POLITECNICO DI TORINO

Master Degree in Communications and Computer Networks Engineering

Master Thesis

An Asynchronous Framework to Mitigate the Network Impact on Federated Learning

Supervisors
Prof. Claudio Ettore Casetti
Prof. Paolo Giaccone
Ph.D. Alessandro Cornacchia
Ph.D. Riccardo Rusca

Candidate
Anna Origlia

April 2023

Abstract

Federated learning is a technique which has been introduced to evolve machine learning and to provide a distributed learning structure more suited to be applied to complex environments such as IoT or privacy-preserving applications. The federated aggregation process is controlled by a parameter server, but, differently from centralised ML, an additional on-site step of local training is added in the devices; in this way, data is not disclosed and the communication overhead reduces significantly.

Because of its distributed nature, FL faces a highly heterogeneous environment; clients differ in computational capabilities (especially considering IoT devices), datasets distributions (data is generated or collected on the device, and this may bias or pollute the population), reliability (clients may drop out mid-run). In addition, the network plays an important role in the client-server communication, as it affects the total time needed by a client's update to reach the server. In synchronous FL, slow clients become the bottleneck of the whole process.

In this work, we address the effect that the network bandwidth and latency have on FL rounds and we propose a new framework for coping with it: our system architecture is composed by a centralised server, a pool of available clients and a number of mediators that act as the middle-man between clients and server, coordinating their communication. The mediators are positioned on the cloud edge, so to reduce as much as possible the clients' updates transmission latency; also, we consider that the bandwidth and delay between the mediators and the server is fixed and known. Mediators can run a request-acknowledgement procedure with each client before sending the round training instructions: this procedure is used to estimate each client's network conditions and to modify its behaviour to compensate for the network delay. The proposed framework uses an asynchronous configuration: the aggregation strategy is an asynchronous version of FedAvg, with the advantage of being tolerant to stragglers; stragglers updates are scaled based on their staleness degree and then included by the server.

Acknowledgements

I would like to warmly thank my supervisors, Claudio Ettore Casetti and Paolo Giaccone, for their support, effort and patience. I am thankful to Riccardo Rusca and Alessandro Cornacchia for their precious encouragement, help and support.

My heartfelt gratitude goes also to my family and my friends.

Contents

List of Figures

List of Tables

Acronyms

asyncFedAvg asynchronous Federated Averaging.

DNN Deep Neural Network.

FedAvg Federated Averaging.

FL Federated Learning.

IID Independent and Identically Distributed.

IoT Internet-of-Things.

ML Machine Learning.

NN Neural Network.

RTT Round Trip Time.

SGD Stochastic Gradient Algorithm.

Chapter 1

Introduction

Machine learning is a computer science field that deals with programming softwares which are able to learn from experience by training models on a real-world dataset. Thanks to its ability to adapt to different scenarios, machine learning is employed in the most diverse areas: from social media recommendation engines to self-driving cars, health and care, speech recognition, finance, just to mention a few. In recent years, ML research started to move towards a distributed environment and to explore solutions to train models by exploiting multiple devices and their heterogeneous data: the explosion of the Internet-of-Things market made it the favourite candidate application scenario for distributed machine learning, or *federated learning*.

Internet-of-Things (IoT) technology represents a complex system composed by a huge number of devices, each able to retrieve and elaborate data; these devices are interconnected through the internet and able to exchange information between them. Lately, onboard computational capabilities have increased considerably while maintaining a low cost, making the IoT devices suitable to perform complex tasks like federated learning training.

Federated learning is a machine learning technique where multiple devices collaborate to train a shared machine learning model. Differently from centralised ML, this technique does not need to collect in a single entity the datasets owned by the devices to be used in the model training, but devices are deputed to perform the training step themselves and just send their local model parameters: in this way, data disclosure is not necessary (preserving data privacy) and the communication overhead is reduced. However, FL presents new challenges with respect to centralised ML, as it involves multiple devices with heterogeneous resources and the communication through a network.

In a synchronous FL architecture, a round ends when the server collects all the results from the selected participants: each round completion time is ruled by the slowest client, which becomes the bottleneck. The client slowness may be due to the poor computational capacity of the device, to a poor network connection, or to a combination of both. In this work, we focus on the network delay and bandwidth effect on the server-client communication and the impact this has on a FL round.

In order to counteract the network variability, we adopted an asynchronous model: the server sets a round timeout and clients who cannot be compliant with it are considered stragglers. The second step was to adapt the FedAvg aggregation strategy in order to provide tolerance towards stragglers: updates that are able to reach the server in a later round are still included in the aggregation, and their weight is scaled proportionally to their degree of staleness.

We also modified the classic centralised FL structure by adding an additional player: the mediator. A number of mediators is inserted between the server and the clients and they coordinate the communication between the two parties. The mediators are positioned on the cloud edge, so to reduce as much as possible the clients' updates transmission latency; also, we consider that the bandwidth and delay between the mediators and the server is fixed and known.

To directly address the network delay, a request-acknowledge procedure is implemented between mediators and clients. After selecting a client, the mediator sends a request frame to the client, and the client replies with an ack frame: the RTT time is measured and, based on this knowledge, the mediator modifies the client instructions to control its behaviour and make it compliant with respect to the timeout.

The proposed framework was implemented by integrating Docker [2], Flower [3] and OMNeT++/INET [4] [5] as main components; the platform was then experimentally evaluated against a synchronous FL setup with FedAvg aggregation strategy.

The rest of this work is organised as follows.

Chapter 2 introduces the main characteristics of machine learning and shows the most relevant solutions that have been proposed in literature to move the process from a centralised approach towards a distributed one.

The chapter then describes in greater detail the characteristics of the federated learning approach to machine learning, its resources and the open problems it offers. We present the main state-of-the-art solutions that have been proposed in the literature, analysing the network topology, the synchronism of the architecture, the aggregation scaling rules and the FL frameworks. Finally, we review the main papers which have been the starting point for this work.

In chapter 3, we explain the network impact issue of federated learning, which is the main focus of this work: network delays and small bandwidths may lengthen the duration of a round. We then delineate our model architecture, along with the related design choices. We brought two main contributions with our system: we firstly modified the classic FL structure by introducing a new system component, the mediator, which is an additional intelligent piece inserted in between the server and the clients. The mediator behaviour can be completely controlled by the server or it can have perform independent decisions on how to handle the clients connected to it; in our framework, we implemented an ack procedure between mediators and clients to estimate and compensate the link delay. We also built *asyncFedAvg*, an asynchronous aggregation strategy able to cope with stragglers.

Chapter 4 describes how the system model was implemented in an experimental environment using three main components. We started from the Flower federated learning

library and we customised it to implement our structure. We encapsulated the FL workers inside Docker containers to control the network connectivity and the CPU usage. We eventually connected the workers to a network emulator based on the OMNeT++/INET library so to obtain a flexible network topology.

In chapter 5, we analysed the single framework components and their performances in isolation, to find the experimental limitations. We then describe the experimental setup of the simulations and we show and comment the results. We performed three kinds of simulations: we firstly compared the synchronous and the double-layer asynchronous architectures (without the ack procedure); next, we evaluated the influence of the position of the mediator in the edge-cloud continuum on the FL process; the last experiment was to evaluate the complete framework with the ack procedure in a heterogeneous scenario, providing both network and computational capability variability.

