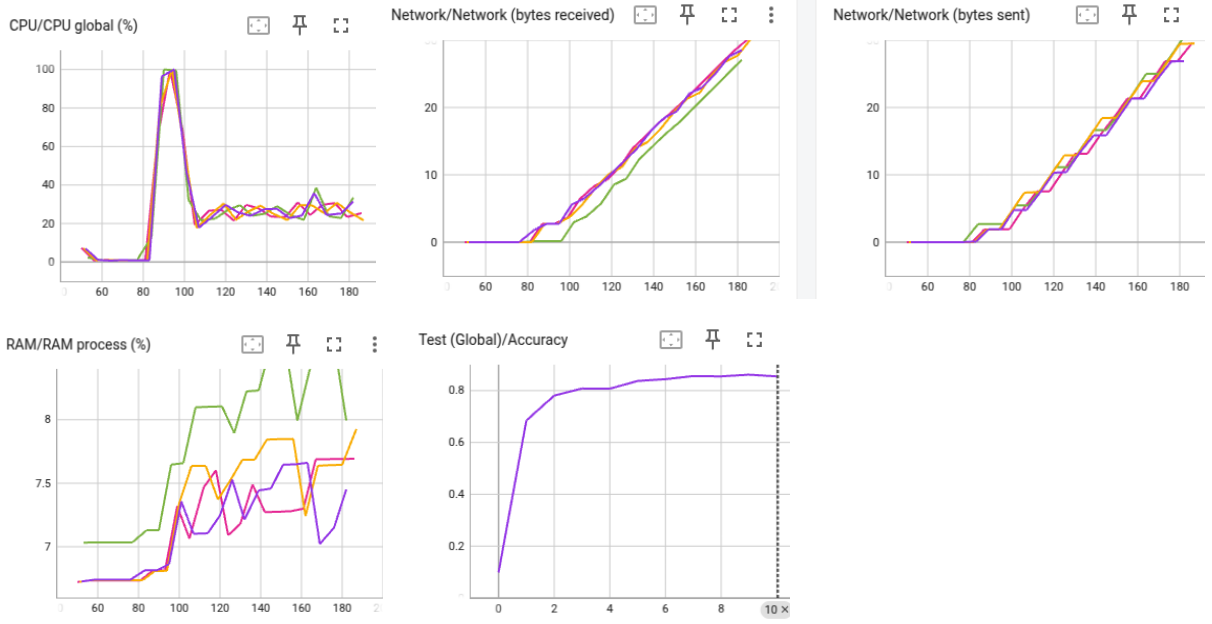


Figure 5.10: Results of the virtualized scenario with a fully connected topology evaluated with the MNIST dataset.

Table 5.2 summarizes the results of the evaluation. The results demonstrate that the choice of network topology significantly influences both learning speed and model performance. The fully-connected topology consistently achieved the highest F_1 -scores on both MNIST and Fashion-MNIST. In contrast, sparser topologies such as the star or ring topology exhibited greater variability in F_1 -scores among participants, likely due to fewer parameter exchanges between neighbours.

In the star topology, the central node consistently outperformed other participants, achieving the highest F_1 -scores in both the MNIST and Fashion-MNIST experiments. Overall the star topology reached nearly the same F_1 -score as fully-connected. To converge, it may require slightly more rounds due to the indirect communication of updates through the central node. The ring topology demonstrated stable convergence, with F_1 -scores ranging between 76% and 82% on the MNIST dataset. However, for Fashion-MNIST, it recorded the lowest F_1 -scores for some nodes across all experiments. The random topology showed high variability in performance, with some edge nodes struggling to reach high accuracy within 10 training rounds. Interestingly, one edge node achieved the highest F_1 -score (78%) on the Fashion-MNIST dataset. A possible explanation for some of these rather unexpected results may be the non-IID nature of the data, which can have a significant effect on the model’s accuracy [29].

Resource utilization remained largely consistent across experiments. CPU usage followed the FL training cycle, exhibiting 10 distinct peaks and troughs. Across all experiments, average CPU usage ranged between 25% and 29%. Notably, the star topology with the MNIST dataset had the highest average CPU usage, while the fully connected topology with MNIST had the lowest.

