UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

FELIPE MACHADO SCHWANCK

A Framework for testing Federated Learning algorithms using an edge-like environment

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Juliano Wickboldt
Coadvisor: Prof. Dr. Joel Luís Carbonera

Porto Alegre
April 2023

“Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.”

— THOMAS EDISON

ACKNOWLEDGEMENTS

First, to the tax-paying Brazilian people who support the public education system in the country, which today allows me to enjoy high-quality education. I emphasize how necessary it is that we, as part of the Brazilian nation, make sure that public money is correctly applied to the development of education in the country. Only through education can we have a more just, developed and humane nation.

To UFRGS, essential in my professional training process, for their dedication and for everything I learned over the years of the course.

To my supervisor, Juliano Wickboldt, and co-supervisor, Joel Luis Carbonera, for participating in this turbulent moment of my life in a completely welcoming way. I had no idea how I was going to carry out this work at the beginning. Your mentorship was always proactive, with important feedback, interesting ideas and constructive debates that guided me in the conception of this work. I am grateful to have had you as mentors.

To all my classmates who shared the adventure of the undergraduate course with me.

To my grandparents, Nilza and Alvino, for always rooting for me and spoiling me. I love you with all my heart.

To my dear brother, Vinícius, for always taking care of me and wanting my best. I love you very much and know that you can always count on me, even if the distance gets in the way. It will be an honor to receive the dreamed "straw" from you, from engineer to engineer.

To my parents, Iara and Osmar, for always providing me with all the best. If today I am able to achieve this goal, it is largely due to the support I received from you.

To my parents-in-law, Alissa and Marcos, for all the affection and support they have given me over the last few months. I will never forget what you did for me.

In particular, to my wife, Lívia, for having lived with me every part of the conception of this work. Thank you for all the care, love, patience and understanding. I know it wasn't easy to put up with being stressed and extremely busy right now. May we have a life full of dreams in Portugal and may we realize them together, side by side. I love you.

AGRADECIMENTOS

Primeiramente, ao povo brasileiro contribuinte de impostos que sustenta o sistema de educação público no país que me permite hoje usufruir de alta qualidade de ensino. Ressalto o quão necessário é que nós, como parte da nação brasileira, certifiquemo-nos de que o dinheiro público seja corretamente aplicado ao desenvolvimento da educação no país. Somente através da educação podemos ter um país mais justo, desenvolvido e humano.

À UFRGS, essencial no meu processo de formação profissional, pela dedicação, e por tudo o que aprendi ao longo dos anos do curso.

Para meu orientador, Juliano Wickboldt, e co-orientador, Joel Luis Carbonera, por participarem deste momento turbolento da minha vida de maneira completamente acolhedora. Eu não tinha a menor ideia de como realizaria este trabalho no seu começo. A mentoria de vocês sempre foi pró-ativa, com feedbacks importantes, ideias interessantes e debates construtivos que guiaram-me na concepção deste trabalho. Sou grato de poder ter tido vocês como mentores.

À todos os meus colegas de curso que conviveram comigo a aventura do curso de graduação.

À meus avós, Nilza e Alvino, por sempre torcerem por mim e me mimar. Amo vocês com todo meu coração.

À meu querido irmão, Vinícius, por sempre cuidar de mim e querer meu bem. Eu te amo muito e saiba que sempre poderás contar comigo, mesmo que a distância atrapalhe. Será uma honra receber o tão sonhado "canudo" de ti, de engenheiro para engenheiro.

À meus pais, Iara e Osmar, por sempre me proporcionarem com tudo de bom e do melhor. Se hoje estou conseguindo alcançar este objetivo, muito é devido ao suporte que recebi de vocês.

Aos meus sogros, Alissa e Marcos, por todo apoio, acolhimento, carinho e suporte que me deram nos últimos meses. Nunca esquecerei o que fizeram por mim.

Em especial, à minha esposa, Lívia, por ter vivido comigo cada parte da concepção deste trabalho. Obrigado por todo o carinho, amor, paciência e compreensão. Sei que não foi fácil me aturar estressado e extremamente atarefado neste momento. Que tenhamos uma vida repleta de sonhos em Portugal e que possamos realiza-los juntos, lado à lado. Eu te amo.

ABSTRACT

Federated Learning [FL] is a machine learning paradigm where many clients cooperatively train a single centralized model while keeping their data private and decentralized. FL is commonly used in edge computing, which involves placing computer workloads (both hardware and software) as close as possible to the edge, where the data is being created and where actions are occurring, enabling faster response times, greater data privacy, and reduced data transfer costs. However, due to the heterogeneous data distributions/contents of clients, it is non-trivial to accurately evaluate the contributions of local models in global centralized model aggregation. This has been a major challenge in FL, commonly known as data imbalance or class imbalance. Previous work has been proposed to address this issue such as Deep Reinforcement Learning (DRL) algorithms to dynamically learn the weight of the contributions of each client at each round. Testing and assessing this FL algorithms can be a very difficult and complex task due to the distributed nature of the systems. In this work, a literature review of these concepts is presented in order to introduce the reader enough context of this challenge and a distributed edge-like environment framework is proposed to assess FL algorithms in a more easy and scalable way.

Keywords: Federated Learning. Edge Computing. Kubernetes. Microservices. Framework.

Um framework para testar algoritmos de aprendizadem federada usando um ambiente de computação de borda

RESUMO

O aprendizado federado é um paradigma de aprendizado de máquina em que muitos clientes treinam cooperativamente um único modelo centralizado, mantendo seus dados privados e descentralizados. Ele é comumente usado em computação de borda, que consiste em colocar as cargas de trabalho do computador (hardware e software) o mais próximo possível da borda, onde os dados estão sendo criados e onde as ações estão ocorrendo, permitindo tempos de resposta mais rápidos, maior privacidade de dados e custos de transferência de dados reduzidos. No entanto, devido às distribuições/conteúdos de dados heterogêneos dos clientes, não é trivial avaliar com precisão as contribuições de modelos locais na agregação de modelos centralizados globais. Este tem sido um grande desafio para o paradigma, comumente conhecido como desequilíbrio de dados ou desequilíbrio de classes. Trabalhos anteriores foram propostos para resolver esse problema, como algoritmos de Deep Reinforcement Learning (DRL) para aprender dinamicamente o peso das contribuições de cada cliente em cada rodada. Testar e avaliar esses algoritmos FL pode ser uma tarefa muito difícil e complexa devido à natureza distribuída dos sistemas. Neste trabalho, uma revisão da literatura desses conceitos é apresentada a fim de apresentar ao leitor o contexto suficiente desse desafio e uma estrutura de ambiente semelhante a uma borda distribuída é proposta para avaliar o desempenho de algoritmos FL de uma maneira mais fácil e escalável .

Palavras-chave: Aprendizagem Federada, Computação de Borda, Kubernetes, Microsserviços, Framework.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artifical Intelligence
API	Application Programming Interface
CNN	Convolution Neural Network
CPU	Central Processing Unit
CIFAR	Canadian Institute for Advanced Research
DNN	Deep Neural Network
DRL	Deep Reinforcement Learning
FedAvg	Federated Averaging
FedAvgM	Federated Averaging with Momentum
FedOpt	Advanced Federated Optimization
FedYogi	Advanced Federated Optimization with Yogi
FMNIST	Fashion Modified National Institute of Standards and Technology
GB	Gigabyte
GiB	Gibibyte
gRPC	Google Remote Procedure Call
ID	Identifier
IID	Independent and identically distributed
IoT	Internet of Things
MB	Megabyte
ML	Machine Learning
MEC	Multi-access Edge Computing
MNIST	Modified National Institute of Standards and Technology
RAM	Random-Access Memory
RPC	Remote Procedure Call

TiB	Tebibyte
VM	Virtual Machine

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Federated Learning (FL) is a machine learning solution designed to train machine learning models while keeping the data private and decentralized (MCMAHAN et al., 2017). The main idea of FL is for each client to train its own local model using its own data and, afterwards, upload the generated local model to a single centralized server, where the local models of the participant clients will be aggregated and weighted to create a global model. FL is commonly used in edge computing, which involves placing computer workloads (both hardware and software) as close as possible to the edge, where the data is being created and where actions are occurring, enabling faster response times, greater data privacy, and reduced data transfer costs (DARLING, 2022b). Thus, FL can be used with a variety of clients such as smartphones, sensors, IoT devices and data silos (distributed databases that need to keep their data private with the outside world). However, as seen in (DUAN et al., 2021), in general, the data distribution of the mobile systems and other similar settings is imbalanced, which can increase the bias of model and impacting in a negative way its performance. Different approaches have been proposed, such as Deep Reinforcement Learning (DRL) (PLAAT, 2022), as a solution for this problem.

Although, regular centralized machine learning may outperform FL in terms of prediction performance (NILSSON et al., 2018), it requires the entire dataset to be shared. Its first application was in Google GBoard (HARD et al., 2018), which learns from every smartphone using Gboard without sharing user data. Since then, FL applicability has advanced to a variety of fields such as autonomous vehicles, traffic prediction and monitoring, healthcare, telecom, IoT, pharmaceutics, industrial management, industrial IoT, and healthcare and medical AI (SHAHEEN et al., 2022). Moreover, the number of publications in the academy that has FL as the main subject has increased significantly since its first proposal (MCMAHAN et al., 2017).

FL has been greatly enabled by the vast increase of IoT devices. The total number of IoT connections will reach 83 billion by 2024, rising from 35 billion connections in 2020; a growth of 130% over the next four years. The industrial sector has been identified as a key driver of this growth. Expansion will be driven by the increasing use of private networks that leverage cellular networks standards (RESEARCH, 2020). The evolution of the IoT devices computational power has also enabled FL, since an edge computer can process data locally, its sensors (such as cameras) could collect samples (such as images or frames) at a higher resolution, and at a higher frequency (such as frame rate) than would be possible if the data had to be sent to the cloud for processing (DARLING, 2022a).

One of the many challenges that have come up with the advance of FL is how to deal with data imbalance and heterogeneity, as seen in (SHAHEEN et al., 2022). In FL, using their local data, each edge node trains a shared model. As a result, the distribution of data from those edge devices is based on their many uses. Imagine a scenario, for example, of the distribution of cameras in a surveillance system. In comparison to cameras located in the wild, cameras in the park, for example, capture more photographs of humans. Also, the size of the dataset that each one of those cameras will have to train their local models might differ by a large magnitude since a park might have much more data to input than a camera in the wild. Furthermore, an approach that has been proposed in many studies to dynamically address weight for the local models of clients participating in the FL global model and, therefore, deal with the heterogeneous data is DRL (GUO; WU, 2022; HUANG et al., 2022; SUN et al., 2022; ZHENG et al., 2022).

DRL has gathered much attention recently. Recent studies have shown impressive results in activities as diverse as autonomous driving (LIU et al., 2022), game playing (WANG et al., 2021), molecular recombination (SRIDHARAN et al., 2022), and robotics (RUDIN et al., 2022). In all those applications, it has been used in computer programs to teach them how to solve difficult problems, for example, how to fly model helicopters and perform aerobatic maneuvers, and, in some applications, it has already outsmarted some of the most skilled humans, such as in Atari, Go, poker and StarCraft.

This work proposes a framework architecture for FL algorithms testing that enables users to easily create different scenarios of training by simply changing parameters. These include computing and data distributions, datasets, global model aggregations, client models and server and client training parameters. The framework is also capable of collecting training results and resource usage metrics. A set of experiments have been conducted varying these parameters on top of the proposed architecture in order to demonstrate the capabilities of the framework. Collected data has been analyzed in order to understand how the framework performed in each scenario.

The main goal of this work is to first build a Proof-of-Concept (PoC) of the conceptual framework architecture. Secondly, compare the conceptual framework with the PoC and run the experiments to demonstrate its capabilities. Finally, analyze how the PoC can be enhanced in further development continuation.

The remainder of this work is organized as follows. In section 2 a literature review is presented for edge computing and FL respectively. In section 3 it is presented the tools and frameworks used in the proposed PoC. In section 4 the conceptual framework architecture is proposed. Section 5 presents the PoC developed. Section 6 presents the experiments and the obtained results. Section 7 concludes the work with improvements that can be done in the PoC solution provided.

2 LITERATURE REVIEW

This chapter presents the main topics of this work, providing an overview of the current state of the art of edge computing, federated learning and deep reinforcement learning.

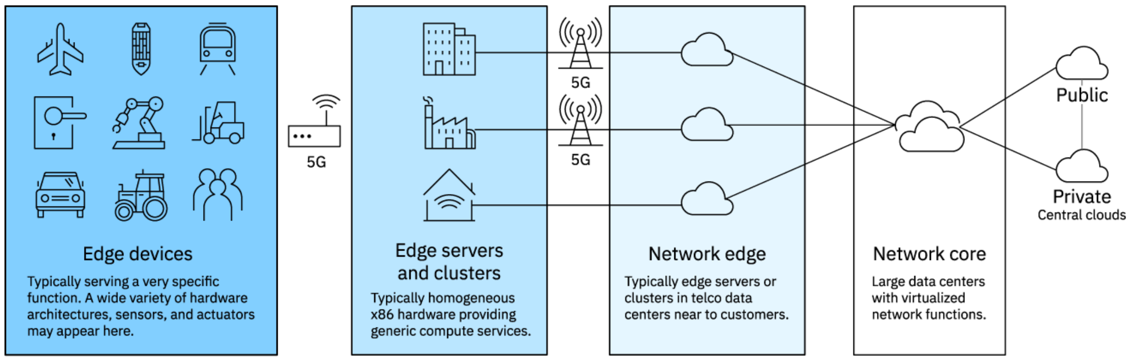
2.1 Edge Computing

Data is increasingly produced at the edge of the network, therefore, it would be more efficient to also process the data at the edge of the network. With the advancement of telecommunication services and the increase of the necessity for low latency computing, the edge computing paradigm has been motivated. In edge computing, instead of having computer workloads (both hardware and software) centralized in a data center (cloud), we have them as close as possible to the edge, where the data is being created and where actions are occurring, thus benefiting lower latency, greater data privacy and reduced data transfer costs. For (SHI et al., 2016), edge computing refers to the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services. Therefore, edge devices can be any device with Internet access such as smartphones, smart cars, or other IoT devices.

Multi-access Edge Computing (MEC) (GIUST; COSTA-PEREZ; REZNIK, 2017) is proposed as a key solution that enables operators to open their networks to new services and IT ecosystems and leverage edge-cloud benefits in their networks and systems, since it places storage and computation at the network edge. The close proximity from the end users and connected devices provides extremely low latency and high bandwidth while minimizing centralized cloud limitations such as delay, access bottlenecks, and single points of failure. A MEC infrastructure representation is shown in Figure 2.1, where 5G IoT devices are connected to a local access network for high throughput and massive data volume in low-latency applications, while also utilizing a centralized cloud for latency-tolerant applications.

MEC use cases can be seen in real-time traffic monitoring (WAN; DING; CHEN, 2022) and autonomous vehicles (LIU et al., 2019). In traffic monitoring, real-time and accurate video analysis are very important and challenging work, especially in situations with complex street scenes, therefore, edge computing based video pre-processing is pro-

Figure 2.1: Multi-access Edge Computing Infrastructure


Edge devices
Typically serving a very specific function. A wide variety of hardware architectures, sensors, and actuators may appear here.
5G
Edge servers and clusters
Typically homogeneous x86 hardware providing generic compute services.
5G
5G
Network edge
Typically edge servers or clusters in telco data centers near to customers.
Network core
Large data centers with virtualized network functions.
Public
Private
Central clouds

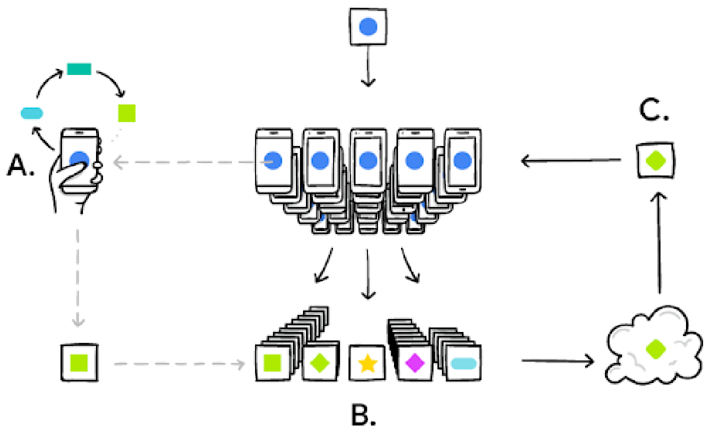
Source: (DARLING, 2022b)

posed to eliminate the redundant frames that edge devices need to process, since a huge amount of vehicle video data is generated. Also, the decentralized and highly available nature of multi-access edge computing is taken advantage to collect, store, and analyze city traffic data in multiple sensors. For autonomous vehicles, large amount of data processing from different sensors at high speed in real time is needed in order to guarantee driver safety.

2.2 Federated Learning

Federated learning (FL) is a distributed form of machine learning proposed by Google (MCMAHAN et al., 2017) to train models at scale while allowing the user data to be private. In Federated Averaging (FedAvg), the server aggregates the model updates using simple averaging and returns the new model parameters to the client devices, which continue training using the updated model parameters. Google's proposal provided the first definition of federated learning, as well as the Federated Optimization (KONEčNý et al., 2016) approach to further improve these federated algorithms. Advanced Federated Optimization (REDDI et al., 2021) is a variant of the Stochastic Gradient Descent (SGD) algorithm, which is commonly used in centralized training. In FedOpt, each local node applies SGD to its local data to compute the gradients and then sends the gradients to a central server. The server then aggregates the gradients from all the nodes to update the global model. Adaptive Federated Optimization with Yogi (FedYogi) incorporates a momentum-based optimizer called Yogi after the central server aggregates the model updates using the FedAvg algorithm to improve the convergence rate. Federated Averaging with Momentum (HSU; QI; BROWN, 2019) incorporates a momentum-based optimizer, similar to the Yogi optimizer in FedYogi. The key difference is that it uses a combination of the gradients from the current iteration and the gradients from the previous iteration to update the model parameters. This allows the optimizer to maintain a direction of movement even when the gradient changes direction, which helps to smooth out noisy gradients and accelerate convergence. In the "Federated Learning: Collaborative Machine Learning without Centralized Training Data" blog post (GOOGLE, 2017), Google explains how FL is enabling mobile phones to collaboratively learn a shared prediction model while keeping all the training data on device, decoupling the ability to do machine learning from the need to store the data in the cloud. In addition, is cited the current use of FL to predict keyboard words in Google's Gboard (HARD et al., 2018) and how it can be also used for photo ranking and further improving language models. Figure 2.2 illustrates Google's FL workflow.