In conclusion, chapter 6 summarises the main results of this work and presents some possible future directions.

Chapter 2

Background: federated learning

This chapter briefly introduces the core aspects of machine learning in section 2.1, discussing the directions taken in literature to move from centralised ML towards distributed computing. It then focuses on the explanation of the federated learning approach on machine learning in section 2.2, considering its strengths and weaknesses (described in section 2.2.1) and then presenting the main characteristics of the process. In conclusion, the chapter ends with a brief description of the papers which have been our main starting points (section 2.2.2).

2.1 Introduction on machine learning

Differently from traditional computer programming, where a set of rules specifies the behaviour of the software, *machine learning* is generally known as the practice of letting a software learn from experience and execute an assignment without explicitly defining its actions beforehand [6]. ML is able to perform this complex action by carrying out data analysis on a real-world dataset and learning to forecast patterns by building a prediction model: this model is considered the final product of the process and, once trained, is aimed at autonomously making classifications or predictions on unknown data.

Thanks to its ability to adapt to different scenarios, ML is employed in the most diverse fields: from social media recommendation engines to self-driving cars, health and care, speech recognition, finance, just to mention a few.

The function of machine learning worldwide has become more and more important in the last years, and literature is copious. A number of algorithms has been proposed and is commonly used in ML, depending on the nature of the problem to solve and on its complexity: in this work, we chose to use a deep neural network because of its easy-to-understand underlying concepts and the straightforward deployment. The next section is dedicated to explaining its characteristics.

Deep Neural Networks

A neural network aims at artificially imitating the way the human brain works by using a mathematical model that simulates the interconnections between neurons. The nodes in the model are called *perceptrons* and their aggregated form is called network layer. The structure of a perceptron is shown in fig. 2.1 and its operation consists of two steps:

1. **weighted sum**: weights are necessary to determine and assign the importance of the connection with another perceptron with respect to the other connections. The output of this first step is defined by the equation

$$z = \sum_{j=1}^{n} i_j \cdot w_j \tag{2.1}$$

   where i_j is the j-th input, w_j is its weight and n is the total number of inputs belonging to the perceptron.

2. **activation function**: the activation function is an output-limited function that takes the general form $o = f(z + bias)$ and modifies the result of the weighted sum z, forcing it to be within a specified range of values. Many activation functions with different shapes have been developed to provide varied behaviours. Apart from the shape, activation functions have another characteristic parameter called *bias*, which is responsible for dynamically shifting the function on the x-axis during training.

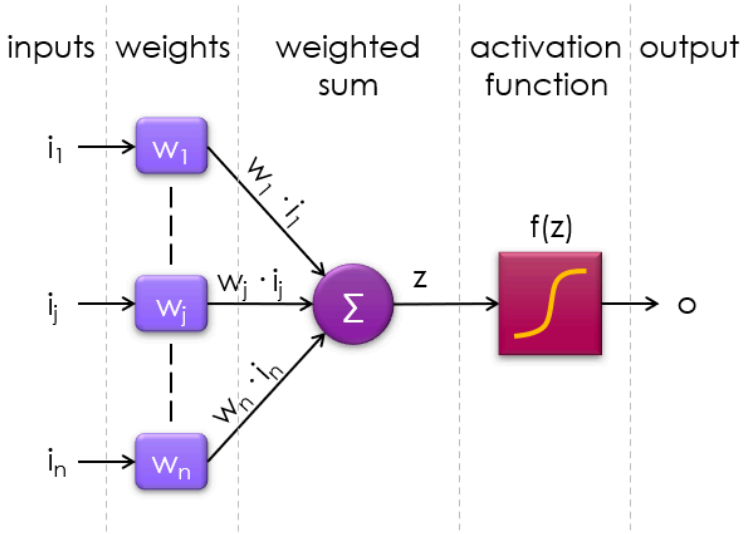

Figure 2.1. Schematic of the perceptron.

The structure of an artificial neural network consists of an input layer, a number of hidden layers and an output layer (fig. 2.2): if the amount of hidden layer (n in the figure) is greater than 3, the network is called *deep* neural network (DNN). Each layer is composed by many perceptrons, whose inputs are interconnected with the previous layer outputs, and whose outputs represent the inputs of the next layer. Generally, the perceptrons belonging to the same layer behave in the same way, which means they have the same activation function.

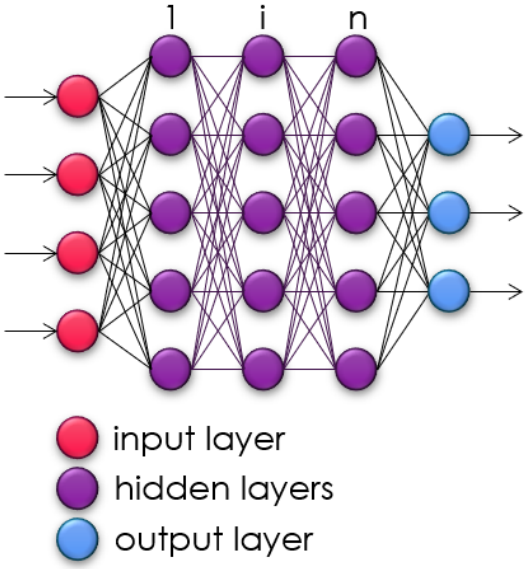

Figure 2.2. Schematic of a deep neural network: the nodes are the perceptrons depicted in fig. 2.1. n indicates the number of hidden layers.

The weights and biases previously described represent the network parameters that define a trained model. *Backpropagation* is a method of adjusting and fine-tuning these parameters. The training happens at first by processing the dataset *forwards* in the network, that is, from the input layer across the hidden layers to the output layer. In general, at the beginning, the weights are randomly chosen and the biases are null. After each iteration (*epoch*), the difference between the actual output and the desired output is computed: this error measure is used to revisit and recompute all the weights and biases in the network in the *backwards* direction, that is, from the output layer towards the input one. The objective of this phase is to reduce the error in the next epochs. At the end of the learning process, thanks to backpropagation the network parameters will be trained and the final model will have reached the desired value of accuracy (the fraction of overall correct predictions)[1].

[1] Accuracy is not the only metric that can be used to evaluate a model: other metrics exist in literature and they should be chosen in relation to the classification problem under analysis.

Because of the challenges that ML poses when applied to IoT or privacy-preserving scenarios, different approaches are required to better adapt the learning to these distributed environments. The new methods should be communication-efficient, computationally conservative and they should also provide tolerance towards delays and unstable connections, in addition to being privacy compliant: to achieve this, the idea is to move the training closer to the data. The literature has proposed several solutions to bridge the gap between centralised ML and distributed ML: the main techniques, described in the next sections, are distributed on-site learning, split learning and federated learning. Figure 2.7 shows a synthetic schematic of these strategies.

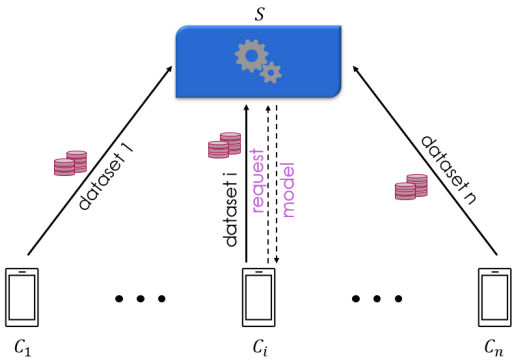

Figure 2.3. Centralised machine learning.

