Centralized and Multimaster Federated Learning Cloud Computing

Mario Edoardo Pandolfo[1], **Nicola Grieco**[1], **Davide Vigneri**[1]

[1]University of Rome La Sapienza, Italy

pandolfo.1835189@studenti.uniroma1.it
grieco.2081607@studenti.uniroma1.it
vigneri.2058036@studenti.uniroma1.it

Abstract

This study explores two distinct implementations of Federated Learning, specifically Centralized Federated Learning and Multi-Master Federated Learning, applied to a dataset of hospital data for predicting the outcomes of individuals affected by diabetes.

1 Description of the problem

In the realm of AI technology, we are witnessing remarkable advancements across various industries and aspects of our daily lives. Nevertheless, this landscape presents two significant challenges.

The first challenge revolves around the critical issue of data governance. Data often contains sensitive, private information that is subject to legal constraints, necessitating proper authorization or consent for usage. In numerous machine learning applications, the common practice of centralizing data from multiple sources raises concerns about data security and privacy.

The second challenge is tied to the growing storage and computational capabilities of devices in distributed networks. This trend presents an opportunity to leverage local resources, potentially addressing some of the limitations associated with centralized data storage and processing.

To tackle these issues, Federated Learning emerges as a promising alternative. This collaborative technique promotes cooperation among devices and organizations. Instead of sharing raw local data, Federated Learning focuses on aggregating model parameters from local models. This approach enhances privacy, security, and the development of robust machine learning models.

Historically, Centralized Federated Learning (CFL) has been the dominant approach. In CFL, a central entity manages a global model. However, this centralized paradigm suffers from shortcomings like elevated latency, system failure susceptibility, and doubts about the trustworthiness of the central entity. As a solution, Decentralized Federated Learning (DFL) has emerged, championing a decentralized model aggregation approach.

Our project evaluates the efficacy of Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL) in performing a classification task on healthcare data.

The healthcare sector is one of the many applications in which FL presents itself as a valid solution to issues intrinsic to such a distinctive landscape. In this intricate environment, in which data is fragmented to abide by stringent privacy requirements, individual healthcare institutions maintain separate data silos. This fragmentation hinders the opportunity to leverage local resources.

In the upcoming sections, we will introduce custom solutions for healthcare data challenges, implement them practically, explore deployment options, and discuss experimental methodology. We will then unveil the results of our experiments, comparing Centralized Federated Learning (CFL) and Decentralized Federated Learning (DFL) in healthcare data classification. This revised structure offers a more coherent and cohesive narrative.

2 Design of the solution

Our project addresses the challenge of logistic classification on a dataset containing patient information, particularly a categorical variable indicating patient mortality (0 for survival, 1 for death). The data originate from numerous hospitals across different locations in America. However, due to privacy and access constraints, we do not have direct access to geographical information. To address this problem, we adopted two distinctive design solutions: Centralized Federated Learning and a variant of Decentralized Federated Learning

known as Multi-Master Federated Learning.

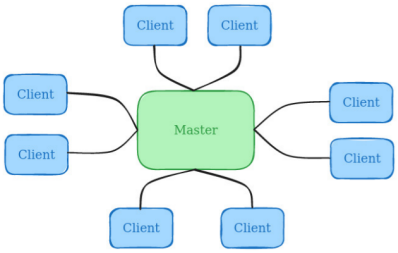


Figure 1: Centralized architecture

Centralized Federated Learning embodies a classical approach with a central server assuming responsibility for the orchestration of the learning process. This design was chosen primarily for its streamlined coordination and the assurance it provides in terms of data privacy. Under this approach, the central server accumulates and aggregates model updates from all participating clients. The centralization of control makes monitoring and management of the learning process more straightforward, enhancing transparency and security. However, it is not without its limitations. The central server presents a single point of failure, and the scalability of the system can be constrained. This approach is best suited when data volumes are manageable, and privacy concerns can be adequately addressed through encryption, secure communication, and stringent access controls. Additionally, Centralized Federated Learning shines in scenarios where a singular coordinating entity possesses the authority and trust to manage the learning process efficiently.