Power consumption closely mirrored CPU activity, highlighting its cyclical nature in the FL scenario. Average power usage across all setups ranged from 3 to 3.5 watts, with

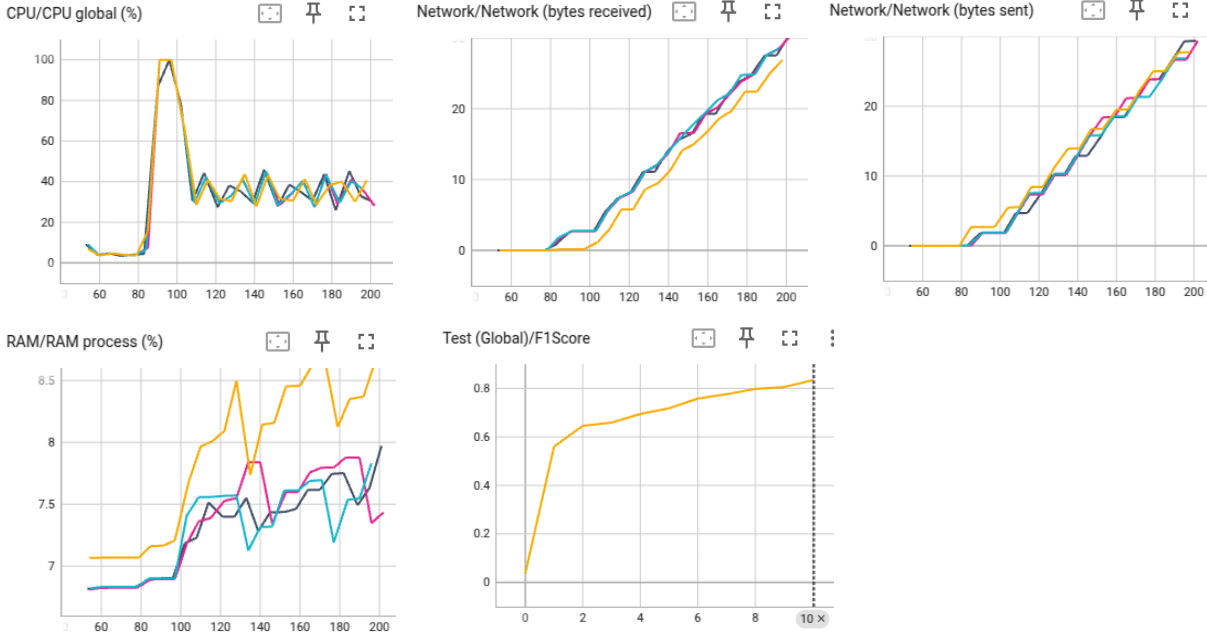


Figure 5.11: Results of the virtualized scenario with a fully connected topology evaluated with the Fashion MNIST dataset.

the fully connected topology on Fashion-MNIST consuming the most power and the star topology on MNIST consuming the least. Total energy consumption varied between 1251 joules and 1404 joules, depending on the topology and dataset. The highest overall energy consumption was observed in the fully-connected topology with MNIST and the ring topology with Fashion-MNIST, while the lowest was recorded for the ring topology with MNIST. The average power consumption and CPU usage were generally slightly higher with the Fashion-MNIST dataset than with the MNIST dataset.

RAM usage remained consistent across topologies, showing minimal variation. The NVIDIA Jetson Nano consistently exhibited slightly higher average RAM usage compared to the two Raspberry Pis with 4 GB RAM, due to its role as the starting node in the FL process.

Network traffic closely followed the connectivity of each topology. The fully connected topology required the highest communication bandwidth, as every node exchanged model updates with every other node in each round. The star topology generated less total traffic than the fully connected topology, but since all communication passed through the central node, it handled the same amount of traffic as an individual node in a fully connected setup. The ring and random topologies required significantly less communication. In the ring topology, each node only communicated with its two neighbours, minimizing bandwidth usage. In the random topology, traffic varied depending on the number of connections each node had.

