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

FERNANDO FERREIRA REMDE

Analyzing Federated Learning Performance in Distributed Edge Scenarios

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

Advisor: Prof. Dr. Juliano Wickboldt

Porto Alegre
May 2022

“Achievement is its own reward.”

— DAVID LYNCH

ACKNOWLEDGEMENTS

First of all, to the government and the Brazilian people. I have been benefiting from free education of the highest quality for over ten years now – ever since high school. This is something I will never take for granted and will always fight for as a basic right. Education changes lives. It certainly changed mine.

To my advisor, Juliano Wickboldt, for the guidance throughout this work. Surely the best advisor I could ask for; always open to discussions, always able to provide key feedback and suggestions, super knowledgeable and insightful. I started this journey knowing nothing or very little about federated learning or edge computing, or even about academic research or machine learning, and I was able to accomplish something I can be proud of, much thanks to him.

To my sisters, Monica and Miriam, to my brother Marcos, and the *aggregates*: husbands, wife, kids. Thank you for being my support network. I know you are proud of me as I am proud of you. I feel very privileged to have you as my family, and I'm happy that the family keeps growing.

To my lovely and wonderful girlfriend, Lia, who supported me from the beginning. Thanks for putting up with me. I know sometimes it's not easy, especially this last month, but I feel grateful to always be able to count on you, and to have you always supporting me no matter the situation or what I choose to do.

To all my friends, all the professors, all my colleagues, everyone that is part of UFRGS in any level: thank you for making the university what it is. I spent some of the best years of my life there, and I will forever cherish those moments. The university was not only the basis for my technical knowledge, it molded me as a person.

Above everything else, thank you to my father, Luiz Fernando, and thank you to my mother, Sofia, for giving me everything I needed (and much more) so I could pursue my dreams. Everything I am or will ever be is because of you.

AGRADECIMENTOS

Primeiramente, ao governo e ao povo Brasileiro. Eu venho usufruindo de educação de graça da mais alta qualidade por mais de 10 anos – desde o ensino médio. Isso é algo que eu não tomo por garantido e vou para sempre defender como um direito básico. Educação muda vidas. Certamente mudou a minha.

Para meu orientador, Juliano Wickboldt, pela orientação nesse trabalho. Certamente o melhor orientador que eu poderia desejar; sempre aberto a discussões, sempre capaz de fornecer feedbacks e sugestões chave, super inteligente e perspicaz. Eu comecei nessa jornada sabendo pouco ou quase nada de aprendizagem federada e computação de borda, e até mesmo de pesquisa acadêmica e aprendizado de máquina, e fui capaz de chegar em um trabalho do qual eu posso me orgulhar. Muito graças a ele.

Para minhas irmãs, Mônica e Míriam, para o meu irmão Marcos, para seus filhos, meus sobrinhos, e para os *agregados*: maridos, esposa. Muito obrigado por serem minha rede de apoio. Eu sei que vocês estão orgulhosos de mim, assim como eu sempre fico de vocês. Me sinto muito privilegiado por ter vocês como minha família, e fico ainda mais feliz de saber que a família continua crescendo.

Para a minha amável e maravilhosa namorada, Lia, por me apoiar desde o início. Obrigado por me aguentar. Eu sei que às vezes não é muito facil, principalmente nesse último mês, mas eu sou muito grato por sempre poder contar contigo, e por te ter me apoiando não importa a situação ou o que eu decida fazer.

Para todos meus amigos, todos os professores, todos meus colegas, todos os servidores, todos que são parte da UFRGS em qualquer nível: obrigado por fazerem da universidade o que ela é. Esses anos de faculdade foram alguns dos melhores da minha vida, e eu vou pra sempre celebrar esses momentos e carregar com orgulho a bandeira da UFRGS onde quer que eu vá. A universidade foi não só a base do meu conhecimento técnico, ela me moldou como pessoa.

Acima de tudo, obrigado ao meu pai, Luiz Fernando, e obrigado a minha mãe, Sofia, por terem me dado tudo que eu precisava (e muito mais) para que eu pudesse ir atrás dos meus sonhos. Tudo que eu sou e serei é por causa de vocês.

ABSTRACT

Federated learning is a machine learning paradigm where many clients cooperatively train a single centralized model while keeping their data private and decentralized. This novel paradigm imposes many challenges, such as dealing with data that is not independent and identically distributed, spread among multiple clients that are not synchronized and may have limited computing power. These clients are often edge devices such as smartphones and sensors, which form a system that is heterogeneous, highly distributed by nature and difficult to manage. This work proposes an architecture for running federated learning experiments in a distributed edge-like environment. Based on this architecture, a set of experiments are conducted to analyze how the overall system performance is affected by different configuration parameters and varied number of connected clients.

Keywords: Federated Learning. Edge Computing. Observability. Microservices. Performance.

Analisando o Desempenho de Aprendizagem Federada Para Cenários Distribuídos de Computação de Borda

RESUMO

Aprendizagem federada é um paradigma de aprendizagem de máquina onde diversos clientes treinam um único modelo de forma cooperativa enquanto mantêm seus dados privados e decentralizados. Esse paradigma inovador impõe muitos desafios, como lidar com dados que não são independentes e igualmente distribuídos, divididos entre clientes que não estão sincronizados e possuem poder de computação limitado. Esses clientes normalmente são dispositivos de borda, como celulares e sensores, que formam um sistema que é heteorgêneo, altamente distribuído por natureza e de difícil administração. Este trabalho propõe uma arquitetura para rodar experimentos de aprendizagem federada em um ambiente distribuído com poder de computação limitado. Baseado nessa arquitetura, uma série de experimentos são conduzidos para analisar como o desempenho do sistema é afetado pelas diferentes de parâmetros e pelo variado número de clientes.

Palavras-chave: Aprendizagem Federada, Computação de Borda, Observabilidade, Microsserviços, Desempenho.

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
FedAvg	Federated Average
GB	Gigabytes
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
ID	Identifier
IID	Independent and identically distributed
IoT	Internet of Things
ML	Machine Learning
MEC	Multi-access Edge Computing
MNIST	Modified National Institute of Standards and Technology
RAM	Random-Access Memory
RPC	Remote Procedure Call
VM	Virtual Machine
vCPU	Virtual Central Processing Unit

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Federated Learning is a solution proposed by (MCMAHAN et al., 2016) in order to train machine learning models while keeping the data private and decentralized. It works by having different clients training their own models with their own data, then averaging these trained models in a single centralized server, creating a global model. These clients can be separated into two categories. They are either end devices, consisting of single devices such as a smartphone, a single sensor, or another IoT device with internet connection such as a smart TV or a smart car; or they are data silos, distributed databases that need to keep their data private with the outside world, but are comprised of multiple devices. Data silos examples include hospitals (XU et al., 2021) and farms (DURRANT et al., 2021).

The clients, either data silos or end devices, train their models with their own data and, after finished, the models are uploaded to a central server, where all the client models are aggregated, converging to a single final model made from the uploaded models. While regular centralized machine learning may outperform federated learning in terms of accuracy (NILSSON et al., 2018), it requires the entire data set to be public. Federated learning allows large scale data sets to be used for training models while keeping the data private to each client. A use case example is Google's mobile keyboard prediction (HARD et al., 2018), which learns from every smartphone using Gboard – the Google keyboard – without sharing user data.

Federated learning has been greatly enabled by the vast advances and abundance of IoT devices. According to (CISCO, 2020), the number of devices connected to IP networks will be more than three times the global population by 2023, there will be 3.6 networked devices per capita by 2023, up from 2.4 networked devices per capita in 2018, and there will be 29.3 billion networked devices by 2023, up from 18.4 billion in 2018. This represents a rapid increase of potential federated learning use cases, considering the plenitude of client data to be trained in order to produce high accuracy machine learning models.

Additionally, edge computing has been greatly enabled by the advancement in technologies and tools such as Docker[1], which allows an application to be containerized, i.e., turned into a container, a package containing the application code and all its dependencies. This way, the application can run reliably in any computing environment.

[1]https://www.docker.com/

Besides portability, another important aspect of Docker vital for edge scenarios is its small overhead (AVINO et al., 2018), making the container as lightweight as possible and empowering it to quickly be deployed and run in every environment.

Besides Docker for containerization, there is also the need to manage and orchestrate these created containers as simply and seamlessly as possible. Tools such as Kubernetes have become the industry standard for orchestration, although there are multiple options including Docker Swarm, which is used for this work. Container orchestration leverages Docker container capabilities and allows the client to control the whole lifecycle of the container: creating, updating, monitoring, and destroying. Additionally, it also enables fast deployments for different scenarios with transparent load balancing and horizontal scaling – adding more nodes instead of more computing power.

This work proposes an architecture that is able to run federated learning algorithms in a distributed environment, where the clients are machines with limited power, similar to edge devices. Then, on top of this architecture, a set of experiments have been conducted with different configurations of data distribution for a varying number of clients, while also changing parameters for both the client and the server. Collected data has been analyzed in order to understand how to optimize for model accuracy while minimizing computing resource usage on client-side.

The goal of this work is first to build an end-to-end federated learning platform that is easy to deploy and modify. The source code for the implemented solution will also be made available in order to facilitate extensibility. Second, the data analyzed in the experiments will provide an understanding on how to properly increase accuracy without overloading the limited power devices which are part of the federated learning process. This analysis can be useful for further development in federated learning technologies, such as orchestration between edge devices and cloud for federated learning use cases.

The remainder of this work is organized as follows. In Section 2 an overview for edge cloud orchestration and federated learning is presented. In Section 3 the tools and frameworks that enable the fundamental building blocks of the solution are discussed. In Section 4 the conceptual architecture of the proposed solution is presented. Section 5 approaches the layout of the experiments and presents the obtained results. Section 6 concludes the work outlining opportunities for future research.

2 EDGE CLOUD ORCHESTRATION AND FEDERATED LEARNING

This section approaches the main topics of this work, providing an overview of the current state of the art of edge computing and federated learning.

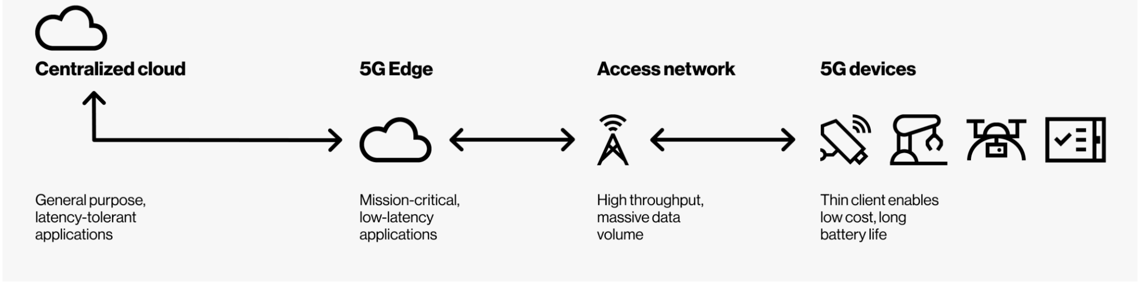
2.1 Edge Computing

The need for low latency computing along the rapid advancements in telecommunication services motivated the edge computing paradigm. In edge computing, instead of having the computing resources centralized in a data center – a cloud –, the idea is to distribute these resources in devices – the edge – closer to the final user, thus allowing lower latency and faster connections. These edge devices can be any device with Internet access such as smartphones, smart cars, or other IoT devices.

The concept of Multi-access Edge Computing (MEC) (GIUST; COSTA-PEREZ; REZNIK, 2017) proposes an infrastructure that places storage and computation at the network edge, benefiting situations where real time access to the server is required or preferred, while also reducing the traffic to a centralized network. This creates a heavily decentralized infrastructure that provides low latency connection to the end user, all while minimizing centralized cloud limitations such as delay, access bottlenecks, and single points of failure.

A MEC infrastructure is shown in Figure 2.1, where 5G IoT devices connect 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.

Figure 2.1: Multi-access Edge Computing Infrastructure


Centralized cloud
5G Edge
Access network
5G devices
General purpose, latency-tolerant applications
Mission-critical, low-latency applications
High throughput, massive data volume
Thin client enables low cost, long battery life

Source: Verizon[1]

Multi-access Edge Computing use cases that fully benefit from these concepts

[1]https://www.verizon.com/business/solutions/5g/edge-computing/multi-access-edge-computing/ (accessed October 19th, 2021)

include autonomous driving (LIU et al., 2019), where edge devices need to process a large amount of data from different sensors at high speed in real time in order to guarantee the safety of the drivers; and smart city traffic monitoring (BARTHéLEMY et al., 2019), where 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.

More recently, many advances connecting edge computing to artificial intelligence were made. (ZHOU et al., 2019) define Edge Intelligence as the union between AI and edge computing, an opportunity that rose in virtue of the abundance of devices connected to the internet that generate huge amounts of data on a daily basis. Edge Intelligence aims to capitalize on this data to train machine learning models, using concepts such as Deep Learning and Federated Learning.

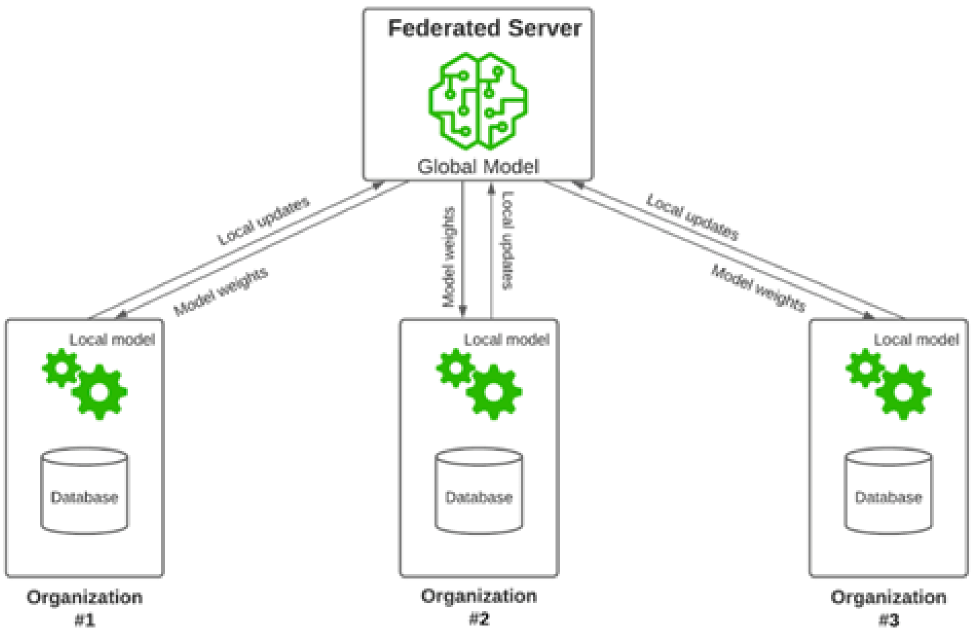
Additionally, (DENG et al., 2020) further expand on Edge Intelligence and propose a conceptual difference between artificial intelligence for edge and artificial intelligence on edge. The former encompasses intelligence-enabled edge computing that provides solutions to edge computing problems by utilizing artificial intelligence, while the latter encompasses how to run artificial intelligence models on edge devices, extracting insights from its distributed data nature. For this work, we are more interested in AI on edge.

2.2 Federated Learning

Federated learning is a decentralized form of machine learning used to train models at scale while allowing the user data to be private. Federated learning was first introduced by Google (MCMAHAN et al., 2016), which provided the first definition of federated learning, as well as the Federated Optimization (KONEčNý et al., 2016) approach to further improve these federated algorithms. Google also explains in further detail the concept of federated learning in the Federated Learning: Collaborative Machine Learning without Centralized Training Data blog post (GOOGLE, 2017), stating the usage to predict keyboard words as seen in (HARD et al., 2018) and planning to also use federated learning for photo ranking and further improving language models.

A generic federated learning infrastructure can be seen in Figure 2.2. Organizations 1, 2, and 3 represent three different clients which create their own models using private data. These models are then sent to the Federated Server which handles the aggregating and has no access to any kind of data. Then, this central server updates each model's parameters with the updated values.

Figure 2.2: Federated Learning Architecture



Source: Sparkd.AI[2]

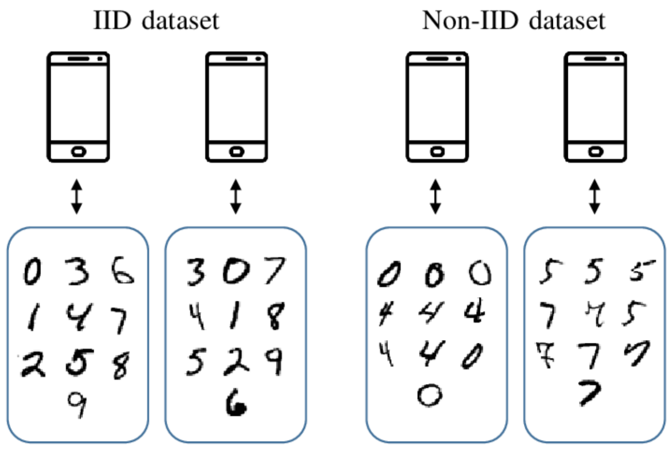
Federated learning deals with data that is not independent and identically distributed (non-IID). Since the data samples are distributed among multiple devices that are not synchronized and may have limited connectivity, this also means that, besides being non-IID, federated learning cannot simply train these devices in parallel and aggregate them, as it cannot guarantee the device availability or the homogeneity of the data. While advances have been made to understand the heterogeneous nature of the data samples (ZHU et al., 2021) and how to properly average them (WANG et al., 2020), this remains the main challenge for federated learning.

The difference between an IID dataset as usually found in standard machine learning scenarios, and a non-IID dataset, found in federated learning scenarios, is illustrated in Figure 2.3. In the left is illustrated the standard IID dataset: every data point is represented for every client, meaning that every client will have data for how the numbers from 0 to 9 look like. In the right we see the opposite; for the Non-IID dataset, clients may have heterogeneous data samples with few or no matches between clients.

