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 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.

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.

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.

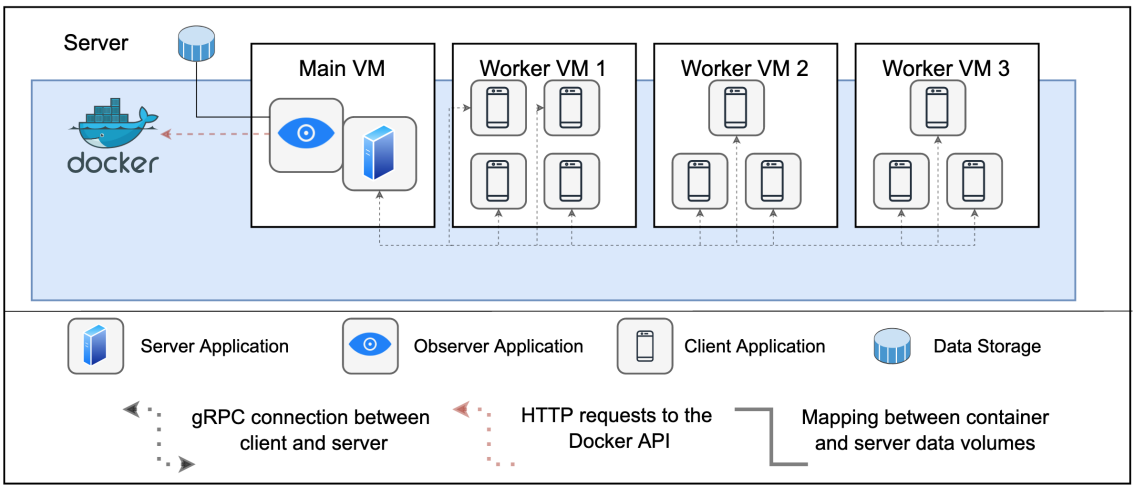

Figure 1. High-level diagram of the implemented solution when running with ten clients

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 framework, 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.

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

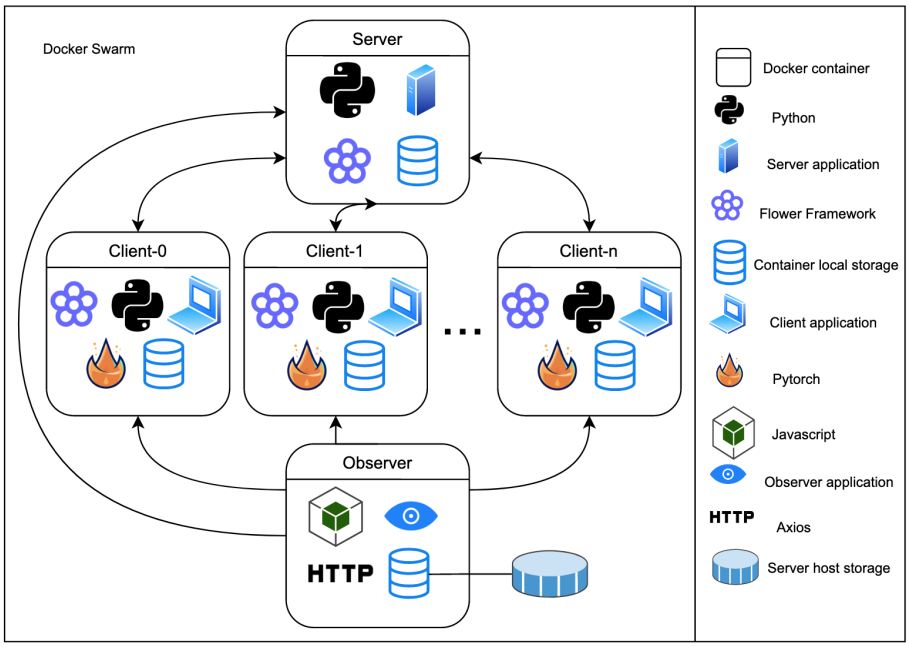

Figure 2. Solution architecture with the services and their dependencies

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 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].

[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)

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

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.

4.2. 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.