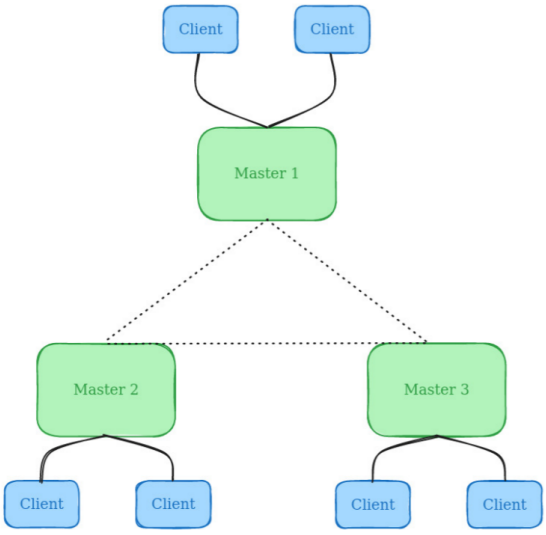


Figure 2: Multi-master architecture

On the other hand, the **Multi-Master Federated Learning** architecture introduces multiple coordinators, each responsible for a subset of clients. This design enhances fault tolerance, scalability, and redundancy, offering an appealing trade-off between centralization and decentralization. Multi-Master Federated Learning is particularly advantageous when dealing with large datasets and distributed systems. It minimizes the risk of a central point of failure while ensuring robust coordination. The decision to adopt this approach is motivated by its suitability for the unique dataset we are working with—data from multiple hospitals with diverse characteristics. It offers the flexibility to manage the learning process across these diverse data sources without a central entity having complete access to all data, thereby preserving data privacy.

Our dataset's unique structure, originating from different hospitals with distinct characteristics, presented us with a challenge data partitioning. Given our limited access to geographical information, we chose to simulate the partitioning based on the Hospital ID. This choice was not driven by geographic considerations but was made to mimic realistic data partitioning scenarios where hospitals are distinct entities with their unique characteristics. This simulated partitioning approach ensures that each node primarily interacts with data from a specific cluster of hospitals, preserving individual data source characteristics.

Below, we will outline the steps applied in each of the two chosen approaches:

2.1 Centralized Federated Learning

1. **Data Preprocessing**: After the partitioning of the data and distribution to the clients, a crucial data preprocessing step is performed. This preprocessing phase involves the standardization of continuous variables and the application of one-hot encoding for categorical variables. However, a challenge arises in this step due to the inherent variation in the number of features across clients. The number of output features from the one-hot encoding procedure is directly related to the number of unique classes within each client's dataset. This variability poses a unique challenge that needs to be addressed during model training.

2. **Model Training with PyTorch**: Model

training for logistic classification is conducted using the PyTorch framework. Each client takes its preprocessed dataset and performs model training independently. This results in a set of model weights that are specific to each client. The goal is to achieve localized model updates while preserving data privacy.

3. **Handling Missing Values**: One of the key challenges arises due to the variability in the number of features after one-hot encoding. When applying logistic regression, each client obtains a vector of model weights. In cases where certain categorical classes are absent from a client's dataset, it leads to the creation of missing values within the weight vector. To address this issue, a placeholder (missing value) is inserted in the weight vector to indicate the absence of specific categorical classes. This ensures that each client's weight vector is of uniform length, with placeholders indicating missing classes.

4. **Aggregation at the Central Master Node**: Following local model training, each client sends its weight vector, which may contain placeholders for missing values, to the central master node. The master node is responsible for aggregating these weight vectors and performing further computations to derive a collective model.

5. **Weighted Bayesian Bootstrap**: At the central master node, a weighted Bayesian bootstrap method is employed to evaluate the average weights for each parameter. This approach handles missing values and accounts for the associated uncertainties in a statistically robust manner. The Bayesian Bootstrap method involves drawing samples from a Dirichlet distribution and calculating posterior means and variances. This process allows for the estimation of the average weights while accommodating the presence of placeholders representing missing values.

2.2 Multi-Master Federated Learning

1. **Data Partitioning within Clusters**: In the Multi-Master Federated Learning architecture, the process of data partitioning begins by grouping clients into clusters based on a predefined criterion. Each cluster represents a group of clients that share similar data characteristics. The primary objective of clustering is to ensure that data privacy is preserved while promoting efficient collaboration. Data partitioning within clusters is random and is not based on geographic regions due to the unavailability of this information.

2. **Localized Data Preprocessing**: Within each cluster, clients perform a localized data preprocessing step. This process involves type casting for features, standardizing continuous variables, and one-hot encoding for categorical variables. As in the Centralized Federated Learning approach, one-hot encoding may result in varying feature lengths due to the differing number of classes in each client's dataset. Localized preprocessing ensures that data is in a consistent format for subsequent model training.

3. **Model Training with PyTorch**: Clients within each cluster independently execute model training using the PyTorch framework. This localized model training results in a set of model weights for each client within the cluster. The clustering strategy facilitates the sharing of knowledge among clients with similar data characteristics, further enhancing the learning process.

4. **Handling Missing Values within Clusters**: Similar to the Centralized Federated Learning approach, the Multi-Master Federated Learning architecture may encounter missing values due to varying feature lengths after one-hot encoding. Each client within a cluster addresses this challenge by inserting a placeholder (missing value) in the weight vector, corresponding to the missing categorical classes. As a result, all clients within a cluster maintain weight vectors of uniform length, which includes placeholders for missing classes.