In order to properly define and further advance in the non-IID subject, the FedML Research Library and Benchmark (HE et al., 2020) has been proposed to facilitate federated algorithm development and performance comparison. FedML provides an open-

[2]https://sparkd.ai/archives/4444 (accessed April 8th, 2022)

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



Source: (HELLSTRöM et al., 2020)

source framework that allows the development and evaluation of novel federated algorithms. Similarly, the Flower Learning Research Framework (BEUTEL et al., 2021) also provides heterogeneous environments that allow experimentation with non-IID data and algorithms. FlowerML is used as the federated learning infrastructure facilitator for this work.

Despite the novelty and the myriad of challenges, federated learning adoption is rising. Besides Google's keyboard, (JI et al., 2019) proposes a novel optimized model aggregation for keyboard suggestion that considers each client contribution to the global model and weighs them instead of simply averaging. Federated learning is also key to fully utilize machine learning capabilities in scenarios where the data cannot be shared due to sensitivity such as for the health industry (RIEKE et al., 2020).

Google, along researchers from many universities, at the workshop on federated learning and analytics, states that federated learning is inherently interdisciplinary (KAIROUZ et al., 2021), encompassing techniques and methods from other fields such as cryptography, security, differential privacy, fairness, compressed sensing, systems, information theory, and more, requiring a collaborative effort in order to further advance the subject. For this work, we are notably interested in the intersection that federated learning has with edge computing.

3 TOOLS AND FRAMEWORKS

This section provides an overview of the tools and frameworks that are used to enable the fundamental architectural blocks for the solution.

3.1 Docker

Docker is an open platform for developing, shipping, and running applications[1]. Docker is a staple for microservices development as it enables to separate the applications from the infrastructure so the software can be delivered regardless of the underlying host machine.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, making it a good and portable solution to run applications on different edge devices.

Docker application specifications are called *images*. Images are templates with instructions to create the containers, which are usually based on another images. The images created for this work, for example, are based either on top of the Python or Javascript image which, in turn, may be based on Alpine Linux or other lightweight operating system as such.

To build an image, it is require to specify a *Dockerfile* with the steps to create it. There, it is specified the base image and other dependencies such as packages or data. Each instruction in a Dockerfile creates a layer in the image. When a Dockerfile is changed, only those layers which have changed are rebuilt, which allows for faster development iterations.

Containers are runnable instances of these images. Many tools are available to better manage the containers. For this work, Docker Swarm[2], which is included with Docker, is used due to its simplicity and light weight. As the use cases here are mainly for easiness to deploy and observation purposes, there is no need to use more complex and robust solution such as Kubernetes.

A Docker Swarm is a group of machines that are running the Docker containers

[1]https://docs.docker.com/get-started/overview/
[2]https://docs.docker.com/engine/swarm/ (Accessed April 8th, 2022)

and have been configured to form a cluster together. Once the machines are clustered, every Docker command can be carried out to the other machines in the cluster. Machines can be set to be workers or managers and, from the manager machine, the whole cluster can be accessed. This, in practice, allows for one-line deployments, while also enabling simpler observability as it can be done from one node only.

What makes Docker Swarm and Docker even more powerful is that containers can be connected together in a network managed by the Swarm. After defining the network, the services can communicate with each other using their canonical names (e.g. my-server, my-client). Swarm specifically creates a network in *overlay* mode, which enables communication through different host machines clustered together. Then, Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container. This makes deploying a full end-to-end federated learning system, after creating the image for each service and creating a cluster, as easy as declaring in a single file which images will be used, the name of the containers, the network, and telling docker to deploy them.

3.2 Flower

The birth of federated learning and its subsequent adoption stems from the need to utilize data from multiple devices while maintaining it private at the same time, as mentioned throughout this work. However, despite usual machine learning which is already a complex subject, other two fundamental issues are introduced when it comes to federated learning experiments: scaling to multiple clients and data heterogeneity.

Flower[3] (flwr) is an open-source framework for building federated learning systems which aims to solve these problems. Flower delivers two main interfaces: the client and the server. These interfaces allow for usual, standard centralized machine learning solutions to be easily decentralized by implementing the required methods.

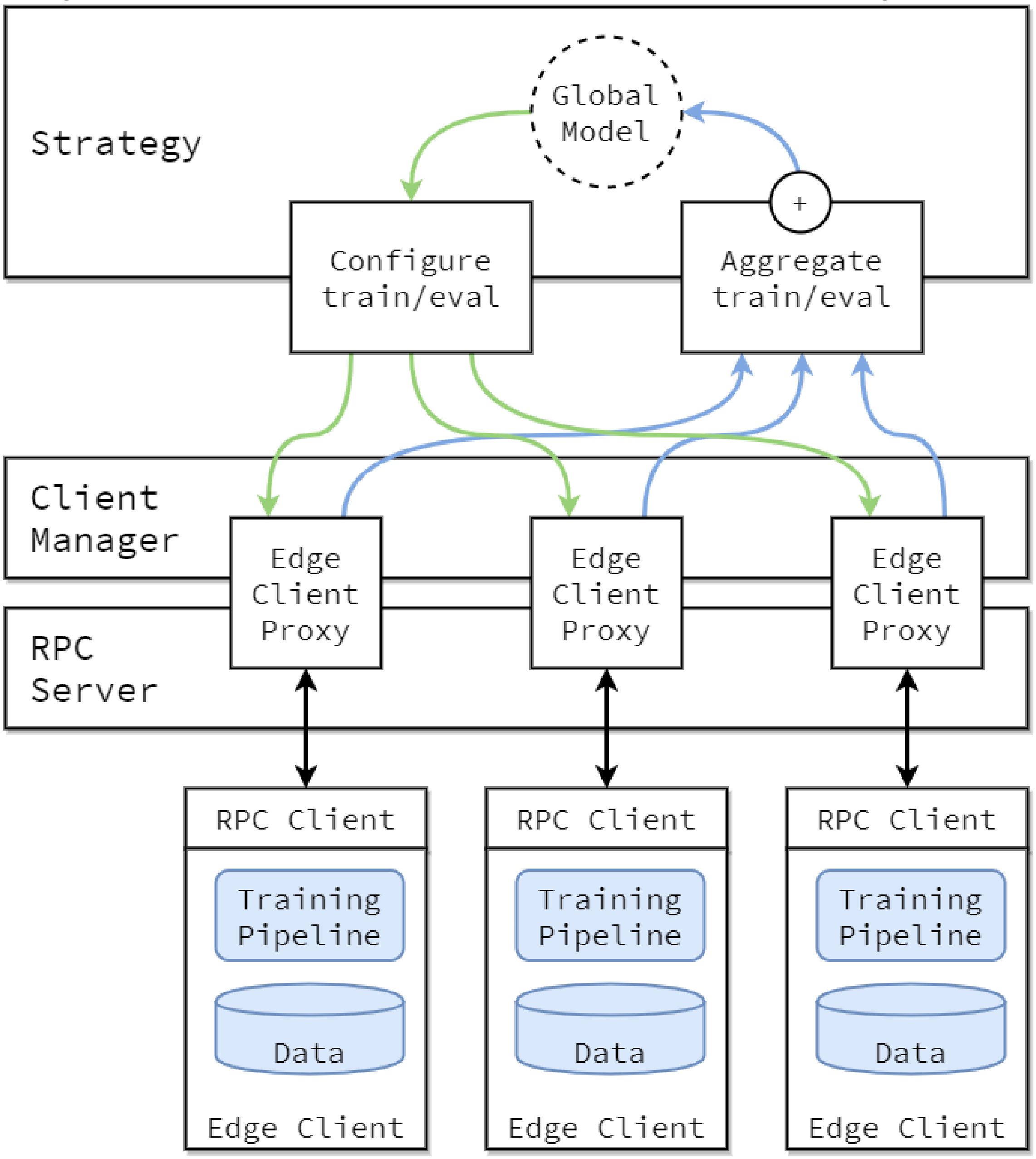
By having a centralized machine learning algorithm implement the federated client interface, and running a server containing an *averaging strategy*, also provided by the Flower Framework, it makes it possible to run this client multiple times – even on the same machine – and Flower handles the client-server communication, as well as the averaging, parameter updates, and accuracy measurements.

Figure 3.1 shows the Flower Federated framework architecture when running with

[3]https://flower.dev (Accessed November 22nd, 2021)

multiple edge clients. Each device is running a Flower client containing a regular, centralized machine learning solution while implementing the needed methods for the client interface. Flower provides a transparent connection from the clients to the server via the Edge Client Proxy utilizing an RPC protocol – gRPC, for instance. The server application is responsible for the top half of the diagram: it creates the global model by averaging the parameters from the clients utilizing a pre-defined strategy, and communicates these parameters back to the clients through the Edge Client Proxy connection.

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



Source: Flower Documentation[4]

[4]https://flower.dev/docs/architecture.html (accessed April 18th, 2022)

It is also key to note the importance of Flower being open source and the importance for open source tools in general when dealing with federated learning: as privacy is the main concern, the usage of tools that contain open code is vital to ensure user trust and enable anyone interested in checking and evaluating whether the tool actually guarantees data privacy among clients.

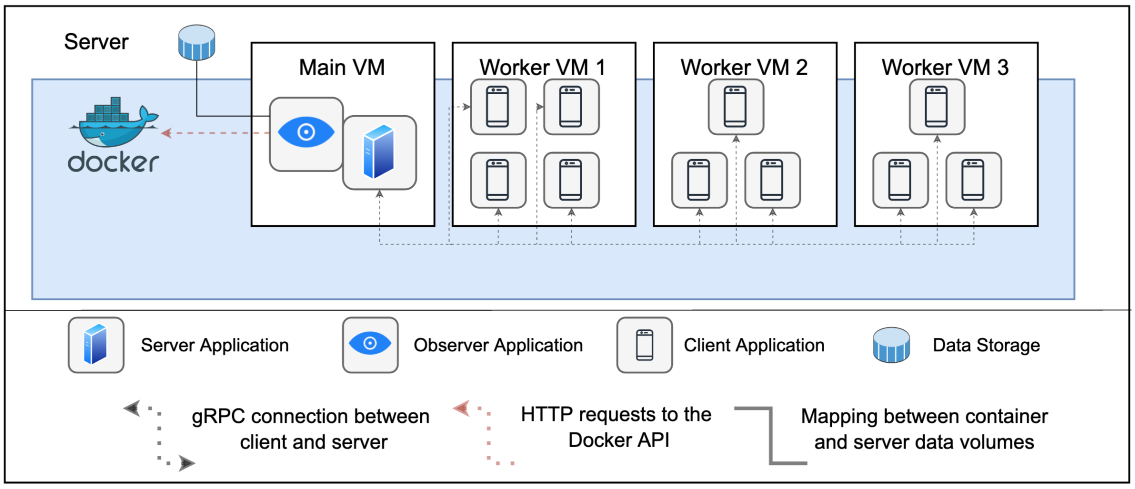
Flower originated from a research project at the Univerity of Oxford, so it was built with AI research in mind. It enables experiments with federated learning without the budget of a major technology company, allowing researchers around the world to further advance federated learning knowledge.

4 SOLUTION

This section presents the solution implemented in order to run fully functional federated learning scenarios with multiple clients with different data distributions. It includes details about the underlying server infrastructure, the implemented architecture with containerized microservices, including how the communication and observability is done, and the data that has been used for training.

Figure 4.1 shows the full solution diagram in a scenario running with ten different clients. We can see the underlying infrastructure, i.e., the server and the VMs, and on top of that the docker containers running each application: the server, the client, and the observer. Further subsections will further expand on each part of this diagram. First, the underlying infrastructure where the solution is running is presented. Then, the solution architecture and its components are presented. Finally, the data that will be used for the experiments is covered.

Figure 4.1: High-level diagram of the implemented solution when running with ten clients


Server
docker
Main VM
Worker VM 1
Worker VM 2
Worker VM 3
Server Application
Observer Application
Client Application
Data Storage
gRPC connection between client and server
HTTP requests to the Docker API
Mapping between container and server data volumes

Source: Image provided by the author

4.1 Infrastructure

To achieve the desired infrastructure, multiple machines are required. For this work, we are interested in both how the centralized server and how the distributed clients function.

For the server, ideally we have a powerful machine. The centralized server, in an ideal federated learning scenario, runs on a traditional cloud; a powerful computer capable of more complex operations. To achieve this, a VM *main* runs exclusively the server and observer applications and is twice as powerful as the VMs running the clients, having 4GB RAM.

For the clients, as we are simulating an edge environment, we will utilize VMs with limited power which run more than one client at once. We will call them *workers*; machines with 2GB RAM that will run from zero to four federated clients at once depending on the experiment.

The infrastructure components are described on table 4.1, which highlights the available computing resources for each machine used for the implemented solution.

Table 4.1: VMs used for the solution infrastructure

VM	**RAM**	**CPUs**	**Disk space**
main	4GB	2vCPU	50GB
worker1	2GB	1vCPU	50GB
worker2	2GB	1vCPU	50GB
worker3	2GB	1vCPU	50GB

Source: Table provided by author

4.2 Architecture

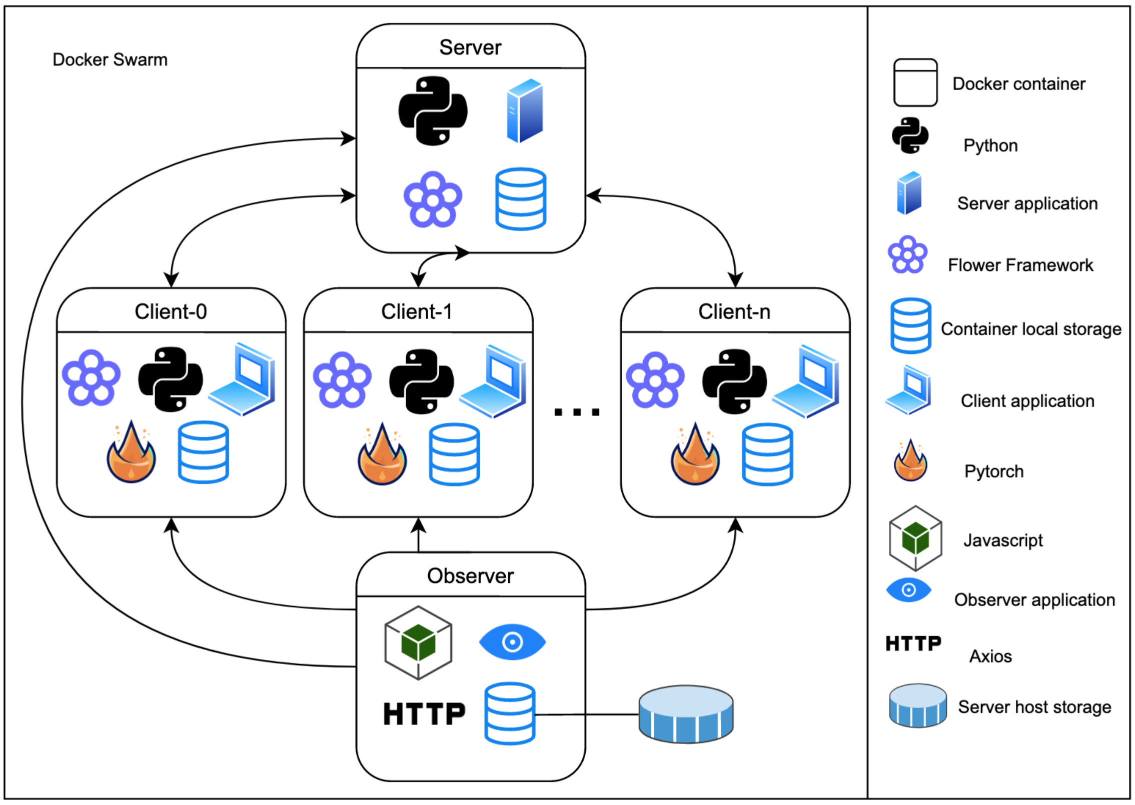
The solution has three architectural components. These are the server, the client, and the observer services, which are all containerized services that can run in any underlying infrastructure. Each client trains a local model and uploads this model to the server which averages and returns the updated parameters. Meanwhile, the observer retrieves and persists to a volume metrics of every running container besides itself.

Figure 4.2 shows the high-level architecture of the solution, highlighting each of the aforementioned architectural components and their dependencies. The microservices with their respective applications and dependencies are individually defined and deployed together in a Docker Swarm using a *docker-compose* file. To the right in the figure, each dependency for the Docker containers is explicit.

It should be noted that, while there will be multiple client services and only one server and one observer, the clients will be treated as a single service while explaining in this section for simplicity purposes. Since the only difference, application-wise, between one client and the other is the data in the storage, there is no need to analyze them differently when looking at them on a service level.

These services will be then deployed to the infrastructure previously defined.

Figure 4.2: Solution architecture with the services and their dependencies



Source: Image provided by the author

Server and observer services are exclusively deployed to the *main* VM, while the client services are distributed among the *workers*.

In the following sections, the individual components of the architecture will be further presented.

4.2.1 Server

The server is a containerized Python application with the FlowerML server framework as dependency. The server is responsible for receiving the models being trained by the clients, averaging the received parameters, then updating the clients' models with the averaged parameters. The average is done by the server using Federated Average – FedAvg –, which is a standard federated learning averaging method, also used by Google Keyboard, which simply averages the parameters received by the clients without attributing weights. The server connects to the clients through a gRPC connection and performs a pre-defined number of averaging rounds. The algorithm is outlined in Algorithm 1.

Algorithm 1: Server service loop

```
1 open connection in $address$;
2 while $minimum\_clients$ have not yet connected do
3     wait;
4 for $i \leftarrow rounds$ to $0$ do
5     receive parameters from clients;
6     average client parameters;
7     send updated parameters back to clients;
8     evaluate current accuracy from two random clients;
9     log timestamp and accuracy;
```

The server address, number of rounds, and minimum number of connected clients are parameterized for the server application and can be changed in each deployment. The full code for the server can be found in the *server* module in the Github repository[1].