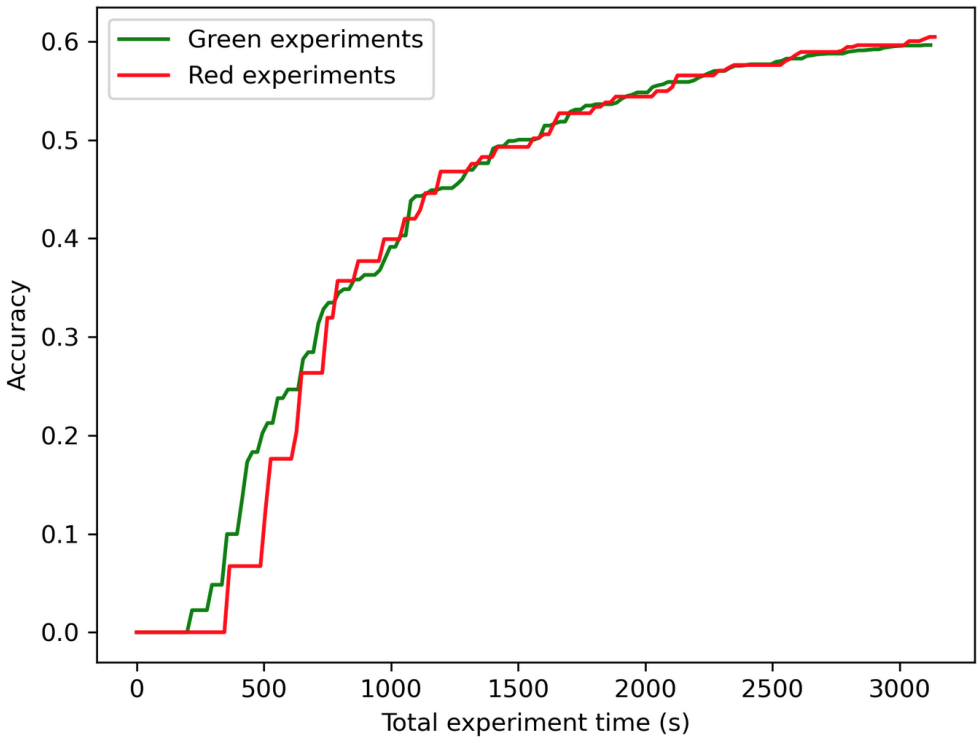

Figure 3. Average accuracy with ten clients through time

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.

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.

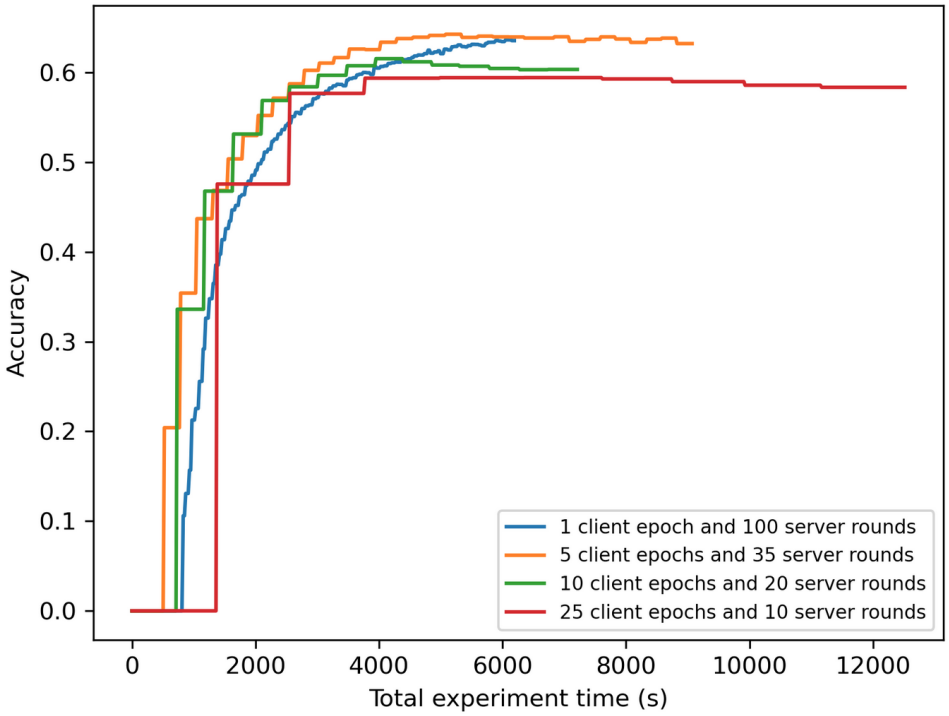

Figure 4. Average accuracy with ten clients split by client epoch and server round configurations through time