5. **Aggregation at Cluster Masters**: Each cluster of clients has a designated master node that coordinates the aggregation of weight vectors within the cluster. The master node collects weight vectors from all clients, considering the presence of placeholders. This aggregation occurs independently for each cluster.

6. **Weighted Bayesian Bootstrap within Clusters**: Within each cluster, the master node employs the weighted Bayesian bootstrap method to evaluate the average weights for each parameter. This method handles missing values and quantifies uncertainties stemming from placeholders within the weight vectors. It includes the sampling of posterior distributions using a Dirichlet distribution and the calculation of means and variances. This process is performed within each cluster, independently.

7. **Master-to-Master Aggregation**: Following the aggregation of average weight vectors within clusters, the final step involves the aggregation of these cluster-level weight vectors at the central master node. The central master node performs a further level of aggregation and computes the ultimate average weight vector for the entire federated learning model.

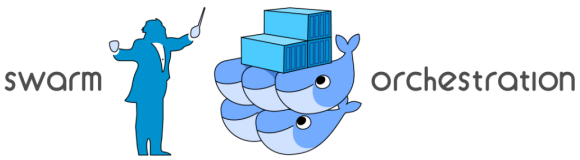
3 Implementation

In this section, we provide a detailed overview of the implementation of our federated learning project, including the setup of Docker containers, orchestration using Docker Swarm, and the key components of our architecture.

3.1 Docker Container Setup

We chose to containerize our federated learning system using Docker for its efficiency in managing isolated environments. Two main Docker containers were employed: one for the master node and another for the clients. These containers are based on the PyTorch Docker image [1], which is well-suited for deep learning tasks. The containers are configured with the necessary software dependencies, libraries, and code to facilitate the federated learning process.

3.2 Docker Swarm Orchestration



[1]Specifically, we employed the image version pytorch/pytorch:2.0.0-cuda11.7-cudnn8-devel.

To orchestrate and manage the deployment of our federated learning architecture, we leveraged Docker Swarm, a native Docker orchestration tool. Docker Swarm allows us to create a cluster of Docker nodes and deploy services across them seamlessly. This provides scalability and resilience for our federated learning system.

The orchestration process is defined in a Docker Compose file, which specifies the services, networks, and volumes required for the system. This file simplifies the deployment process, making it easy to manage the entire architecture as a single entity.

3.3 Architecture Overview

Our federated learning architecture comprises the following components:

- **Master Node Container**: The master node container is responsible for coordinating the federated learning process: it keeps track of the subscribed clients and in the end it aggregates the clients model weights by performing a *Weighted Bayesian Bootstrap*.

- **Client Node Containers**: Client node containers simulate the individual devices or users in the federated learning scenario. The clients have a common volume in which they have access to their specific data partition, on which they train their local model. Upon the completion of the training phase, every client transmits its model weights back to the master node.

- **Communication Channels**: Docker Swarm inherently manages communication between the master and client nodes, safeguarding data privacy and security. While we haven't currently enabled encrypted communication, Docker Swarm offers the capability to implement it seamlessly. This is achieved through the utilization of APIs and network configurations within Docker containers.

3.4 Deployment Script

To facilitate the setup and deployment of the entire federated learning system, we provide a `run.sh` script. This script automates the following tasks:

1. Downloads the needed python dependencies via a `pip install -r requirements.txt`.

2. Downloads and partitions the dataset.

3. Builds the two docker images, one for the master the other for the clients.

4. Initializes a Docker Swarm Cluster with `docker swarm init`.

5. Deploys our stack with the `docker-compose.yml` file.

3.5 Simulation Script

To run a federated learning simulation we provide a Python script called `main.py`. This script automates the following tasks:

1. Hard resets the master node.

2. Initializes k client nodes, where k is a passed parameter representing the desired number of clients (default 10).

To run the script with k clients just run in a terminal:

```
python master.py -c k
```

4 Deployment on AWS

Under the permissions granted, we executed the deployment of our solution on AWS via EC2 instances.

4.1 Create the EC2 instance

Each EC2 instance was configured with a t2.large instance and initialized with a bash script incorporated into the user data, as outlined below:

```
#!/bin/bash
sudo yum update
sudo yum -y install docker
 ↪  python git htop
service docker start
usermod -a -G docker ec2-user
chkconfig docker on
pip3 install docker-compose
```

4.2 Security Group

The security group functions as the firewall for the instances. In order to enable cluster communication and remote instance access, we need to open the following ports:

- TCP port 2376: This port is used for secure Docker client communication, and it is necessary for Docker Machine to function properly. Docker Machine is employed to orchestrate Docker hosts.