4.2.2 Client

The client is a containerized Python application with the FlowerML client framework, Pytorch[2], and the data that will be used as dependencies. The client is responsible for training a local model with local data in a convolution neural network, then uploading the model to a server which in turn returns updated values for the sent parameters.

The client connects to the server through a gRPC connection and performs a pre-defined number of epochs, 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

[1]https://github.com/remde/federated-learning/tree/main/server
[2]An open source machine learning framework: https://pytorch.org/ (Accessed April 21st, 2022)

for the client service can be seen in Algorithm 2.

Algorithm 2: Service service loop

```
1 start connection with server in $address$;
2 while server connection is open do
3     for $i \leftarrow epochs$ to $0$ do
4         iterates through dataset training the local model;
5     upload model to server;
6     receive updated parameters;
7     update local parameters;
```

It should be noted that, while the above algorithm is a good representation of the client service lifecycle, the client is ultimately controlled by the server. The client will only upload the model when the server requests the models, and the loop will end when the server finishes its rounds.

The client epochs and the address which they will connect are parameterized for the client application and can be changed in each deployment. The code for the client can be found in the *client* module in the Github repository[3], which contains two client examples without any data.

4.2.3 Observer

The observer is a containerized JavaScript application with axios[4] as dependency. The observer is responsible for logging information about the other containers – namely the server and clients. The observer is the only container which does not stop unless when forced to do so; it keeps continuously retrieving data of every other container in the Swarm and persisting the data to the Docker volume when these other containers are stopped.

The observer has a volume which is mapped directly to a folder in the main VM. In practice, every data that is persisted to an observer container is also persisted to the host VM. Observer services allow for an *EXPERIMENT_NAME* parameter, which is the folder where the data will be saved. This can be changed in each deployment. The observer loop

[3]https://github.com/remde/federated-learning/tree/main/client

[4]Promise based HTTP client for the browser and node.js: https://axios-http.com/ (Accessed April 18th, 2022)

can be seen in Algorithm 3.

Algorithm 3: Observer service loop

```
1  initialization;
2  previous ← get active container IDs;
3  while any container is running do
4      current ← get active container IDs;
5      get current container stats for each ID;
6      if ID in current is not in previous then
           /* Container retrieved for the first time          */
7          create entry in hashmap ID: stats[];
8      else
           /* Existing container was retrieved                */
9          append stats to existing entry in hashmap;
       /* If container stopped between loop executions, stats are
          persisted to disk                                   */
10     if ID in previous is not in current then
11         get container service name;
12         get container host;
13         get current timestamp;
14         create folder EXPERIMENT_NAME/SERVICE;
15         persist hashmap entry with timestamp-host filename;
16         remove entry from hashmap;
17     previous ← current;
```

To collect containers stats, the observer leverages the Docker stats API[5], with the server accuracy being a separate case. As the accuracy is a metric created by the server application, to retrieve the accuracy the observer looks at the container logs API instead of using the stats API. Every request is done via HTTP requests do the Docker APIs, being completely decoupled from the other services logic. The logic used to retrieve container

[5]https://docs.docker.com/engine/api/v1.21/ (accessed February 22nd, 2022)

stats is deeper described in Algorithm 4.

Algorithm 4: Retrieving container information

```
1  initialize hashmap;
2  for $host$ in $hosts$ do
3      retrieve $containers$ in $host$;
4      for $container$ in $containers$ do
5          get stats for $container$;
6          if $container = server$ then
7              retrieve server container logs;
8              extract timestamp and accuracy values;
9              add values to stats object;
10         save values to hashmap entry;
11 return hashmap;
```

The stats retrieved by the Docker API include, but are not limited to, network information such as bytes received and transmitted, and memory and CPU usage information. The result of each observer run is a HashMap which holds the container IDs as keys mapping to an array of stats of the container through time. These entries are then persisted to a JSON file according to their experiment name, separated in folders according to the service name (e.g. server, client-0, client-1...), with the timestamp and host as the filename.

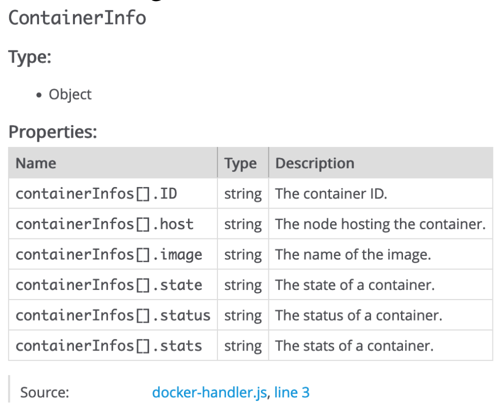
This allows for an easy visibility of the logs and, since the observer never dies, it also allows for multiple runs of the same experiment without having to worry about reboots: the other containers start and stop automatically as the observer obtains metrics for all of them. The full code for the observer can be found in the observability module in the Github repository[6]. It should be noted for module reusability and future work purposes that the docker-service.js file, responsible for the majority of the important logic such as retrieving container stats, is thoroughly documented using JSDocs[7].

Figure 4.3 shows an example of a docker-service API documentation generated using JSDocs. getAllContainerInfos, on the right, is the method used to retrieve a list of the container information, i.e., Array<ContainerInfo>. The ContainerInfo custom type is explicit in the left; also generated with JSDocs.

[6]https://github.com/remde/federated-learning/tree/main/observability

[7]Markup language used to annotate JavaScript source code files: https://jsdoc.app/ (Accessed February 23rd, 2022)

Figure 4.3: docker-service API documentation generated with JSDocs


ContainerInfo

Type:

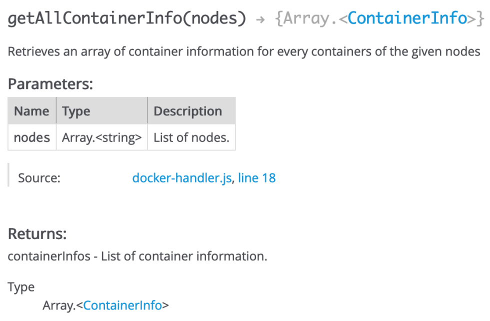
- Object

Properties:

Name	Type	Description
containerInfos[].ID	string	The container ID.
containerInfos[].host	string	The node hosting the container.
containerInfos[].image	string	The name of the image.
containerInfos[].state	string	The state of a container.
containerInfos[].status	string	The status of a container.
containerInfos[].stats	string	The stats of a container.

Source: docker-handler.js, line 3

(a) ContainerInfo type definition, used for the return type of getAllContainerInfos


getAllContainerInfo(nodes) → {Array.<ContainerInfo>}

Retrieves an array of container information for every containers of the given nodes

Parameters:

Name	Type	Description
nodes	Array.<string>	List of nodes.

Source: docker-handler.js, line 18

Returns:

containerInfos - List of container information.

Type

Array.<ContainerInfo>

(b) getAllContainerInfos JSDocs documentation

Source: Images provided by author

4.3 Data

The dataset used for the experiments is the CIFAR dataset (KRIZHEVSKY; HINTON et al., 2009). The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The original dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. A dataset sample with the classes can be seen in figure 4.4, with 10 rows of images, one row for each class as stated in the first column.

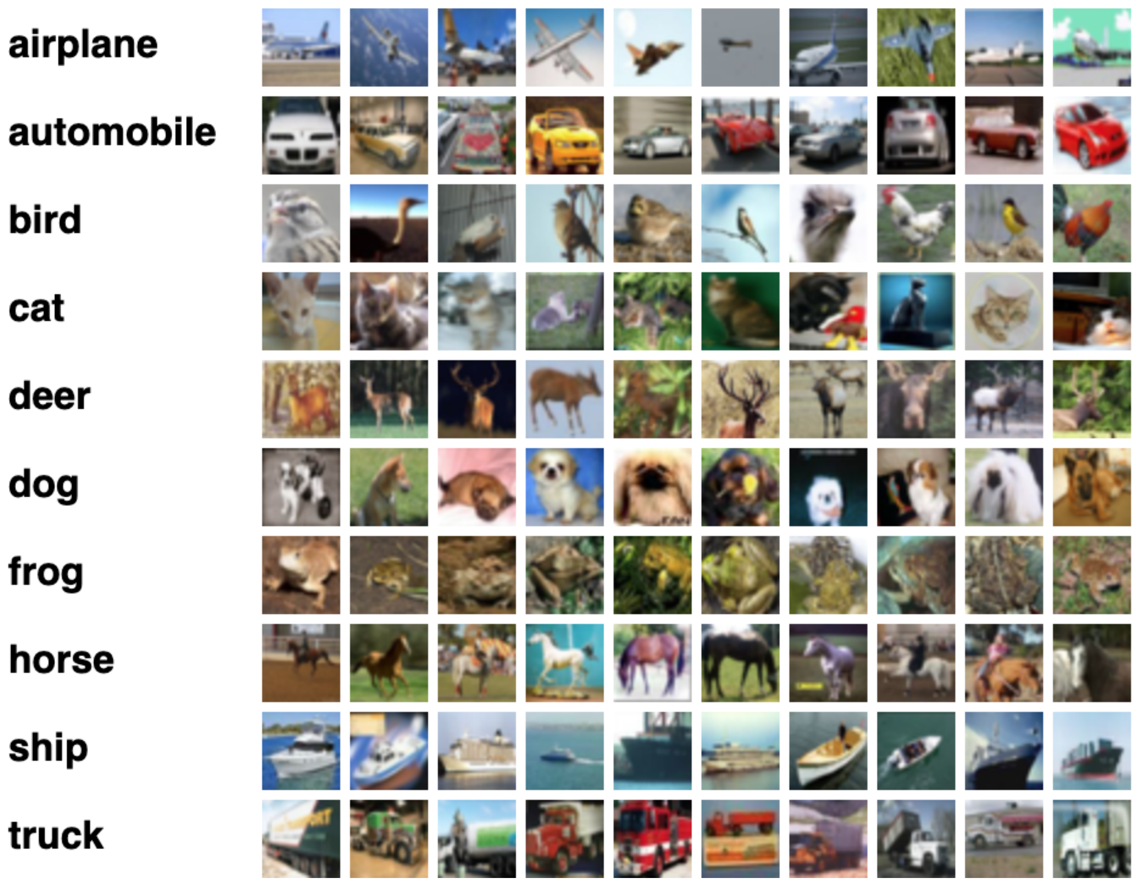
In order to properly divide this data between multiple clients, the five original data batches have been divided into 500. This enables a higher level of granularity for testing federated learning scenarios, as we are able to precisely state a percentage of data overlap between clients. Additionally, for scenarios of more than five clients, the original five batches wouldn't be sufficient for having every client with a different dataset.

To separate the batches, the Python package Pickle[9] was used for the marshalling and unmarshalling of the data. In the new batches, similar to the original five, each of

[8]https://www.cs.toronto.edu/ kriz/cifar.html (accessed April 10th, 2022)

[9]Module for serializing and de-serializing Python object structures: https://docs.python.org/3/library/pickle.html (Accessed February 2nd, 2022)

Figure 4.4: CIFAR-10 Dataset Sample



Source: University of Toronto Department of Computer Science[8]

the 500 batches has exactly the same amount of images – 100 per batch, with classes randomly distributed among them. The total number of images remains the same, 50000 total, being 5000 images from each class. The test batch remain the same. The code can be found in the *utilities* module in the Github repository[10].

[10]https://github.com/remde/federated-learning/tree/main/utilities

5 EXPERIMENTS

5.1 Layout

Each experiment has been ran at least two times, with some being ran for up to sixteen times. The results have been averaged to try to minimize any randomness. Every experiment utilizes homogeneous client configurations, which means that every client involved in a given experiment is exactly the same, except for the data that it contains and the host in which it runs. Since every worker VM has the same specification, we can assume that only the data is different between clients for any given experiment.

It also should be noted that, while absolute accuracy is an extremely important metric for any machine learning algorithm, it isn't the focus of this work. The goal is to analyze how different parameters impact accuracy and resource usage. Higher accuracy numbers 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 the CNN, which is out of scope as far as we are concerned.

There are different parameters that can be modified in order to test the system performance which will be covered in the next sections. These are the number of clients, the data distribution, the client epochs, and the server rounds.

Each experiment will run for 2, 4, 6, 8, and 10 different clients. They will be named client-0, client-1, up to 9. When not all clients are used (e.g. when only 4 clients are part of the experiment), the clients that are part of the experiment will be chosen randomly.

The data distribution has been separated into color codes: green and red. Both green and red will have available the 500 batches mentioned in the previous section. What differs between them is the amount of data each client will have. Green clients have 50 batches, which means that for the experiment with 10 clients, every batch will be used without any overlap. Red clients have 100 batches, so there will be data overlap between clients, with 50% of the data being replicated for the 10 clients participating in the experiment. Table 5.1 shows the data distribution layouts for the ran experiments as described.

The number of epochs of the client is how many times the whole dataset is passed in the CNN. Four different configurations will be tested: 1, 5, 10, and 25 epochs. Ad-

Table 5.1: Data distribution layouts for the ran experiments

Experiment color	Batches per client	Total batches	Maximum data overlap
Green	50	500	0%
Red	100	500	50%

ditionally, each number of epochs configuration will have a matching number of server rounds. Respectively, the maximum number of rounds is 100, 35, 20, and 10. This matching is mainly to guarantee that every experiment converges, but also that the experiments do not run for over one hour due to time constraints. Table 5.2 shows the layouts for number of rounds and number of epochs for the ran experiments as described.

Table 5.2: Layouts for number of rounds and number of epochs for the ran experiments

Number of client epochs	Number of server rounds
1	100
5	35
10	20
25	10

There are two different data distributions, five number of clients possibilities, and four layouts for number of rounds and epochs. In total, 40 different experiments layouts have been ran, with 270 total ran experiments as most layouts were executed multiple times, generating over 1.25GB of plain text files containing observability logs.

5.2 Results

In total, 1633 containers of a client or a server application were ran. For each one of these, a log file was generated containing stats information through time. From these 1633 files, 80 were created to generate the results.

As previously mentioned, when it comes to computing resource usage, we can safely assume that clients are homogenous. In an experiment, while the data content is different between clients, the amount of data is the same and, while the host machine is different, clients are always ran on worker VMs which have the same specs and servers are always ran on the main VM.

This makes so that one client file and one server file were generated for each of the 40 experiments, making 80 total files. To achieve this, using Python scripts and pandas[1], the logs for each were all averaged and converged into one file. This one file also contains

[1]Open source data analysis and manipulation tool: https://pandas.pydata.org/ (accessed April 17th, 2022)

an array of stats through time, but instead of the entire log that Docker generates, only 8 properties were inserted. These are:

- memory_percentage: current memory percentage usage. This is not originally retrieved by the Docker Stats API, it is calculated from the logs using resource usage formulas[2].
- cpu_percentage: current CPU percentage usage. Also calculated using resource usage formulas.
- time_from_start: time in seconds since the start of the experiment. Calculated using the timestamps retrieved with the Docker Stats API.
- bytes_received: current amount of received bytes.
- bytes_transmitted: current amount of transmitted bytes.
- packets_received: current amount of received packets.
- packets_transmitted: current amount of transmitted packets.
- accuracy: current accuracy. This stat is only shown for server logs. Retrieved using Docker Logs API.

The code for extracting information from the logs, averaging, as well as for plotting, can be found in the *plotting* module in the Github repository[3].

5.3 System Accuracy

5.3.1 Effects of Replicating Data Between Clients

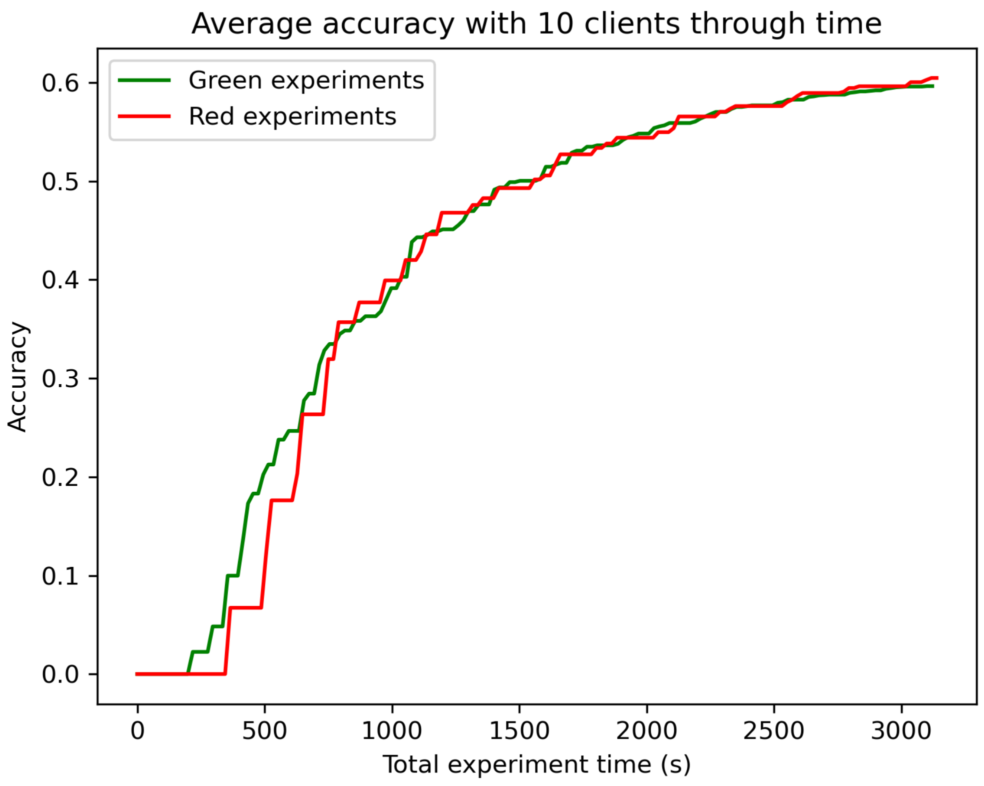
The first question to be answered regarding server is if data replication impacts the accuracy of the system. To answer this, a plot of accuracy through time has been done for both red and green data distributions with ten clients. Green will have the 500 batches with 50 unique batches per client, totaling the 500; and red, while having the same 500 batches available, will have 100 batches per client, with 50% of them being overlap data.