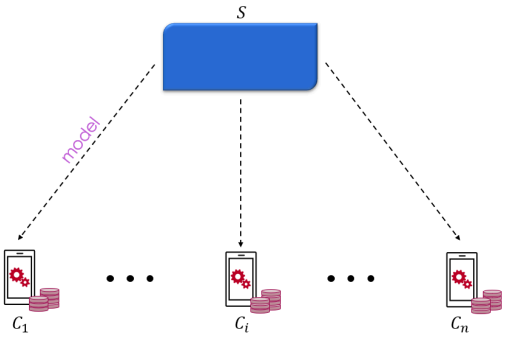

Figure 2.4. Distributed on-site learning.

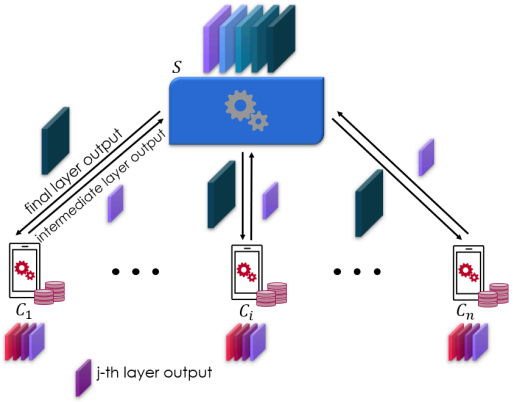

Figure 2.5. Split learning.

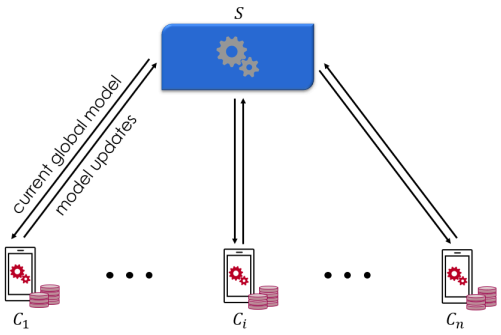

Figure 2.6. Federated learning (star topology).

Figure 2.7. Schematic of centralised ML (fig. 2.3), distributed on-site learning (fig. 2.4), split learning (fig. 2.5) and federated learning (fig. 2.6) in their most common configuration.

2.1.1 Centralised machine learning

In the classic schematic of *centralised machine learning*, the involved devices generate their datasets and send them towards a high performance server; the server uses the obtained data to build the final model, which is subsequently sold as a service (fig. 2.3).

ML techniques can have either a supervised or an unsupervised paradigm. *Supervised*

learning builds the prediction model based on both an input dataset and known responses to the data (output): in other words, it is able to measure the difference between the model output and the desired model output, which is then used to correct and adjust the model parameters. This approach can use classification techniques, where data is split into categories, or regression techniques, where the response is in a continuous range. On the other hand, *unsupervised learning* only uses unlabeled input data to draw its prediction model without referencing the desired output: here, the human classification of data is not necessary. The most used unsupervised technique is the clustering one, which is able to autonomously find hidden patterns in the data by grouping it into bundles.

In addition to supervised and unsupervised learning, a third paradigm of ML, called *reinforcement learning*, uses unlabelled data to achieve the goal of maximizing some form of reward, learning through trial and error. It can be used in a real-time fashion as it does not need to rely on a fixed set of static data, and it does not need human supervision. Each decision is either rewarded, if it corresponds to a desired behaviour, or punished with a penalty, if it corresponds to a negative behaviour. Neural networks, described in a previous section, can be used with supervised, unsupervised and reinforcement learning paradigms, which makes them very flexible.

The main advantage of centralised ML is the possibility to use a powerful server which is omniscient with respect to the overall training dataset. The server can exploit all the available data, without having to choose blindly between subsets of it; outliers may be easier to identify and remove with such a visibility. Also, the server does not suffer from low computational capacity and battery consumption issues typical of IoT devices. Once the server has collected all data, the network variability does not come into play anymore and there is no additional communication overhead.

A ML server needs to have powerful equipment to achieve competitive performances: fast CPUs and GPUs are necessary, along with a large system memory to store the great amount of training data. To avoid the need to design and build on-premises workstations, there are many solutions to obtain machine learning in the cloud. This helps to reduce the deployment costs by scaling the ML projects as they go from a laboratory environment into production. Also, cloud ML providers offer pre-packaged distributions that do not require previous expertise in the artificial intelligence field.

There are also some downsides tied to the centralised-server ML approach, in particular:

- data privacy is at risk and users may be reluctant to participate in the process;
- the data communication overhead is high because the data volume may be large;
- the latency may be high, as the data may cross the globe before reaching the server. This makes centralised ML unsuitable for latency-constrained applications, typical for the mobile devices.

In order to reduce latency and communication cost, *edge computing* has been introduced, which moves the server closer to the edge in the edge-cloud continuum and consequently closer to the clients, too. However, to better address and solve these problems, especially data privacy, another approach has been introduced: the *distributed learning*.

2.1.2 Distributed on-site learning

The first evolution of ML, the so-called *distributed on-site learning* [7], moves some of the intelligence of the system inside the devices: a pre-trained or general model is shared by the server and it is then customised internally by the clients by using their own training datasets (fig. 2.4).

One of the disadvantages of distributed on-site learning is that knowledge is not shared across different devices and the final model may be biased due to small or incomplete local datasets. However, data privacy is clearly preserved, and there is virtually no communication overhead because there are no client-server messages (except for the initial model download).

2.1.3 Split learning

Split learning is a hybrid approach, collocated in between centralised machine learning and edge computing: this technique proposes to divide (*split*) the DNN models into head and tail parts [8] [9]; the computationally weaker device performs the initial calculations based on its data, then the results are sent to the more powerful edge server, which executes the remaining computations (fig. 2.5). In this way, the overall computational load is better distributed between clients and server.

Only the intermediate model parameters are transmitted over the network, which reduces the communication time, in comparison with the time needed to send over the whole dataset. To further reduce the communication time, split computing literature has also been oriented to produce task-oriented compression schemes for these intermediate updates.

The decision of where to split the network introduces a tradeoff between performance (model accuracy) and the size of the intermediate model parameters, which is proportional to the communication time: depending on the design of the DNN, the initial layers may amplify the size of the parameters; the model may also branch to produce intermediate results for later layers, increasing the number of parameters to be sent.

2.2 Federated learning: state-of-the-art

Federated learning is a process which has been recently introduced to evolve machine learning and bypass some of the problems it involves: it can be considered as privacy-preserving, decentralised machine learning in a collaborative environment [10]. As in centralised ML, the aggregation process is usually controlled by a parameter server, but now an additional step of local training is added in the devices. In this way, data is not sent over and the communication overhead reduces significantly.

The general idea behind federated learning is that the participant clients are non-trusting and do not disclose their own datasets. They train the global model on their local data and just sent the model parameter updates over the network; these parameters will then be aggregated to update the global model. This process will be repeated until some predefined metric is achieved (e.g., a certain level of accuracy is obtained).