- TCP port 2377: This port is crucial for communication between the nodes in a Docker Swarm or cluster. It should only be opened on manager nodes.

- TCP and UDP port 7946: These ports are essential for communication among nodes, specifically for container network discovery.

- UDP port 4789: This port is vital for overlay network traffic, particularly for container ingress networking.

- TCP port 22: This port allows secure SSH access to our instances for remote management.

4.3 Join the Swarm

The integration of each client node into the swarm was made effortlessly through the execution of the following command:

```
docker swarm join --token
 ↪  <SPECIFIC-TOKEN>
 ↪  <IP-ADDRESS>:2377
```

Please note that the complete command can be conveniently copied from the master node after executing the `docker swarm init` command.

In the **Multi-Master approach**, you need to initialize as many Docker Swarms as the number of master nodes required for redundancy and high availability.

5 Experimental Design

In this section, we present the experimental design for testing and evaluating our Federated Learning solutions. Our design encompasses two key aspects: the experimentation with Centralized Federated Learning and Multi-Master Federated Learning.

5.1 Centralized Federated Learning

We assess the performance and scalability of Centralized Federated Learning through a series of structured trials. Our approach involves running

the model under varying instances to simulate system scaling and we maintained a constant number of clients throughout the experiments, fixing it at 10 adn a master. The experiments are structured as follows:

- Single Instance Run: In this configuration, a single instance is utilized, representing the base case of our system.
- Five Instances Run: To gauge the system's behavior as it scales, we deploy the model across five instances, simulating increased computational load.
- Nine Instances Run: The final run involves nine instances, providing a more comprehensive assessment of system performance under scaling conditions.

Crucially, in each of these runs,there is an additional element designated as the master, enabling us to analyze the impact of scaling and the presence of a central coordinator on system performance.

5.2 Multi-Master Federated Learning

The Multi-Master Federated Learning architecture undergoes evaluation through a distinct set of experiments. In this configuration, we deploy a system with three masters, with each master overseeing two nodes. This design emphasizes the robustness and fault tolerance of the multi-master approach, enabling us to assess its performance under various operational scenarios.

For the Multi-Master Federated Learning experiments, we deploy three masters, each associated with a distinct Docker Swarm using always 9 instances. These Docker Swarms are designed to accommodate different numbers of clients, with the first Swarm hosting three clients, the second Swarm featuring three clients, and the third Swarm consisting of four clients. In each Docker Swarm, the combination of masters and clients totals ten, thereby maintaining a consistent client count across all Multi-Master Federated Learning experiments. This configuration allows us to explore the system's performance and resilience while effectively distributing computational tasks across a diverse set of clients.

The unique arrangement of three masters within separate Docker Swarms further accentuates the multi-master approach's adaptability, particularly in scenarios where heterogeneous client distributions or varying operational conditions may be encountered. These experiments offer a comprehensive assessment of the architecture's ability to coordinate and manage Federated Learning tasks, even in situations with diverse client loads and operational environments.

5.3 Metrics and Monitoring

The metrics selected for analysis are integral to our evaluation process. We focus on the following key metrics:

- CPU Utilization Percentage: An indicator of the computational efficiency of our models.
- CPU Costs: Including credit balance and usage, which provide insights into resource consumption.
- Status Check Failures: A measure of system health and reliability.
- Network Packets: We analyze the number of network packets both inbound and outbound, considering both the total count and the average, to assess data transfer efficiency.

To ensure accurate and real-time data collection during our experiments, we leverage AWS CloudWatch, which provides robust monitoring capabilities for these metrics.

This structured experimental design enables us to draw informed comparisons between our Centralized and Multi-Master Federated Learning solutions. It offers a comprehensive assessment of their respective strengths and trade-offs, considering factors such as scaling, performance, and robustness in real-world operational scenarios.

6 Experimental results

In this section, we delve into the results of our experiments, offering a detailed analysis of the conditions under which our Federated Learning solutions were evaluated. The core of our investigation in the Centralized approach revolved around understanding the system's behavior as it scales with varying computational resources. To achieve this, we maintained a constant number of clients throughout the experiments, fixing it at 10. This consistent client count allows us to isolate the effects of scaling on the system's performance and efficiency. To systematically assess the impact

of computational capacity, we varied the number of instances, which represent the CPU capabilities connected to the swarm. This controlled experimentation approach provides valuable insights into how the system's scalability influences the Federated Learning process and its resource utilization. In the following sections, we present the outcomes of these experiments and offer detailed insights into our findings.

The assessment includes two architectures: Centralized Federated Learning and Multi-Master Federated Learning.