The primary difference between the physical and virtual scenarios was the training time. While both scenarios achieved the same F₁-score, the physical setup took significantly longer. This was expected, as modern laptops (see section 5.1) have far superior computational resources compared to Raspberry Pi devices or the NVIDIA Jetson Nano.

Interestingly, average CPU usage was higher in the physical scenario when tested with

the Fashion-MNIST dataset. This is likely because running all participants on a single machine increased its overall CPU load. Finally, RAM usage in the physical scenario was significantly lower, due to the machine's larger memory capacity.

Chapter 6

Conclusion

This work presents the successful deployment of NEBULA, a platform enabling decentralized training of FL models, onto resource-constrained devices such as Raspberry Pis and Nvidia Jetson Nano. This deployment demonstrates the feasibility of executing a full-fledged FL platform on edge devices and establishes a realistic testbed for DFL systems. To enable deployment and real-time monitoring in a physical environment, REST-based communication was implemented to transmit configuration data and performance metrics between the controller and participant nodes. Furthermore, to enhance the real-time capabilities and investigate the feasibility of the presented endeavour, a mechanism to track and record each device’s energy and power consumption throughout the platform’s operation was implemented using JT-TC66C, a USB-C multimeter. The implementation was evaluated across two federated scenarios: 1) Physical Deployment, a federation comprising four Raspberry Pis (three as participant nodes and one as a controller) and one Nvidia Jetson Nano, and 2) Virtualized Deployment, a federation consisting of four Docker containers running on a modern laptop.

The results indicate that in a fully connected topology, the final F_1 -score was identical to that of the virtualized deployment, despite the training process taking 2–3 times longer. This demonstrates that NEBULA can effectively train accurate models on resource-constrained devices. More constrained topologies, like ring and star, showed slightly slower improvements and more variability in accuracy across different nodes.

Resource utilization in the physical experiments stayed within acceptable bounds and showed that NEBULA can run on resource-constrained devices without exceeding their CPU, memory, or power capacities. Although FL training did stress CPU usage, it did not cause excessive strain, with CPU utilization peaking at $\sim 40\text{--}43\%$ and averaging $\sim 25\text{--}29\%$ across devices. The power consumption remained low throughout the FL operations, with roughly 3–4 watts per device on average, and the device consumed around 1.2–1.4 kilojoule of energy. The network overhead did not bottleneck the training process but may be a factor to consider in larger federations or longer training sessions.

As future work, there remains a multitude of avenues for exploration. Expanding the implementation to a significantly larger number of devices would allow for a more comprehensive stress test of resource utilization and provide deeper insights into the network

overhead associated with larger federations. Furthermore, more work may be needed to optimize the training process on edge devices, accelerating the training times. Additionally, while this work performed experiments with fully connected, star, ring, and random topologies, the number of available devices severely restricted the scenarios possible to test. Future research could explore alternative network topologies, such as cluster-based topologies, where designated proxy nodes enable communication between clusters. And finally, a natural extension of this study would involve deploying NEBULA in real-world applications, such as on smartphones or IoT devices, to evaluate its practical feasibility and performance under realistic constraints.

Bibliography

- [1] A. Imteaj, U. Thakker, S. Wang, J. Li, and M. H. Amini, “Federated learning for resource-constrained iot devices: Panoramas and state-of-the-art”, *CoRR*, vol. abs/2002.10610, 2020. arXiv: 2002.10610. [Online]. Available: <https://arxiv.org/abs/2002.10610>.
- [2] E. T. Martínez Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, “Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges”, *IEEE Communications Surveys & Tutorials*, vol. 25, no. 4, pp. 2983–3013, 2023, ISSN: 2373-745X. DOI: 10.1109/comst.2023.3315746. [Online]. Available: <http://dx.doi.org/10.1109/COMST.2023.3315746>.
- [3] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, “Federated optimization: Distributed machine learning for on-device intelligence”, *arXiv preprint arXiv:1610.02527*, 2016.
- [4] E. T. Martínez Beltrán, M. Quiles Pérez, P. M. Sánchez Sánchez, *et al.*, “Decentralized Federated Learning: Fundamentals, State of the Art, Frameworks, Trends, and Challenges”, *IEEE Communications Surveys & Tutorials*, vol. 25, no. 4, pp. 2983–3013, 2023. DOI: 10.1109/COMST.2023.3315746.
- [5] E. T. M. Beltrán, Á. L. P. Gómez, C. Feng, *et al.*, “Fedstellar: A platform for decentralized federated learning”, *Expert Systems with Applications*, vol. 242, p. 122861, 2024.
- [6] A. Huertas Celdrán, P. M. Sánchez Sánchez, C. Feng, G. Bovet, G. M. Pérez, and B. Stiller, “Privacy-preserving and syscall-based intrusion detection system for iot spectrum sensors affected by data falsification attacks”, *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8408–8415, 2023. DOI: 10.1109/JIOT.2022.3213889.
- [7] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, *Communication-efficient learning of deep networks from decentralized data*, 2023. arXiv: 1602.05629 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1602.05629>.
- [8] 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.
- [9] 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.