Because of its distinctive features, federated learning is particularly indicated for IoT networks [11], as it can provide data privacy enhancement, low-latency network communication (with respect to centralised machine learning), and a higher learning quality. Another benefit of using FL in an IoT scenario is that real-time learning can be achieved: models are continuously refined with local data, which is a far more responsive technique than centralised ML. This work is targeted towards IoT applications (which introduce distinctive restrictions on the devices capabilities) and also towards mobile edge computing.

The main aspects of a FL system are introduced in the next section.

2.2.1 Available resources and open problems

Due to the dynamic nature of the environment where federated learning is applied, the available resources are afflicted by an unpredictable quality which makes the coordination of the learning process challenging. Also, FL shows different weaknesses with respect to centralised machine learning, and they need to be properly tackled not to impact the final model KPIs [12].

The main variables involved in FL are briefly described in the following:

- *clients' capabilities*: the devices usually have different capacities in terms of storage, computation and battery; the available CPU that each device devotes to FL may vary a lot depending on the time of day, current usage of the device, etc.

- *clients' datasets*: the datasets are generated and collected locally by the devices, therefore they could be non-IID and affected by noise, outliers, etc. The global data distributions may therefore be unbalanced and biased. However, a unique and divergent dataset is very valuable for the global model accuracy.

- *clients' reliability*: the devices' connection to the server should be considered unreliable, as they could drop out of a round mid-run due to energy or network limitations (batteries may run out, mobile devices may fall out of the coverage range).

- *network characteristics*: the network can be either wireless, wired, mobile, ad hoc or it can be a sensor network, each with their own peculiarities; network bandwidth and delay may not be deterministic, but vary across devices and time. Environmental characteristics will therefore cast a different shape on the round computation time, in addition to the effect brought about by client heterogeneity.

- *communication cost*: network connections are metered, and, for this reason and considering that the number of participants may be huge, it is important to limit the number of message exchanges.

- *vulnerabilities*: a cooperative model like the federated learning process may be subject to security risks on multiple fronts; clients may be passively malicious and listen in to "steal" model updates, or actively malicious by poisoning either the dataset or the model update.

To address these new challenges, many aspects of FL have been analysed: the literature has been focusing mainly on the network configuration, the synchronous/asynchronous behaviour of the learning, the learning strategies and the client selection. The most relevant FL architecture categories are described in the next sections.

Network topology

There are two main network topologies that have been explored in federated learning literature:

- *mesh topology*: in this configuration, there is no central coordination unit and devices are connected with peer-to-peer links, as shown in fig. 2.8. Each round, every client performs its local training (client selection is not enforced) and the global update is obtained by collecting the neighbouring model parameters.
- *star topology*: a centralised server is connected to a pool of available clients (shown in fig. 2.9). The server acts as a controller by selecting which devices will participate in a round and coordinates both the communication and the aggregation processes. In this architecture, the server supervises and manages the behaviour of the clients.

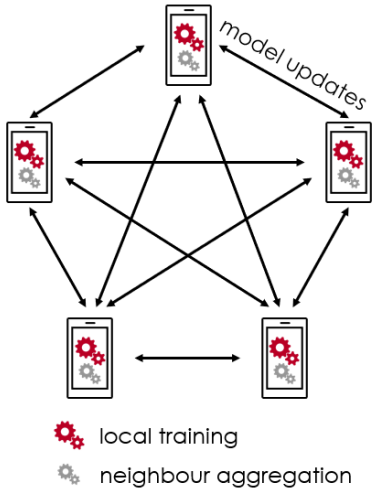

Figure 2.8. Mesh topology

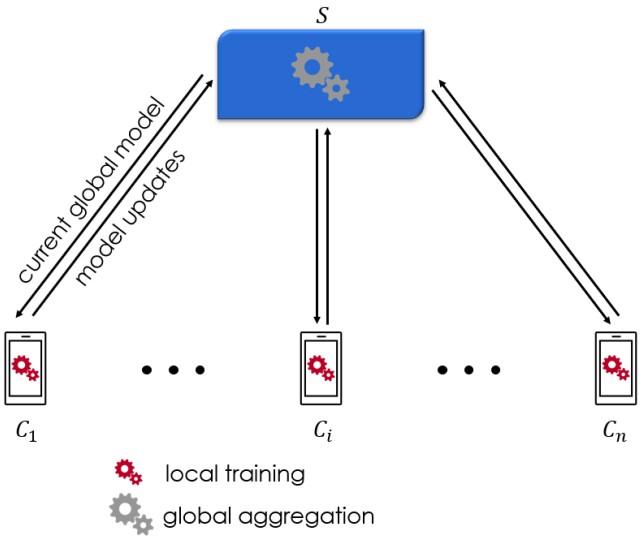

Figure 2.9. Star topology

Figure 2.10. Schematic of the mesh and star topologies for FL applications.

In the following, we will focus on the star topology configuration, which is the most used one in literature because of the significant system control capability it can provide.

Synchronous versus asynchronous architectures

A typical FL run consists of a series of federated rounds; each of them involves a *training phase*, during which the on-device training is performed on the selected clients, alternated

with the *central aggregation phase*, where the updates are collected and aggregated by the server. A federated round may then end with the current model *federated evaluation phase*: another subset of selected clients receives and evaluates the current model parameters and the evaluation metrics are again collected by the server.

A federated round is characterised by a timeout set by the server: this timeout represents the system time unit and it can be considered either a hard or soft limit for updates to be received, based on the system synchronism. With respect to the server timeout, a *straggler* can be defined as a client which was selected for a training round, but missed the round timeout and the server could not receive its update in time to aggregate it. Different solutions have been proposed in FL literature with respect to timeouts and stragglers, in particular:

- *synchronous training round*: in a synchronous round, shown in fig. 2.11, the server selects K participants out of the pool of available clients and collects their results: the round ends when the last update is collected. As a worst case, the server sets a timeout T: if less than K devices have committed their results before T expires, the round fails and is rescheduled. In this scenario, the presence of stragglers impacts the training round tragically, since they may cause the round to fail; also, stale updates are wasted even if they reach the server. To counteract stragglers, the usual solution is to overcommit the number of participants by a certain percentage (typically, the 30% of the target K), so not to depend too much on the crashed or slow devices by selecting additional backup workers.