6.1 Centralized Federated Learning

The primary focus of our analysis lies in the comparison of different Centralized Federated Learning experiments.

6.1.1 CPU Utilization

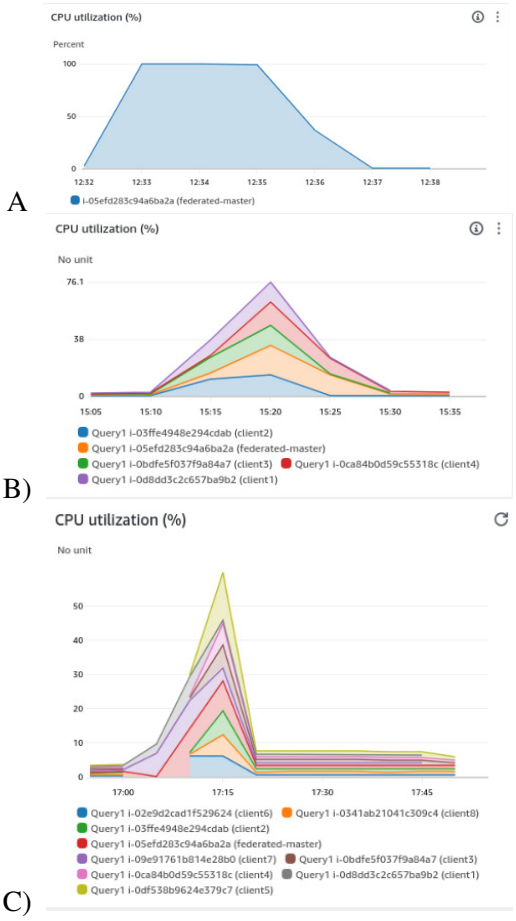


Figure 3: CPU Utilization: A) 1 instance, B) 5 instances, C) 9 instances

One notable observation from the results is the evident trend in CPU utilization percentage as the number of instances and clients increases. In the case of using a single instance, the CPU utilization reaches its maximum capacity, operating at 100%. However, as we scale the system to four instances, we observe a decline in the percentage of CPU utilization. The maximum utilization drops to approximately 76%. With the further increase to nine instances, the maximum CPU utilization reaches around 55%. This trend is indicative of the system's improved efficiency as more computational resources become available.

6.1.2 Status check failed

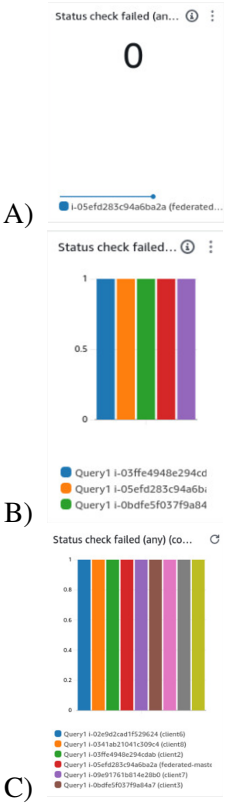


Figure 4: Number of failures: A) 1 instance, B) 5 instances, C) 9 instances

During the course of our experiments, it is worth noting that we encountered minimal issues and observed a specific behavior that required clarification. An interesting aspect is that we did not recognize substantial errors in the performance of our Federated Learning system. However, we observed that, at times, the Docker Swarm, used for orchestrating our computational tasks, would conclude its operation. While this occurrence might initially seem like an error, it was not the case. Our meticulous examination revealed that the Docker Swarm was effectively finishing its work as intended.

Interestingly, AWS CloudWatch, our chosen monitoring tool, occasionally recognized the termination of Docker Swarm tasks as errors. These notifications were not indicative of issues within our system but rather a result of the job's natural completion. Therefore, it was crucial to distinguish these false positives and prevent any misconceptions regarding the system's behavior. Our experiments further exemplify the importance of considering the context and the underlying processes, ensuring that the recognition of errors aligns with the actual performance of the Federated Learning system.