Figure 2.2: Illustration of Google's FL Workflow

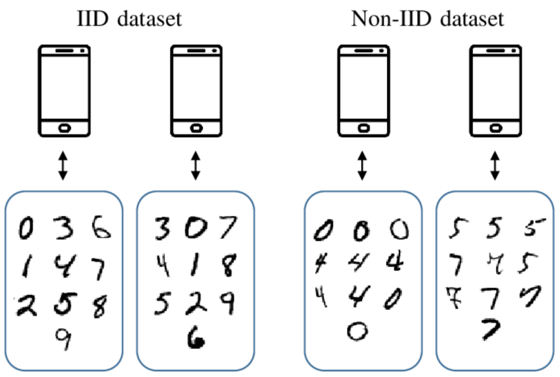


Source: (GOOGLE, 2017)

FL usually deals with data distributed across multiple devices. In such settings, data is usually non-independently and identically distributed (i.e., non-IID). One of the main challenge in FL is how to deal with the heterogeneity of the data distribution among the parties, since the distribution of data from those edge devices is based on their many uses (SHAHEEN et al., 2022). The difference between an IID dataset and a non-IID dataset is illustrated in Figure 2.3 where, from left to right, the first two mobile devices contains data that is equally distributed (both have the numbers 1 to 9) whilst the last two contains an unequal distribution (one has the numbers 0 and 4 and the other has the numbers 5 and 7).

Furthermore, a lot of use cases in FL have data samples that are distributed among

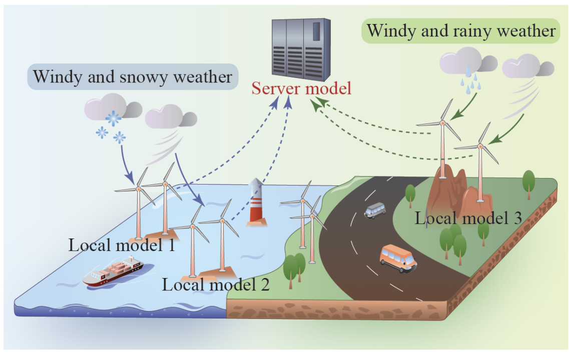
Figure 2.3: Illustration of IID vs. non-IID for MNIST dataset



Source: (HELLSTRöM et al., 2020)

multiple devices which are not always synchronized and may have limited connectivity. Thus, it cannot simply train these devices in parallel and aggregate them in a direct way, as it cannot guarantee the device availability or the homogeneity of the data. Understanding how to properly select clients and correctly weight each of those clients contributions in the global model remains an open problem in FL. Example of a practical scenario of data imbalance and heterogeneity in which DRL was used in FL as a solution can be seen in recent work proposed to deal with blade icing detection in distributed wind turbines (CHENG et al., 2022). Wind turbines that are closer to the sea experience windy and snowy weather whilst those more close to the continent deal with windy and rainy conditions. This heterogeneity introduces a bias in the local models since one might be more susceptible to icing than another. Therefore, since our objective is to identify icing, clients that experience more icing should have a different weight assigned to its contribution in the global model than others that have not. Figure 2.4 illustrates this scenario of data imbalance and heterogeneity whereas data from local model 1 and 2 is unbalanced in comparison with data from local model 3 due to the distribution of the clients.

Figure 2.4: Wind turbines use case with data imbalance



Source: (CHENG et al., 2022)

3 TOOLS AND FRAMEWORKS

This section provides an overview of the tools and frameworks used to implement the final solution. In section 3.1, are presented orchestration, deployment and building tools. In section 3.2, are presented libraries and frameworks for FL. Data storage tools are presented in section 3.3 . Finally, monitoring and data visualization tools are presented in section 3.4

3.1 Orchestration, Deployment and Building

This section present tools related to orchestration, deployment and building of containers.

3.1.1 Docker

Docker[1] provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows running many containers simultaneously on a given host.

Containers provide a robust solution for bundling software and its dependencies into a transportable unit that can run seamlessly across diverse computing environments. With a containerized application, all its libraries, settings, and tools are encapsulated within an isolated environment. Developers can avoid the arduous task of addressing compatibility issues with varied hardware and software and concentrate solely on the application's functionality. This inherent portability of containers eradicates the need to reconfigure applications for distinct environments, ensuring uniformity and dependability.

Docker Compose is a tool used for defining and running multi-container Docker applications. It is a way to define and configure multiple containers that work together to provide a complete application stack. Docker Compose is particularly useful for development and testing environments, where you need to quickly spin up a complex stack of services. By defining all the services in a single YAML file, you can easily reproduce the same environment on any machine that has Docker installed.

[1]https://docs.docker.com/get-started/overview/ (accessed September 27th, 2022)

3.1.2 Kubernetes

Kubernetes[2] is a portable, extensible, open source platform for managing containerized workloads and services, which facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

A Kubernetes cluster[3] consists of a set of worker machines, called nodes, which run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. Heterogeneous edge-computing devices can easily be integrated and managed in the Kubernetes cluster as computing nodes in an easy and scalable way. This also facilitates the deployment of different FL algorithms and the creation of different scenarios of testing in a distributed infrastructure environment.

3.1.3 Lens Desktop

Lens[4] provides a comprehensive set of features that make it easier for developers, DevOps teams, and Kubernetes administrators to manage Kubernetes clusters. It was used in this work as a front-end way to interact with Kubernetes and create deployment in clusters.

3.2 Libraries and Frameworks

This section approaches the utilized libraries and frameworks to run federated learning (FL).

3.2.1 PyTorch

PyTorch[5] is a popular open-source machine learning library that was created by Meta's AI research team. It is used to develop and train deep learning models and is

[2]https://kubernetes.io/docs/concepts/overview/ (accessed September 27th, 2022)
[3]https://kubernetes.io/docs/concepts/overview/components/ (accessed September 27th, 2022)
[4]https://k8slens.dev/ (accessed September 27th, 2022)
[5]https://pytorch.org/get-started/pytorch-2.0/ (accessed March 17th, 2023)

written in Python, which makes it easy to use and integrate with other Python libraries.

PyTorch was used as the main library for training and testing the FL models used in this work.

3.2.2 Poetry

Poetry[6] is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. Poetry offers a lockfile to ensure repeatable installs, and can build your project for distribution. It was used in this work to create the packages of the FL client and server applications.

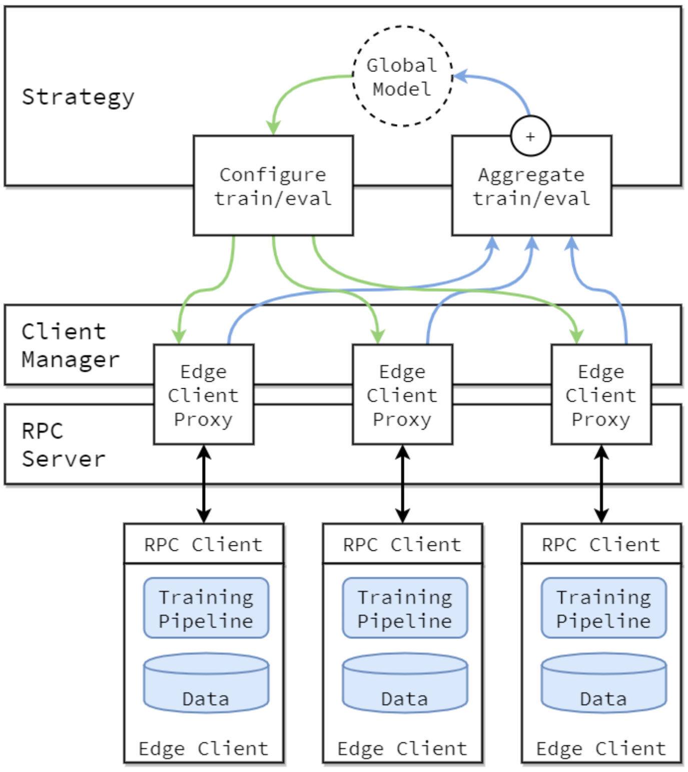
3.2.3 Flower

The concept of federated learning emerged in response to the need to leverage data from multiple devices while ensuring its privacy, as discussed in this work. However, federated learning introduces two additional challenges that are not present in traditional machine learning: scaling to multiple clients and dealing with data heterogeneity.

To address these challenges, as proposed in (BEUTEL et al., 2022), Flower[8] (flwr) has been developed as an open-source framework for building federated learning systems. Flower provides two main interfaces: the client and the server. These interfaces enable the decentralization of standard centralized machine learning solutions by implementing the necessary methods, making it easier to build and deploy federated learning systems. Figure 3.1 shows the Flower Federated framework architecture when running with multiple edge clients. Each edge device participating in the training process runs a Flower client that contains a local machine learning model. To ensure connectivity between the clients and the server, Flower provides a transparent connection via the Edge Client Proxy using an RPC protocol such as gRPC.

[6]https://python-poetry.org/docs/ (accessed March 17th, 2023)
[7]https://flower.dev/docs/architecture.html (accessed March 18th, 2023)
[8]https://flower.dev (Accessed March 17th, 2023)

Figure 3.1: Flower Federated framework client-server architecture for edge devices



Source: Flower Documentation[7]

3.3 Data Storage

This section present the utilized tools for storage.

3.3.1 Rook Ceph

Rook Ceph[9] is a storage solution that combines the capabilities of the Rook storage orchestrator with the Ceph distributed storage system. Rook is an open-source tool for managing storage systems on Kubernetes, while Ceph is a distributed object and file storage system that provides scalability, reliability, and performance. Rook Ceph was used as the persistent storage for the client and server application.

3.4 Monitoring and Visualization

Finally, in this section are presented monitoring and data visualization tools.

[9]https://rook.io/ (accessed March 17th, 2023)

3.4.1 Prometheus

Prometheus[10] is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. Prometheus was used as the main application for monitoring usage of resources in the PoC solution of this work.

3.4.2 Grafana

Grafana[11] is a data visualization tool commonly used with Prometheus that allows you to query, visualize, alert on and understand your metrics. It was used as the main tool to visualize resource usage of Prometheus.

[10]https://prometheus.io/docs/introduction/overview/ (accessed September 27th, 2022)
[11]https://grafana.com/oss/grafana/ (accessed September 27th, 2022)

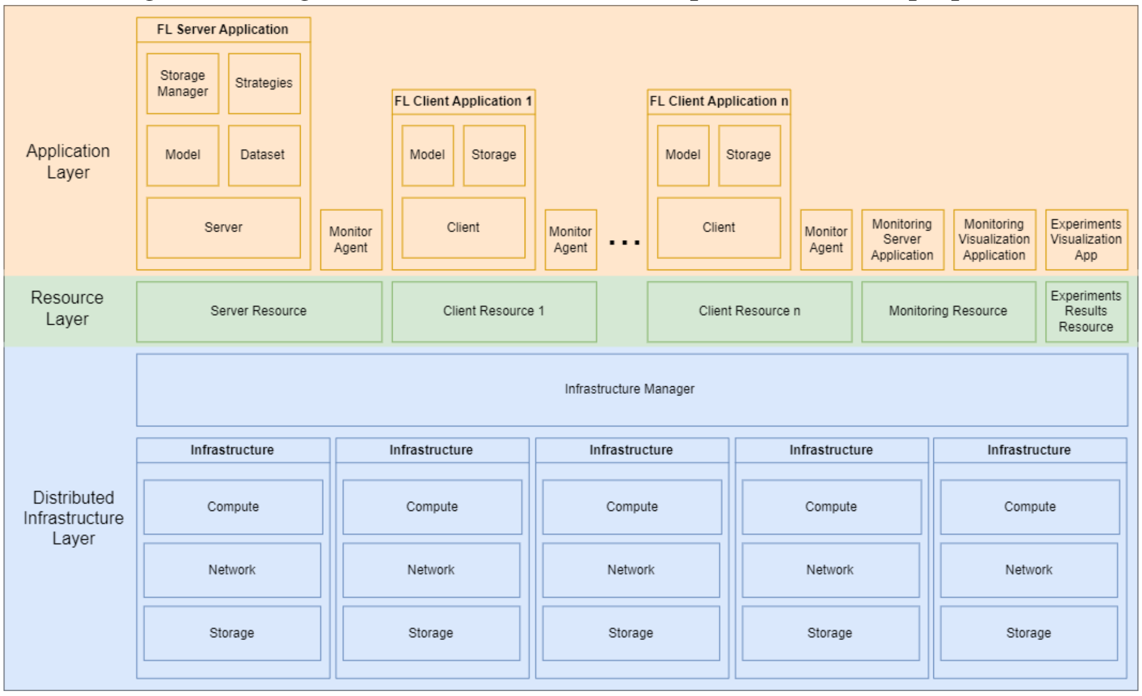
4 CONCEPTUAL FRAMEWORK

This section presents the conceptual framework proposed for testing FL algorithms. In section 4.1 is presented the architecture overview . The following sections detail the layers presented in the first section whereas section 4.2 presents the distributed infrastructure layer, section 4.3 details the resource layer and finally section 4.4 details the application layer.

4.1 Architecture Overview

The main objective of the proposed architecture is to enable FL testing in a platform where users can easily change parameters to create different types of scenarios in a distributed computing environment. These include different computing and data distributions, datasets, global model aggregations, client models and server and client training parameters. The framework should also be capable of collecting training results and resource usage metrics. Figure 4.1 presents an overview of the proposed architecture.

Figure 4.1: High-level overview of the conceptual framework proposed



Source: Image provided by the author

The reason that the framework is designed in these layers is to have the highest level of independence within them. This enables easier development and segregation of functions, so if developments are done in one layer, it does not affect the other. Furthermore, sections 4.2 to 4.4 details each one of the components of the layers.

4.2 Distributed Infrastructure Layer

This is the bottom layer which should contain all the infrastructure needed to create a distributed computing environment. It is responsible for scaling computing, storage and networking from a pool of resources in order to meet the user inputted parameters. The user parameters will reflect on how the resource layer will be laid out to create the scenario desired for testing. For example, let's say a user wants to configure a scenario of FL using ten clients whereas each client has a specific computing requirement of four CPUs cores and 4GB of RAM per client. The distributed infrastructure manager will then configure the resource layer from its pool of resources (compute, storage and networking) with ten clients, each one with the specified specifications to meet user requirements. Having independent infrastructure from the resource and application layer enables usage of different types of computational devices together, since application only sees the resource layer. This will be fundamental to enable different type of scenarios where different types of edge computing devices can be used as clients.

4.3 Resource Layer

The resource layer can be found on top of the distributed infrastructure layer and will match the configuration set by the user to setup the FL training scenario as mentioned before. We can categorize the distribution of resources in four major categories: server resource, client resources, monitoring resource and experiment results resource. The server resource will contain all the necessary resources to run the FL server matching the user configuration and the same can be said to the clients resources. The monitoring resource will contain all the resources needed to run the monitoring server which will monitor the distributed infrastructure layer and record the usage of the resources. The experiment results resource will have all the resources necessary to run an application to display the results of the experiments run in the framework.

4.4 Application Layer

The application layer will run on top of the resource layer in each correspondent resource. It contains all the applications necessary to run a FL algorithm in the framework, such as, the FL server application, FL client application, monitor agent application, monitor server application and the experiment results visualization application. Subsections 4.4.1 to 4.4.4 details each one of the mentioned applications.

4.4.1 FL Server Application

The FL Server application can be divided in five major components that enable FL testing in a distributed environment.

Server is responsible for communicating with the clients and controlling the entire FL learning process which includes selecting available clients to start training; control of the number of FL server rounds and round timeout; global model parameters aggregation and distribution and retrieval of model parameters.

Model is responsible for server-side initialization of global model parameters since some global model aggregation strategies need it to start FL learning training and also to enable the usage of different models to learn weight balancing of client models to distribute parameters.

Strategies is responsible for selecting and configuring supported global model aggregation strategies from the user specified configuration.

Dataset is a collection of supported dataset by the framework which is used by the application to handle the dataset configuration in memory and also how to properly obtain the raw data of the dataset to create distributions versions of it via the storage manager..

Storage Manager is responsible for distributing the raw data of the configured dataset throughout the distributed infrastructure storage in order to achieve user requirements for testing. For example, if a user wants to test a FL algorithm in a dataset using an unbalanced non-iid data distribution with ten clients, the storage manager will separate the data in ten unbalanced and biased non-iid data parts. The storage manager of the server application is also responsible for managing where each experiment will write its data and also where the client will output their results data in the system.

4.4.2 FL Client Application

The FL client application can be divided in three major components.

Client is responsible for the training and testing algorithms run in the client side and the connection with the server.

Model is responsible for selecting the desired model configured by the user to be used by the client component to train it.

Storage Manager is responsible for loading the distributed data for training in the client component and handling the experiments test results of the client in the storage.

4.4.3 Monitoring Application

The monitoring application should be responsible for gathering data from resource usage of the distributed infrastructure and storing it for further visualization and analysis. The monitoring agent is responsible for gathering the resource usage data from the resources and sending the metrics to the monitoring server, which will then be able to storage the data in a database and enable easy to understand visualization for users of the framework.