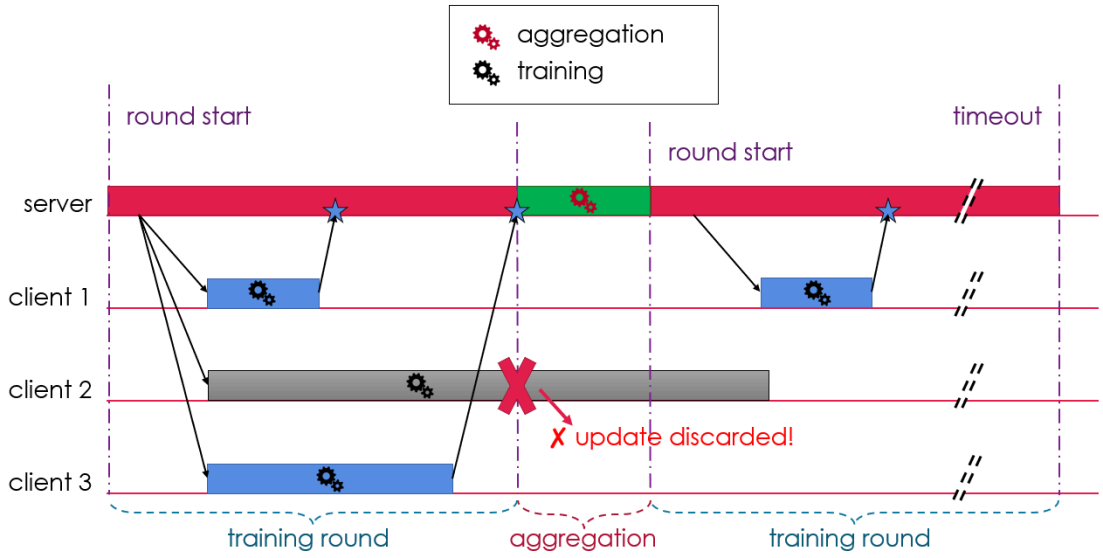

Figure 2.11. FL round with synchronous communication: two results are needed to close a round in this example. Detail of a straggler participant.

- *asynchronous training round*: asynchronous communication is similar to the synchronous one, as in both of them the server sets the round timeout, but the two differ in the way straggler updates are treated. Figure 2.12 shows the same scenario as in the synchronous case, but now the stale update from client 2 is kept in memory and aggregated in the second round. Asynchronous aggregation, however, must take into consideration the staleness degree of the updates: different aggregation strategies have been proposed in literature to take care of this staleness issue.

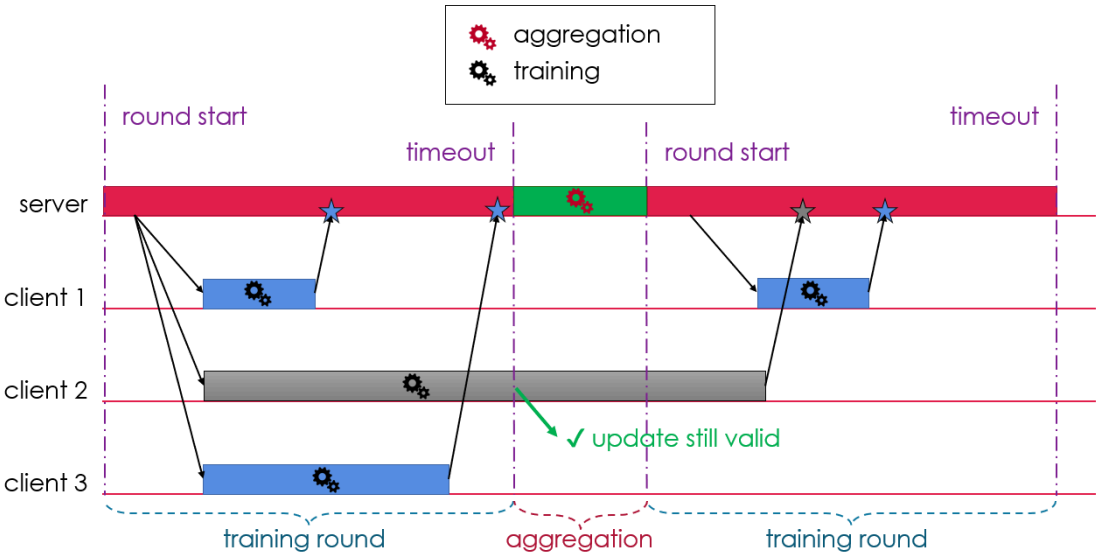

Figure 2.12. FL round with asynchronous communication. Detail of a straggler participant.

- *asynchronous training round (Astraea definition)*: there exists an additional definition of asynchronous training round provided by Astraea [1]. This paper proposes a sequential round training, where one selected client provides its model parameters, which are updated inside the centralised controller (the mediator acts as the server in this case, but does not perform any aggregation); the controller then selects the next client, which will perform the training on these intermediate model updates, and so on (see fig. 2.13). However, this solution does not exploit parallel computation on multiple devices.

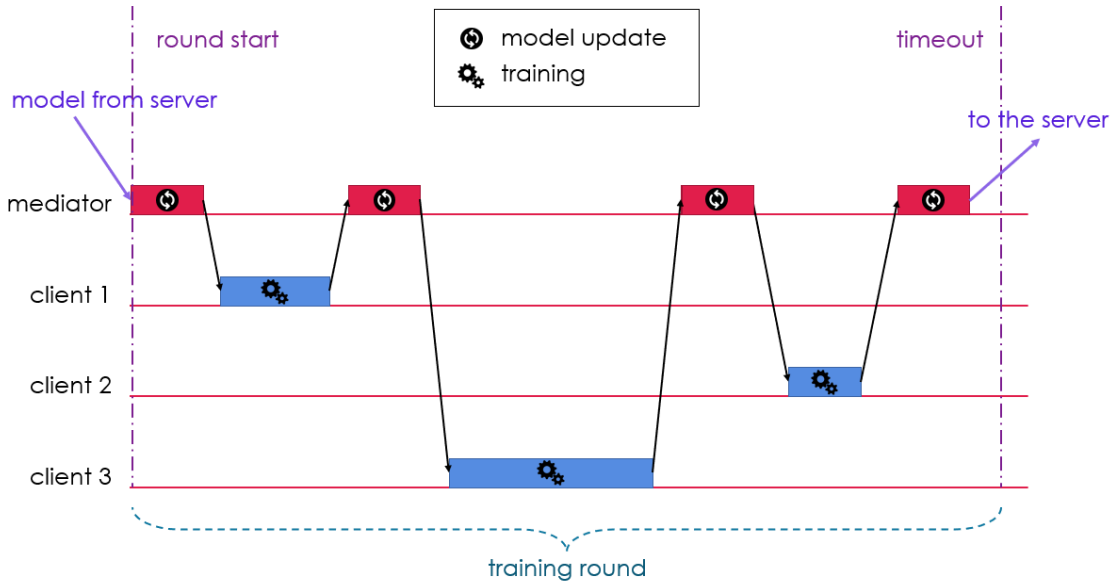

Figure 2.13. Astraea [1] definition of asynchronous round.

In general, both synchronous and asynchronous approaches are used in FL. Synchronous schemes are simpler to analyse and the updates are always serially equivalent; also, they provide better protection from privacy-related attacks. However, they waste stale updates which are still able to reach the server and may contain valuable information. Asynchronous communication is useful in highly heterogeneous scenarios, where the probability of stragglers cannot be neglected, but it implies troubles regarding privacy and scalability.

In the asynchronous architecture, stale updates are built on an older version of the current model and their weight in comparison to up-to-date results should be modified accordingly. The asynchronous aggregation techniques usually rely on bounded-delay assumptions to limit the degree of staleness. The following list presents the state-of-the-art scaling rules (which measure the importance of an update based on its staleness and possibly other parameters) that have been proposed in the FL literature:

- **SSGD**: stalenes-aware SGD algorithm, a modified version of the classic SGD typically used in machine learning. The stale updates and the up-to-date ones have the same weight[2].

$$w_S = 1 \tag{2.2}$$

where w_S is the straggler's weight.