- [10] Y. Liu, Y. Kang, C. Xing, T. Chen, and Q. Yang, “A secure federated transfer learning framework”, *IEEE Intelligent Systems*, vol. 35, no. 4, pp. 70–82, Jul. 2020, ISSN: 1941-1294. DOI: 10.1109/mis.2020.2988525. [Online]. Available: <http://dx.doi.org/10.1109/MIS.2020.2988525>.
- [11] Q. Jing, W. Wang, J. Zhang, H. Tian, and K. Chen, *Quantifying the performance of federated transfer learning*, 2019. arXiv: 1912.12795 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/1912.12795>.
- [12] S. Feng and H. Yu, *Multi-participant multi-class vertical federated learning*, 2020. arXiv: 2001.11154 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2001.11154>.
- [13] T.-D. Cao, T. Truong-Huu, H. Tran, and K. Tran, *A federated deep learning framework for privacy preservation and communication efficiency*, 2022. DOI: <https://doi.org/10.1016/j.sysarc.2022.102413>. arXiv: 2001.09782 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/2001.09782>.
- [14] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions”, *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, May 2020, ISSN: 1558-0792. DOI: 10.1109/msp.2020.2975749. [Online]. Available: <http://dx.doi.org/10.1109/MSP.2020.2975749>.
- [15] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, *Data poisoning attacks against federated learning systems*, 2020. arXiv: 2007.08432 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2007.08432>.
- [16] J. Wen, Z. Zhang, Y. Lan, Z. Cui, J. Cai, and W. Zhang, “A survey on federated learning: Challenges and applications”, *International Journal of Machine Learning and Cybernetics*, vol. 14, pp. 1–23, Nov. 2022. DOI: 10.1007/s13042-022-01647-y.
- [17] L. Ridolfi, D. Naseh, S. S. Shinde, and D. Tarchi, “Implementation and evaluation of a federated learning framework on raspberry pi platforms for iot 6g applications”, *Future Internet*, vol. 15, no. 11, 2023, ISSN: 1999-5903. DOI: 10.3390/fi15110358. [Online]. Available: <https://www.mdpi.com/1999-5903/15/11/358>.
- [18] A. Karras, C. Karras, K. C. Giotopoulos, D. Tsolis, K. Oikonomou, and S. Sioutas, “Federated edge intelligence and edge caching mechanisms”, *Information*, vol. 14, no. 7, 2023, ISSN: 2078-2489. DOI: 10.3390/info14070414. [Online]. Available: <https://www.mdpi.com/2078-2489/14/7/414>.
- [19] D. Biskup, “Flexible and lightweight toolbox for federated learning on edge devices”, Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2022.
- [20] A. Karras, C. Karras, K. Giotopoulos, D. Tsolis, K. Oikonomou, and S. Sioutas, “Peer to peer federated learning: Towards decentralized machine learning on edge devices”, Sep. 2022, pp. 1–9. DOI: 10.1109/SEEDA-CECNSM57760.2022.9932980.
- [21] R. P. Foundation, *Raspberry pi 4 product brief*, Accessed: 2025-02-25, 2019. [Online]. Available: <https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf>.
- [22] N. Corporation, *Jetson nano developer kit: Technical specifications*, Accessed: 2025-02-25, 2025. [Online]. Available: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/>.