6.1.3 CPU Credit Balance

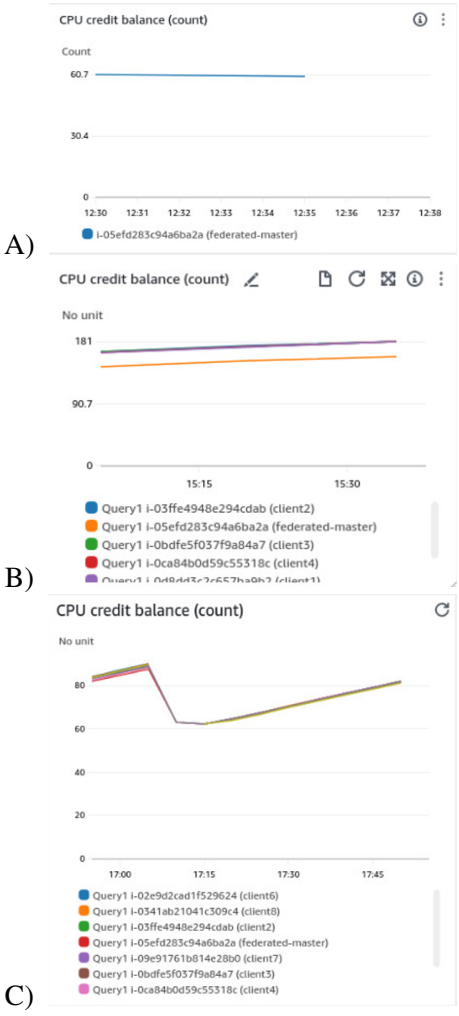


Figure 5: CPU Credit Balance: A) 1 instance, B) 5 instances, C) 9 instances

In our Centralized Federated Learning experiments, the CPU credit balance is an indicator of how well our system optimizes its CPU usage. It is noteworthy that AWS uses CPU credits to measure and manage CPU utilization. For a single instance, we observe a CPU credit usage of approximately 60, reflecting standard usage patterns. However, as we scale the system to four instances, we witness a substantial increase in credit usage. This is expected since more instances demand additional CPU credits to operate efficiently. Surprisingly, with nine instances, the credit balance per instance is much lower, hovering around 80. This deviation from expectations can be attributed to the system's improved efficiency as it scales. The increased number of instances leads to a more distributed workload, reducing the demand for CPU credits per instance.

6.1.4 CPU Credit Usage

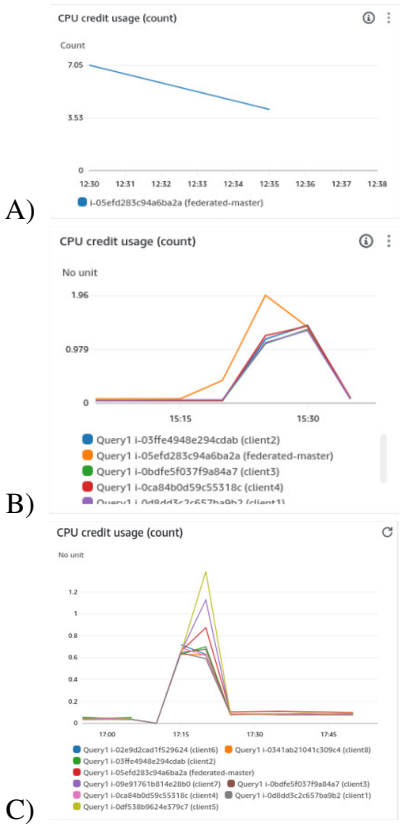
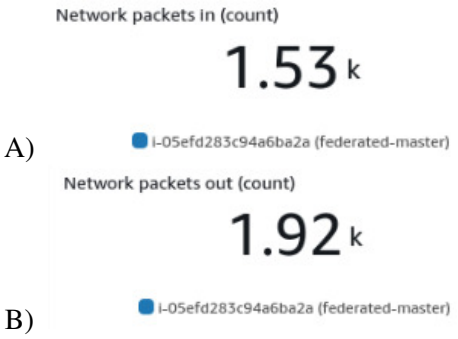


Figure 6: CPU Credit Usage: A) only master, B) 4 clients, C) 8 clients

The credit usage metric is another essential component of our analysis, focusing on the efficiency of CPU credit utilization. Remarkably, as we increase the number of instances, we observe a consistent decrease in credit usage per instance. This trend highlights an inherent advantage of scalability by distributing the computational workload across a greater number of instances, each instance requires fewer CPU credits for efficient operation. This can be attributed to the distributed nature of

Federated Learning, which effectively leverages computational resources, leading to a reduction in credit usage.

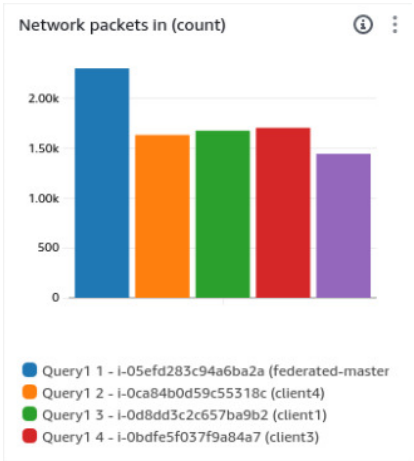
6.1.5 Network packets in and out



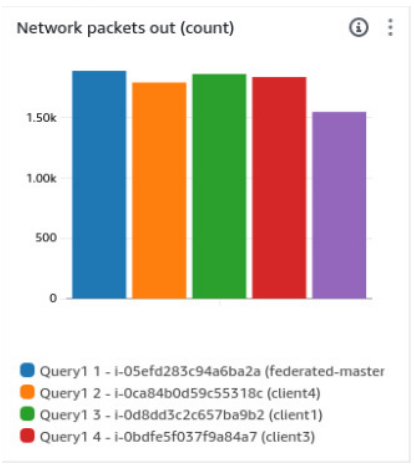
A)