- **DynSGD** [13]: the stale updates' weight decreases proportionally to their degree of staleness.

$$w_S = \frac{1}{\tau_S + 1} \tag{2.3}$$

where τ_S is the number of round timeouts that the client has missed before providing its update (id est, the number of staleness rounds).

- **AdaSGD** [14]: the stale updates' weight decreases exponentially with respect to their degree of staleness, but they are enhanced if they contain relevant information.

$$w_S = e^{-(\tau_S+1)} + boosting \tag{2.4}$$

where the exponential term is the staleness-aware dampening factor and the boosting is computed as $boosting = \frac{1}{sim(x_S)}$, based on a similarity function which gives an indication of the data novelty of the client.

- **RELAY** [15]: the scaling rule of DynSGD is riproposed, together with an exponential enhancement factor. A tunable parameter β is responsible for tuning the weight of the stale updates, balancing the dampening with the boosting.

$$w_S = (1-\beta)\frac{1}{\tau_S + 1} + \beta(1 - e^{-\frac{\Lambda_S}{\Lambda_{max}}}) \tag{2.5}$$

where the boosting factor is based on the term Λ_S (the deviation of the stale update from the average of the fresh updates), divided by the maximum deviation from all the stale results, Λ_{max}. If an update is very different from the average, with this scaling rule its weight increases even if its staleness is high.

Federated learning frameworks

Many FL frameworks have been proposed, both for research and real deployment purposes. A concise description of the most relevant ones follows:

- *Flower* [3]: Flower is an open-source platform-independent framework which is particularly relevant for its attribute of easy customisation with respect to all aspects of federated learning: the implementation of client/server behaviour, the exchanged messages, the learning strategies and even the whole system architecture can be modified and shaped using Flower to explore new FL scenarios.

[2]The assumption is that the up-to-date parameters have weight 1.

- *IBMFL* [16]: IBMFL is a black-box Python framework realised by IBM. Here, only the dataset and the neural network model are customisable. IBMFL provides a fast start-up time for real application deployment and also an straightforward experimental environment to test ML models in a federated setting.

- *FLSlim* [17]: FLSim is a standalone library supported by Facebook research that provides high-level API calls to manage the FL process; it is scalable and open-source, therefore also simple to customise. It is written using the PyTorch library, which denotes an object-oriented approach.

It is worth noting that all the presented frameworks adopt the centralised FL architecture with the star topology.

2.2.2 Related works

The following list presents the most relevant ideas introduced in the papers that have been the main starting point for this work.

- ***Oort*** [18]: Oort is an informed participant selection framework that preferentially selects learners with a higher *utility*. Utility is defined as the combination of statistical utility (measured using the training loss as a proxy, to favour valuable clients) and system utility (measured as a function of the completion time, to favour fast clients). Oort also uses a pacer algorithm that can trade a longer round duration to include unexplored or slow learners, when required for statistical efficiency.

  The chosen evaluation metric is time-to-accuracy, that is total completion time with respect to final model accuracy, and the selection algorithm selected for comparison is the random selection. Some of the downsides of Oort are the need for an accurate estimation of the clients' utility and the lack of data diversity.

- ***SAFA*** [19]: SAFA is a semi-asynchronous protocol that enables updates from straggling participants. It flips the participant selection process of FedAvg strategy (see section 3.2.1 for more details on this strategy): training is run on all learners, and a round is ended when a pre-set percentage of them returns their updates (process called *post-training client selection*, combined afterwards with discriminative aggregation). Participants are allowed to report after the round deadline and their updates are cached and applied in a later round; however, SAFA only tolerates updates from learners that are within a bounded staleness threshold. Therefore, the round duration is reduced by only waiting for a fraction of the participants, while the cache ensures that the computational effort of straggling participants is not entirely wasted and is still able to boost the statistical efficiency.

  The evaluation metric is time-to-accuracy; FedAvg, FedCS [20] and fully local training are the benchmark strategies for comparison. The main disadvantages of SAFA are the huge waste of resources as the number of total clients increases, and the bias introduced in the client selection due to the divergence between speed and importance (in terms of valuable local data) of the learners.

- ***RELAY*** [15]: RELAY proposes to maximise the system resource efficiency in terms of cumulative hours spent in training. It is based on *intelligent participant selection*: the participants target becomes adaptive, that means that the default number of polled clients may vary based on the stragglers behaviour during a round; also, least available prioritisation is applied to boost data diversity, where learners availability is defined as when a device is connected to a charger, idle and using an unmetered network (availability is usually limited in time). In addition to this client selection algorithm, RELAY proposes a *staleness-aware aggregation*, a stale synchronous version of the FedAvg aggregation strategy combined with a scaling factor to properly weigh stale updates based on their similarity to the model.

  The comparison metrics are the training performance in non-IID settings in terms of cumulative resource usage and model accuracy. Some of the weaknesses of RELAY may be that the learners are supposed to periodically train their future availability model and to estimate their remaining training time when they become stragglers.

- ***FLEET*** [14]: FLEET enables stale updates robustness by introducing AdaSGD, an ad hoc version of the classic Stochastic Gradient Algorithm, which adopts a dampening factor on out-of-date results to give them a smaller weight as their staleness increases. This has the advantage of not discarding updates that exceed the staleness threshold. However, the AdaSGD protocol is not directly compatible with the traditional FL settings such as FedAvg.

  The metrics for comparison of AdaSGD are DynSGD [13], the staleness-aware SGD (SSGD) and the staleness-unaware SGD (FedAvg). One of the downsides of AdaSGD is that the clients have to send their data distribution beforehand, that may affect privacy. Also, differently from FedAvg, FLEET synchronises model gradients after every single mini-batch.

- ***Astraea*** [1]: Astraea is the first paper to propose a different system architecture for FL. In between the clients and the server, a number of *mediators* is added, which have the role of moderating the client-server communication. Before the FL starts, a scheduling phase is performed, where the clients' assignment to the mediators is balanced based on their data distribution: the objective of this is to ensure that each mediator is provided with a diverse pool of client datasets, as unbiased and representative as possible.

  In this framework, the server supports a global model with synchronous updates from the mediators, while the mediators support asynchronous updates from their connected clients. Astraea also proposes an initial rebalancing phase where data augmentation is performed on clients.

  The experimental comparison of Astraea is done with the FedAvg strategy. One of the disadvantages of this approach is that the rebalancing and scheduling phases require additional time to be performed if the data distribution rapidly changes; also, clients may be unwilling to share their data distribution with the server.

Chapter 3

Framework model

This chapter is dedicated to presenting the network impact issue on federated learning in section 3.1, which is the main problem we address in this work, and to describing the system we developed to counteract this issue in section 3.2.

Our main contributions are an asynchronous model tolerant to stragglers, described in section 3.2.2, and the addition of the mediators in the client - server architecture, described in section 3.2.3. Section 3.2.4 explains how our complete federated learning model turns out to be.

3.1 Network effect on federated learning

One of the fundamental aspects of federated learning is that the workers communicate through a complex network, which may have different characteristics and properties. The network plays an important role in the process, since it dynamically affects the system timing.

In this section, we will consider a synchronous architecture with a star topology as our baseline. A FL round ends when the server collects all the results from the K selected participants. In this configuration, each round completion time is ruled by the slowest client: the slowness may be due to the poor computational capacity of the device, to the network delay, or to a combination of both. The slowest selected client sets the round duration as well and represents the round bottleneck.