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 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.

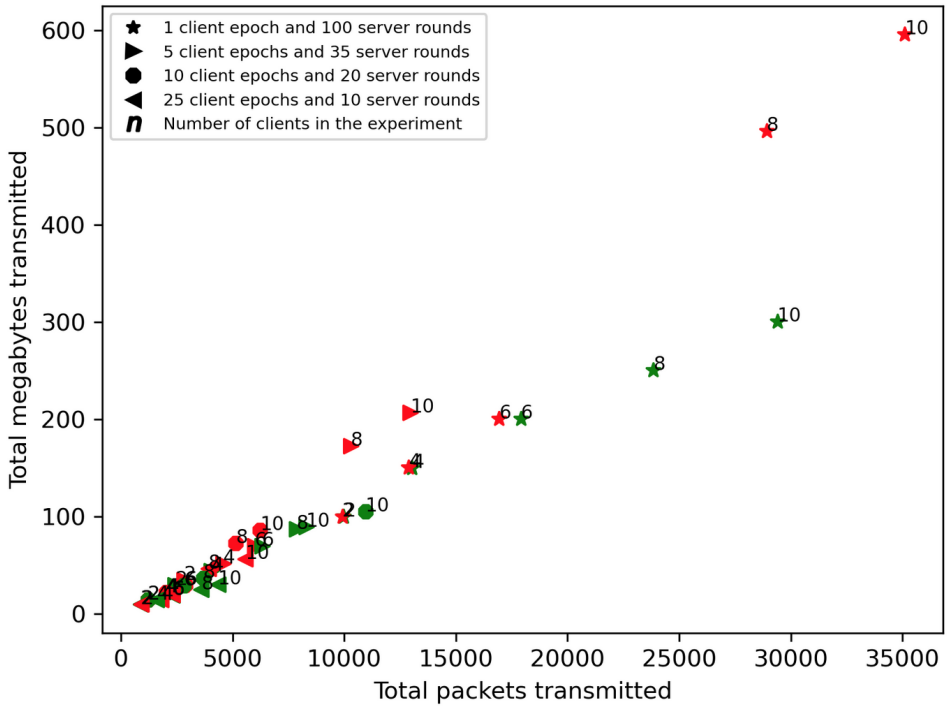

Figure 5. Total amount of bytes and packets transmitted by the server application

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 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.