There were four total experiment layouts for green and four for red that match this specification. The difference between them is the number of client epochs and server rounds. These four averages were further averaged and converged into one, resulting in one for green and one for red.

[2]https://docs.docker.com/engine/api/v1.41/#operation/ContainerStats (accessed February 22th, 2022)
[3]https://github.com/remde/federated-learning/tree/main/plotting

Figure 5.1 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates the accuracy. There is a red line, which represents the averages for the red experiments with ten clients; and a green line which represents the averages for the green experiments with ten clients. Even with twice the number of batches, there wasn't significant difference for the red experiments, not even in experiment time. The curves are essentially the same. This indicates that what matters to the total accuracy is the unique data spread among clients.

Figure 5.1: Plot for the average accuracy for red and green experiments with ten clients through time

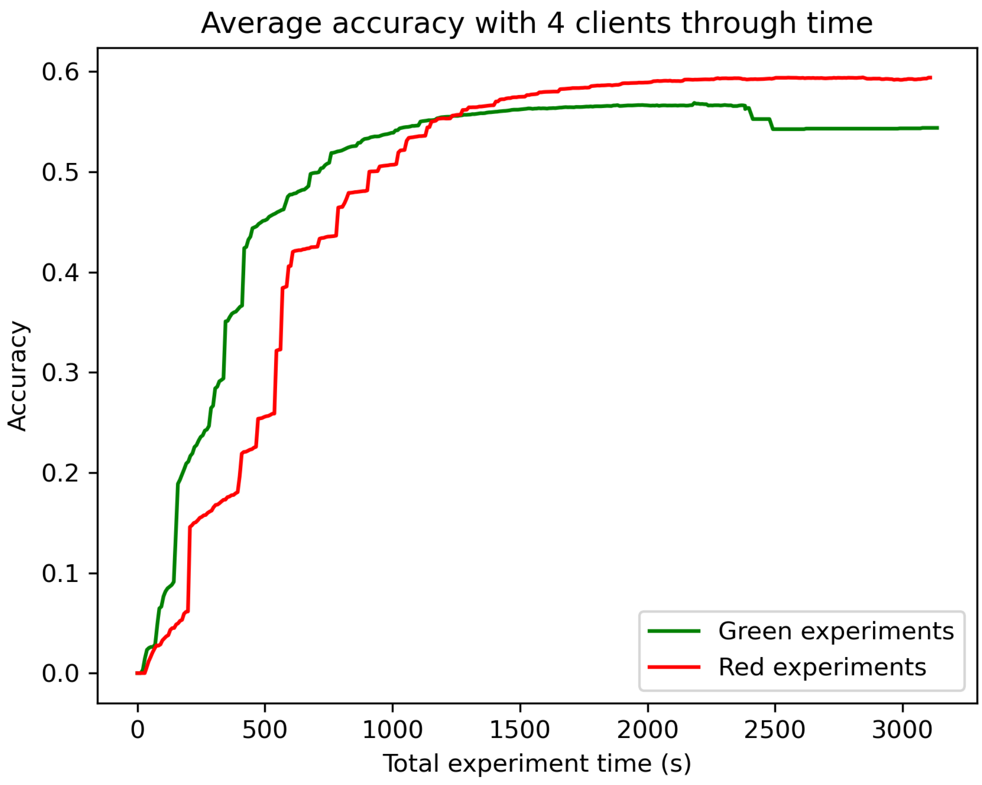


Source: Image provided by the author

To further confirm this, a similar experiment will be made but this time using red and green experiments containing four clients. Four has been chosen because it is a middle point. Green will contain 200 batches total, and red 400, both with no overlap whatsoever. Similar to the previous experiments, there are four green and four red layouts with four clients which have been averaged and converged to one for each data distribution.

Figure 5.2 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates the accuracy. There is a red line, which represents the averages for the red experiments with four clients; and a green line which represents the averages for the green experiments with four clients.

Figure 5.2: Plot for the average accuracy for red and green experiments with four clients through time



Source: Image provided by the author

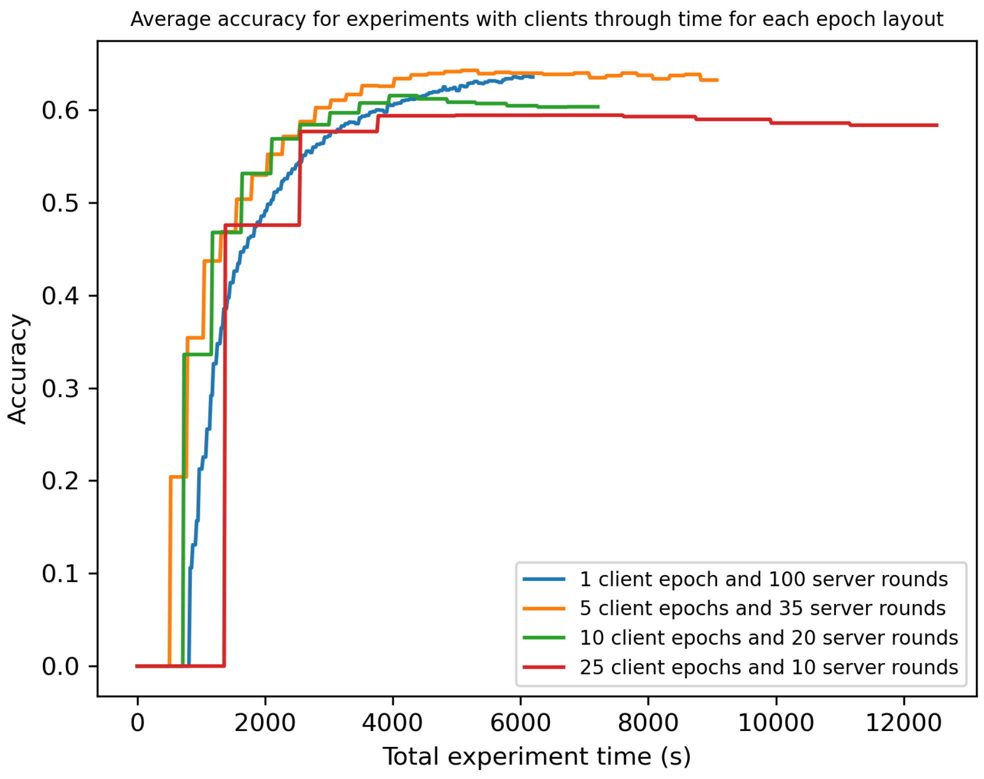
The behavior seen for this experiment is much different. The green experiments, due to having less data overall, increase their accuracy faster. However, red then surpasses green and outperforms it by a significant margin, being 10% superior in the second 3000 (50 minutes). This confirms the proposed idea that the impact in accuracy is from total unique data spread among clients.

5.3.2 Effects of Client Epochs and Server Rounds Layouts

Another experiment done to understand impact in accuracy is according to client epochs and server rounds. To accomplish this, using exclusively experiments with ten clients, red and green were averaged and split into four categories, one for each client epoch and server round configuration.

Figure 5.3 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates the accuracy. There is a blue line, which represents the averages for the experiments with ten clients with 1 client epoch and 100 server rounds; an orange line which represents the averages for the experiments with ten clients with 5 client epochs and 35 server rounds; a green line which represents the averages for the experiments with ten clients with 10 client epochs and 20 server rounds; and a red line which represents the averages for the experiments with ten clients with 25 client epochs and 10 server rounds.

Figure 5.3: Plot for the average accuracy for experiments with ten clients split by client epoch and server round configurations through time



Source: Image provided by the author

First thing that can be noted is that client epoch per client and server rounds are not equal; at least not in a one-to-one relation. We cannot simplify the rounds as a multiplication of client epochs and server rounds. If they had the same effect on accuracy and experiment time, the plots would indicate the green line as having the same result as the blue one in half the time, for instance.

By looking more closely at the red line, it rises to a maximum accuracy in very few server rounds, three or four, but the accuracy doesn't reach the same values as the other configurations. Moreover, it is possible to observe that the blue and orange line outperform the green and red lines by a wide margin. The blue line, despite having the least *epochsXrounds*, takes the least time and achieves the highest frequency. Additionally, different from the others, the experiment appears to not have reached the maximum accuracy by the end of its run.

The mentioned points indicate that, to achieve the maximum accuracy, and disregarding other aspects that may come with this decision, if choosing between client epochs and server rounds, one should increase server rounds as it is the more effective option. It should be noted that this assumption for now only holds truth when observing impact on accuracy in a vacuum.

5.4 Transmitted Data

The first analysis for network stats will be done by understanding how the server transmits data, i.e., packets and bytes, and if it is possible to understand a trend while changing number of clients, epochs, and amount of data in the system.

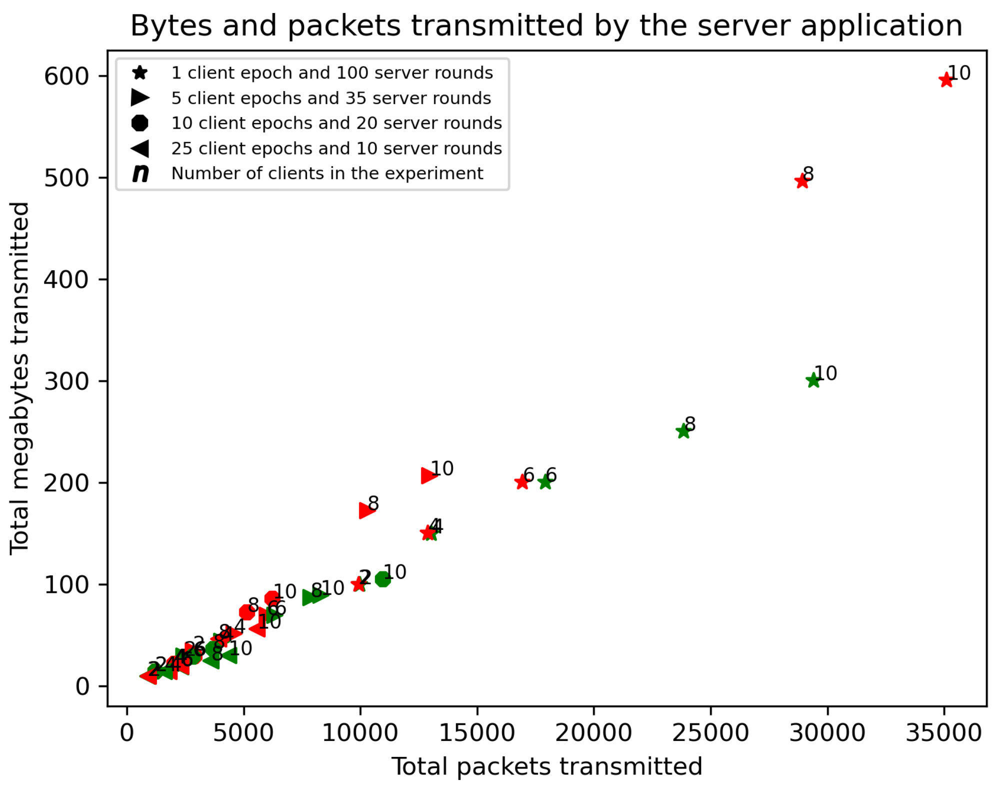
Figure 5.4 shows a scatter plot for the many different server layouts that were run. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. The different epochs and server round layouts are organized as follows: 1 client epoch with 100 server rounds is represented by a star. 5 client epochs with 35 server rounds is represented by an arrow pointing right. 10 client epochs with 20 server rounds is represented by an octagon. 25 client epochs with 10 server rounds is represented by an arrow pointing left. Green data distributions are in the color green, while red data distributions are in the color red. Finally, on top of each scatter, there is the amount of clients involved in the experiment.

This plot, while containing a lot of information, can be broken down and analyzed. Three different aspects can be inferred that contribute to higher data transmission.

5.4.1 Effects of Number of Clients Involved

The more clients involved in the experiment, the more total data is transmitted. To an experiment with the same layout for data distribution, client epochs and server rounds, there will be a linear proportion relation between the number of clients involved in the experiment and the total amount of transmitted data. This happens because, with more clients involved, the server has more clients to update parameters after having them

Figure 5.4: Plot for the total amount of bytes and packets transmitted by the server applications



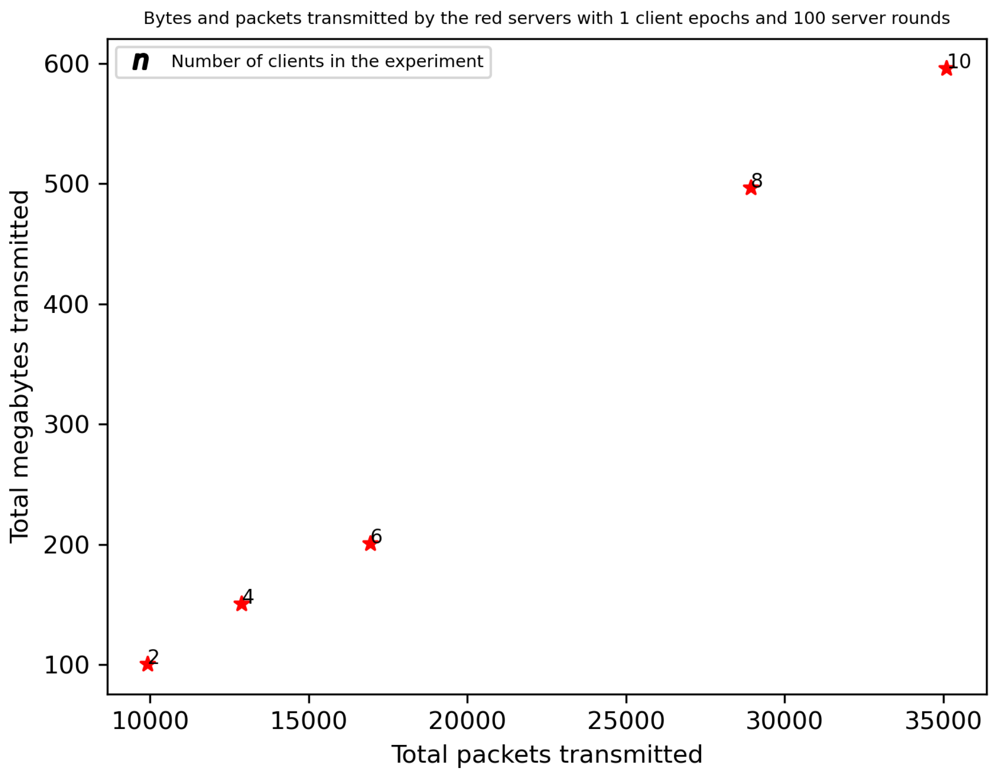
Source: Image provided by the author

averaged, requiring a higher amount of network usage.

Figure 5.5 further demonstrates this point, breaking down the original plot in Figure 5.4, showing a scatter plot for only the red configuration with 1 client epoch and 100 server rounds. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. On top of each dot, there is the amount of clients involved in the experiment. In this plot it is possible to see more clearly the aforementioned linear relationship between transmitted data and clients, as a virtually straight line can be crossed between the five points.

The decision for the client epochs and server rounds layout chosen being 1 and 100, as well as the decision to use red data distributions, is random – it is merely to pin a variable to prove the affirmation with a simpler plot. This tendency can be analyzed for any client epoch and server round configurations, both green and red.

Figure 5.5: Plot for the total amount of bytes and packets transmitted by the server applications with red data distributions, 1 client epoch and 100 server rounds



Source: Image provided by the author

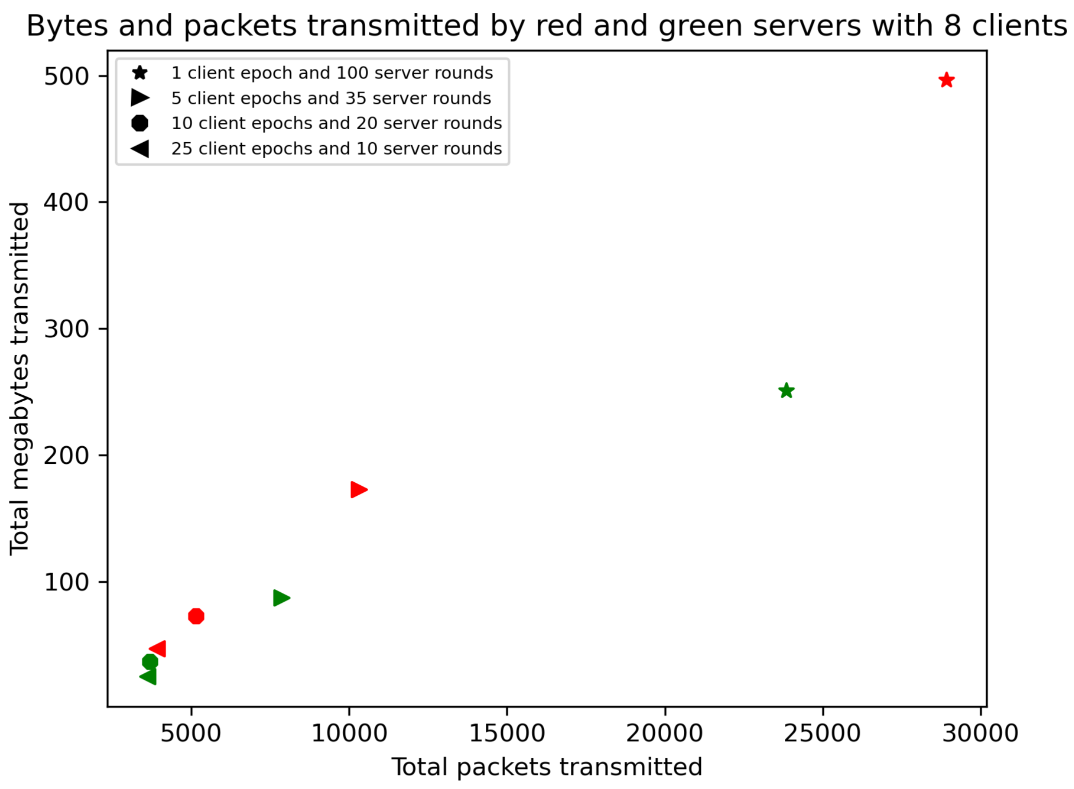
5.4.2 Effects of Total Amount of Data

The higher the total amount of data involved in the system, the higher is the total transmitted data by the server. This can be seen because red experiments are transmitting more data than their counterpart green experiments. Having this difference in the experiment with ten different clients as well means that every data matters for the increase in transmitted data, not only unique data as previously seen in the accuracy results. This may happen because the amount of parameters to update for clients with models that were built with more data are larger than similar ones built with less data.