Figure 3.1 shows an example of a pool of six available clients with different hardware and network characteristics; the y-axis represents the total time needed by a client to compute and send back the model parameters. Instead of random client selection, assume that the server is omniscient and has perfect knowledge about the training speed of the clients, and it will use this knowledge for an informed client selection; also, suppose that the server needs three participants out of six available clients to complete a FL round.

By looking at the computational capabilities only, the server would choose the fastest clients so to reduce as much as possible the training round duration, and the choice would be 2, 3 and 6; however, even in this ideal scenario, the network effect would impair the round duration, since the best network conditions are held by clients 1, 3 and 4. Considering both the computational and transmission times, the *joint* overall lowest round duration is achieved by selecting clients 3, 4 and 6, which would be the optimal choice.

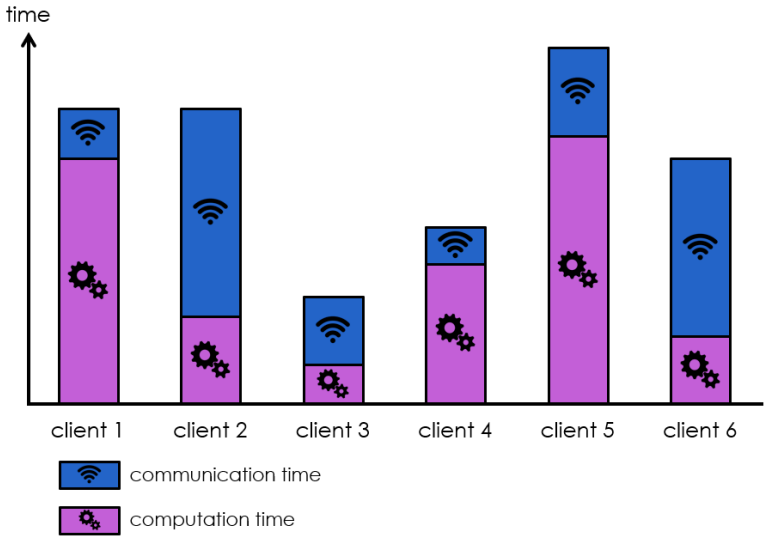

Figure 3.1. Computational capabilities versus communication time for a pool of six clients.

When talking about synchronous FL, causes for troubles due to the network effect include the fact that the network *uplink/downlink bandwidth* for each client may vary in time, or be statically allocated; *network delays* also play an important role in the server-client communication. Issues arise when the delay is combined with the slower training participants, that consequently become the round bottlenecks.

From our research, in the literature the network problem in FL has not been directly addressed. The main focus of this work is to consider and investigate the effect of the network bandwidth and delay on a FL round and to try and develop a model which compensates for it in an efficient way, so to reduce its impact on the computation time of the whole learning process.

3.2 Model description

As explained in the previous section, the synchronous FL architecture is afflicted by the bottleneck of the slowest client. In order to try and bypass this problem, we propose a completely asynchronous architecture which is governed by the server timeout as the system time measure, and client updates who miss the timeout can still be incorporated in the model aggregation. Also, we propose a new structure, shown in fig. 3.2: a number of components named *mediators* is added between the clients and the centralised server, whose purpose is to mediate the communication between server and clients by reducing the overall time each update needs to reach the FL infrastructure.

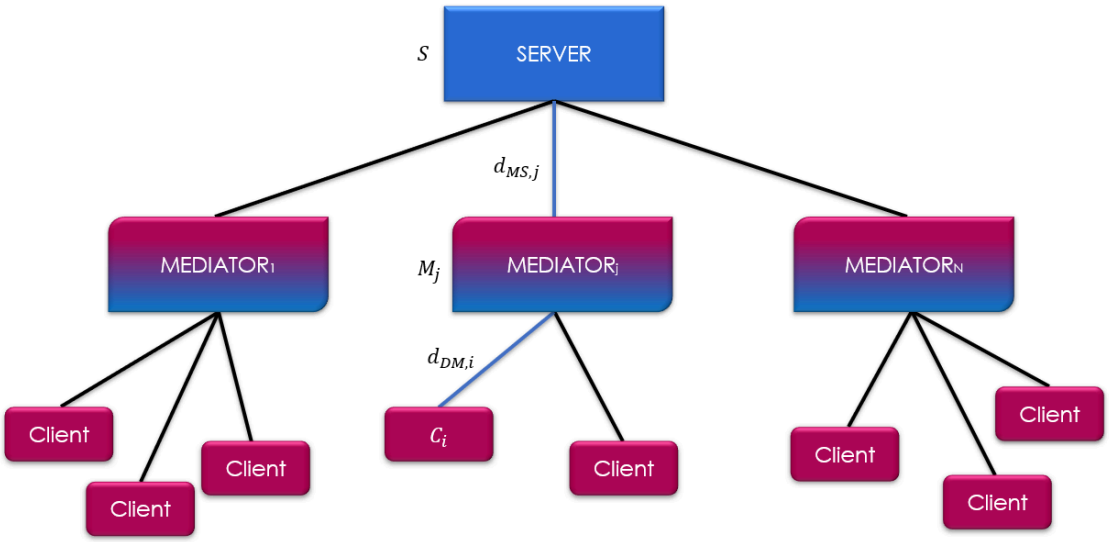

Figure 3.2. Schematic of the double-layer architecture.

3.2.1 *FedAvg* strategy and client selection characteristics

One of the first FL aggregation strategies proposed in literature is the FedAvg strategy [21]: it is often used in practical applications and it represents a classical comparison benchmark for presenting new strategies. FedAvg involves a first local SGD[1] training step on the data owned by the devices, and a later centralised aggregation step which happens inside the server; this aggregation consists of a weighted average of the local models. FedAvg works very well in IID scenarios, but is less performant in a heterogeneous environment, when devices generate non-IID data distributions. This strategy is

[1]SGD (Stochastic Gradient Descent) is an optimisation method that aims at finding a local minimum of a function by computing an estimate of its gradient. It can be used to train a model and has become very popular in machine learning as it is a cost-efficient approach, when applied to large scale learning problems.

synchronous: a round closes until all chosen devices (or a percentage of the total number of participants) have returned their results.

Client selection characteristics

In general, the client selection algorithm is not part of the aggregation strategy, as they happen in two different moments of the FL process: aggregation strategies can be combined with different client selection techniques.

FedAvg operates with a *ahead-of-training selection*: the client selection happens before the training starts, and the fit instructions are sent only to the chosen participants. The success of a round, however, depends on the availability of the chosen participants, which are unreliable and may crash mid-run: a usual technique is to overcommit the number of clients with respect to the number of needed results, so to statically compensate for crashed/delayed clients. Other approaches, like [19], flip the two FL steps: local training is run on all clients and the selection is applied a posteriori on the returned results. The obvious disadvantage of this method is the waste of resources, as a portion of the results are not aggregated in the final model.

Focusing on the ahead-of-training selection, which is the most explored in literature, there are many possible alternatives: however, an obvious limit of many selection algorithms is that they require some prior knowledge on the clients' characteristics, which is a data privacy weakness; another frequent feature is that they may infer clients' data quality, but this usually introduces biases in the selection. Random selection, instead, has the dual advantage of being fair and not requiring any extra information on the participants. This passive selection algorithm is the most used and also the simplest one.