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

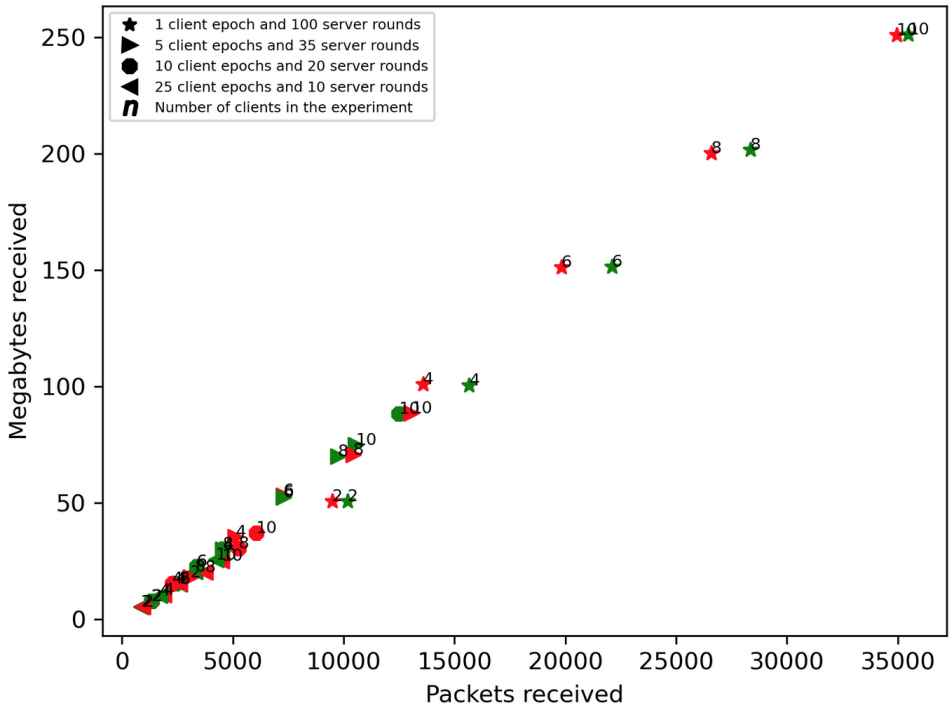

Figure 6. Average total amount of bytes and packets received by the server applications

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

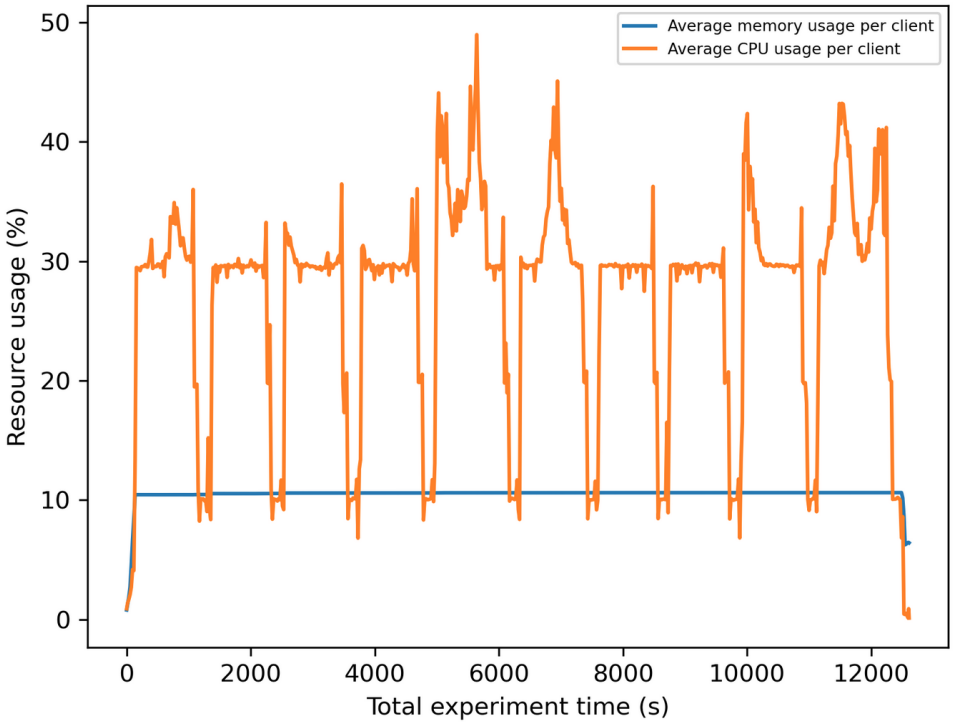

Figure 7. Average CPU and memory used by clients with red data distribution with 25 client epochs and 10 server rounds

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 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,