B)

Figure 7: Number of packets in and out: A) number packets in (1 instance), B) number packets out(1 instance)

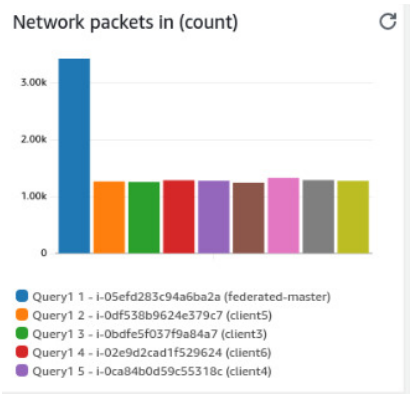


E)

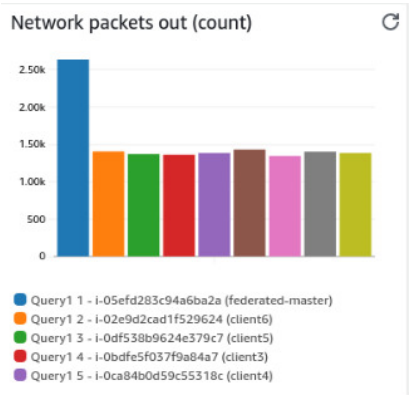


F)

Figure 8: Number of packets in and out: E) number packets in(5 instances), F) 4 clients number packets out(5 instances)



A)



B)

Figure 9: Number of packets in and out: A) number packets(9 instances) in, B) number packets out(9 instances)

One key aspect of our experiments involved the analysis of network packets exchanged within our Federated Learning system. As anticipated, the number of packets exchanged by the master was significantly greater than the packets transmitted by the clients. This observation aligns with expectations, as the master coordinates and manages the communication across multiple clients.

A noteworthy trend emerged as we increased the system's scaling with more instances. While the messages were distributed efficiently among the instances, we observed a considerable increase in the number of packets exchanged by the master. This finding raises a potential concern—the master's role as a single point of control could also become a single point of failure in the system. This vulnerability could have implications for the overall robustness of the architecture, particularly in cases where the master faces disruptions.

To mitigate this challenge, we explore the application of the Multi-Master Federated Learning architecture. By distributing control among multiple masters, we aim to enhance the system's fault tolerance, reduce the risk of single points of failure, and bolster the overall robustness of the Federated

Learning process.

6.2 Multi Master Federated Learning

6.2.1 CPU Utilization

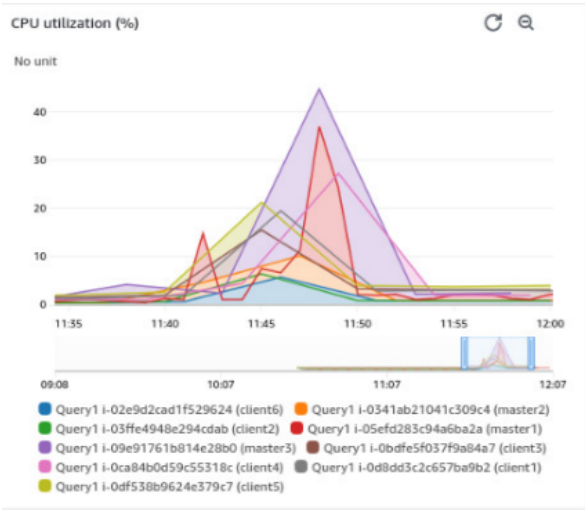


Figure 10: Multi-master CPU Utilization

In our exploration of the Multi-Master Federated Learning architecture, we uncover intriguing insights into its performance and resilience. Notably, the maximum CPU utilization of the instances remains slightly above 40%. While this utilization is slightly higher than in the previous Centralized Federated Learning experiments, it is indicative of efficient resource management within this architecture.