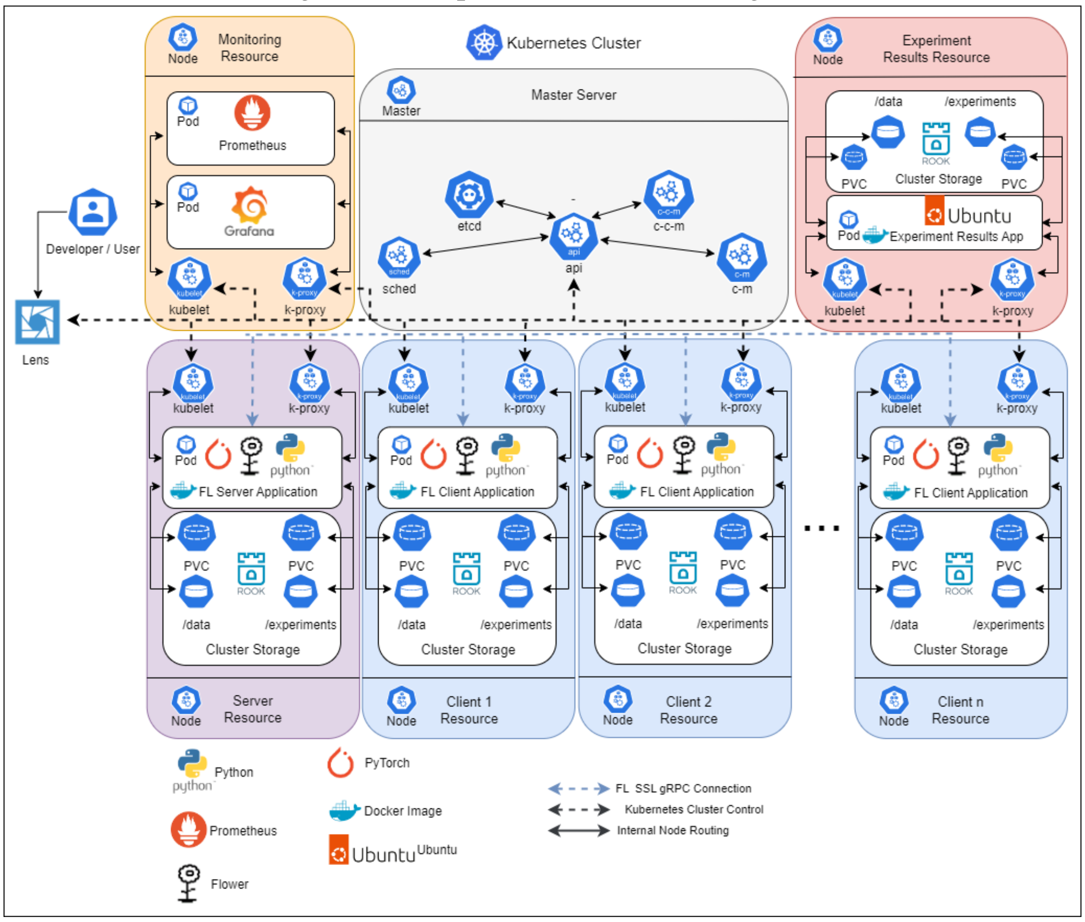
4.4.4 Experiment Visualization Application

This application is responsible for accessing data from already run experiments and enable user to visualize the data for each steps of the FL training experiment.

5 SOLUTION

The previous chapter detailed the conceptual framework proposed, which can be implemented in several ways. This chapter presents the current PoC solution implemented of the conceptual framework in order to run fully functional federated learning scenarios in a distributed infrastructure with multiple clients and with different data distributions .

Figure 5.1: Implemented solution diagram



Source: Image provided by the author

The PoC consists in an end-to-end edge-like environment solution using edge computing devices orchestrated by Kubernetes to run the FL applications, the server and the clients, and also the applications for monitoring the resources usage. Figure 5.1 shows the full diagram of the PoC solution implemented. The distributed infrastructure layer is composed by the Kubernetes cluster of the Institute of Informatics which serves as foundation for each one of the resources. The application layer can be visualized on top of the resources as containerized Docker images of each application running in pods of the cluster. Further sections will go through more details about the layers, including how they were assembled with the tools and frameworks presented in Chapter 3, the underlying infrastructure of the Kubernetes cluster and how each one of the applications of the solution works.

5.1 Distributed Infrastructure Layer

The Kubernetes cluster at the Institute of Informatics boasts impressive computing resources, with 352 CPUs, 544 GiB of RAM, and 6.3 TiB of disk space spread across 43 distinct nodes, as indicated in Table 5.1. These nodes are conveniently categorized into three classifications: "computer," "edge," and "server," enabling experimentation with pods that are constrained to specific machines with hardware that closely matches the real-life devices being simulated. For instance, the "edge" label encompasses a total of twenty-three Raspberry Pi devices (three Raspberry Pi 4s and twenty Raspberry Pi 3s) with less powerful hardware specifications relative to the "computer" and "server" machines.

Table 5.1: Institute of Informatics Kubernetes Cluster Capacity

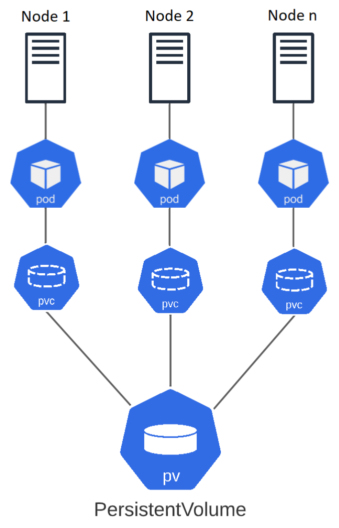
Type	**Nodes**	**Total RAM**	**Total CPUs cores**	**Total storage space**
Computer	14	110.9GiB	56	5.2TiB
Edge	23	40.6GiB	92	559.2GiB
Server	6	392.4GiB	204	587.2GiB

Source: Table provided by the author

The cluster nodes are interconnected by a network switch and has distributed storage configured using Rook Ceph Filesystem (subsection 3.3.1) to build a layer on top of the storage resources. This enables the mounting of Persistent Volumes (PV) from the storage pool via a Persistent Storage Claim (PVC) into the container images that can be shared between applications to enable data distribution by the server and saving experiments results from each client. When a pod requests storage resources, it does so by creating a PVC that specifies the amount of storage required and any other requirements, such as access mode and storage class. The Kubernetes scheduler then looks for an available PV that matches the requirements specified in the PVC. If a matching PV is found, it is bound to the PVC, and the pod can use the storage resource provided by the PV. Figure 5.2 illustrates the interaction between a PVC and a PV. The advantage of using PVs and PVCs is that they provide a level of abstraction between the pod and the underlying storage infrastructure. This allows pods to request storage resources without having to know the details of the underlying storage infrastructure.

As seen in subsection 3.1.3, Lens is a Kubernetes platform manager and is used

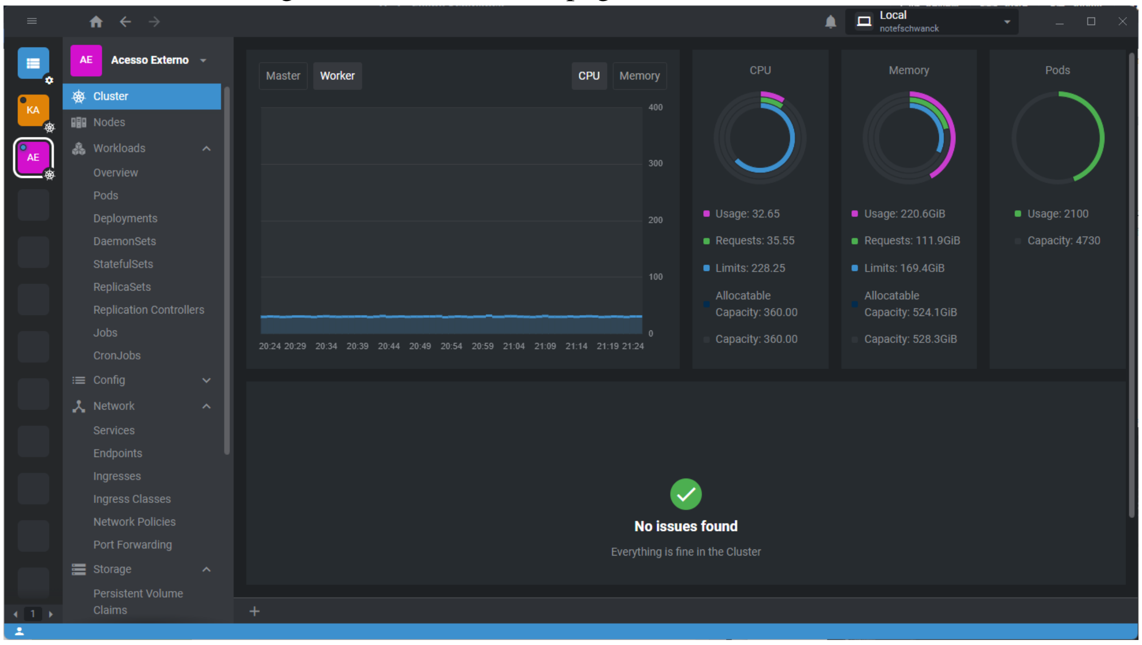
Figure 5.2: PVC and PV illustration



Source: Image provided by the author

to managed the Kubernetes cluster, providing an easy to use interface, enabling easy deployment of containers in the platform and visibility of ongoing experiments. Figure 5.3 shows the cluster page of Lens user interface of the Institute of Informatics cluster. Users can configure a scenario in a Kubernetes deployment file in Lens stating the needed resources and the application parameters via environment variables of the containers.

Figure 5.3: Lens cluster page of the user interface



CPU
- Usage: 32.65
- Requests: 35.55
- Limits: 228.25
- Allocatable Capacity: 360.00
- Capacity: 360.00

Memory
- Usage: 220.6GiB
- Requests: 111.9GiB
- Limits: 169.4GiB
- Allocatable Capacity: 524.1GiB
- Capacity: 528.3GiB

Pods
- Usage: 2100
- Capacity: 4730

No issues found

Everything is fine in the Cluster

Source: Image provided by the author

Knowing the context and making a parallel with the conceptual framework, in the PoC, the infrastructure of the Kubernetes cluster of the Institute of Informatics serves as the distributed infrastructure and Kubernetes plays the role of the infrastructure manager.

5.2 Resource Layer

Kubernetes utilizes labels to organize objects. They are key-value pairs that can be attached to Kubernetes objects such as pods, services, nodes, and deployments and can be used to identify and group related objects together. This is a powerful mechanism for selecting and manipulating subsets of objects based on specific criteria and can also be used to manage nodes in a Kubernetes cluster. Nodes are the worker machines that run containerized applications and services in a Kubernetes cluster. By attaching labels to nodes, we can assign specific roles or attributes to them and use those labels to manage and schedule workloads on those nodes.

For example, you can label nodes based on their hardware characteristics, such as CPU or memory capacity, and then use those labels to schedule workloads that require specific hardware requirements. The Kubernetes Cluster of the Institute of Informatics uses the "node-type" label to identify if a node is a "computer", "server" or "edge" type of computational device.

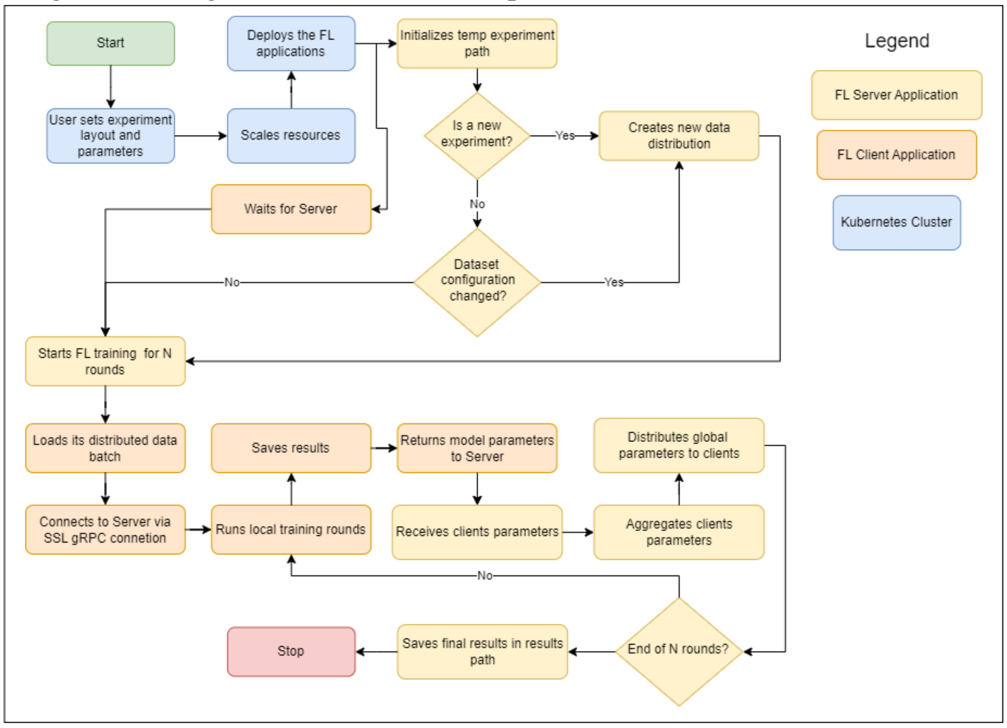
Finally, in the PoC solution, the resource layer used is a total reflex of the configuration set in the Kubernetes deployment file. Labels were used to assign where each containerized application will run and also how much resource of the node will be available for them to use.

5.3 Application Layer

The application layer of the PoC solution is composed by all the docker images built by the author, such as the FL server, client and experiment results images, which contains all the implemented logic for FL algorithms testing. For monitoring purposes, were used the Prometheus monitoring installed in the Kubernetes Cluster to capture resource usage by the applications and also Grafana to visualize the data in charts.

Figure 5.4 shows how an experiment starts from the beginning to the end in the PoC solution from a high-level overview. In the blue boxes we can see the actions of the Kubernetes cluster, in the yellow ones from the FL Server application and in orange from the FL clients. From a high-level perspective, the user configures a scenario in Lens for deployment, Kubernetes scales the necessary resources and deploys the applications. Afterwards, the FL server handles the data distribution of the dataset and starts the FL training algorithm. The client that was waiting for the server connects with it, runs local training rounds and return the model parameters back to the server. The server receives those parameters, aggregates them using the configured aggregation method by the user in the experiment layout and distributes them back to the clients, which will start local training rounds again with the new parameters. After all server rounds are done, the FL server will save the results in the correct output folder of the experiment and the experiment will be over.

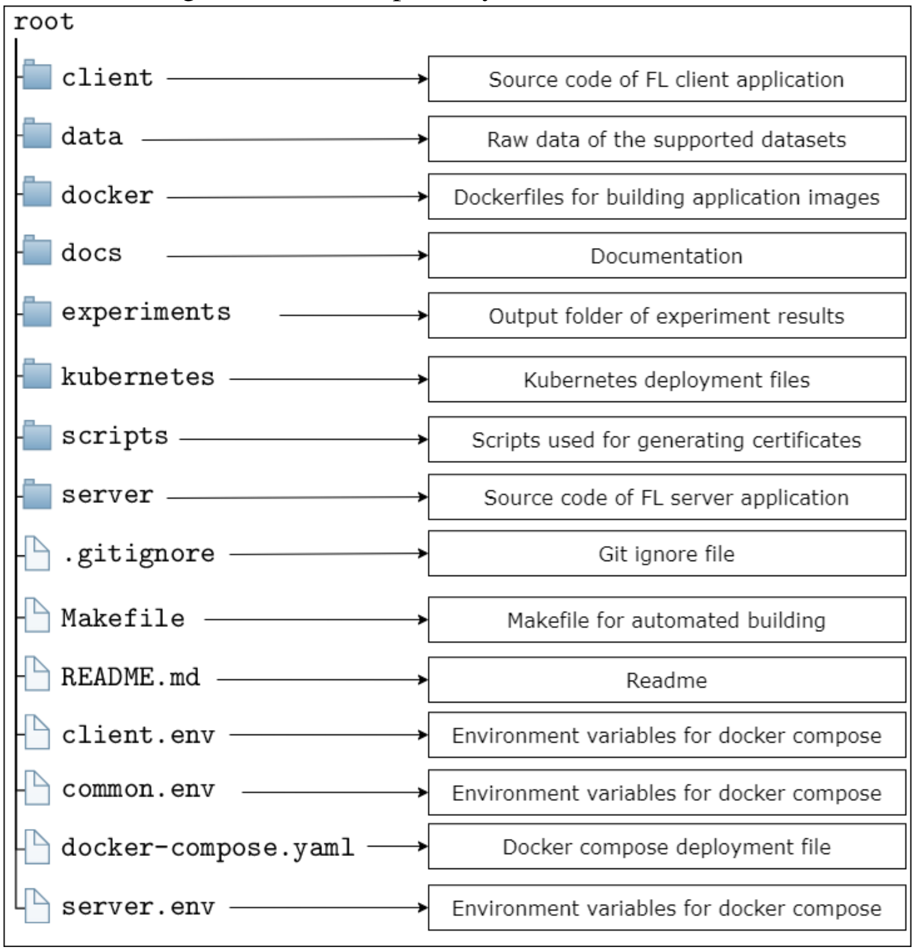
Figure 5.4: High-level flowchart of experiment execution in the PoC solution


Start
User sets experiment layout and parameters
Scales resources
Deploys the FL applications
Initializes temp experiment path
Is a new experiment?
Yes
Creates new data distribution
No
Waits for Server
Dataset configuration changed?
No
Yes
Starts FL training for N rounds
Loads its distributed data batch
Connects to Server via SSL gRPC connetion
Runs local training rounds
Saves results
Returns model parameters to Server
Receives clients parameters
Aggregates clients parameters
Distributes global parameters to clients
End of N rounds?
No
Saves final results in results path
Stop
Legend
FL Server Application
FL Client Application
Kubernetes Cluster

Source: Image provided by the author

The repository used in the development of the solution was organized into isolated packages containing all necessary code, data or declarations of each component in order to maintain independence and enable better re-usability. The applications of the FL server and client were also developed in order to run in bare-metal environments. In addition, to enable easy testing of the application during development, you can option to run docker compose environment to deploy the application locally using the docker images built. Figure 6.2 illustrates the code repository root path organization with the description of each folder and file. Further subsections will detail each one of the applications and their role in the PoC solution.