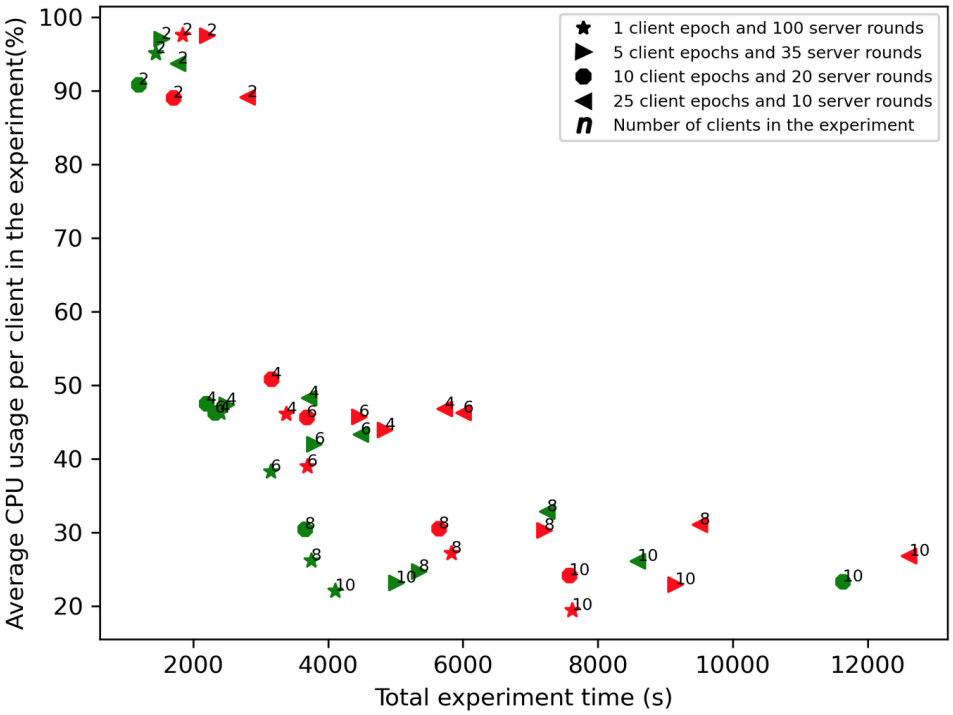

Figure 8. Average CPU used by clients according to total experiment time

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 thoroughly developed with the possibility of further advancement in mind.

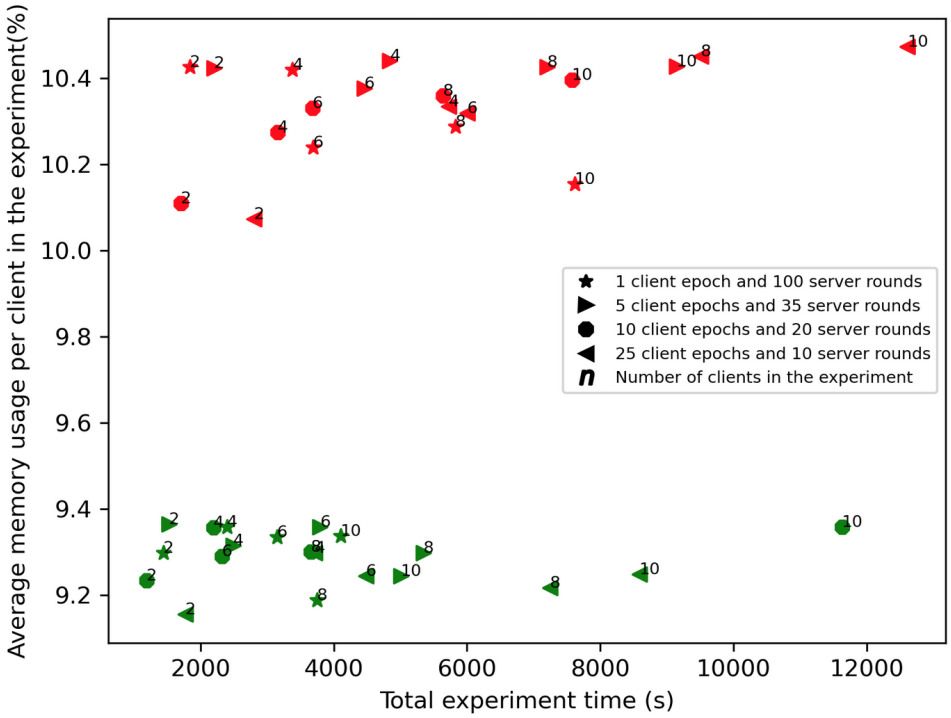

Figure 9. Average memory used by clients according to total experiment time

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.