Figure 5.6 further demonstrates this point, breaking down the original plot in Figure 5.4, showing a scatter plot for both data distributions with eight clients involved. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. Additionally, green dots represent green data distribution experiments, while red dots represent red data distribution experiments. In this plot it is possible to see the aforementioned relationship between transmitted data and total amount of data in the system, as the red experiment always transmits more bytes and packets than its green counterpart since it contains more data batches per client.

The decision for the number of clients involved being 8 is random – it is merely to pin a variable to prove the affirmation with a simpler plot. This tendency can be analyzed for any number of clients.

Figure 5.6: Plot for the total amount of bytes and packets transmitted by the server applications with green and red data distributions with 8 clients involved



Source: Image provided by the author

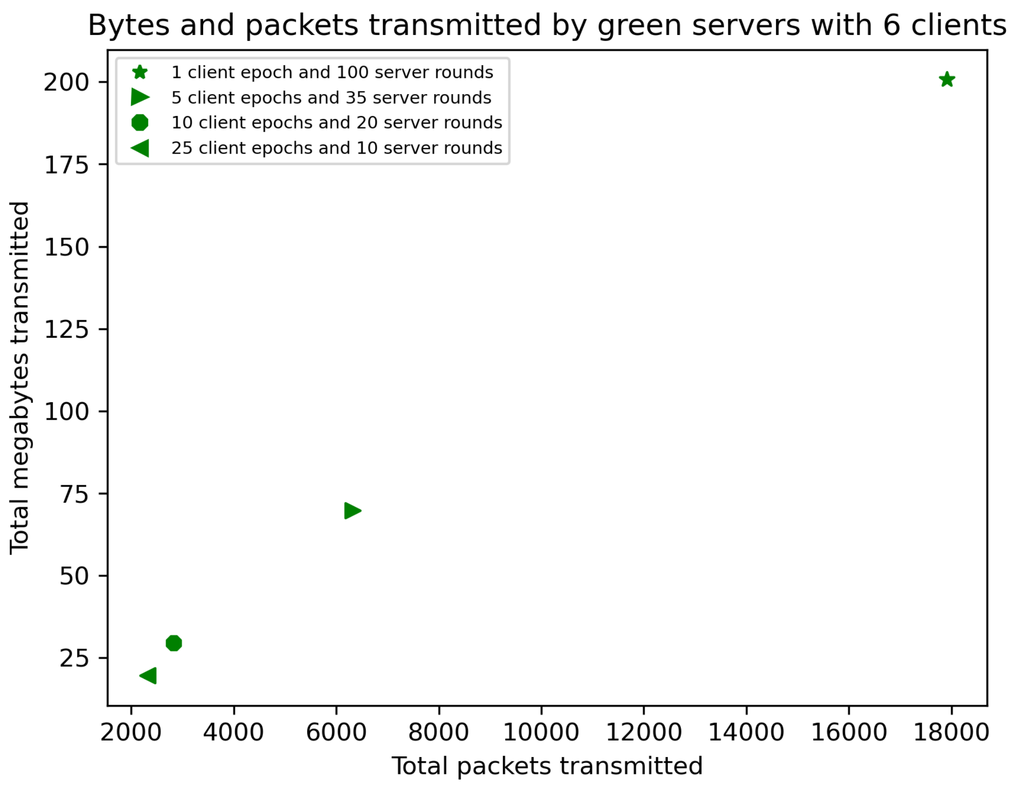
5.4.3 Effects of Server Rounds

A higher amount of server rounds accounts to a higher amount of data transmitted by the server. The impact client epochs may have for server transmission data will be mostly indirectly related, as the client epochs will affect the local model, which may affect the server data that is transmitted. Server rounds, however, is the moment the server updates the parameters in the clients, so there is a greater impact in the total amount of transmitted data.

Figure 5.7 further demonstrates this point, breaking down the original plot in Figure 5.4, showing a scatter plot for the green data distribution with six clients involved for varying client epochs and server rounds configurations. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. Further client epochs and server rounds legends can be seen in the upper left corner.

The decision for the number of clients involved being 6, as well as the decision to use green data distributions, is random – it is merely to pin a variable to prove the affirmation with a simpler plot. This tendency can be analyzed for any number of clients involved, both green and red. In this plot it is possible to see more clearly the aforementioned linear relationship between transmitted data and number of server rounds, as a virtually straight line can be crossed between the four points.

Figure 5.7: Plot for the total amount of bytes and packets transmitted by the server applications with green data distribution with 6 clients involved



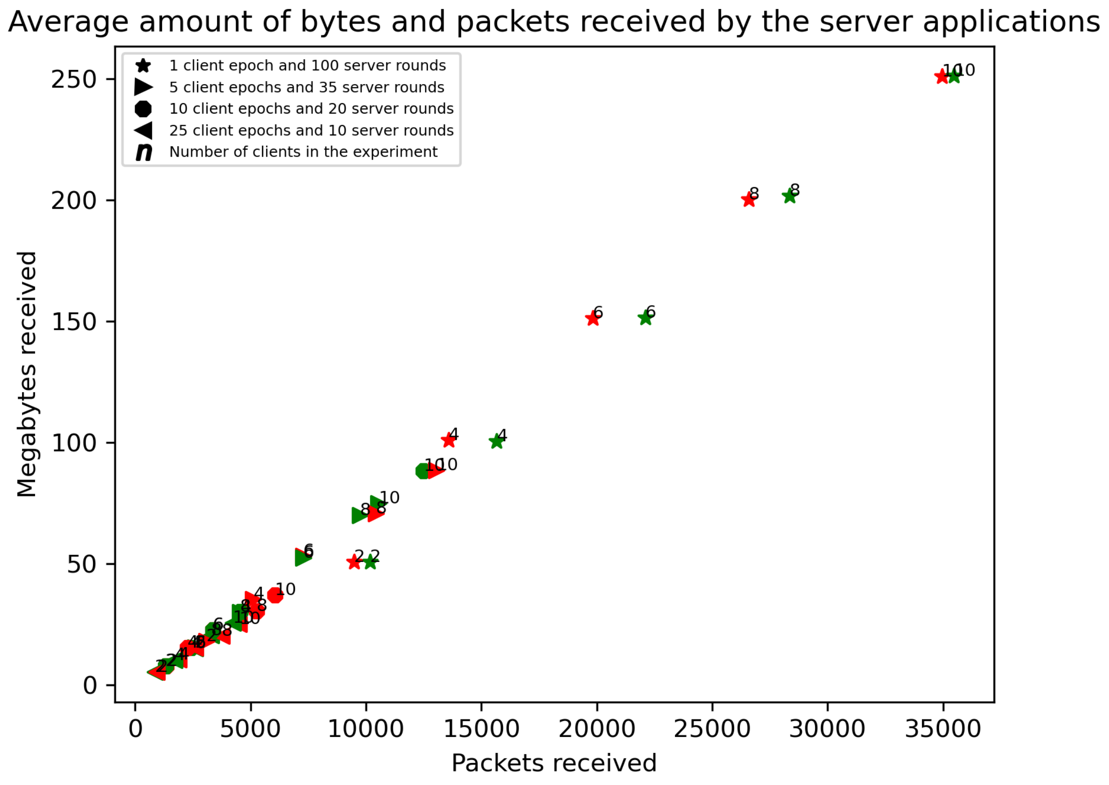
Source: Image provided by the author

5.5 Received Data

Similar to transmitted data, a second analysis for network stats will be done by understanding how the server receives data, i.e., packets and bytes, and if it is possible to understand a trend while changing number of clients, epochs, and amount of data in the system.

Figure 5.8 shows a scatter plot for the many different server layouts that were run. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. The different epochs and server round layouts are organized as follows: 1 client epoch with 100 server rounds is represented by a star. 5 client epochs with 35 server rounds is represented by an arrow pointing right. 10 client epochs with 20 server rounds is represented by an octagon. 25 client epochs with 10 server rounds is represented by an arrow pointing left. Green data distributions are in the color green, while red data distributions are in the color red. Finally, on top of each scatter, there is the amount of clients involved in the experiment.

Figure 5.8: Plot for the total amount of bytes and packets received by the server applications

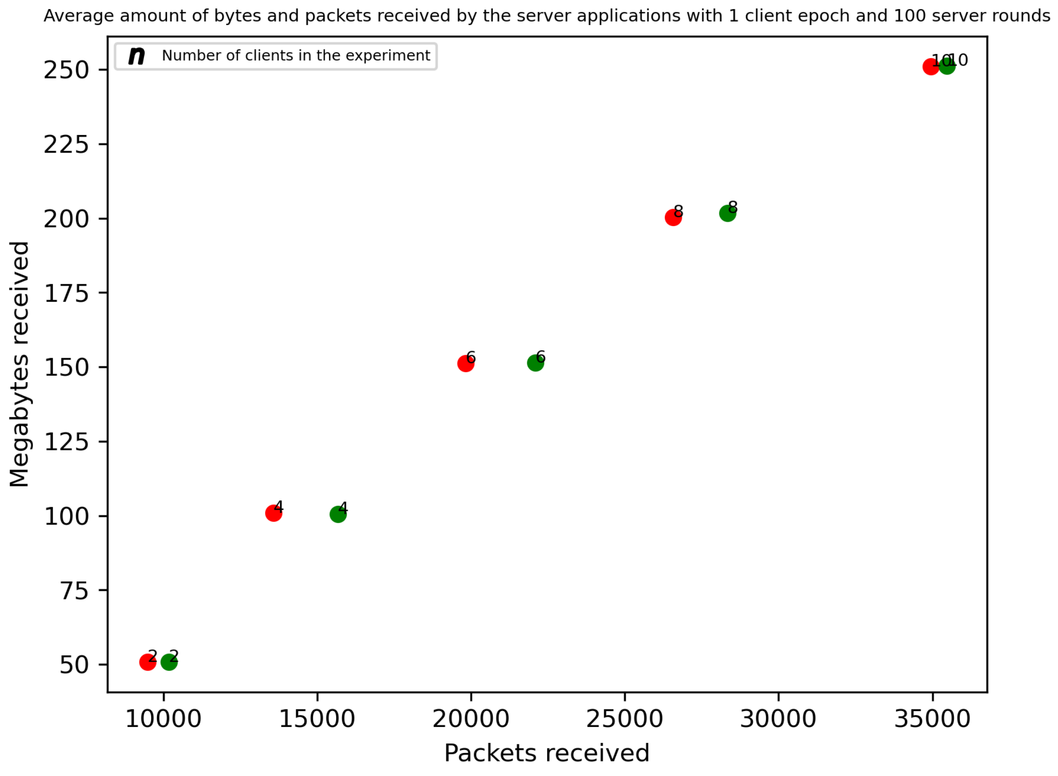


Source: Image provided by the author

Received data follows a similar pattern that transmitted does. Data received scales linearly with the number of clients involved in the experiment, as the more clients, the more parameters the server receives. Similarly, the more server rounds in the experiment, the more are the times where the server receives the data from the clients in order to average the data and update the clients.

Figure 5.9 shows data received more clearly, by showing only for red and green data distributions with 1 client epoch and 100 server rounds. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. Green data distributions are in the color green, while red data distributions are in the color red. On top of each scatter, there is the amount of clients involved in the experiment.

Figure 5.9: Plot for the total amount of bytes and packets received by the server applications with 1 client epoch and 100 server rounds



Source: Image provided by the author

The decision for the number of client epochs and server rounds being 1 and 100 is random – it is merely to pin a variable to prove the affirmation with a simpler plot. This tendency can be analyzed for any layout of client epochs and server rounds.

The only difference from the data transmitted is that the amount of data in the clients has no effect in the bytes received by the server, only slightly in the packets received. This can be seen by noticing that, for each red marker, there is a green one right next to it with the exact same configuration. This discrepancy in packets received may be related to how the clients batch their data to send to the server, although it is curious as to why there is no effect in received data, only when transmitting afterwards.

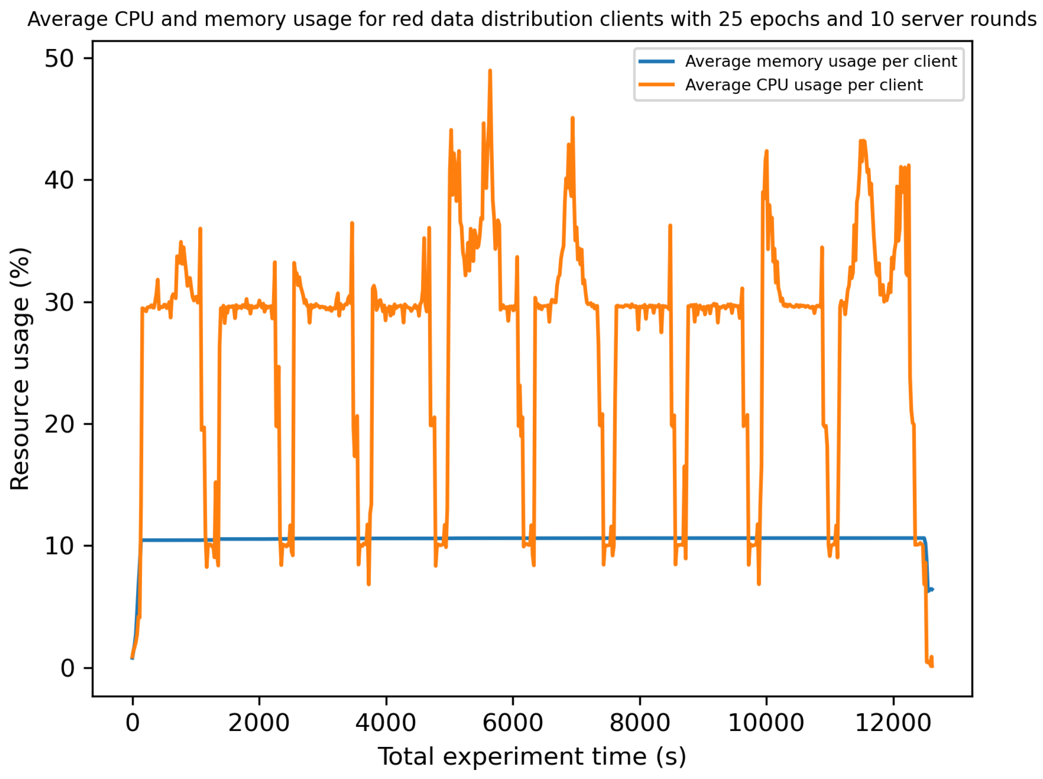
5.6 CPU and Memory

To better understand CPU and memory usage on the client side, the chosen experiment is the red data distribution with ten clients, 25 client epochs and 10 server rounds. The client epochs and server round configuration has been chosen as it contains the smaller numbers, making the plot more easily understandable. Red data distribution and ten clients were chosen as it contains the more available data in the experiment, so theoretically it is the experiment which stressed the applications the most regarding resource usage.

Figure 5.10 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates average resource usage in percentage. There is an orange line, which represents the average CPU usage for the clients of the given experiment layout; and a blue line which represents the average memory usage for the clients of the given experiment layout.

The CPU usage in the plot varies from 10% to 30%, with some spikes going up to 50%. These spikes might have been caused by some external factor such as the server being overloaded, or some other outlier being accounted for in the averages. However, there is a tendency of periods in the CPU usage that lasts for approximately 1000s that the usage is increased, then it drops to 10%, then repeat. There are ten of these periods in the experiment, which is the number of server rounds. When the CPU is 30%, the client is training the local model; when it is 10%, it is sending or receiving data from the server. The memory, on the other hand, demonstrates a flat usage throughout the experiment, being close to 10% from start to finish.

Figure 5.10: Plot for the average CPU and memory used by clients with red data distribution with 25 client epochs and 10 server rounds



Source: Image provided by the author

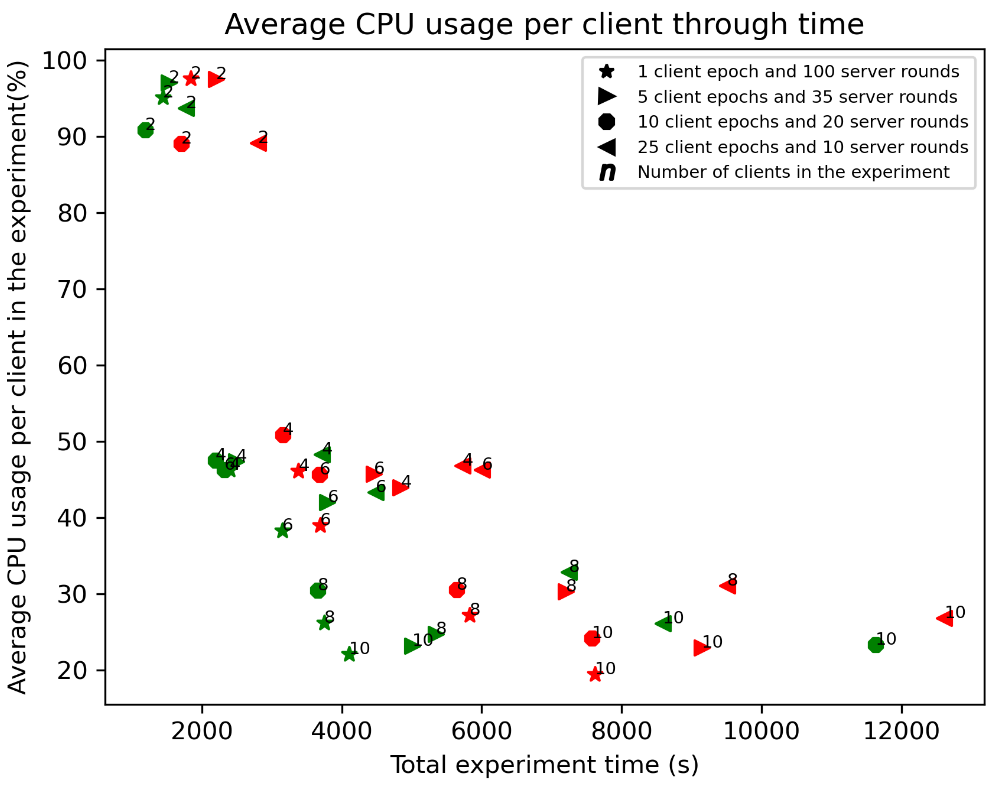
5.6.1 Client CPU Usage

To better understand tendencies regarding client CPU usage, a deeper dive will be taken taking into account every experiment scenario.