Figure 5.5: PoC repository structure at the time


root
client
Source code of FL client application
data
Raw data of the supported datasets
docker
Dockerfiles for building application images
docs
Documentation
experiments
Output folder of experiment results
kubernetes
Kubernetes deployment files
scripts
Scripts used for generating certificates
server
Source code of FL server application
.gitignore
Git ignore file
Makefile
Makefile for automated building
README.md
Readme
client.env
Environment variables for docker compose
common.env
Environment variables for docker compose
docker-compose.yaml
Docker compose deployment file
server.env
Environment variables for docker compose

Source: Image provided by the author

5.3.1 Server Application

The server is a containerized Python application with the FlowerML server framework as dependency. The server storage manager is responsible for initializing the experiment root path in the distributed storage, where the results and logs of the current deployed run of each client will be saved. It will also be responsible for receiving the models being trained by the clients, averaging the received parameters using the selected strategy, then updating the clients' models with the averaged parameters. The connection is done to the clients through a gRPC connection with SSL encryption. The algorithm is outlined in

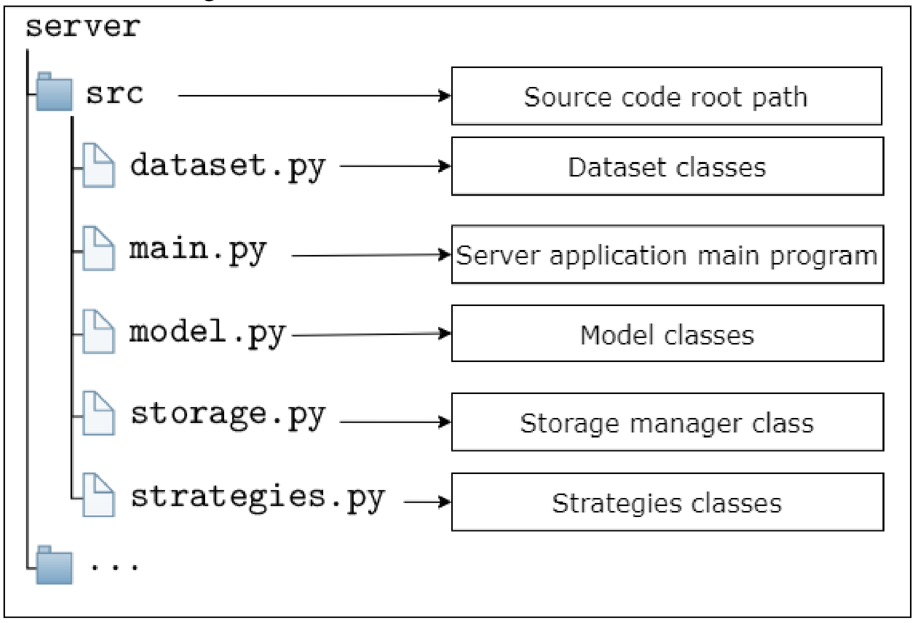
Algorithm 1.

Algorithm 1: Server application

```
 1 load environment variables with user parameters;
 2 if $experiment\_path$ don't exists then
 3     initialize $experiment\_path$ in storage;
 4     create temporary folder for current experiment run;
 5     initialize experiment configuration file ;
 6 load experiment configuration file;
 7 if dataset configuration changed or is empty then
 8     load raw data of the dataset;
 9     create new data distribution using user parameters;
10     save the data batches in the experiment path ;
11     save the new dataset configuration in configuration file;
12 else
13     use current dataset configuration;
14 configure server strategy with the selected $fl\_strategy$;
15 initialize server global model parameters;
16 open connection in $server\_address$;
17 while $minimum\_clients$ have not yet connected do
18     wait;
19 for $i \leftarrow rounds$ to $0$ do
20     receive parameters from clients;
21     aggregate client parameters;
22     send updated parameters back to clients;
23     evaluate current accuracy from clients;
24 save temporary experiment folder with results to current experiment run path;
25 update experiment configuration file;
```

The server address, number of server rounds, dataset, data distribution, global model aggregation strategy, client local rounds, model and minimum number of connected clients are parameterized for the server application and can be changed in each deployment. Figure 5.6 illustrates how the source code of the application was broken down. It is easy to correlate each part of the application with the conceptual framework since each file of the source code encapsulate its correspondent component.

Figure 5.6: Server code structure at the time



Source: Image provided by the author

dataset.py contains all the implemented classes of the supported datasets by the framework which are used by the application to handle the dataset configuration in memory and also how to properly obtain the raw data of the dataset. At the time of this work, the **CIFAR-10**, **CIFAR-100** and **FMNIST** dataset where implemented. Further development of this class can enable any dataset to be compatible with the framework. Subsection 6.2 will detail each one of the current supported datasets.

main.py contains the main program loop and the server component of the application which is responsible for communicating with the clients and controlling the entire FL learning process which includes selecting available clients to start training; control of the number of FL server rounds and round timeout; global model parameters aggregation and distribution and retrieval of model parameters. Here is where all the implemented packages are included and the main server program runs as demonstrated in Algorithm 1.

model.py contains the supported models and is responsible for server-side initialization of global model parameters since some global model aggregation strategies need it. At the time of this work, the current supported models are **CNN**, a **DNN**, **googlenet** (SZEGEDY et al., 2014) and **resnet** (HE et al., 2015). Further development of this class can enable any models to be compatible with the framework.

storage.py contains the storage manager class which is responsible for distributing the raw data of the configured dataset throughout the distributed infrastructure storage in order to achieve user requirements for testing. The storage manager of the server application is also responsible for managing where each experiment will write its data and also where the client will output their results data in the system. At the time of this work, the storage manager can create data distributions by changing the following parameters:

Balance boolean parameter that specifies if the data should be balanced or not between clients. If $true$, the dataset is balanced, meaning that the data batches have the same number of samples. Otherwise, if $false$, the number of data samples is different between them.

Non-IID boolean parameter that specifies if the data should be Non-IID or IID between clients. If $true$, the dataset is Non-IID, meaning that their is class imbalance between the clients. Otherwise, if $false$, the dataset has a balanced class distribution.

Distribution this parameter specifies how data should be distributed regarding the classes of the dataset if $Non-IID = true$. If set to **pat** it will generate a pathological scenario of class imbalance whereas each client will have only a subset of classes. If set to **dir**, it will use a heterogeneous unbalanced Dirichlet distribution of the classes, similar as seen in (YUROCHKIN et al., 2019), that can be modified by an α parameter to change the distribution aspects. Further development of this class can enable more data distributions to be compatible with the framework.

Further development of this class can enable more types of data distributions to be compatible with the framework.

strategies.py contains the strategies for supported global model aggregation strategies from the user specified configuration. At the time of this work, the supported strategies implemented are **FedAvg**, **FedAvgM**,**FedYogi** and **FedOpt**. Further development of this class can enable more strategies to be compatible with the framework.

The full code for the server can be found in the *server* module in the Github repository[1].

[1]https://github.com/fschwanck/tcc/tree/main/server/src

5.3.2 Client Application

The client waits for the server storage manager to initialize the experiment path in the distributed storage, connects to the server through a gRPC connection encrypted with SSL and performs the pre-defined number of epochs received from the server, which is the number of times that the data set passes through the neural network. When multiple clients are running, an individual client is oblivious to the existence of the other clients – it can only communicate with the server. The algorithm for the client application can be seen in Algorithm 2.

Algorithm 2: Service service loop

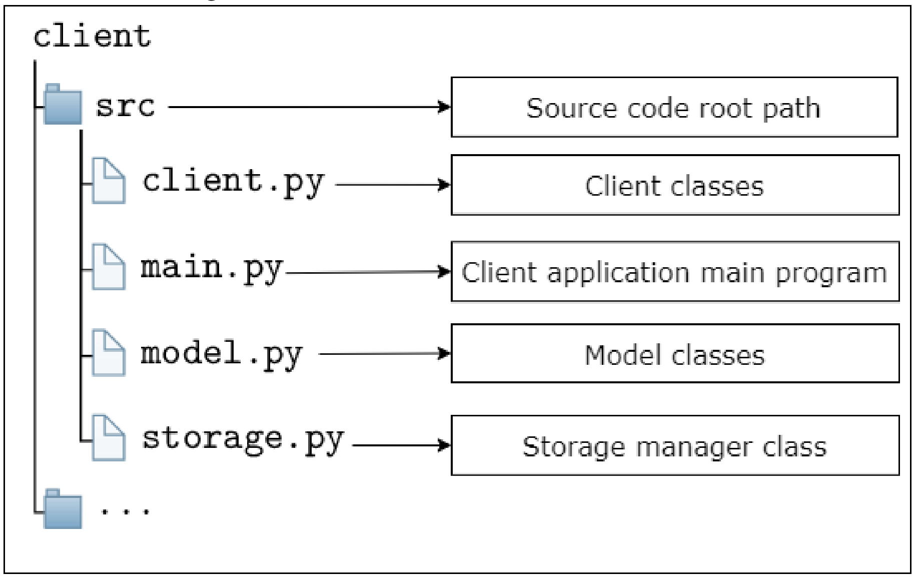
```
1  load environment variables with user parameters while
     $experiment\_path$ not initialized by the Server do
2      wait;
3  start connection with server in $address$;
4  while server connection is open do
5      for $i \leftarrow epochs$ to 0 do
6          iterates through the dataset batch training the local model;
7          saves current results of round in experiment path temporary folder;
8      upload model to server;
9      receive updated parameters;
10     update local parameters;
```

The client will only upload the model when the server requests the models, and the loop will end when the server finishes its rounds. Figure 5.7 illustrates how the source code of the application was broken down. It is easy to correlate each part of the application with the conceptual framework since each file of the source code encapsulate its correspondent component. The client's models and the address which they will connect are parameterized for the client application and can be changed in each deployment.

client.py contains the client classes with the utilized training and testing algorithms. At the time of this work it would only supported one PyTorch training algorithm.

main.py contains the client application main loop described in Algorithm 2 and is responsible for initializing the environment variables used to set parameters, such as the local model utilized for training and also common variables such as the experiment path to save results.

Figure 5.7: Client code structure at the time



Source: Image provided by the author

model.py contains all the supported models already described in the server application section.

storage.py contains the storage manager class of the client which is responsible for loading the distributed data batch for training and testing of the distributed storage and saving the experiments test results of the client in the storage.

The code for the client can be found in the *client* module in the Github repository[2].

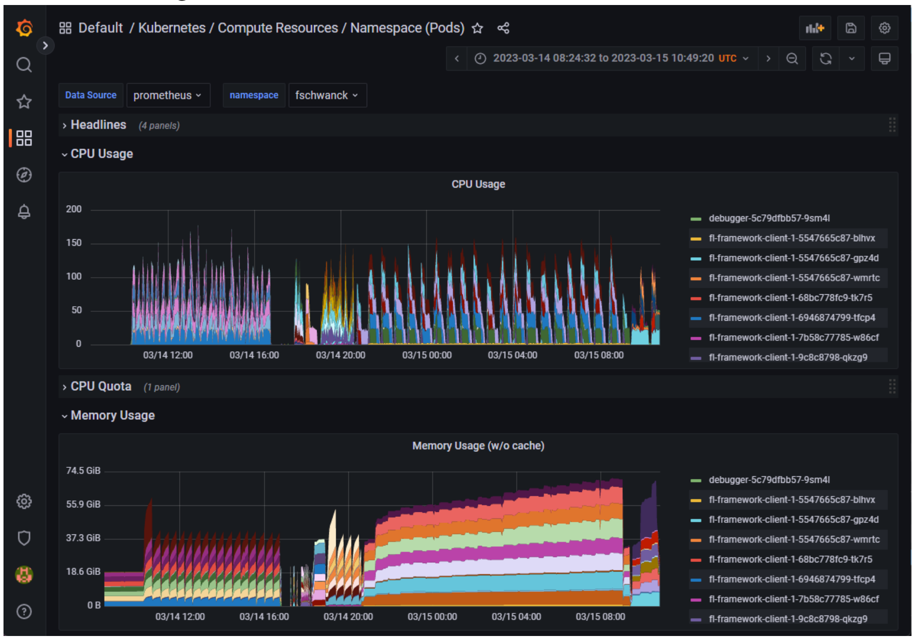
5.3.3 Monitoring Application

Prometheus uses the Kubernetes API to discover the various resources that it needs to monitor in the cluster, such as pods, services, deployments, nodes, and more. It does this by querying the Kubernetes API server to get information about the desired resources and then collecting metrics data from these resources.

For example, to monitor a Kubernetes pod, Prometheus will query the Kubernetes API server to get information about the pod's name, namespace, labels, and other metadata. Prometheus will then use this information to collect metrics data from the pod, such as CPU and memory usage, network traffic, and other metrics. Grafana queries the data from Prometheus database in order to enable real-time visualization of resource usage in dashboards. Figure 5.8 is a screenshot of the Kubernetes cluster Grafana interface.

[2]https://github.com/fschwanck/tcc/tree/main/client/src

Figure 5.8: Kubernetes cluster Grafana interface


Default / Kubernetes / Compute Resources / Namespace (Pods)
2023-03-14 08:24:32 to 2023-03-15 10:49:20 UTC
Data Source
prometheus
namespace
fschwanck
Headlines (4 panels)
CPU Usage
CPU Usage
200
150
100
50
0
03/14 12:00
03/14 16:00
03/14 20:00
03/15 00:00
03/15 04:00
03/15 08:00
debugger-5c79dfbb57-9sm4l
fl-framework-client-1-5547665c87-blhvx
fl-framework-client-1-5547665c87-gpz4d
fl-framework-client-1-5547665c87-wmrtc
fl-framework-client-1-68bc778fc9-tk7r5
fl-framework-client-1-6946874799-tfcp4
fl-framework-client-1-7b58c77785-w86cf
fl-framework-client-1-9c8c8798-qkzg9
CPU Quota (1 panel)
Memory Usage
Memory Usage (w/o cache)
74.5 GiB
55.9 GiB
37.3 GiB
18.6 GiB
0 B
03/14 12:00
03/14 16:00
03/14 20:00
03/15 00:00
03/15 04:00
03/15 08:00
debugger-5c79dfbb57-9sm4l
fl-framework-client-1-5547665c87-blhvx
fl-framework-client-1-5547665c87-gpz4d
fl-framework-client-1-5547665c87-wmrtc
fl-framework-client-1-68bc778fc9-tk7r5
fl-framework-client-1-6946874799-tfcp4
fl-framework-client-1-7b58c77785-w86cf
fl-framework-client-1-9c8c8798-qkzg9

Source: Image provided by the author

5.3.4 Experiments Results Application

To retrieve the experiments results from the Kubernetes cluster, a container running an Ubuntu base image from Docker was used to mount the used PVs and access the results. Since the results are stored in files, we can copy the results to the local machine to read and interpret them. Figure 5.9 illustrates how the experiments that run in the PoC are saved in the distributed storage.

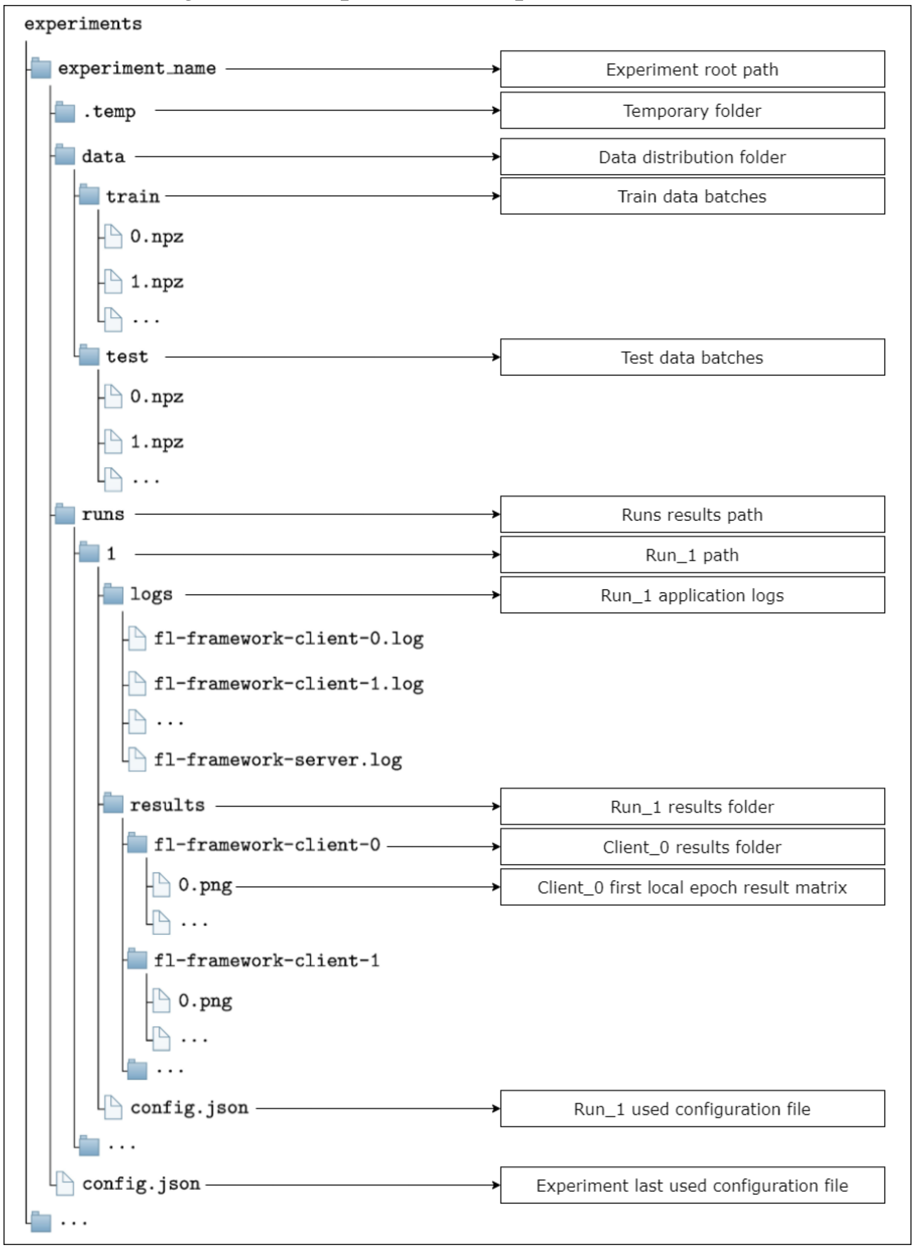
.temp contains the partial results whilst the applications is running. This directory is deleted after the server application storage manager has moved the finalized run into its correct directory.

data contains the training and testing data batches of each client used in the experiment.

runs contains the results for each run of the experiment. The logs subfolder of a run holds all the logs from the server and client applications and the results subfolder contains the evaluation matrix for each local epoch ran in the clients. Figure 5.10 illustrates how results for each round are saved in the results folder. A copy of the configuration used in the experiment run is saved to enable visualization of how the data was distributed in each client, how the classes were distributed inside each data batch and which model and global aggregation strategy was used.