- [23] Joy-IT, *Jt-tc66c usb-c multimeter specifications*, Accessed: 2025-02-25, 2025. [Online]. Available: <https://joy-it.net/de/products/JT-TC66C>.
- [24] MatsLindh, *What is the purpose of uunicorn?*, Accessed: 2025-02-26, 2022. [Online]. Available: <https://stackoverflow.com/questions/71435960/what-is-the-purpose-of-uunicorn>.
- [25] TheHWCave, *Tc66 - python utility for ruideng tc66 usb tester*, Accessed: 2025-02-23, 2022. [Online]. Available: <https://github.com/TheHWCave/TC66>.
- [26] L. Deng, “The mnist database of handwritten digit images for machine learning research [best of the web]”, *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.
- [27] H. Xiao, K. Rasul, and R. Vollgraf. “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms”. arXiv: [cs.LG/1708.07747](https://arxiv.org/abs/1708.07747) [cs.LG]. (Aug. 28, 2017).
- [28] A. A. Heidari, H. Faris, S. Mirjalili, I. Aljarah, and M. Mafarja, “Ant lion optimizer: Theory, literature review, and application in multi-layer perceptron neural networks”, *Nature-inspired optimizers: theories, literature reviews and applications*, pp. 23–46, 2020.
- [29] 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.

Abbreviations

FL	Federated Learning
CFL	Centralized Federated Learning
DFL	Decentralized Federated Learning
ML	Machine Learning
IID	Independent and Identically Distributed
FedAvg	Federated Averaging
SDG	stochastic gradient descent
HFL	Horizontal Federated Learning
VFL	Vertical Federated Learning
FTL	Federated Transfer Learning
MMVFL	Multi-class Vertical Federated Learning
DP	Differential Privacy
SMC	Secure Multiparty Computation
TEE	Trusted Execution Environment
HE	Homomorphic Encryption
DL	Deep Learning
V	voltage
A	ampere
W	watt
MLP	Multi-Layer Perceptron
MB	Megabyte

List of Figures

2.1	Architecture for a horizontal federated learning system [8]	7
2.2	Architecture for a vertical federated learning system [8]	8
2.3	Overall architecture of NEBULA [5]	15
2.4	Communication process in NEBULA [5]	16
2.5	Federated participant cycle in NEBULA [5]	17
2.6	Sequence diagram showing the interaction of NEBULAs (previously Fed-stellar) components [5]	18
3.1	Overall architecture of NEBULA on Raspberry Pis	22
3.2	Sequence diagram showing the simplified interaction between NEBULAs frontend, controller, and nodes.	23
5.1	Network topologies evaluated in NEBULA. From left to right: Fully connected, ring, star, random (evaluated with MNIST), and random (evaluated with Fashion-MNIST).	39
5.2	Results of the physical scenario with a fully connected topology evaluated with the MNIST dataset.	40
5.3	Results of the physical scenario with a fully connected topology evaluated with the Fashion-MNIST dataset.	41
5.4	Results of the physical scenario with a star topology evaluated with the MNIST dataset.	42
5.5	Results of the physical scenario with a star topology evaluated with the Fashion MNIST dataset.	43
5.6	Results of the physical scenario with a ring topology evaluated with the MNIST dataset.	44
5.7	Results of the physical scenario with a ring topology evaluated with the Fashion MNIST dataset.	45

5.8	Results of the physical scenario with a random topology evaluated with the MNIST dataset.	46
5.9	Results of the physical scenario with a random topology evaluated with the Fashion MNIST dataset.	48
5.10	Results of the virtualized scenario with a fully connected topology evaluated with the MNIST dataset.	49
5.11	Results of the virtualized scenario with a fully connected topology evaluated with the Fashion MNIST dataset.	50

List of Tables

5.1	Comparison of validation scenarios in NEBULA using physical and virtualized environments.	37
5.2	Summary of Experimental Results for Physical and Virtualized Scenarios .	48

Appendix A

Deployment

Currently, to run NEBULA on physical devices, you have to first select the 'physical' button, then create a new scenario, and only then adjust the Federations' configurations (e.g., IP address).