Figure 5.11 shows a scatter plot with the proposed experiment. The X axis indicates the total time in seconds that the experiment took, while the Y axis indicates average CPU used in percentage throughout the experiment. The different epochs and server round layouts are organized as follows: 1 client epoch with 100 server rounds is represented by a star. 5 client epochs with 35 server rounds is represented by an arrow pointing right. 10 client epochs with 20 server rounds is represented by an octagon. 25 client epochs with 10 server rounds is represented by an arrow pointing left. Green data distributions are in the color green, while red data distributions are in the color red. Finally, on top of each scatter, there is the amount of clients involved in the experiment.

The plot indicates a correlation between clients involved in the experiment, CPU usage and experiment time. Experiments with fewer clients are able to utilize more CPU

Figure 5.11: Plot for the average CPU used by clients according to total experiment time



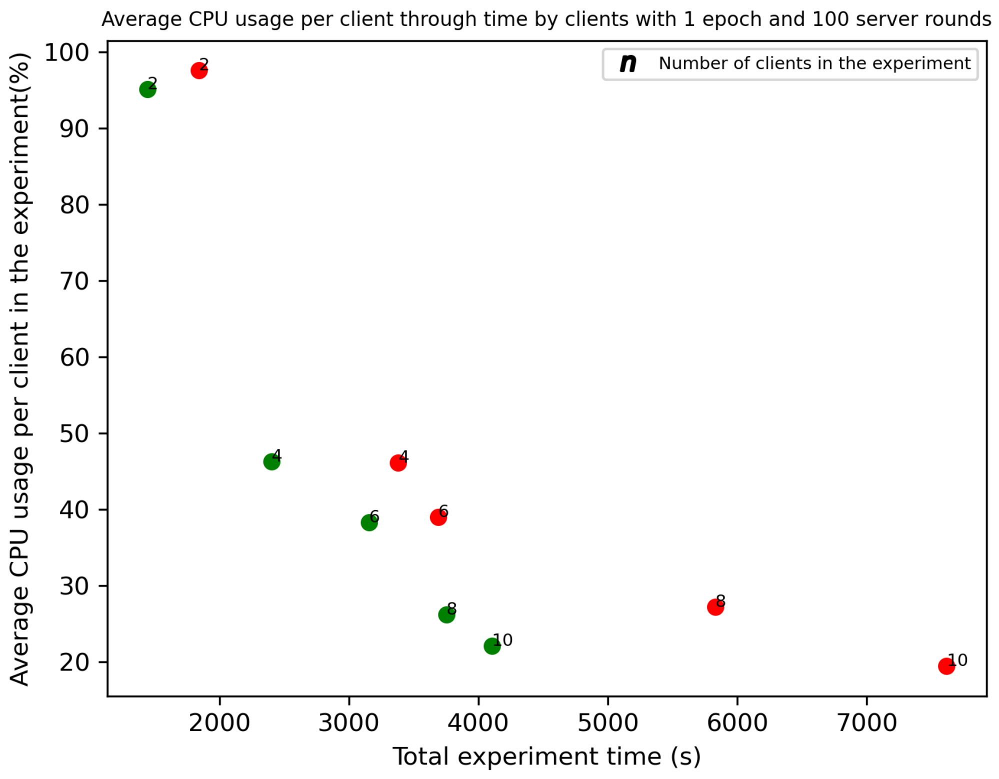
Source: Image provided by the author

power, allowing the experiment to finish faster. This can be seen, for example, looking at the cluster in the top left corner, all of them being experiments with two clients. The faster to finish, and using nearly 100% CPU. From there, around the 50% mark is the experiments with four clients. Bottom right shows mainly the experiments with ten clients, taking longer to finish and using less CPU. This makes clear the relationship between CPU usage and time to finish the experiment.

Figure 5.12 shows this relationship more clearly by showing only experiments with 1 client epoch and 100 server rounds. The X axis indicates the total time in seconds that the experiment took, while the Y axis indicates average CPU used in percentage throughout the experiment. Green data distributions are in the color green, while red data distributions are in the color red. On top of each marker, there is the amount of clients involved in the experiment.

The tendency appears to be logarithmic curve: if there was an experiment with only one client, it would use 200% of the CPU and finish in half the time. Likewise, if there was an experiment with eleven clients, it wouldn't be that much different from the experiment with ten.

Figure 5.12: Plot for the average CPU used by clients according to total experiment time for clients with 1 epoch and 100 server rounds



Source: Image provided by the author

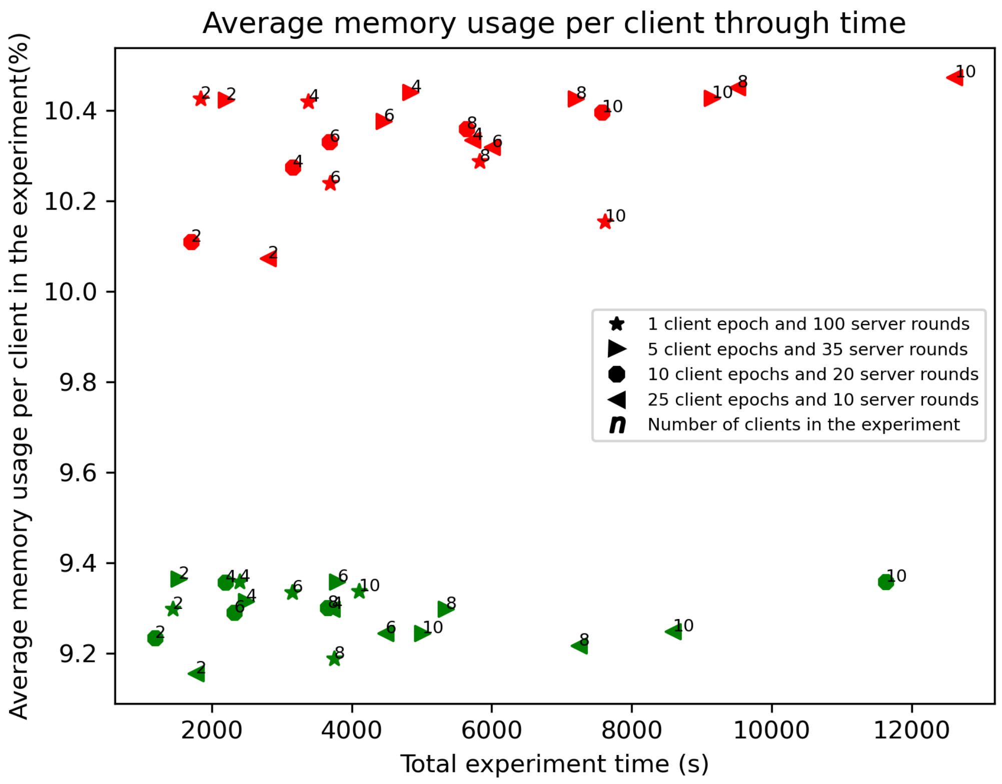
As noted in Figure 5.10, the CPU is mainly used in the period where the clients are training their models locally. With more available CPU power, the clients are able to finish their local training faster, making the client-server exchange faster and ending the experiment sooner.

5.6.2 Client Memory Usage

To better understand tendencies regarding client memory usage, a deeper dive will be taken taking into account every experiment scenario.

Figure 5.13 shows a scatter plot with the proposed experiment. The X axis indicates the total time in seconds that the experiment took, while the Y axis indicates average memory used in percentage throughout the experiment. The different epochs and server round layouts are organized as follows: 1 client epoch with 100 server rounds is represented by a star. 5 client epochs with 35 server rounds is represented by an arrow pointing right. 10 client epochs with 20 server rounds is represented by an octagon. 25 client epochs with 10 server rounds is represented by an arrow pointing left. Green data distributions are in the color green, while red data distributions are in the color red. Finally, on top of each marker, there is the amount of clients involved in the experiment.

Figure 5.13: Plot for the average memory used by clients according to total experiment time



Source: Image provided by the author

For memory, there is a clear difference between red and green experiments. As the red clients carry more data, they require more memory to perform, with a difference of up to 10% to their green counterparts. However, apart from that, there is not much to be taken from the plot. The difference in experiment time for more clients is due to the CPU, as previously observed, so it reflects in memory but is not directly related to it.

6 CONCLUSION

This work achieves the intended proposal by laying the foundation for federated learning experiments and further understanding how the system scales when adding more clients in regards to computing resources, and how this affects the overall system accuracy. The findings can be useful to further improve federated learning technologies, using the knowledge on the application behavior to build, for instance, an edge-cloud orchestrator specifically for federated learning use cases. Moreover, it helps the design of federated learning systems that may scale to multiple clients.

The main takeaway point is that the system allows for horizontal scaling by adding more client nodes in order to utilize more unique data and achieve higher accuracy numbers. By adding more nodes, however, the load in the server application will keep scaling linearly, to the point where it may become the bottleneck for the system. Server rounds, while previously mentioned to be more effective to increase accuracy, not only will even further increase the server load as the server will have to average and update parameters more frequently, but it will also make so that the clients will have to send their data more frequently as well. The latter can be an issue especially for edge devices with limited connectivity, because it cannot be guaranteed continuous access internet access in order to match the frequency the server will expect.

All of the above indicate that, in order to have good federated learning use cases, first a good strategy to train the local client model is required. For scenarios with a huge amount of clients, placing some of the load on the clients and improving their local model training allows the clients to make fewer connections, even if it may not be optimal considering only the accuracy as previously analyzed. This also alleviates the load on the server that having these many clients involved in the system will cause.

This trade-off, however, has to be considered on a case by case basis. If the clients are guaranteed to always have good internet connectivity, for instance, and the cost of maintaining a server that is powerful enough to scale as the number of clients increase is not an issue, the load can be better placed in the server, allowing the clients to train less rounds themselves. This option may also make sense to federated learning scenario with fewer number of clients, e.g., the clients being a few different sensors in a farm.

This work also opens up opportunities to further experiment with federated learning scenarios. All of the code is publicly available and has been developed with extensibility in mind, with well defined methods while prioritizing clean code, modularity, and documentation as much as the time constraints allowed. The observability and plotting modules have been thoroughly developed with the possibility of further advancement in mind.

Regarding limited connectivity scenarios, one path would be to see how the system would behave for clients with slow internet connection, and that may lose internet connection between training. Fault tolerance, especially in the server, is an important aspect, since as it is right now the server application is a single point of failure in the system, and if one of the clients fail the experiment stops until the connection is retrieved. An optimization idea is for the server to be a distributed system that retrieves client information based on geographic proximity to alleviate network calls for clients while avoiding the single point of failure, then averages the data.

Regarding client configurations, an interesting study is to create heterogeneous clients and see the effects on accuracy and resource usage. In the real world, the devices will hardly have the same specifications, so it is important to properly understand if, for instance, a device much slower than the rest becomes a bottleneck for the experiment running time. Utilizing datasets with real world data, as well as improving the model, may lead to additional findings regarding system accuracy. Additionally, scaling to a larger amount of clients, e.g., thousands, may also provide more data to analyze federated leraning in large scale distributed systems.

REFERENCES

AVINO, G. et al. **Characterizing Docker Overhead in Mobile Edge Computing Scenarios**. arXiv, 2018. Available from Internet: <https://arxiv.org/abs/1801.08843>.

BARTHéLEMY, J. et al. Edge-computing video analytics for real-time traffic monitoring in a smart city. **Sensors**, v. 19, n. 9, 2019. ISSN 1424-8220.

BEUTEL, D. J. et al. **Flower: A Friendly Federated Learning Research Framework**. 2021.

CISCO. **Cisco Annual Internet Report (2018–2023) White Paper**. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (accessed October 16th, 2021), 2020.

DENG, S. et al. Edge intelligence: The confluence of edge computing and artificial intelligence. **IEEE Internet of Things Journal**, v. 7, n. 8, p. 7457–7469, 2020.

DURRANT, A. et al. The role of cross-silo federated learning in facilitating data sharing in the agri-food sector. **arXiv preprint arXiv:2104.07468**, 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.

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

HE, C. et al. **FedML: A Research Library and Benchmark for Federated Machine Learning**. 2020.

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

JI, S. et al. Learning private neural language modeling with attentive aggregation. **2019 International Joint Conference on Neural Networks (IJCNN)**, IEEE, Jul 2019. Available from Internet: <http://dx.doi.org/10.1109/IJCNN.2019.8852464>.

KAIROUZ, P. et al. **Advances and Open Problems in Federated Learning**. 2021.

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.

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**. [S.l.], 2016.

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.

RIEKE, N. et al. The future of digital health with federated learning. **npj Digital Medicine**, Springer Science and Business Media LLC, v. 3, n. 1, Sep 2020. ISSN 2398-6352. Available from Internet: <http://dx.doi.org/10.1038/s41746-020-00323-1>.

WANG, H. et al. Optimizing federated learning on non-iid data with reinforcement learning. In: **IEEE INFOCOM 2020 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2020. p. 1698–1707.

XU, J. et al. Federated learning for healthcare informatics. **Journal of Healthcare Informatics Research**, Springer, v. 5, n. 1, p. 1–19, 2021.

ZHOU, Z. et al. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. **Proceedings of the IEEE**, v. 107, n. 8, p. 1738–1762, 2019.

ZHU, H. et al. **Federated Learning on Non-IID Data: A Survey**. 2021.

APPENDIX A — WRGS PAPER SUBMISSION

The following appended document is currently a working paper, submitted to the XXVII Workshop de Gerência e Operação de Redes e Serviços (WGRS) that has been pre-approved for publication.

WGRS is part of the SBRC, Brazilian Symposium on Computer Networks and Distributed Systems, which is an annual event held by the Brazilian Computer Society (SBC) and the National Laboratory of Computer Networks (LARC). WGRS has the goal to showcase research and relevant activities related to the management and operation of networks and services.

Analyzing Federated Learning Performance in Distributed Edge Scenarios

Fernando Remde[1], **Juliano Wickboldt**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves, 9500 - Porto Alegre, RS - Brasil

`{ffremde, jwickboldt}@inf.ufrgs.br`

Abstract. *Federated Learning is a machine learning paradigm where many clients cooperatively train a single centralized model while keeping their data private and decentralized. This novel paradigm imposes many challenges, such as dealing with data that is not independent and identically distributed, spread among multiple clients that are not synchronized and may have limited computing power. These clients are often edge devices such as smartphones and sensors, which form a system that is heterogeneous, highly distributed by nature and difficult to manage. This work proposes an architecture for running federated learning experiments in a distributed edge-like environment. Based on this architecture, a set of experiments are conducted to analyze how the overall system performance is affected by different configuration parameters and varied number of connected clients.*

1. Introduction

Federated Learning is a solution proposed by [McMahan et al. 2016] in order to train machine learning models while keeping the data private and decentralized. It works by having different clients training their own models with their own data, then averaging these trained models in a single centralized server, creating a global model. These clients can be consisted of smartphones, a single sensor, or any IoT device with internet connection such as a smart TV or a smart car.

The clients train their models with their own data and, after finished, the models are uploaded to a central server, where all the client models are weighted and averaged, converging to a single final model made from the uploaded models. While regular centralized machine learning may outperform federated learning in terms of accuracy [Nilsson et al. 2018], it requires the entire data set to be public. Federated learning allows large scale data sets to be used for training models while keeping the data private to each client.

Federated learning has been greatly enabled by the vast advances and abundance of IoT devices. According to [Cisco 2020], the number of devices connected to IP networks will be more than three times the global population by 2023, there will be 3.6 networked devices per capita by 2023, up from 2.4 networked devices per capita in 2018, and there will be 29.3 billion networked devices by 2023, up from 18.4 billion in 2018. This represents a rapid increase of potential federated learning use cases, considering the plenitude of client data to be trained in order to produce high accuracy machine learning models.

This work proposes an architecture that is able to run federated learning algorithms in a distributed environment, where the clients are machines with limited power, similar to edge devices. Then, on top of this architecture, a set of experiments have been conducted with different configurations of data distribution for a varying number of clients, while also changing parameters for both the client and the server. Collected data has been analyzed in order to understand how to optimize for model accuracy while minimizing computing resource usage on client-side. This analysis can be useful for further development in federated learning technologies, such as orchestration between edge devices and cloud for federated learning use cases.

The remainder of this work is organized as follows. In Section 2 an overview for edge cloud orchestration and federated learning is presented. In Section 3 the conceptual architecture of the proposed solution is presented. Section 4 approaches the layout of the experiments and presents the obtained results. Section 5 concludes the paper outlining opportunities for future research.

2. Edge Cloud Orchestration and Federated Learning

This section approaches the main topics of this work, providing an overview of the current state of the art of edge computing and federated learning.