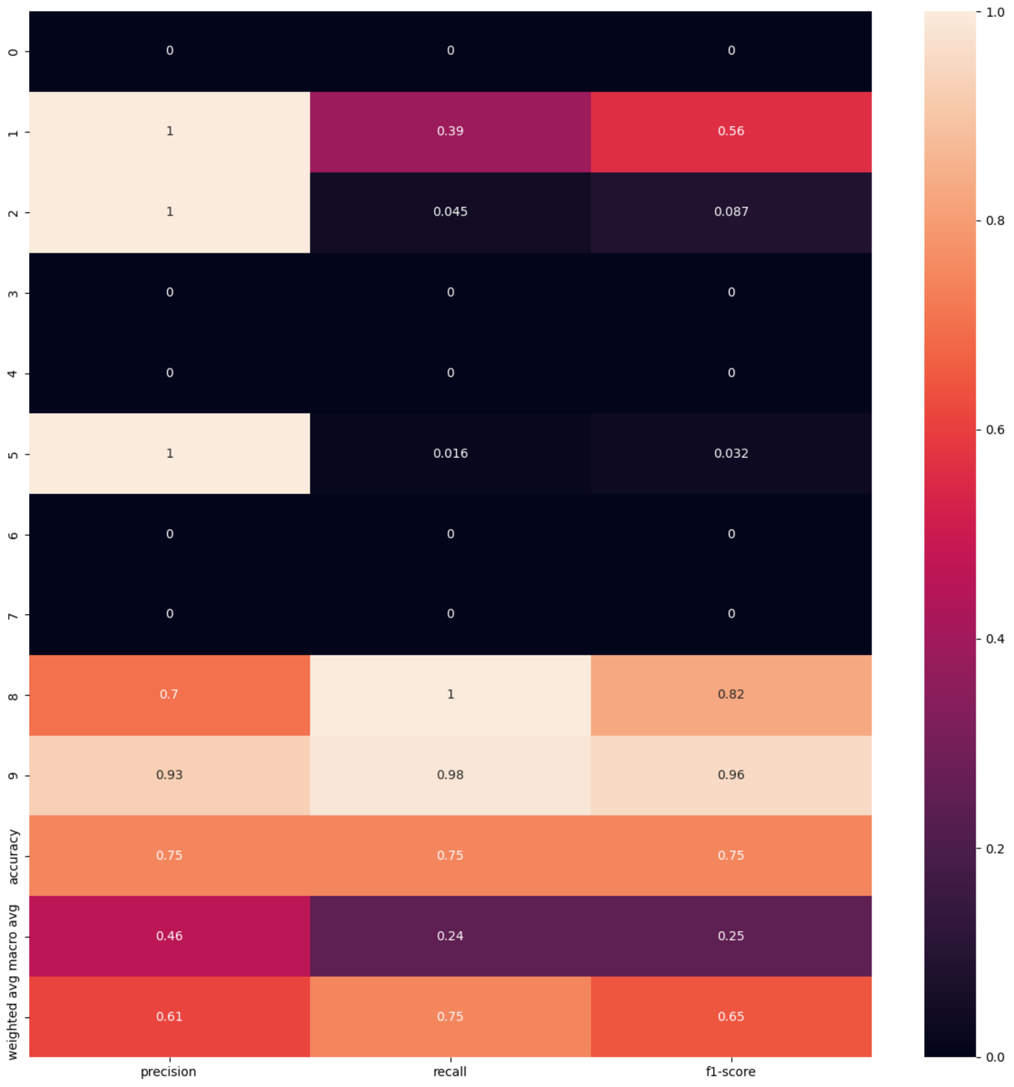
config.json contains the last configuration used in the experiment. This enables testing of a same data distribution using several different parameters, strategies and models.

Figure 5.9: Experiments output folder structure



Source: Image provided by the author

Figure 5.10: Experiment result evaluation matrix



	precision	recall	f1-score
0	0	0	0
1	1	0.39	0.56
2	1	0.045	0.087
3	0	0	0
4	0	0	0
5	1	0.016	0.032
6	0	0	0
7	0	0	0
8	0.7	1	0.82
9	0.93	0.98	0.96
accuracy	0.75	0.75	0.75
macro avg	0.46	0.24	0.25
weighted avg	0.61	0.75	0.65

Source: Image provided by the author

6 EXPERIMENTS

This chapter covers the experiments run in the PoC solution in order to demonstrate its capabilities. The first and second section will describe the experiments layout used and the third section will show the results obtained from them. Demonstration of the visualization of resource usage in the PoC solution will be presented in the last section.

6.1 Layout

The results where collected directly from the PoC solution results output folder of the last experiment run of each experiment and where aggregated into better to visualize charts since the number of output files can be exponentially big. Every experiment utilizes ten heterogeneous client configurations from the "computer" type of computational devices of the Kubernetes cluster, which means that every client involved in a given experiment might change from one run to another. This is a clear demonstration on how the framework can be useful to reproduce real-life scenarios.

It also should be noted that, while prediction performance is an extremely important metric for any machine learning algorithm, it isn't the focus of this work. The main objective of the experiments is to demonstrate the capabilities of the framework and how easy is to completely change a FL testing scenario by only changing input parameters in the deployment file. Higher prediction performance would require a much higher amount of training time, which would make having this many experiments not viable due to the time constraints of writing this article. It would also require further tweaks on each model ran, which is out of scope as far as we are concerned.

There are different parameters that can be modified in order to test the power of the framework. These are the data distribution, the dataset, the global model aggregation, the local client model, the client epochs, and the server rounds. Each experiment will run with 10 different clients. They will be named client-0, client-1, up to 9. The fraction fit will be set to 1 and the minimum available clients parameter will be set to 10. This will force the experiment to run each server round with all the clients participating.

Besides the output of the results being saved in performance score matrices, they where also written in the log files. F1 score is a useful metric for evaluating classification models that takes into account both precision and recall, and is particularly useful in situations with imbalanced data. For binary classification problems is calculated as followed:

$$f1score = 2 * (precision * recall)/(precision + recall)$$

Precision is the proportion of true positives (TP) out of all predicted positives (TP + false positives, FP). It measures how often the model correctly predicts a positive class. Recall is the proportion of true positives (TP) out of all actual positives (TP + false negatives, FN). It measures how well the model is able to detect positive classes.

The macro and weighted F1 measure are commonly used as performance metrics for evaluating the performance of a multiclass classification model. The first one is calculated by first computing the F1 score for each class in the dataset, and then taking the average of these scores across all classes. The weighted F1 measure takes into account the class distribution by computing the F1 measure for each class separately and then weighting the average F1 score by the number of instances in each class.

In total, 143 containers of a client or a server application were ran in the cluster nodes. For each one of these, a log file was generated containing stats information through each server round. Each run of the experiments generated 11 files, which sums to a total of 143 files. Since only were analyzed the last run of each experiment, 33 files were utilized to generate the results .

To manipulate all the log files containing the results, Power BI Desktop[1] was used to manipulate the data and display it into easy to visualize charts. Future development of the framework could include automatic chart generation from the performance measures generated from the experiments. The resource usage metrics where retrieved directly from the Grafana application running in the cluster. Here are the metrics analyzed:

Accuracy: it measures the accuracy of the model using a ratio of the predicted samples by the number of samples of the test dataset. The client log files contains the Micro average F1 measures.

Macro average F1: useful when the classes in the dataset are balanced or when we want to evaluate the overall performance of the model across all classes equally. The client log files contains the Macro average F1 measures.

Weighted F1 score: useful when the dataset is imbalanced and we want to give more importance to the classes with more samples. The client log files contains the Weighted F1 measures.

[1]https://powerbi.microsoft.com/pt-br/desktop/ (accessed March 28th, 2023)

Losses distributed: the aggregated loss of the clients. The server file contains the average loss of the clients.

Accuracy distributed: the average weighted accuracy of the clients. The clients accuracy are weighted by the number of samples of their dataset and then averaged to evaluate the FL accuracy overall. The server file contains the average accuracy of the clients. .

CPU percentage: current CPU percentage usage. Retrieved from the Grafana application.

Bytes received: current amount of received bytes. Retrieved from the Grafana application.

Bytes transmitted: current amount of transmitted bytes. Retrieved from the Grafana application.

6.2 Datasets

In the following we present the datasets used in the experiments.

6.2.1 CIFAR-10

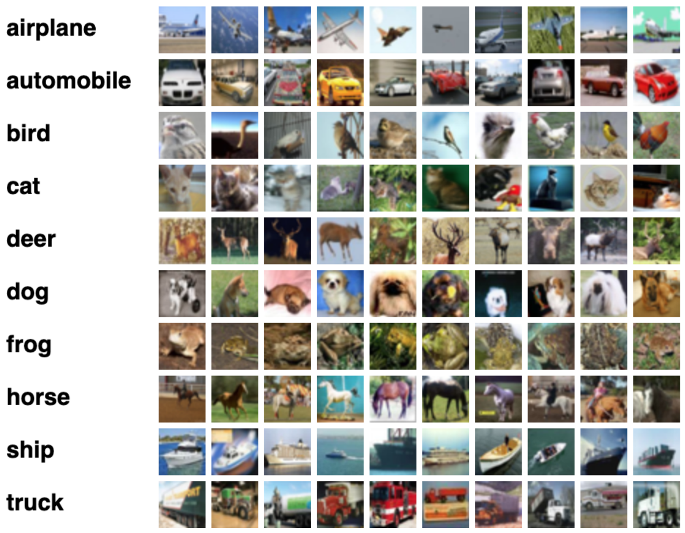
The CIFAR-10 (KRIZHEVSKY; HINTON et al., 2009) dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. A dataset sample with the classes can be seen in figure 6.1, with 10 rows of images, one row for each class as stated in the first column.

6.2.2 CIFAR-100

The CIFAR-100 (KRIZHEVSKY; HINTON et al., 2009) is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

[2]https://www.cs.toronto.edu/ kriz/cifar.html (accessed March 27th, 2023)

Figure 6.1: CIFAR-10 Dataset Sample

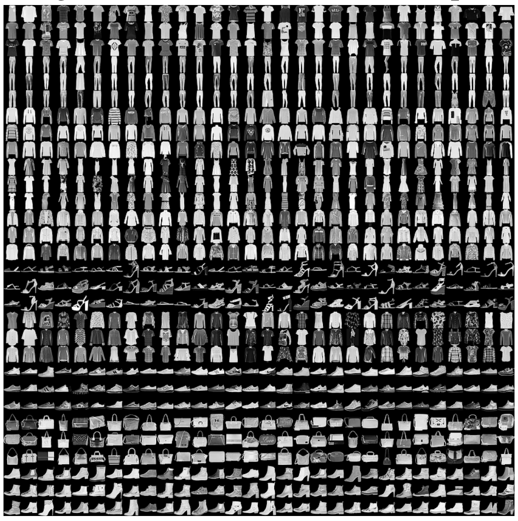


Source: University of Toronto Department of Computer Science[2]

6.2.3 FMNIST

Fashion-MNIST is a dataset of Zalando's[3]article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. We intend Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset (LECUN et al., 1998) for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Figure 6.2: FMNIST dataset sample



Source: Fashion-MNIST GitHub repository[4]

[3]https://jobs.zalando.com/ (acessed March 27th, 2023)
[4]https://github.com/zalandoresearch/fashion-mnist (accessed March 27th, 2023)

6.3 Results

6.3.1 Experiment 1

The first experiment run using the FMNIST dataset with a non-IID and unbalanced data distribution. Also, a Dirichlet distribution of the classes with $\alpha = 0.1$ was used. The global model aggregation used was FedOpt and the experiment run with ten local epochs and ten server rounds. Table 6.5 shows the parameters used in this experiment and Table 6.2 shows the unbalanced data distribution in the clients.

Table 6.1: Parameters selected for experiment 1

Parameter	**Value**
Dataset	FMNIST
Experiment Name	kubernetes-test-1
FL Strategy	FedOpt
Model	DNN
Data Balance	False
Non-IID	True
Class Distribution	Dirchlet
α	0.1
Server Rounds	10
Local Rounds	10

Source provided by the author

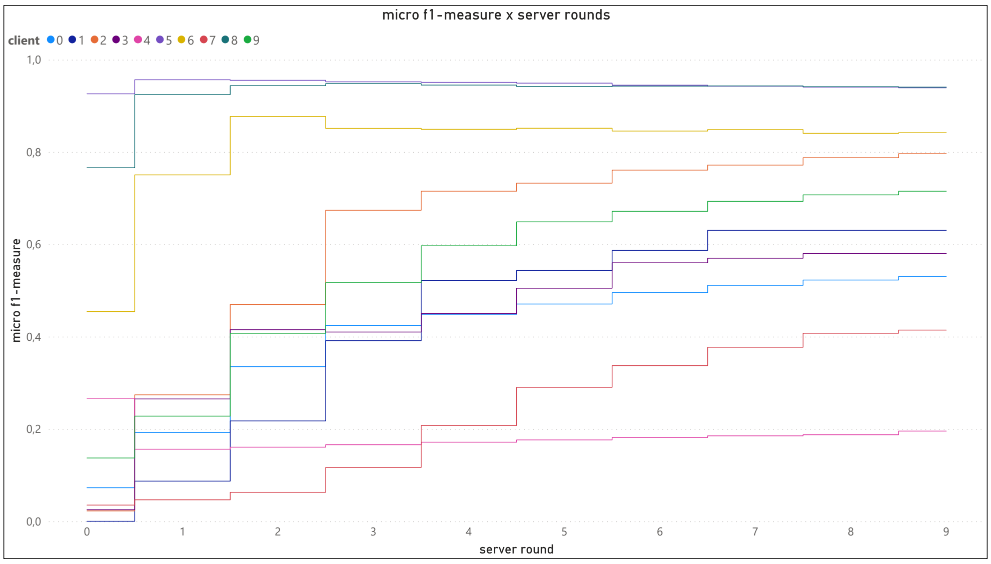
Table 6.2: Data distribution for experiment 1

Client	**Train batch size**	**Test batch size**
client-0	6345	2116
client-1	135	46
client-2	4137	1380
client-3	600	200
client-4	6109	2037
client-5	7389	2464
client-6	8968	2990
client-7	5734	1912
client-8	6665	2222
client-9	6413	2138

Source provided by the author

Accuracy assumes the same importance for all instances and basically calculates the percentage of correctly classified instances regardless of their classes. Figure 6.3 shows the accuracy for each client through the server rounds. The X axis indicates the server round number, while the Y axis indicates the accuracy. The data showed suggests that clients with large amounts of data have a relatively high initial accuracy, already in the first rounds, while the accuracy of clients with little data is initially low. Over the rounds, the chart suggests that the aggregate model has subtly negative impacts on the accuracy of clients with more data available, but that aggregation benefits the accuracy of clients with less data available.

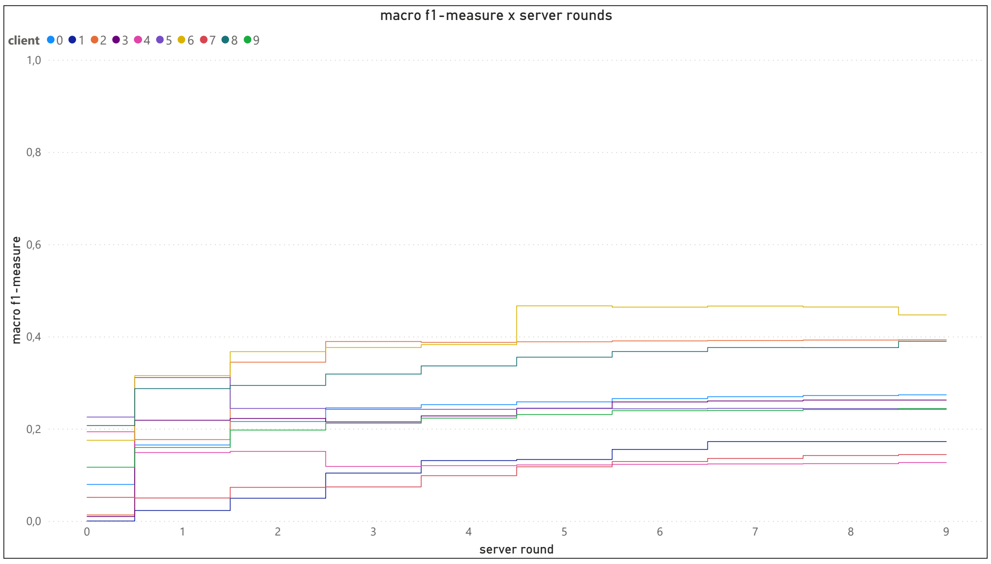
Figure 6.3: Accuracy from each client in experiment 1



Source: Image provided by the author

The f1-measure macro assigns the same importance to all classes and has better results when the performance of all classes in the dataset is better. Figure 6.4 shows the macro F1 measure for each client through the server rounds. The chart suggests that, despite the accuracy being reasonable for each client, as seen in the previous graph, when we seek to understand the average performance focusing on classes and assigning the same importance, the performance is not so good in this case. Even so, the chart shows that over the server rounds, there are performance increases, with some exceptions.