The Nvidia Jetson Nano Developer Kit uses a microSD card as a boot device and for main storage. To prepare the microSD card for use, the Jetson Nano Developer Kit SD Card Image was downloaded and written to it according to the official instructions for Linux.

Similarly, the Raspberry Pi OS (64-bit) was installed through the official Raspberry PI Imager to prepare the microSD card for the Raspberry Pis.

A.1 Deploying on the Controller Node

The setup process for deploying NEBULA on a controller device is the same as for installing it on your own machine. One can follow the official guide on the GitHub repository.

A.1.1 Detailed Code Changes

To run NEBULA on physical devices, the radio button 'Physical devices' had to be made clickable:

```
<input class="form-check-input" type="radio" name="
  deploymentRadioOptions" id="physical-devices-radio"
  value="physical">
<label class="form-check-label" for="physical-devices-radio">Physical
  devices</label>
```

For the button to not jump back to 'process' or 'docker,' the else statement in the `setScenario` function had to be added:

```
function setScenarioData(data) {
  // old implementation
} else {
  //Adjust so that physical button stays checked
  document.getElementById("physical-devices-radio").checked = true
  ;
}
```

For the adjusted IP address to correctly be saved the line `scenarioStorage.replaceScenario()` had to be added in the function `getNumberOfNodesAndUpdate`:

```
//SAVE CHANGES BUTTON FOR PARTICIPANTS DETAILS MODAL
document.getElementById("participant-modal-save").addEventListener("
  click", function () {
    const participant = Graph.graphData().nodes[i];
    participant.ip = document.getElementById("participant-" + i + "-ip")
      .value;
    participant.port = document.getElementById("participant-" + i + "-
      port").value;
    Graph.graphData(Graph.graphData());
    //UPDATE SCENARIO WITH UPDATED NODE IP AND PORT IF NECESSARY
    scenarioStorage.replaceScenario();
  });
```

Changes in `scenario.py`:

For the controller to be able to communicate its IP address to the nodes the IP address needs to be manually changed in the code:

```
# remaining implementation
elif self.scenario.deployment == "process":
    self.controller = f"127.0.0.1:{os.environ.get('
      NEBULA_FRONTEND_PORT')}"
else:
    self.controller = "192.168.204.177:6060"
```

In `init` the following two configurations had to be reloaded so that they are saved correctly:

```
#ADJUST CONFIG IN PARTICIPANTS CONFIG FILE SO THAT THE CONFIGURATION
  MATCHES (until now the default
  #has been set
participant_config["scenario_args"]["deployment"] = self.scenario.
  deployment
participant_config["scenario_args"]["controller"] = self.controller
```

In `load_configurations_and_start_nodes`, the method to 'start' the physical nodes had to be added:

```
if self.scenario.deployment in ["docker", "process"]:
    # ...
else:
    self.start_physical_nodes()
```

The `start_physical_nodes()` function does some necessary configurations and sends the configuration files to the participant nodes.

For the controller to be able to receive the Tensorboard metrics, the following endpoint was added to the `app.py` file:

```
@app.post("/nebula/controller/upload-event-files")
async def upload_event_files(files: list[UploadFile]):
    #location needs to be /nebula/app/logs/'name_of_scenario'/metrics/
    participant_x'
    for file in files:
        path = file.filename
        filename = path.split("/")[-1]
        abs_location = f"''.join(path.split('/')[:-1])/"
        rel_location = f"../../../../{abs_location}"
        os.makedirs(rel_location, exist_ok=True)
        with open(f"{rel_location}{filename}", "wb") as f:
            content = await file.read()
            f.write(content)
    return Response(content="Success", status_code=200)
```

A.2 Deploying on Participant Nodes

To deploy NEBULA on the Raspberry Pis, clone the NEBULA_NODES repository and `cd` into the NEBULA_NODES repo. There, create a python virtual environment with `python3 -m venv venv` activate it with `source venv/bin/activate`.