2.1. Edge Computing

The need for low latency computing along the rapid advancements in telecommunication services motivated the edge computing paradigm. In edge computing, instead of having the computing resources centralized in a data center – a cloud –, the idea is to distribute these resources in devices – the edge – closer to the final user, thus allowing lower latency and faster connections. These edge devices can be any device with Internet access such as smartphones, smart cars, or other IoT devices.

Edge computing use cases include autonomous driving [Liu et al. 2019], where edge devices need to process a large amount of data from different sensors at high speed in real time in order to guarantee the safety of the drivers; and smart city traffic monitoring [Barthélemy et al. 2019], where 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.

More recently, many advances connecting edge computing to artificial intelligence were made. [Zhou et al. 2019] define Edge Intelligence as the union between AI and edge computing, an opportunity that rose in virtue of the abundance of devices connected to the internet that generate huge amounts of data on a daily basis. Edge Intelligence aims to capitalize on this data to train machine learning models, using concepts such as Deep Learning and Federated Learning.

Additionally, [Deng et al. 2020] further expand on Edge Intelligence and propose a conceptual difference between artificial intelligence for edge and artificial intelligence on edge. The former encompasses intelligence-enabled edge computing that provides solutions to edge computing problems by utilizing artificial intelligence, while the latter encompasses how to run artificial intelligence models on edge devices, extracting insights from its distributed data nature. For this work, we are more interested in AI on edge.

2.2. Federated Learning

Federated learning is a decentralized form of machine learning used to train models at scale while allowing the user data to be private. Federated learning was first introduced by Google [McMahan et al. 2016], which provided the first definition of federated learning, as well as the Federated Optimization [Konečný et al. 2016] approach to further improve these federated algorithms. Google also explains in further detail the concept of federated learning in the Federated Learning: Collaborative Machine Learning without Centralized Training Data blog post [Google 2017], stating the usage to predict keyboard words as seen in [Hard et al. 2018] and planning to also use federated learning for photo ranking and further improving language models.

In order to properly define and further advance in federated learning subjects, the FedML Research Library and Benchmark [He et al. 2020] has been proposed to facilitate federated algorithm development and performance comparison. FedML provides an open-source framework that allows the development and evaluation of novel federated algorithms. Similarly, the Flower Learning Research Framework [Beutel et al. 2021] also provides heterogeneous environments that allow experimentation with heterogeneous data and algorithms. FlowerML is used as the federated learning infrastructure facilitator for this work.

Despite the novelty and the myriad of challenges, federated learning adoption is rising. Besides Google's keyboard, [Ji et al. 2019] proposes a novel optimized model aggregation for keyboard suggestion that considers each client contribution to the global model and weighs them instead of simply averaging. Federated learning is also key to fully utilize machine learning capabilities in scenarios where the data cannot be shared due to sensitivity such as for the health industry [Rieke et al. 2020].

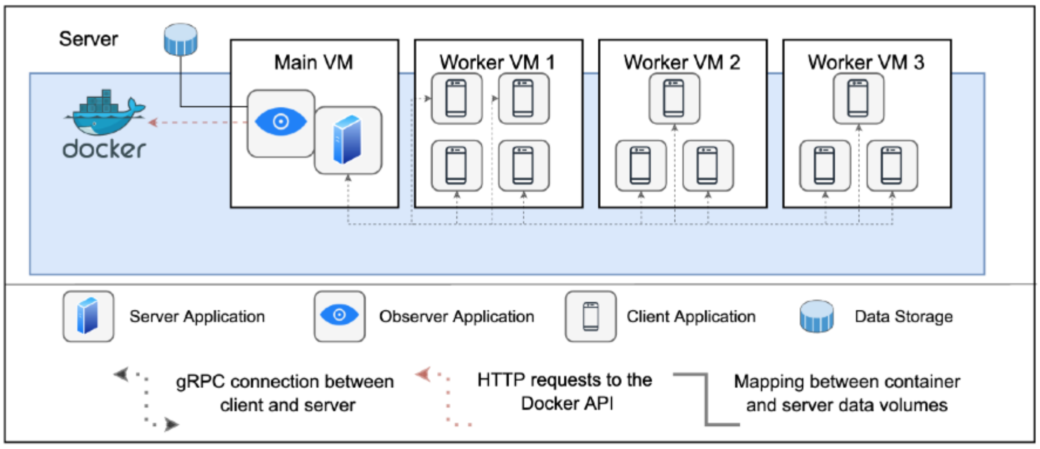
Google, along researchers from many universities, at the workshop on federated learning and analytics, states that federated learning is inherently interdisciplinary [Kairouz et al. 2021], encompassing techniques and methods from other fields such as cryptography, security, differential privacy, fairness, compressed sensing, systems, information theory, and more, requiring a collaborative effort in order to further advance the subject. For this work, we are notably interested in the intersection that federated learning has with edge computing.

3. Solution

Figure 1 shows the full solution diagram in a scenario running with ten different clients. We can see the underlying infrastructure, i.e., the server and the VMs, and on top of that the docker containers running each application: the server, the client, and the observer. Further subsections will further expand on each part of this diagram. First, the underlying infrastructure where the solution is running is presented. Then, the solution architecture and its components are presented. Finally, the data that will be used for the experiments is covered.

3.1. Infrastructure

To achieve the desired infrastructure, multiple machines are required. For this work, we are interested in both how the centralized server and how the distributed clients function.



Server
docker
Main VM
Worker VM 1
Worker VM 2
Worker VM 3
Server Application
Observer Application
Client Application
Data Storage
gRPC connection between client and server
HTTP requests to the Docker API
Mapping between container and server data volumes

Figure 1. High-level diagram of the implemented solution when running with ten clients

The centralized server, in an ideal federated learning scenario, runs on a traditional cloud; a powerful computer capable of more complex operations. To achieve this, a VM *main* runs exclusively the server and observer applications and is twice as powerful as the VMs running the clients, having 4GB RAM.

For the clients, as we are simulating an edge environment, we will utilize VMs with limited power which run more than one client at once. We will call them *workers*; machines with 2GB RAM that will run from zero to four federated clients at once depending on the experiment.

3.2. Architecture

The solution has three architectural components. These are the server, the client, and the observer services, which are all containerized services that can run in any underlying infrastructure. Each client trains a local model and uploads this model to the server which averages and returns the updated parameters. Meanwhile, the observer retrieves and persists to a volume metrics of every running container besides itself.

Figure 2 shows the high-level architecture of the solution, highlighting each of the aforementioned architectural components and their dependencies. The microservices with their respective applications and dependencies are individually defined and deployed together in a Docker Swarm using a *docker-compose* file. To the right in the figure, each dependency for the Docker containers is explicit.

The server is a containerized Python application with the FlowerML server framework as dependency. The server is responsible for receiving the models being trained by the clients, averaging the received parameters, then updating the clients' models with the averaged parameters. The average is done by the server using Federated Average – FedAvg –, which is a standard federated learning averaging method, also used by Google Keyboard, which simply averages the parameters received by the clients without attributing weights. The server connects to the clients through a gRPC connection and performs a pre-defined number of averaging rounds.

The client is a containerized Python application with the FlowerML client frame-

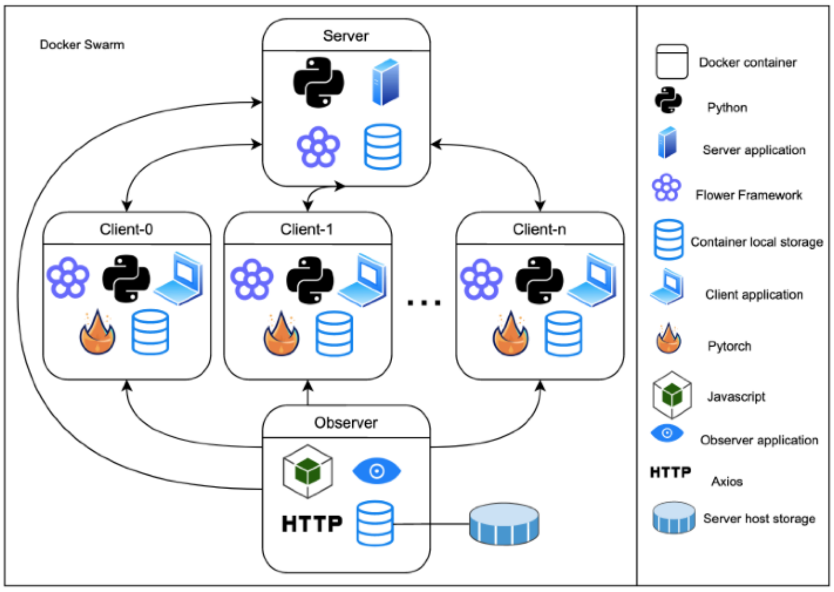


Figure 2. Solution architecture with the services and their dependencies

work, Pytorch[1], and the data that will be used as dependencies. The client is responsible for training a local model with local data in a convolution neural network, then uploading the model to a server which in turn returns updated values for the sent parameters. The client connects to the server through a gRPC connection and performs a pre-defined number of epochs, 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 observer is a containerized JavaScript application with axios[2] as dependency. The observer is responsible for logging information about the other containers – namely the server and clients. The observer is the only container which does not stop unless when forced to do so; it keeps continuously retrieving data of every other container in the Swarm and persisting the data to the Docker volume when these other containers are stopped, allowing for multiple runs of the same experiment without having to worry about reboots. The observer has a volume which is mapped directly to a folder in the main VM. In practice, every data that is persisted to an observer container is also persisted to the host VM.

To collect containers stats, the observer leverages the Docker stats API[3], with the server accuracy being a separate case. As the accuracy is a metric created by the server application, to retrieve the accuracy the observer looks at the container logs API instead of using the stats API. Every request is done via HTTP requests do the Docker APIs, being completely decoupled from the other services logic. The stats retrieved by the Docker API include, but are not limited to, network information such as bytes received

[1]An open source machine learning framework: https://pytorch.org/ (Accessed April 21st, 2022)

[2]Promise based HTTP client for the browser and node.js: https://axios-http.com/ (Accessed April 18th, 2022)

[3]https://docs.docker.com/engine/api/v1.21/ (accessed February 22nd, 2022)

and transmitted, and memory and CPU usage information. The full code for the observer can be found in the observability module in the Github repository[4].

3.3. Data

The dataset used for the experiments is the CIFAR dataset [Krizhevsky et al. 2009]. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The original dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

In order to properly divide this data between multiple clients, the five original data batches have been divided into 500. This enables a higher level of granularity for testing federated learning scenarios, as we are able to precisely state a percentage of data overlap between clients. Additionally, for scenarios of more than five clients, the original five batches wouldn't be sufficient for having every client with a different dataset.

4. Experiments

4.1. Layout

Each experiment has been ran at least two times, with some being ran for up to sixteen times. The results have been averaged to try to minimize any randomness. Every experiment utilizes homogeneous client configurations, which means that every client involved in a given experiment is exactly the same, except for the data that it contains and the host in which it runs. Since every worker VM has the same specification, we can assume that only the data is different between clients for any given experiment.

The data distribution has been separated into color codes: green and red. Both green and red will have available the 500 batches mentioned in the previous section. What differs between them is the amount of data each client will have. Green clients have 50 batches, which means that for the experiment with 10 clients, every batch will be used without any overlap. Red clients have 100 batches, so there will be data overlap between clients, with 50% of the data being replicated for the 10 clients participating in the experiment.

The number of epochs of the client is how many times the whole dataset is passed in the CNN. Four different configurations will be tested: 1, 5, 10, and 25 epochs. Additionally, each number of epochs configuration will have a matching number of server rounds. Respectively, the maximum number of rounds is 100, 35, 20, and 10. This matching is mainly to guarantee that every experiment converges, but also that the experiments do not run for over one hour due to time constraints.

There are two different data distributions, five number of clients possibilities, and four layouts for number of rounds and epochs. In total, 40 different experiments layouts have been ran, with 270 total ran experiments as most layouts were executed multiple times, generating over 1.25GB of plain text files containing observability logs.

[4]https://github.com/remde/federated-learning/tree/main/observability

4.2. System Accuracy

The first question to be answered regarding server is if data replication impacts the accuracy of the system. To answer this, a plot of accuracy through time has been done for both red and green data distributions with ten clients. Green will have the 500 batches with 50 unique batches per client, totaling the 500; and red, while having the same 500 batches available, will have 100 batches per client, with 50% of them being overlap data.

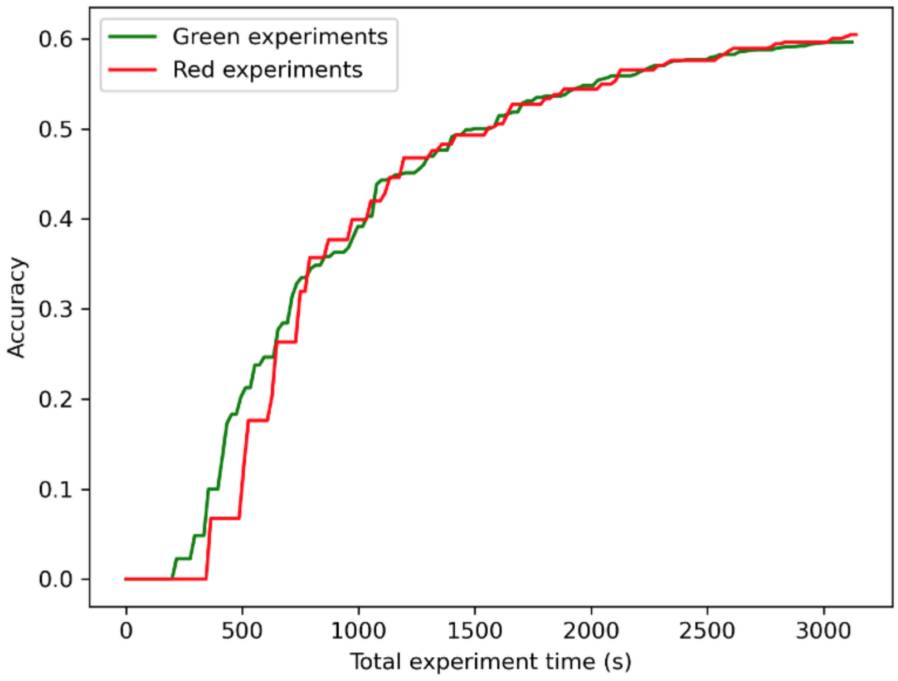


Figure 3. Average accuracy with ten clients through time

Figure 3 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates the accuracy. There is a red line, which represents the averages for the red experiments with ten clients; and a green line which represents the averages for the green experiments with ten clients. Even with twice the number of batches, there wasn't significant difference for the red experiments, not even in experiment time. The curves are essentially the same, indicating that what matters to the total accuracy is the unique data spread among clients.

Another experiment done to understand impact in accuracy is according to client epochs and server rounds. To accomplish this, using exclusively experiments with ten clients, red and green were averaged and split into four categories, one for each client epoch and server round configuration. Figure 4 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates the accuracy.

First thing that can be noted is that client epoch per client and server rounds are not equal; at least not in a one-to-one relation. We cannot simplify the rounds as a multiplication of client epochs and server rounds. If they had the same effect on accuracy and experiment time, the plots would indicate the green line as having the same result as the blue one in half the time, for instance.

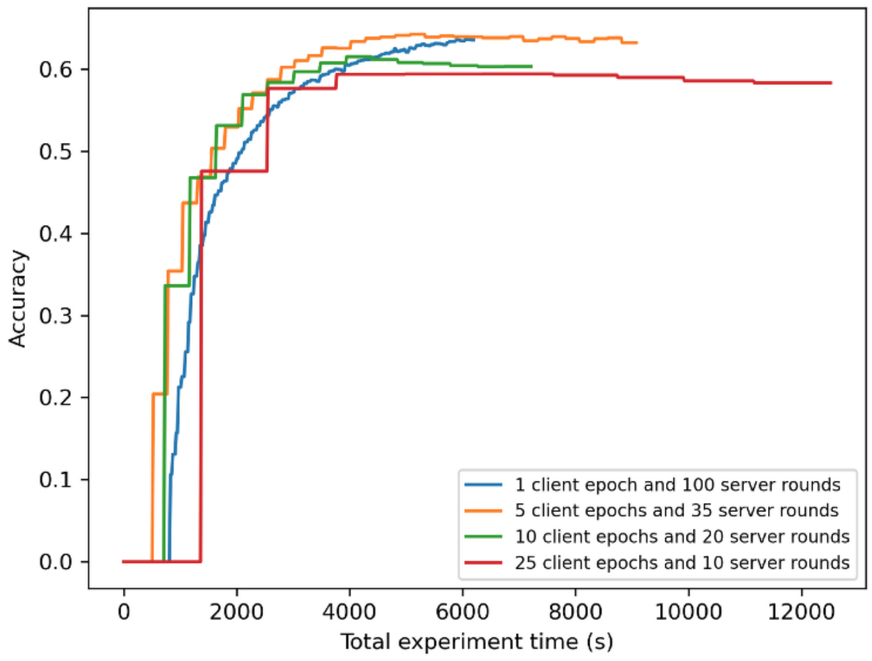


Figure 4. Average accuracy with ten clients split by client epoch and server round configurations through time

To achieve the maximum accuracy, and disregarding other aspects that may come with this decision, if choosing between client epochs and server rounds, one should increase server rounds as it is the more effective option.

4.3. Transmitted Data

The first analysis for network stats will be done by understanding how the server transmits data, i.e., packets and bytes, and if it is possible to understand a trend while changing number of clients, epochs, and amount of data in the system.