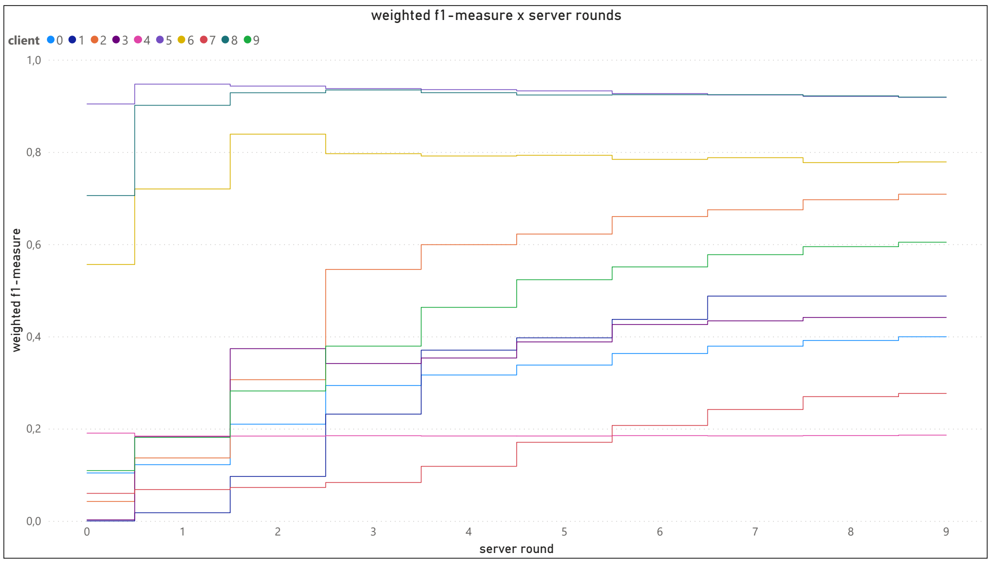
Figure 6.4: Macro f1-measure from each client in experiment 1



Source: Image provided by the author

The weighted f1-measure weights the mean by the proportion of instances in each class. Figure 6.5 shows the weighted F1 measure for each client through the server rounds. In this chart, we notice that some clients (5, 6 and 8) had a relatively high initial performance in terms of weighted f-measure and this performance decreased over time.

Figure 6.5: Weighted f1-measure from each client in experiment 1

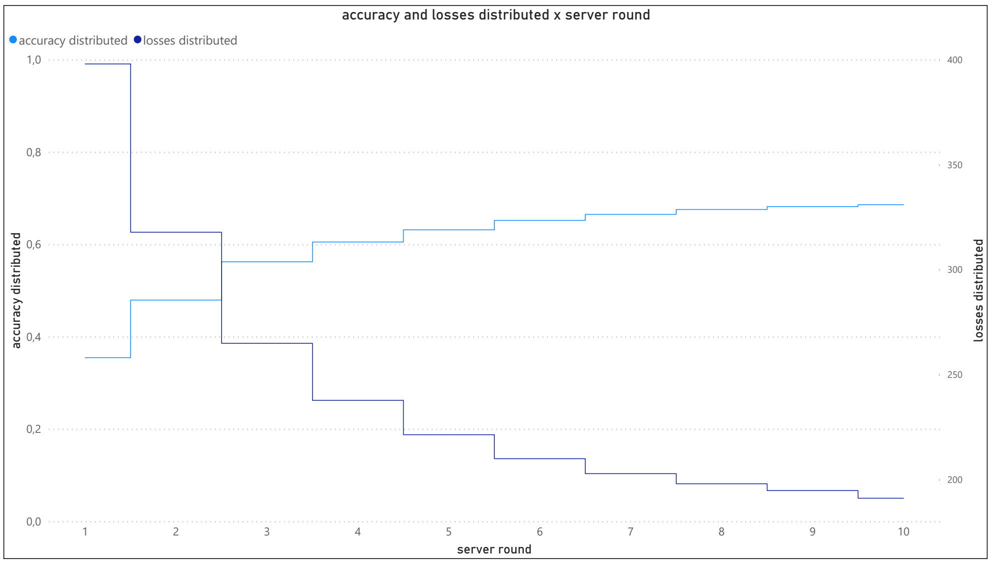


Source: Image provided by the author

Contrasting with Figure 6.4, it is noted that in these clients, the performance in the majority classes (with more instances) is high, in the initial stages, and that over the server runs, the aggregated model worsened the performance in these majority classes, but improved overall performance considering all classes. Observing client 6, for example, in Figure 6.4, it is noted that between rounds 4-5 there is a performance jump in terms of f1-macro, which suggests that the aggregated model improved the overall performance of the model, considering all classes, despite reducing performance for the majority class.

Figure 6.6 shows the aggregated accuracy over all the clients and the aggregated loss of the strategy in each server round. The X axis indicates the server round number, while the left Y axis indicates the accuracy and the second Y axis indicates the losses. It can be noted that the overall performance of the FL algorithm converged between 60-70% accuracy and less 100 loss.

Figure 6.6: accuracy and losses distributed from experiment 1



Source: Image provided by the author

Further analysis could be done using the class distribution of the dataset to better understand how it also impacted in the results. It is important to note that the framework saves this information in the configuration file of each experiment run.

6.3.2 Experiment 2

The second experiment run using the CIFAR-10 dataset with also non-IID and unbalanced data distribution. However, differently from the first experiment, a pathological distribution of the classes of two classes per client was used. The global model aggregation used was FedAvg and the experiment run with fifteen local epochs and ten server rounds. Table 6.3 shows the parameters used in the experiment and table 6.4 shows the unbalanced data distribution in the clients.

Table 6.3: Parameters selected for experiment 2

Parameter	**Value**
Dataset	CIFAR-10
Experiment Name	kubernetes-test-2
FL Strategy	FedAvg
Model	CNN
Data Balance	False
Non-IID	True
Class Distribution	Pathological
Class per Client	2
Server Rounds	10
Local Rounds	15

Source provided by the author

Table 6.4: Data distribution for experiment 2

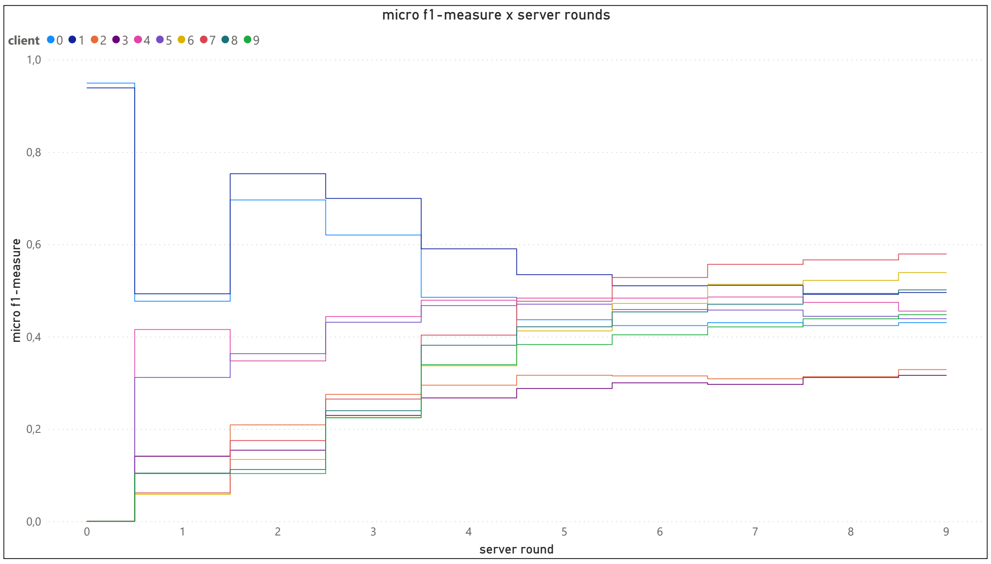
Client	**Train batch size**	**Test batch size**
client-0	1422	474
client-1	7578	2526
client-2	1950	651
client-3	7049	2350
client-4	1278	426
client-5	7722	2574
client-6	2463	822
client-7	6536	2179
client-8	2326	776
client-9	6673	2225

Source provided by the author

The analysis of the results follow the same method used in subsection 6.3.1. First is done an analysis over the accuracy of the clients followed by an analysis over the f1-macro and f1-weighted measures.

Figure 6.7 represents the accuracy of all clients across server rounds. It shows that initially some clients are able to classify a good part of the test instances, while other clients are not able to classify any instances. Over time, the performance of the clients with high accuracy decreases, while the performance of the clients with low accuracy increases, so that the overall accuracy of the clients stabilizes in a medium range, between 30-60%. That is, the aggregation is capable of distributing performance to clients with low initial performance, at the expense of reducing the performance of clients with high performance, in terms of accuracy.

Figure 6.7: Accuracy from each client in experiment 2

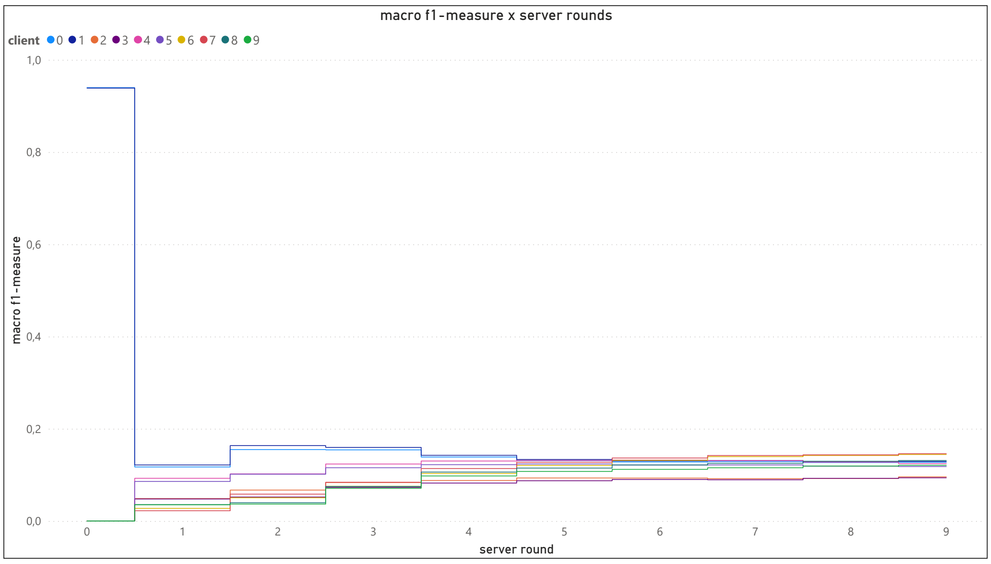


Source: Image provided by the author

Figure 6.8 represents the f1-macro measure of all clients across server rounds. It demonstrates that initially, client 8 managed to obtain a very good performance in the two classes from which it had data and that the aggregation had a very negative impact on its performance, drastically reducing it. A similar behavior occurred with client 0. On the other hand, aggregation had a positive effect on clients that initially performed quite poorly. Note that the general performance of all clients, considering all their classes as important, ends up being very low.

Figure 6.9 represents the f1-weighted measure across server rounds. It suggests more erratic behavior over time. It is noted that clients 0 and 8, which had an initial negative impact on performance considering all classes (Figure 6.7), had a different behavior when we consider the importance of the class proportional to its number of instances. It is noted that the performance of these clients for their majority classes dropped after round

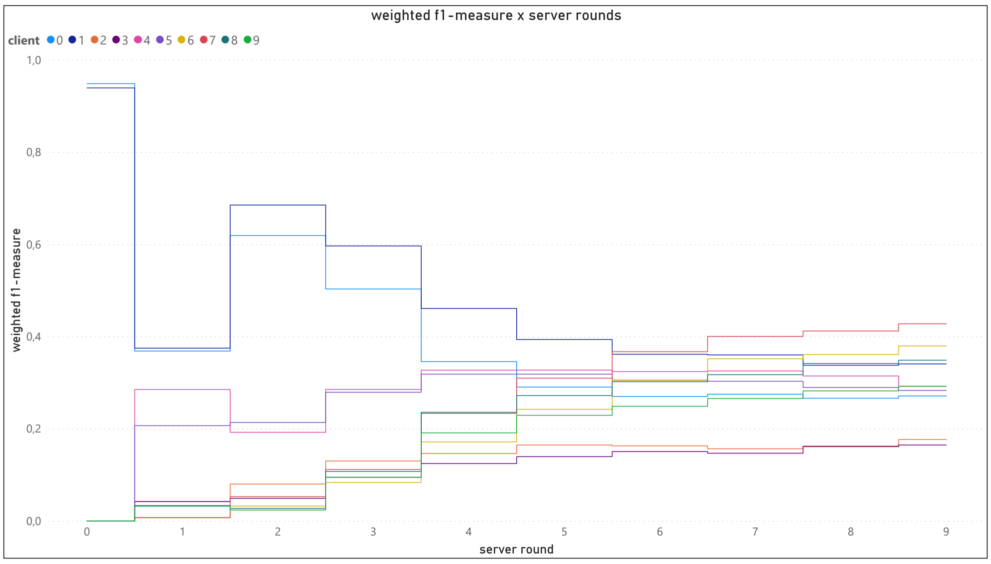
Figure 6.8: Macro f1-measure from each client in experiment 2



Source: Image provided by the author

0 to less than 40%, but increased again after round 1 to 60-70% and then dropped again throughout the rounds. Deeper analyzes must be carried out to investigate this behavior. It is also noted that, on the other hand, the performance of customers who had poor performance at the beginning had performance increases over the rounds. In general, the average performance of all clients has stabilized in the 30-40% range.

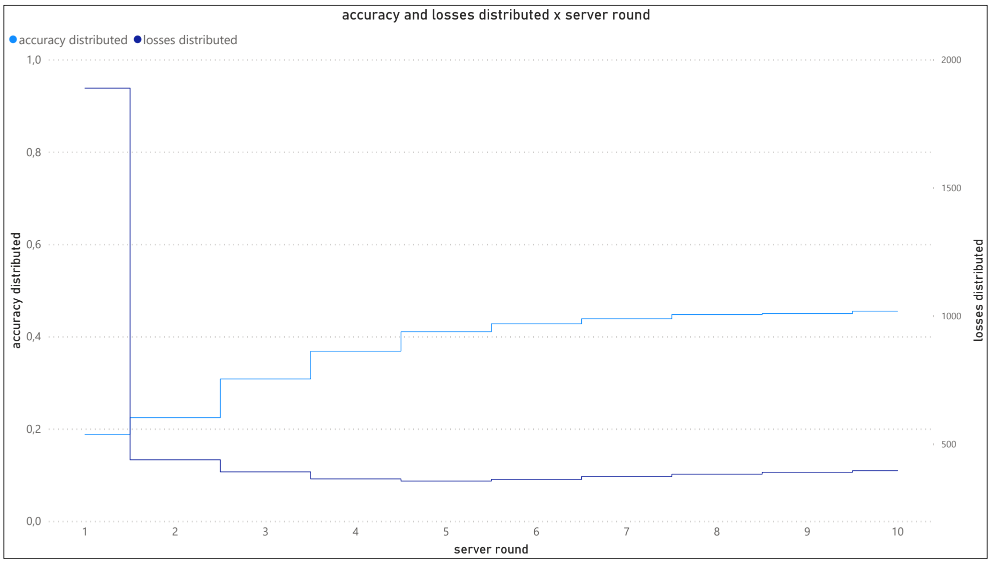
Figure 6.9: Weighted f1-measure from each client in experiment 2



Source: Image provided by the author

Figure 6.10 shows the weighted accuracy over all the clients and the aggregated loss of the strategy in each server round. The X axis indicates the server round number, while the left Y axis indicates the accuracy and the second Y axis indicates the losses. It can be noted that the overall performance of the FL algorithm converged between 40-50% accuracy and closer to 800 of loss.

Figure 6.10: Server accuracy and losses from experiment 2



Source: Image provided by the author

6.3.3 Experiment 3

The last experiment run using the CIFAR-100 dataset with also non-IID but a balanced data distribution. Similar to the second experiment, a pathological distribution of the classes of twenty classes per client was used. The global model aggregation used was FedYogi and the experiment run with ten local epochs and fifteen server rounds. Table 6.3 shows the parameters used in the experiment and table 6.4 shows the balanced data distribution in the clients.

Table 6.5: Parameters selected for experiment 3

Parameter	**Value**
Dataset	CIFAR-100
Experiment Name	kubernetes-test-3
FL Strategy	FedYogi
Model	resnet
Data Balance	True
Non-IID	True
Class Distribution	Pathological
Class per Client	20
Server Rounds	15
Local Rounds	10

Source provided by the author

Table 6.6: Data distribution for experiment 3

Client	**Train batch size**	**Test batch size**
client-0	4500	1500
client-1	4500	1500
client-2	4500	1500
client-3	4500	1500
client-4	4500	1500
client-5	4500	1500
client-6	4500	1500
client-7	4500	1500
client-8	4500	1500
client-9	4500	1500

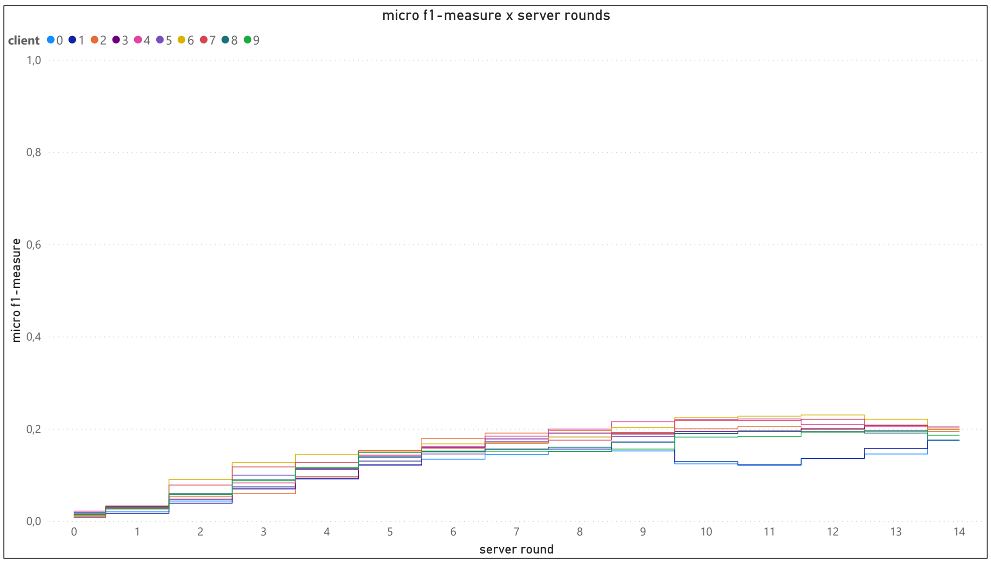
Source provided by the author

The experiment deals with a classification problem that is fairly difficult in itself because there are more classes to predict and originally, there are relatively few instances per class available. Furthermore, in the distributed context, each client has access to only

20% of the classes and less than half of the original dataset samples per class. Due to this, we expect a poor performance in this scenario.