6.2.2 CPU Credit Balance

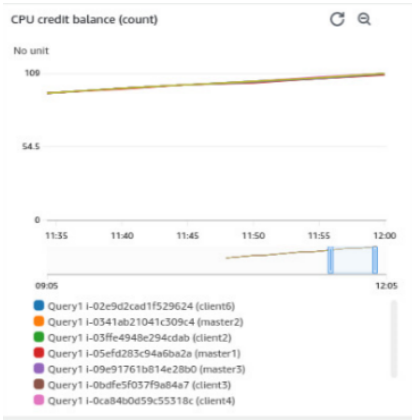


Figure 11: Multi-master CPU Credit Balance

Furthermore, the credit balance for each instance maintains a relatively stable level, hovering around 100. This credit balance is higher compared to the Centralized Federated Learning experiments, demonstrating a robust distribution of computational resources.

6.2.3 CPU Credit Usage

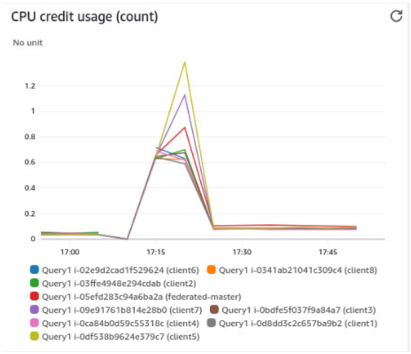
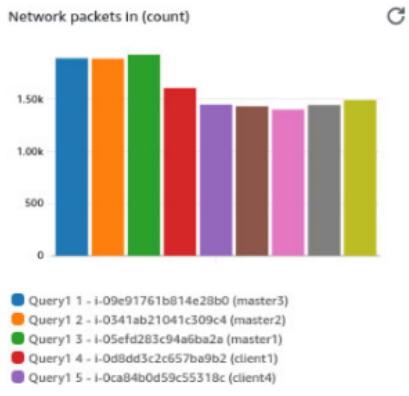


Figure 12: Multi-master CPU Usage

Moreover, it is noteworthy that the credit usage in this experiment is notably lower, approximately 1.2. This reduction in credit usage reflects the efficient allocation of computational resources and cost-effectiveness, adding to the overall advantages of the Multi-Master Federated Learning architecture.

6.2.4 Network packets in and out

A)



B)

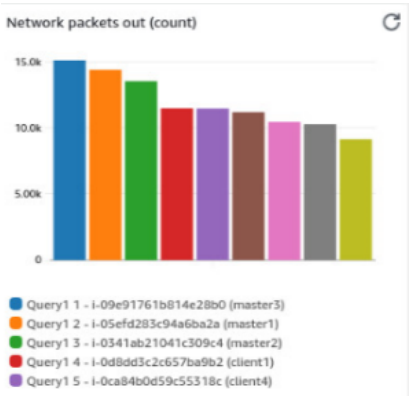


Figure 13: Number of packets in and out: A) number packets in, B) number packets out

One of the most compelling outcomes of the Multi-Master Federated Learning experiment is the observation related to network packets. Unlike the Centralized Federated Learning experiments, where the master exchanged a significant number of packets, here we find a reversal of this trend. In the Multi-Master architecture, the masters consistently handle more packets than the clients, signaling an even distribution of responsibilities. This significant shift is indicative of the successful elimination of single points of failure. The presence of multiple masters not only enhances the fault tolerance of the system but also ensures that the architectural design remains robust in the face of potential disruptions.

7 Final Considerations and Further Improvements

In our Federated Learning experiments, we have explored the potential of Centralized and Multi-Master Federated Learning architectures in the context of healthcare data. Our approach involved partitioning the dataset across multiple clients, each representing a hospital, to train models collectively while preserving data privacy. As we conclude this study, we reflect on the insights gained and contemplate further avenues for improvement.

One crucial aspect of our approach is the computation of average weights among all clients, leading to a final vector of weights. These global weights are subsequently returned to the clients, allowing them to evaluate the prediction of new data using this comprehensive set of parameters. This approach empowers us to detect characteristics that the local weights may not take into consideration. It is a valuable methodology for applications where geographical locations do not significantly impact the dataset.

However, we recognize that in an ideal scenario where access to geographical hospital locations is available, we could further enhance our methodology. By utilizing our Multi-Master architecture, we could compute average weights specifically among clients associated with each master, thereby forming clusters of hospitals located in distinct geographic areas. Each cluster would have its set of weights, corresponding to the number of masters, effectively creating a mechanism to respect the local characteristics of specific regions while taking into account data from hospitals within the same neighborhood. This refined approach could offer valuable insights for healthcare applications operating in diverse geographic contexts.

In conclusion, our Federated Learning experiments have laid a foundation for secure and privacy-preserving model training. The exploration of Centralized and Multi-Master Federated Learning architectures has provided insights into their advantages and trade-offs. As we look to the future, the potential for incorporating geographical data into our methodology represents an exciting avenue for further improvements and enhanced accuracy in healthcare applications.