Figure 5 shows a scatter plot for the many different server layouts that were run. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. This plot, while containing a lot of information, can be broken down and analyzed. Three different aspects can be inferred that contribute to higher data transmission:

1. The more clients involved in the experiment, the more total data is transmitted. To an experiment with the same layout for data distribution, client epochs and server rounds, there will be a linear proportion relation between the number of clients involved in the experiment and the total amount of transmitted data. This happens because, with more clients involved, the server has more clients to update parameters after having them averaged, requiring a higher amount of network usage.
2. The higher the total amount of data involved in the system, the higher is the total transmitted data by the server. This can be seen because red experiments are transmitting more data than their counterpart green experiments. Having this difference in the experiment with ten different clients as well means that every data

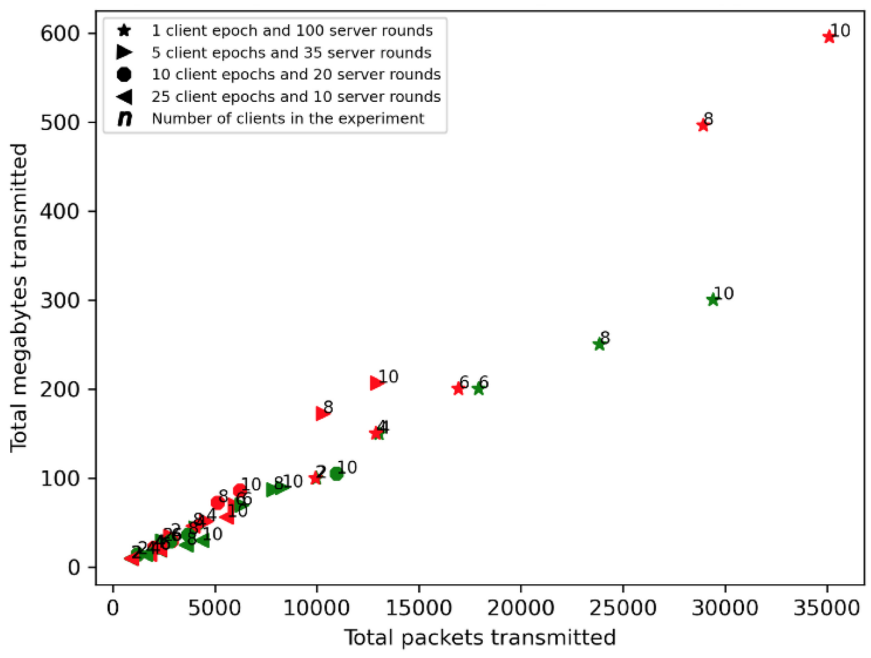


Figure 5. Total amount of bytes and packets transmitted by the server application

matters for the increase in transmitted data, not only unique data as previously seen in the accuracy results. This may happen because the amount of parameters to update for clients with models that were built with more data are larger than similar ones built with less data.

3. A higher amount of server rounds accounts to a higher amount of data transmitted by the server. The impact client epochs may have for server transmission data will be mostly indirectly related, as the client epochs will affect the local model, which may affect the server data that is transmitted. Server rounds, however, is the moment the server updates the parameters in the clients, so there is a greater impact in the total amount of transmitted data.

4.4. Received Data

Fig 6 shows a scatter plot for the many different server layouts that were run. The X axis indicates the total amount of bytes transmitted, while the Y axis indicates the total amount of packets transmitted. Received data follows a similar pattern that transmitted does. Data received scales linearly with the number of clients involved in the experiment, as the more clients, the more parameters the server receives. Similarly, the more server rounds in the experiment, the more are the times where the server receives the data from the clients in order to average the data and update the clients.

The only difference from the data transmitted is that the amount of data in the clients has no effect in the bytes received by the server, only slightly in the packets received. This can be seen by noticing that, for each green marker, there is a green one right next to it with the exact same configuration. This discrepancy in packets received may be related to how the clients batch their data to send to the server, although it is curious as to why there is no effect in received data, only when transmitting afterwards.

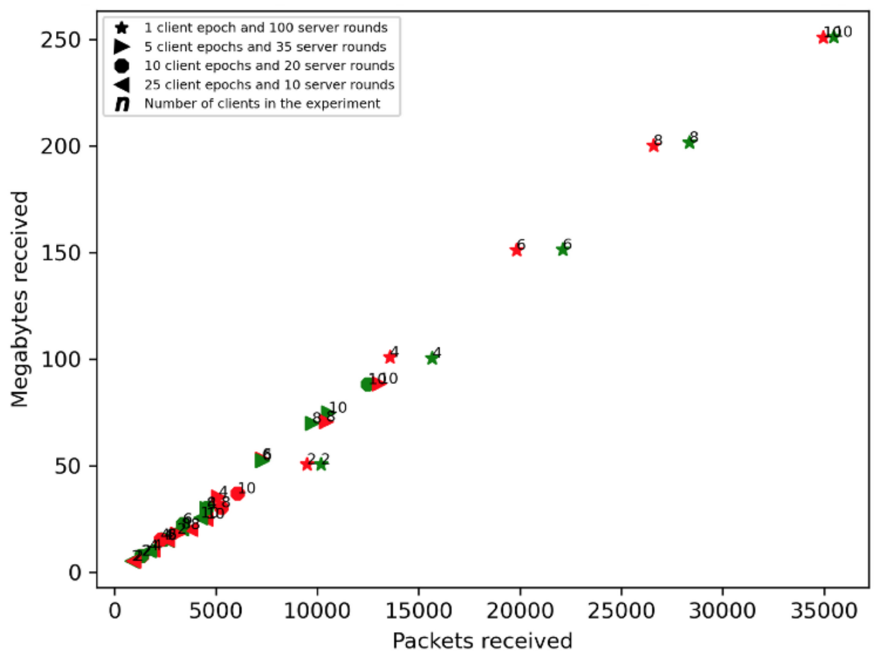


Figure 6. Average total amount of bytes and packets received by the server applications

4.5. CPU and Memory

To better understand CPU and memory usage behavior on the client side, the chosen experiment is the red data distribution with ten clients, 25 client epochs and 10 server rounds. Figure 7 shows the results of the proposed experiment. The X axis indicates the time in seconds, while the Y axis indicates average resource usage in percentage.

The CPU usage varies from 10% to 30%, with some spikes going up to 50%. These spikes might have been caused by some external factor such as the server being overloaded, or some other outlier being accounted for in the averages. However, there is a tendency of periods in the CPU usage that lasts for approximately 1000s that the usage is increased, then it drops to 10%, then repeat. There are ten of these periods in the experiment, which is the number of server rounds. When the CPU is 30%, the client is training the local model; when it is 10%, it is sending or receiving data from the server. The memory, on the other hand, demonstrates a flat usage throughout the experiment, being close to 10% from start to finish.

Figure 8 shows a scatter plot with the CPU usage for every experiment. The X axis indicates the total time in seconds that the experiment took, while the Y axis indicates average CPU used in percentage throughout the experiment.

The plot indicates a correlation between clients involved in the experiment, CPU usage and experiment time. Experiments with fewer clients are able to utilize more CPU power, allowing the experiment to finish faster. This can be seen, for example, looking at the cluster in the top left corner, all of them being experiments with two clients. The faster to finish, and using nearly 100% CPU. From there, around the 50% mark is the experiments with four clients. Bottom right shows mainly the experiments with ten clients,

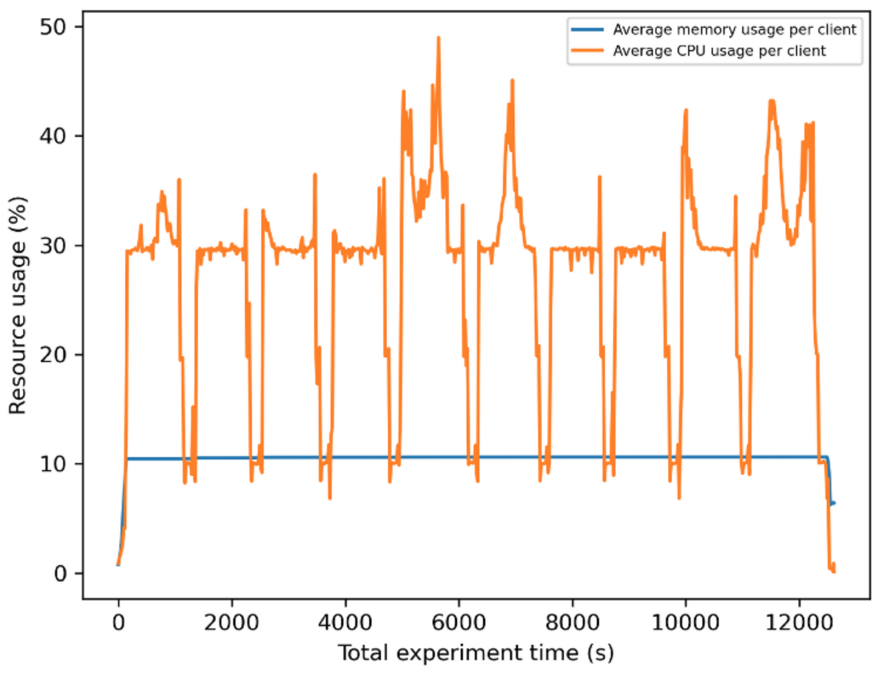


Figure 7. Average CPU and memory used by clients with red data distribution with 25 client epochs and 10 server rounds

taking longer to finish and using less CPU. This makes clear the relationship between CPU usage and time to finish the experiment.

The tendency appears to be logarithmic curve: if there was an experiment with only one client, it would use 200% of the CPU and finish in half the time. Likewise, if there was an experiment with eleven clients, it wouldn't be that much different from the experiment with ten.

Figure 9 shows a scatter plot with the memory usage for every experiment. The X axis indicates the total time in seconds that the experiment took, while the Y axis indicates average memory used in percentage throughout the experiment. For memory, there is a clear difference between red and green experiments. As the red clients carry more data, they require more memory to perform, with a difference of up to 10% to their green counterparts.

5. Conclusion and Future Work

This work achieves the intended proposal by laying the foundation for federated learning experiments and further understanding how the system scales when adding more clients in regards to computing resources, and how this affects the overall system accuracy. The findings can be useful to further improve federated learning technologies, using the knowledge on the application behavior to build, for instance, an edge-cloud orchestrator specifically for federated learning use cases. Moreover, it helps the design of federated learning systems that may scale to multiple clients.

The main takeaway point is that the system allows for horizontal scaling by adding more client nodes in order to utilize more unique data and achieve higher accuracy num-

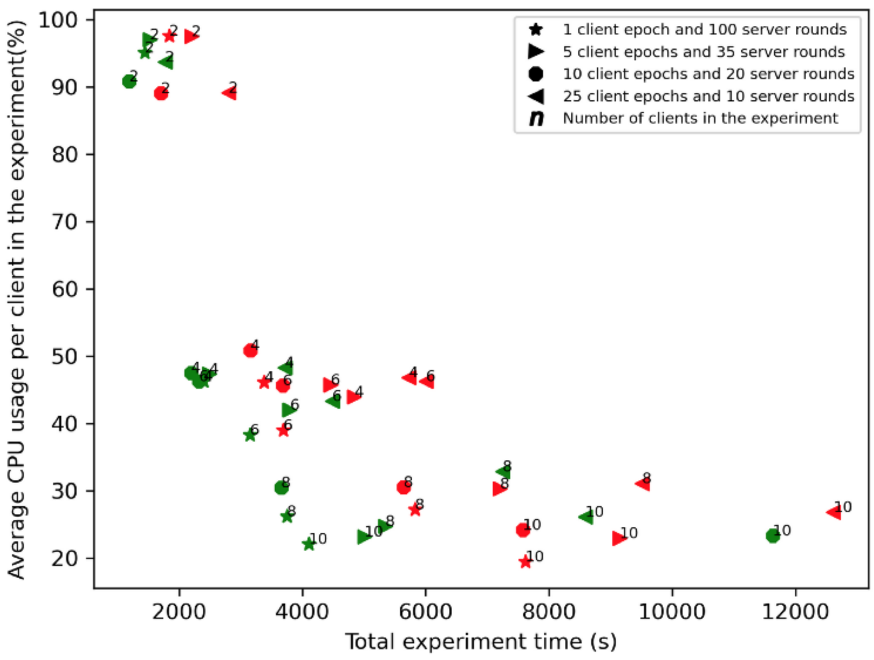


Figure 8. Average CPU used by clients according to total experiment time

bers. By adding more nodes, however, the load in the server application will keep scaling linearly, to the point where it may become the bottleneck for the system. Server rounds, while previously mentioned to be more effective to increase accuracy, not only will even further increase the server load as the server will have to average and update parameters more frequently, but it will also make so that the clients will have to send their data more frequently as well. The latter can be an issue especially for edge devices with limited connectivity, because it cannot be guaranteed continuous access internet access in order to match the frequency the server will expect.

All of the above indicate that, in order to have good federated learning use cases, first a good strategy to train the local client model is required. For scenarios with a huge amount of clients, placing some of the load on the clients and improving their local model training allows the clients to make fewer connections, even if it may not be optimal considering only the accuracy as previously analyzed. This also alleviates the load on the server that having these many clients involved in the system will cause.

This trade-off, however, has to be considered on a case by case basis. If the clients are guaranteed to always have good internet connectivity, for instance, and the cost of maintaining a server that is powerful enough to scale as the number of clients increase is not an issue, the load can be better placed in the server, allowing the clients to train less rounds themselves. This option may also make sense to federated learning scenario with fewer number of clients, e.g., the clients being a few different sensors in a farm.

This work also opens up opportunities to further experiment with federated learning scenarios. All of the code is publicly available and has been developed with extensibility in mind, with well defined methods while prioritizing clean code, modularity, and documentation as much as possible. The observability and plotting modules have been

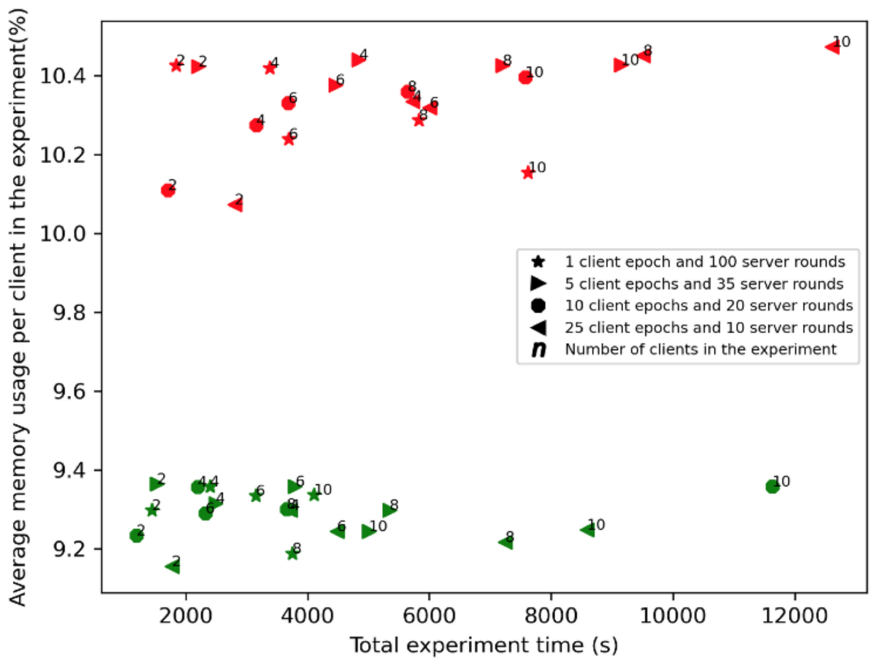


Figure 9. Average memory used by clients according to total experiment time

thoroughly developed with the possibility of further advancement in mind.

Regarding limited connectivity scenarios, one path would be to see how the system would behave for clients with slow internet connection, and that may lose internet connection between training. Fault tolerance, especially in the server, is an important aspect, since as it is right now the server application is a single point of failure in the system, and if one of the clients fail the experiment stops until the connection is retrieved. Regarding client configuration, heterogeneous clients and superior models with real world datasets may also lead to further discoveries.

References

Barthélemy, J., Verstaevel, N., Forehead, H., and Perez, P. (2019). Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors*, 19(9).

Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Parcollet, T., de Gusmão, P. P. B., and Lane, N. D. (2021). Flower: A friendly federated learning research framework.

Cisco (2020). Cisco annual internet report (2018–2023) white paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (accessed October 16th, 2021).

Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S., and Zomaya, A. Y. (2020). Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469.

Google (2017). Federated learning: Collaborative machine learning without centralized training data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html (accessed November 2nd, 2021).

Hard, A., Rao, K., Mathews, R., Ramaswamy, S., Beaufays, F., Augenstein, S., Eichner, H., Kiddon, C., and Ramage, D. (2018). Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*.

He, C., Li, S., So, J., Zeng, X., Zhang, M., Wang, H., Wang, X., Vepakomma, P., Singh, A., Qiu, H., Zhu, X., Wang, J., Shen, L., Zhao, P., Kang, Y., Liu, Y., Raskar, R., Yang, Q., Annavaram, M., and Avestimehr, S. (2020). Fedml: A research library and benchmark for federated machine learning.

Ji, S., Pan, S., Long, G., Li, X., Jiang, J., and Huang, Z. (2019). Learning private neural language modeling with attentive aggregation. *2019 International Joint Conference on Neural Networks (IJCNN)*.

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Eichner, H., Rouayheb, S. E., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konečný, J., Korolova, A., Koushanfar, F., Koyejo, S., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F. X., Yu, H., and Zhao, S. (2021). Advances and open problems in federated learning.

Konečný, J., McMahan, H. B., Ramage, D., and Richtárik, P. (2016). Federated optimization: Distributed machine learning for on-device intelligence. Technical report, Google, Inc.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y., and Shi, W. (2019). Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. (2016). Communication-efficient learning of deep networks from decentralized data. Technical report, Google, Inc.

Nilsson, A., Smith, S., Ulm, G., Gustavsson, E., and Jirstrand, M. (2018). A performance evaluation of federated learning algorithms. *DIDL '18: Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*, pages 1–8.

Rieke, N., Hancox, J., Li, W., Milletarì, F., Roth, H. R., Albarqouni, S., Bakas, S., Galtier, M. N., Landman, B. A., Maier-Hein, K., and et al. (2020). The future of digital health with federated learning. *npj Digital Medicine*, 3(1).

Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., and Zhang, J. (2019). Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762.