The chart in Figure 6.11 shows that the proportion of predicted instances generally increases for all clients, with some exceptions (evident between rounds 9 and 13), demonstrating that, in general, aggregation has beneficial effects for clients over time. rounds.

Figure 6.11: Accuracy from each client in experiment 3

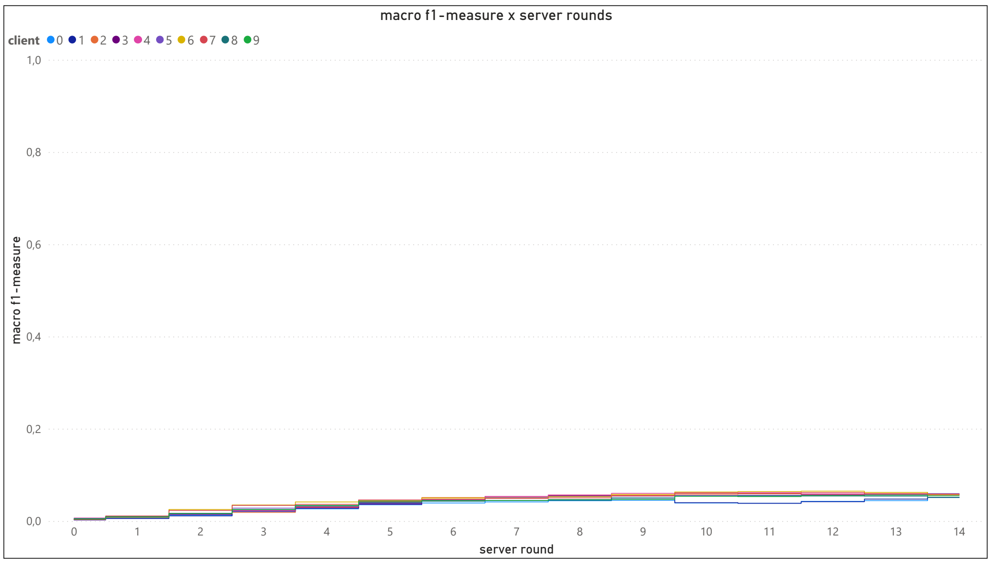


Source: Image provided by the author

Figure 6.12 shows the f1-macro measure. Thus, considering the performance for all classes and those with equal importance, the general performance of all clients is poor, not even reaching 10% of macro f-measure. Although it is possible to verify the general beneficial effects of aggregation throughout the rounds, with some exceptions.

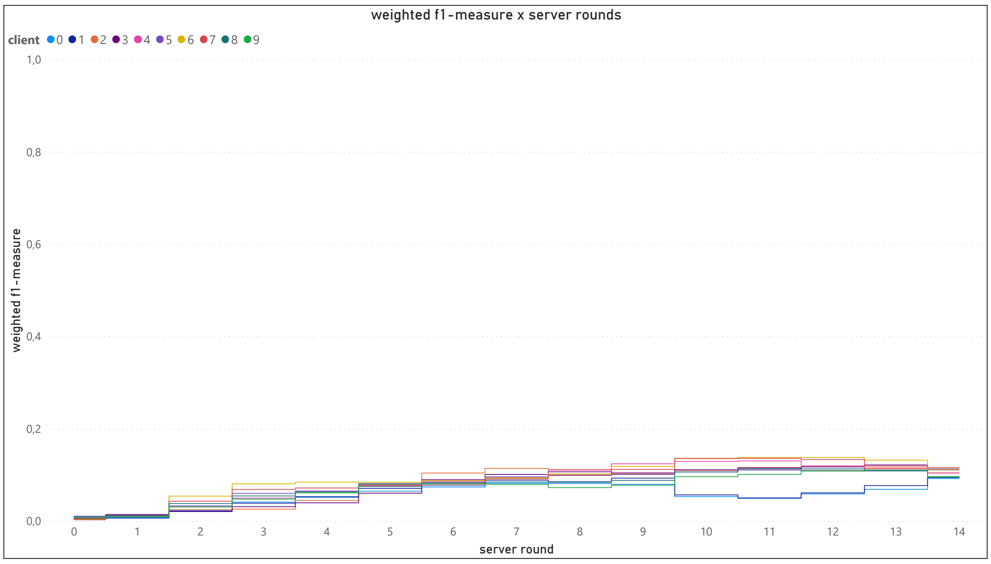
Figure 6.13, with the f1-weighted measure, shows a scenario similar to that seen in charts 6.9 and 6.10. That is, in general, the performance of most clients improves with aggregation, over the rounds, with some exceptions. However, in this scenario, the classes that are represented in each client gain more importance, which is responsible for increasing the performance in this metric, in comparison with the macro average.

Figure 6.12: Macro f1-measure from each client in experiment 3



Source: Image provided by the author

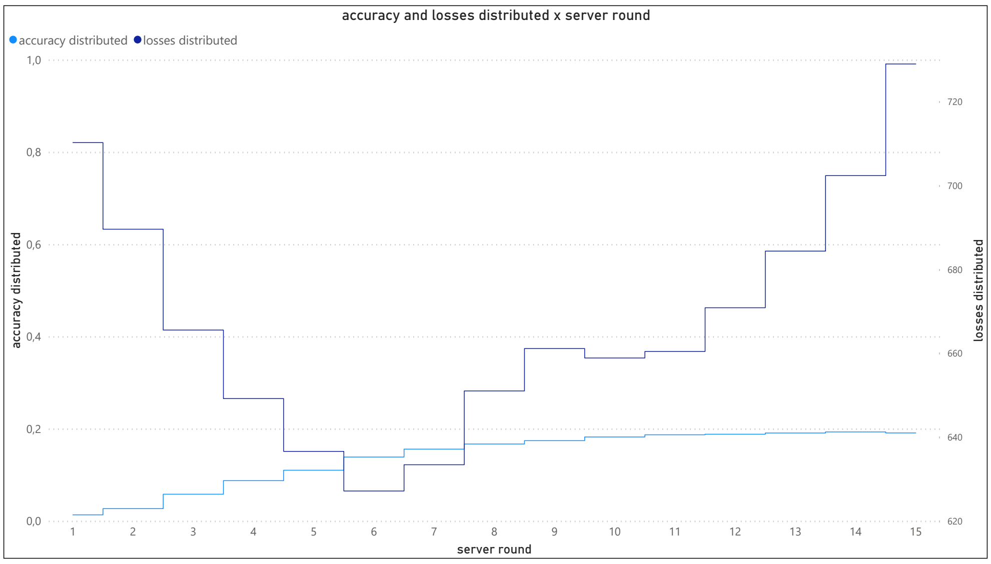
Figure 6.13: Weighted f1-measure from each client in experiment 3



Source: Image provided by the author

Figure 6.14 shows the aggregated accuracy over all the clients and the aggregated loss of the strategy in each server round. The X axis indicates the server round number, while the left Y axis indicates the accuracy and the second Y axis indicates the losses. It can be noted that the overall performance of the FL algorithm converged between 40-45% accuracy and closer to 800 of loss. It it curious how the loss of the strategy decreased in the beginning and afterwards increased to a higher level than it began. It reflects the overall poor performance of the clients.

Figure 6.14: Server accuracy and losses from experiment 3



Source: Image provided by the author

6.4 Resource Usage

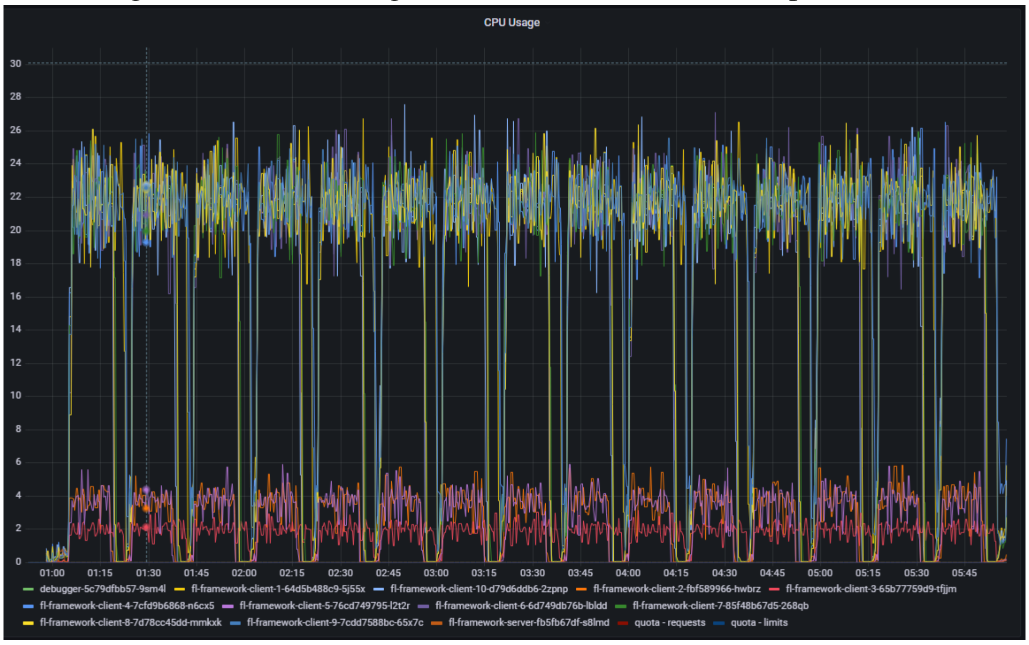
To monitor the resource usage of the experiments, the PoC solution uses the Grafana application of the cluster to easily visualize the data collected by the Prometheus monitoring.

To demonstrate some of the capabilities of the PoC solution on monitoring, an analysis of CPU usage and network traffic was made in Experiment 3 (subsection 6.3.3). It is important, once again, to emphasize the focus on the demonstration of the capabilities and not on the performance. Similar analysis can be done to all the experiments ran in the PoC solution since all the metrics are saved in the Prometheus server database.

6.4.1 CPU

Figure 6.15 shows the CPU resource usage of each client and the server of the third experiment. The X axis indicates the timestamp of the metric and the Y axis the CPU usage in percentage and each data sample has a 10 second interval between one another.

Figure 6.15: CPU Usage from server and clients in experiment 3

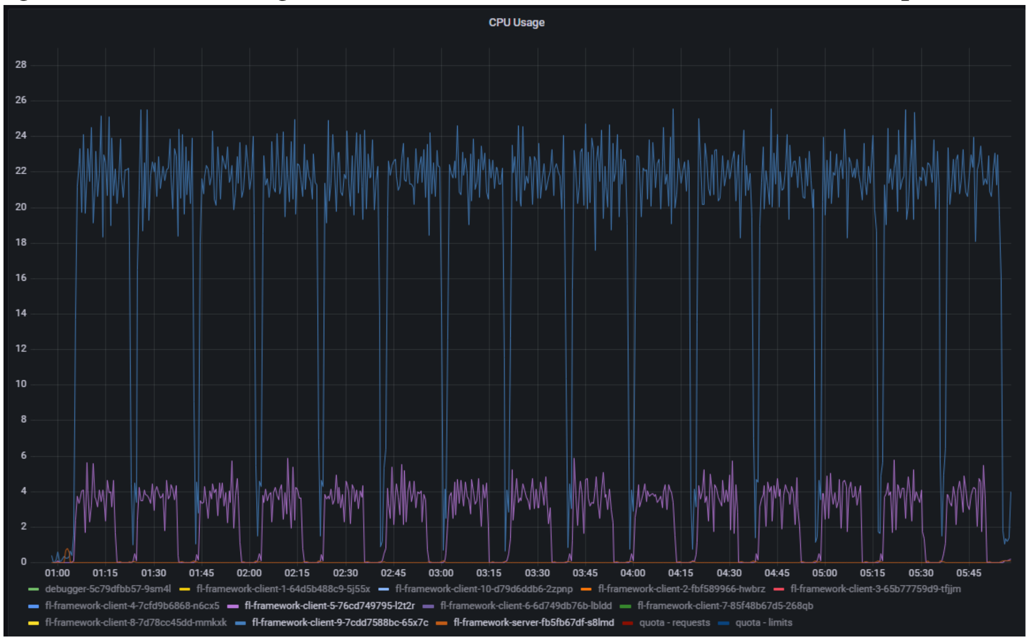


Source: Grafana dashboard of the Kubernetes cluster

We can notice that the figure is quite chaotic, since all the ten clients and the server are plotted in it. However, we can definitely see the execution of each one of the fifteen server round ran in the experiment and also that some of the clients had a CPU usage of 20-30% while others stayed between 1-5%. This can easily be explained by the heterogeneity of the clients CPU processing power, since some clients run in better hardware than others. We can notice also that workload was distributed evenly across the clients, since the data distribution used was balanced in this case.

In Grafana, we can select which application we want to plot in the chart. So, to better analyze the assumption about the first chart, we can select some clients of a specific hardware and other of another specification to compare them. Figure 6.16 demonstrates the same chart with only client-5, client-4 and the server CPU usage being plotted. It becomes evident that client-4 (an Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz) had more resource usage and needed more time to process the same number of samples then client-5 (an Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz).

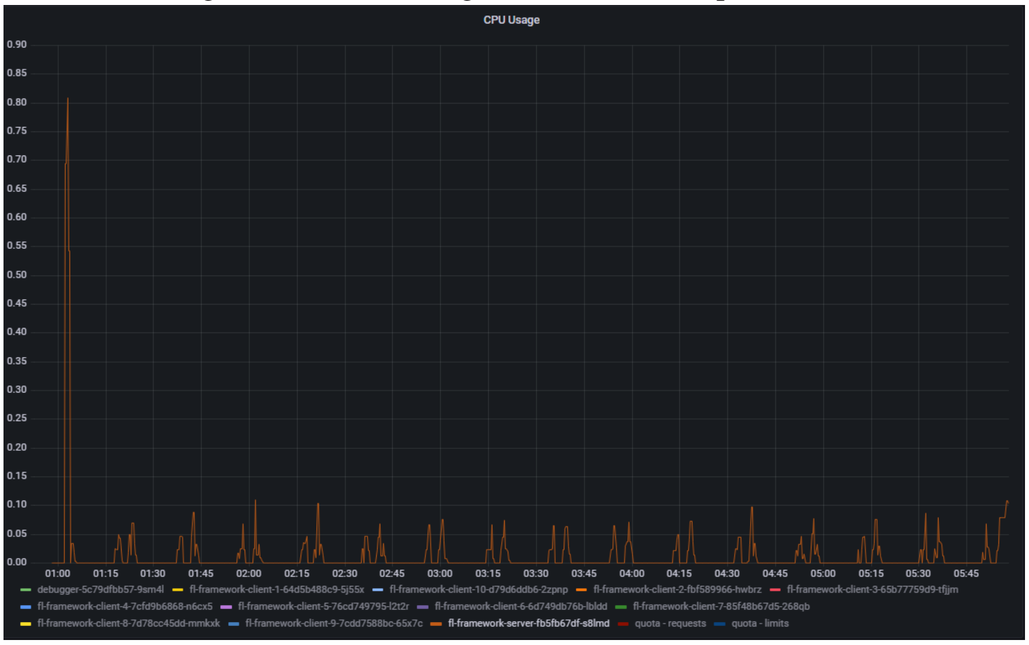
Figure 6.16: CPU Usage from two clients with different hardware in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

We can also select only the server in the chart to understand how was it behavior throughout the experiment. Figure 6.17 demonstrates the same chart with only the server. It noticeable the higher usage of CPU in the beginning of the experiment.