Install FastAPI and uvicorn with: `pip install fastapi` and `pip install uvicorn[standard]`, then run `pip install -upgrade -force-reinstall -r nebula/requirements.txt`. The `nebula/requirements.txt` file has `torchdata==0.8.0` and `docker=7.1.0` commented out, since docker is not needed and the particular torchdata version is not compatible with the python version. Also, run `pip install pycryptodome` and `pip install pyserial`.

On the Raspberry Pis the process can now be started with the command `python3 physical/start.py`.

To deploy NEBULA on the Nvidia Jetson Nano a few extra steps need to be taken and python3.11 needs to be compiled from source. On the JetsonHacksNano GitHub account exists a repository `build_python`, which contains a script that can be used to build a python version available on the JetsonHacksNano account. For this work, the python3.11 repository is needed, specifically the `ubuntu/bionic` branch, which contains the source code for python 3.11. Now follow the following steps:

1. Clone the `build_python` repository: `git clone https://github.com/JetsonHacksNano/build_python`
2. `cd build_python`

3. `bash ./build_python3.sh` This will build the default version 3.11, which is what this work needs. This will take around two hours to build.
4. To make a local apt repo: `bash ./make_apt_repository.sh`
5. Now run: `sudo apt install python3.11-full`
6. Run `python3.11 -m ensurepip -upgrade`

Now clone the NEBULA_NODES repository and `cd` into the NEBULA_NODES repo. There, create a python virtual environment with `python3.11 -m venv venv` activate it with `source venv/bin/activate`.

Install FastAPI and uvicorn with: `pip install fastapi` and `pip install uvicorn[standard]`, then run `pip install -upgrade -force-reinstall -r nebula/requirements.txt`. Also, run `pip install pycryptodome` and `pip install pyserial`.

The process can be started with the command `python3.11 physical/start.py`.

A.2.1 Detailed Code Changes

To start the process on the participant nodes, the `physical` directory was added, which contains the `start.py` and `globals.py` files.

In `engine.py`, the `_send_tensorboard_files` function was added to periodically share the Tensorboard metrics with the controller. Additionally, the following code was added to the `_learning_cycle` function to batch report the detailed power consumption:

```
if self.config.participant["scenario_args"]["deployment"] == "physical":
    self._reporter.log_detailed_power_consumption()
    await self._send_tensorboard_files(last_file)
```

In the `addons` directory, the `TC66C.py` file was added to measure power and energy consumption throughout NEBULAs operations.

In `engine.py`, the `deploy_federation` function was adjusted to wait for the successful establishment of the connection to the start node before sending the `FEDERATION_READY` message to the neighbours:

```
else:
    # we can only send 'FEDERATION_READY' when the connection to start
    # node is established otherwise we will be stuck
    ip_start_node = f"{self.config.participant['network_args']['ip_start']}:8000"
    while not ip_start_node in self.cm.connections:
        print(self.cm.connections)
        await asyncio.sleep(1)
    logging.info(f"Sending FEDERATION_READY to neighbors...")
    message = self.cm.mm.generate_federation_message(nebula_pb2.
        FederationMessage.Action.FEDERATION_READY)
    await self.cm.send_message_to_neighbors(message)
```

```
logging.info(f"␣Waiting␣until␣receiving␣the␣start␣signal␣from␣the␣  
start␣node")
```

In `communications.py` the following code was adjusted in the `handle_incoming_message` function, to forward the `FEDERATION_READY` message to its neighbours:

```
elif message_wrapper.HasField("federation_message"):
    is_federation_ready_msg = message_wrapper.federation_message.action
    == nebula_pb2.FederationMessage.Action.Value("FEDERATION_READY")
    if await self.include_received_message_hash(hashlib.md5(data).
    hexdigest()):
        if self.config.participant["device_args"]["proxy"] or
            message_wrapper.federation_message.action == nebula_pb2.
            FederationMessage.Action.Value("FEDERATION_START") or
            is_federation_ready_msg:
            await self.forwarder.forward(data, addr_from=addr_from)
        await self.handle_federation_message(source, message_wrapper.
        federation_message)
```