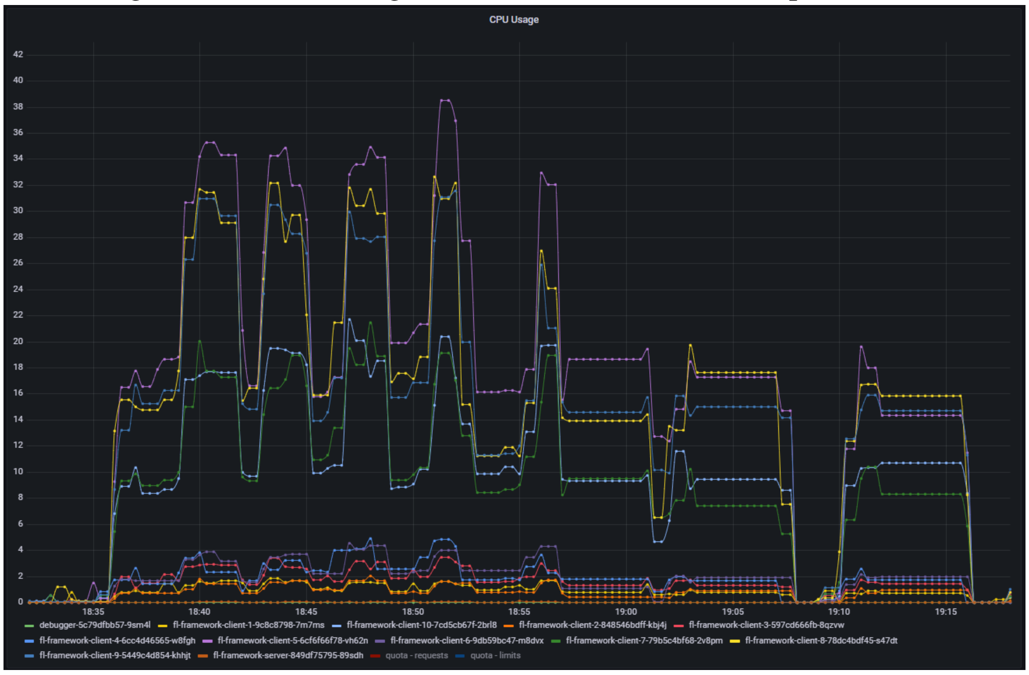
Figure 6.17: CPU Usage from server in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

This can be explained by the fact that the data distribution created by the server is done at the beginning. We can also see that the usage of CPU resource of the server is minimal if compared with any client. The server only uses CPU in the middle of training to aggregate the parameters received from the clients which is a quite simple task if compared with the training done by the clients, thus, explaining this phenomenon.

Figure 6.18: CPU Usage from server and clients in experiment 1



Source: Grafana dashboard of the Kubernetes cluster

To demonstrate that all of the analysis done in experiment 3 could be done to other experiments, we can demonstrate a sample from experiment 1. Figure 6.18 shows the same chart of figure 6.15 plotted with data of the first experiment.

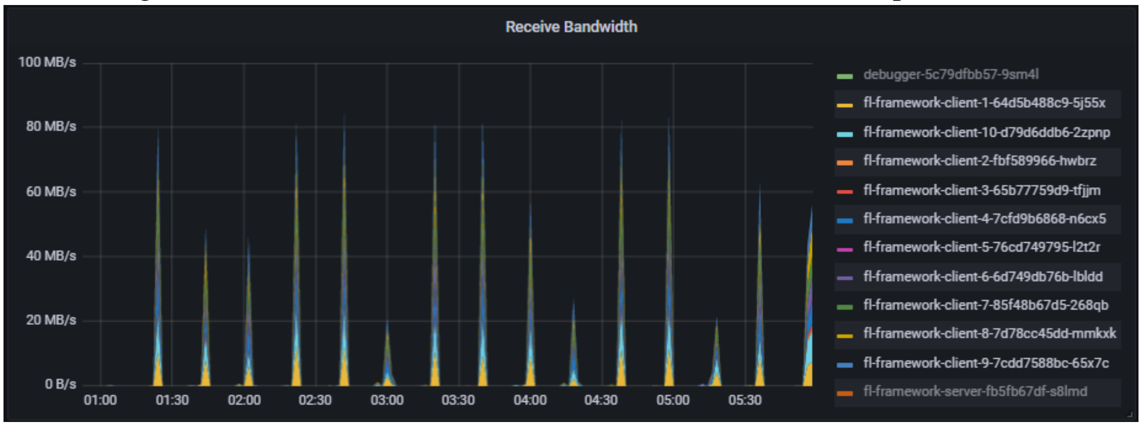
It is curious how the data distribution affects the CPU usage of the clients. We can see that in between all of the them we have gaps due to the size of the samples being different for each one of them, even with some of them running in similar hardware. Further analysis can be done in the Grafana dashboard to understand the behavior of each one of the clients individually and in any time frame specified.

6.4.2 Network traffic

Similar to the charts shown in the previous subsection, the same can be done to visualize the network traffic between the clients and the server. Grafana can enable global filtering across charts in a dashboard. The same time frame used in Figure 6.15, 6.17 and 6.16 was used to demonstrate this capability.

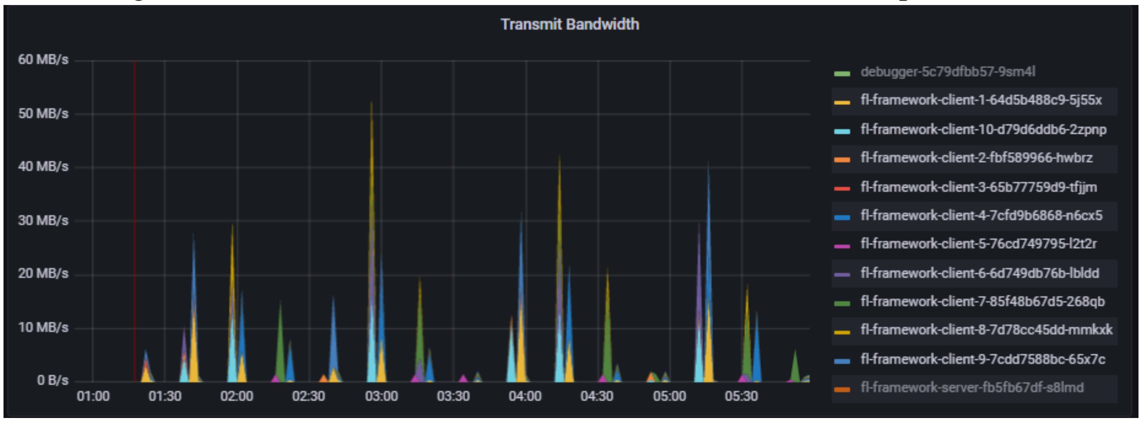
Figure 6.19 shows the received traffic and figure 6.20 the transmitted traffic for each one of the clients and the server stacked. The X axis indicates the timestamp of the metric and the Y axis the network traffic in Megabytes per seconds (MB
s). Each data sample also has a 10 second interval between one another. The first aspect to notice is that we have peaks of traffic in the beginning and end of each round which is expected, since is where communication between client and server should occur. The aggregated traffic peak was of around 50 MB
s. To better understand the behavior of client and server, we can filter one client and the server in the same charts.

Figure 6.19: Receive bandwidth from server and clients in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

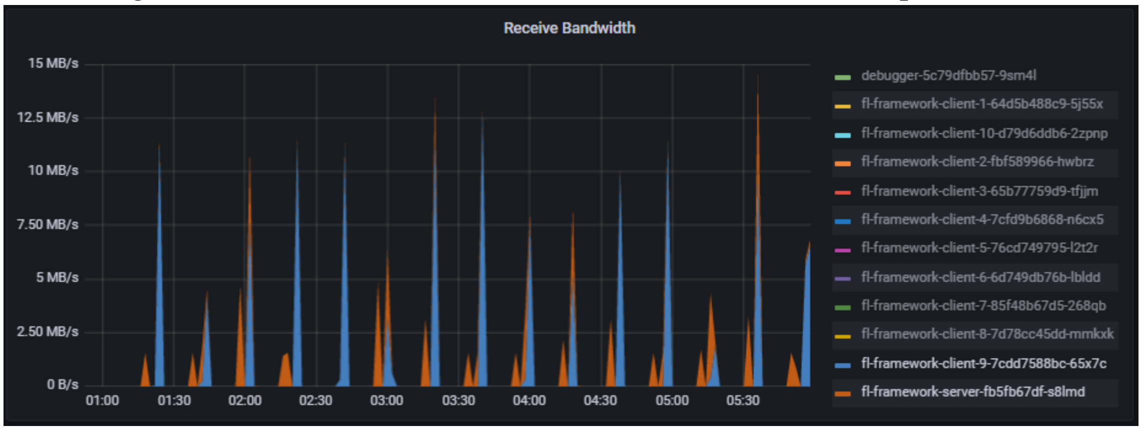
Figure 6.20: Transmit bandwidth from server and clients in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

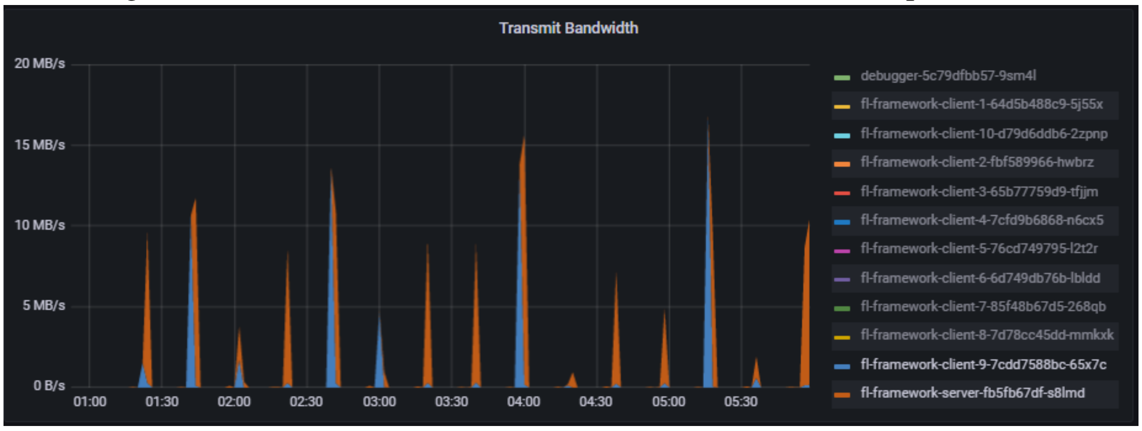
Figure 6.21 shows the traffic received and 6.22 the transmitted in the server and in client-9. We can see that the pattern of communication is always firstly the server receiving data and afterwards the client. This is expected, since initially the clients connect to the server and the server accepts the connection. During training, the pattern is maintained since the server receive the parameters from the clients and afterwards the client receives the aggregated parameters from the server.

Figure 6.21: Receive bandwidth from server and client-9 in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

Figure 6.22: Transmit bandwidth from server and client-9 in experiment 3



Source: Grafana dashboard of the Kubernetes cluster

Many other metrics are collected by the Prometheus monitoring application such as, network packets, memory usage, or any other custom metric from applications that are configured in the system. For the purposes of demonstration, in this work we considered only the CPU and the network due to their relevance in FL.

7 CONCLUSION

FL is a ML paradigm that is constantly being adopted due to its distributed approach, the necessity of data privacy and the vast increase of IoT devices. Heterogeneous data distributions/contents of clients showed to be a real challenge and gained great notoriety over the last years. Proposals, such as DRL algorithms to dynamically learn the weight of the contributions of each client at each round, continue to be developed and will be fundamental to increase accuracy and usability of FL in more applications. Testing and assessing this FL algorithms can be a very difficult and complex task due to the distributed nature of the systems and the several types of scenarios of imbalance possible.

To address the testing of these algorithms, this worked proposed a conceptual framework to facilitate testing federated learning scenarios in distributed computing environments with different types of data distributions. This work achieves the intended proposal by proposing a conceptual framework for testing federated learning scenarios and demonstrating an implementation of those concepts in the PoC solution. The solution developed shows that creating an edge-like FL testing framework that can scale to several different types of real-life scenarios using distributed heterogeneous computing and different data distributions easily is possible, inspiring further development of the concepts and also improvement of the PoC solution itself.

To prove the capabilities of the PoC solution, three experiments with three different scenarios of FL where conducted. The results showed how is possible to analyze the impacts of class and data imbalance in a real-life distributed system of FL through the framework via the f1-measures outputted in the experiments results. It was also possible to see the resource usage of the applications via the monitoring solution and to demonstrate the impact of the underlying heterogeneous infrastructure used.

The key point taken from this work is that designing a solution from the beginning that has independence between infrastructure and applications showed to be very efficient and effective during the development phase. Containers enabled re-usability, isolation and easy testing of the applications despite the environment that they were deployed, either locally or in the distributed cluster. Also, this allowed the system for horizontal scaling by adding more nodes to the cluster from a infrastructure perspective or by creating more application replicas from an application point-of-view.

Further improvement could be done to generate automated visualizations of the aggregated results (section 6.3) of each client and the server. For instance, it should be possible to output the results of the FL algorithms performance to the Prometheus server to be further visualized by Grafana or another visualization tool that has access to its database.

This work also opens new opportunities to develop new experiments with federated learning scenarios. Fault tolerance experiments could also be done in the PoC solution to enable further improvement in the reliability of the framework. From an application point-of-view, one way would be to test limited connectivity scenarios by disconnecting clients between training. Stress testing from a infrastructure point-of-view would also be interesting to understand the limitations of the platform.

All of the code is publicly available and has been developed with extensibility in mind. Future works could also extend the datasets, models, FL strategies supported and better improve the applications as a whole.

REFERENCES

BEUTEL, D. J. et al. **Flower: A Friendly Federated Learning Research Framework**. 2022.

CHENG, X. et al. A class-imbalanced heterogeneous federated learning model for detecting icing on wind turbine blades. **IEEE Transactions on Industrial Informatics**, p. 1–1, 2022.

DARLING, G. **IoT vs Edge Computing**. https://developer.ibm.com/articles/iot-vs-edge-computing/ (accessed September 9th, 2022), 2022.

DARLING, G. **What is edge computing?** https://developer.ibm.com/articles/what-is-edge-computing/ (accessed August 28th, 2022), 2022.

DUAN, M. et al. Self-balancing federated learning with global imbalanced data in mobile systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 32, n. 1, p. 59–71, 2021.

GIUST, F.; COSTA-PEREZ, X.; REZNIK, A. Multi-access edge computing: An overview of etsi mec isg. **IEEE 5G Tech Focus**, v. 1, n. 4, 2017.

GOOGLE. **Federated Learning: Collaborative Machine Learning without Centralized Training Data**. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html (accessed November 2nd, 2021), 2017.

GUO, X. W. E.; WU, W. Adaptive aggregation weight assignment for federated learning: A deep reinforcement learning. **ACM**, p. 1–3, 2022.

HARD, A. et al. Federated learning for mobile keyboard prediction. **arXiv preprint arXiv:1811.03604**, 2018.

HE, K. et al. **Deep Residual Learning for Image Recognition**. 2015.

HELLSTRöM, H. et al. **Wireless for Machine Learning**. 2020.

HSU, T.-M. H.; QI, H.; BROWN, M. **Measuring the Effects of Non-Identical Data Distribution for Federated Visual Classification**. 2019.

HUANG, C. et al. **DearFSAC: An Approach to Optimizing Unreliable Federated Learning via Deep Reinforcement Learning**. 2022.

KONEčNý, J. et al. **Federated Optimization: Distributed Machine Learning for On-Device Intelligence**. [S.l.], 2016.

KRIZHEVSKY, A.; HINTON, G. et al. Learning multiple layers of features from tiny images. Citeseer, 2009.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, 1998.

LIU, H. et al. Improved deep reinforcement learning with expert demonstrations for urban autonomous driving. In: **2022 IEEE Intelligent Vehicles Symposium (IV)**. [S.l.: s.n.], 2022. p. 921–928.

LIU, S. et al. Edge computing for autonomous driving: Opportunities and challenges. **Proceedings of the IEEE**, v. 107, n. 8, p. 1697–1716, 2019.

MCMAHAN, H. B. et al. **Communication-Efficient Learning of Deep Networks from Decentralized Data**. 2017.

NILSSON, A. et al. A performance evaluation of federated learning algorithms. **DIDL '18: Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning**, p. 1–8, 2018.

PLAAT, A. **Deep Reinforcement Learning**. Springer Nature Singapore, 2022. Available from Internet: <https://doi.org/10.1007%2F978-981-19-0638-1>.

REDDI, S. et al. **Adaptive Federated Optimization**. 2021.

RESEARCH, J. **IOT THE INTERNET OF TRANSFORMATION 2020**. https://www.juniperresearch.com/whitepapers/iot-the-internet-of-transformation-2020 (accessed September 9th, 2022), 2020.

RUDIN, N. et al. Learning to walk in minutes using massively parallel deep reinforcement learning. In: FAUST, A.; HSU, D.; NEUMANN, G. (Ed.). **Proceedings of the 5th Conference on Robot Learning**. PMLR, 2022. (Proceedings of Machine Learning Research, v. 164), p. 91–100. Available from Internet: <https://proceedings.mlr.press/v164/rudin22a.html>.

SHAHEEN, M. et al. Applications of federated learning; taxonomy, challenges, and research trends. **Electronics**, v. 11, n. 4, 2022. ISSN 2079-9292. Available from Internet: <https://www.mdpi.com/2079-9292/11/4/670>.

SHI, W. et al. Edge computing: Vision and challenges. **IEEE INTERNET OF THINGS JOURNAL**, v. 3, n. 5, p. 637, 2016.

SRIDHARAN, B. et al. Deep reinforcement learning for molecular inverse problem of nuclear magnetic resonance spectra to molecular structure. **The Journal of Physical Chemistry Letters**, v. 13, n. 22, p. 4924–4933, 2022. PMID: 35635003. Available from Internet: <https://doi.org/10.1021/acs.jpclett.2c00624>.

SUN, Y. et al. **A Fair Federated Learning Framework With Reinforcement Learning**. 2022.

SZEGEDY, C. et al. **Going Deeper with Convolutions**. 2014.

WAN, S.; DING, S.; CHEN, C. Edge computing enabled video segmentation for real-time traffic monitoring in internet of vehicles. **Pattern Recognition**, v. 121, p. 108146, 2022. ISSN 0031-3203. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0031320321003332>.

WANG, X. et al. Scc: an efficient deep reinforcement learning agent mastering the game of starcraft ii. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2021. p. 10905–10915.

YUROCHKIN, M. et al. Bayesian nonparametric federated learning of neural networks. In: CHAUDHURI, K.; SALAKHUTDINOV, R. (Ed.). **Proceedings of the 36th International Conference on Machine Learning**. PMLR, 2019. (Proceedings of Machine Learning Research, v. 97), p. 7252–7261. Available from Internet: <https://proceedings.mlr.press/v97/yurochkin19a.html>.

ZHENG, J. et al. **Exploring Deep Reinforcement Learning-Assisted Federated Learning for Online Resource Allocation in Privacy-Persevering EdgeIoT**. 2022.