3.2.2 Introducing tolerance to stragglers: *asyncFedAvg* strategy

Starting from the baseline provided by the synchronous strategy FedAvg, we build an asynchronous strategy, named *asyncFedAvg*, able to consider and properly treat stragglers. The idea behind an asynchronous strategy is that the server sets a timeout T to the FL round: when the timeout expires, the server closes the round and aggregates the results received up to that point. Stale updates that arrive after the timeout are stored in memory and aggregated in the next round.

In principle, an asynchronous FL round could end when all selected K participants send back their results, exactly as in a synchronous FL architecture, meaning that the server could end the round even before T has expired; however, we choose not to stop rounds earlier than T so to be fair towards straggler devices from earlier rounds. The only case where a round is closed earlier is if all straggler and fresh updates have been received before T. Figure 3.3 shows an example of how an asynchronous round works in our architecture. The server chooses two clients among a pool of three total clients; the results from the current participants are received before the timeout has expired. However, the server still awaits for T to elapse: in this way, the update from the straggling client 2 can be aggregated in the current round and not in the next one, where it would have been penalised with a smaller weight due to its higher staleness degree.

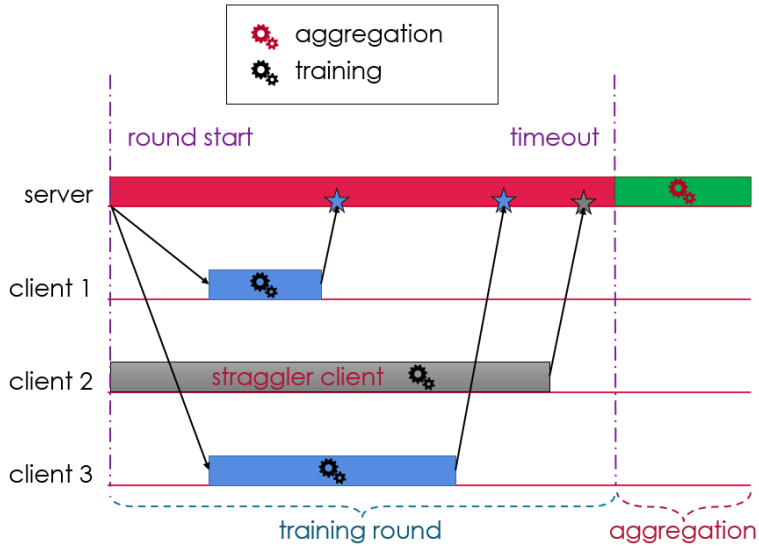

Figure 3.3. Example of an asynchronous round. The server selects 2 clients, therefore it expects 2 fresh updates; however, it still waits for the timeout to expire before closing the round, even if it receives them beforehand.

By applying asyncFedAvg to federated learning, the maximum duration of the FL process depends just on the round timeout T, therefore it can be considered as constant and limited; the system is no longer sensitive to straggler clients. Nevertheless, the macro-parameter T should be set large enough so that, on average, at each round enough clients are able to send their results to the server in time to be aggregated: it is important not to only aggregate together stale updates in a round, but to combine them with fresh ones as well. The reason is that fresh updates incorporate the current model, while stale ones are based on an older version of the learning model, and aggregating only stale updates could compromise the final model.

In addition to the upper limit on the training time, our asynchronous strategy also introduces another advantage: if the timeout is configured properly, there is no need to overcommit the number of participants. Overcommissioning is usually performed in synchronous FL to cope with either straggler and crashed clients: our framework is robust to them and does not need to waste more client resources than what is strictly needed.

The aggregation of the returned results in asyncFedAvg is the same weighted average proposed by FedAvg; here, however, clients' staleness needs to be taken into account. This is done by the aggregation scaling rule, applied before the aggregation step. The next section explains the reasons behind the choice of the scaling rule we selected for asyncFedAvg.

Chosen aggregation scaling rule

Among the aggregation scaling rules proposed in literature (described in section 2.2.1), we chose to adopt the DynSGD rule [13]:

$$w_S = \frac{1}{\tau_S + 1} \tag{3.1}$$

which scales updates based on their staleness. We did not choose the SSGD rule because, with identical weights, a straggler would not be penalised with respect to on-time clients; we excluded AdaSGD [14] and Relay [15] scaling rules because we did not want to bias the strategy evaluation. Our only focus is on the update staleness, not on its uniqueness. As a future addition, more sophisticated rules that take into account the update's similarity with the current parameters can be incorporated in our model.

3.2.3 A new player: the mediator

The mediator is the new system component introduced in our architecture. Its purpose is to ease the communication between clients and server. We suppose that it is collocated in the cloud edge, where the network delays between devices and mediators can be considered small and constant; on the other hand, we can suppose that the end-to-end bandwidth and delay between a mediator and the server are fixed and known.

Figure fig. 3.4 shows a schematic of the constituent stages of a FL training round with our framework architecture. The server S selects the mediators involved in the round, in principle all of them, except stragglers from previous rounds; then it sends the current model parameters and the training instructions to each selected mediator M_j (1). Also, S sets the round timeout T, which is sent to the mediators as well. M_j updates its internal model and selects the round participants by applying its client selection algorithm; it subsequently forwards the instructions to the chosen clients (2). Every chosen client C_i runs the training on its local data and sends back the results to M_j (3). Once T has expired, M_j collects the on-time results and aggregates them (4); the result of this first aggregation is sent back to S. The server awaits for the timeout to expire plus an additional time (named *server tolerance*) needed to take into account the mediator-server delay, after which it collects the received results and aggregates them to produce the new current model (5).

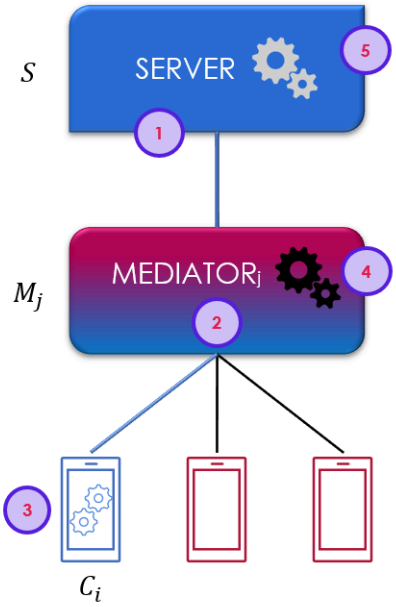

Figure 3.4. Schematic of the five stages that compose a training round in our framework architecture.

Apart from client selection and aggregation, the mediator is a transparent component as it simply forwards instructions and results between server and clients. In the proposed framework, the only "intelligent" decision a mediator takes is the client selection, but this could be delegated to the server as well, so that the mediator behaviour shifts completely under the control of the server. Two important factors are offered with the introduction of the mediators in the FL architecture:

- clients are brought nearer to the FL infrastructure, therefore communication delays can be significantly reduced. This opens new possibilities towards better exploiting clients' resources by taking advantage of their network characteristics.
- the intermediate aggregation step saves communication resources: instead of having K model parameters[2] travelling from clients to the server, only the N intermediate model parameters need to traverse the network[3]. This is a significant boost with respect to classical FL, as model parameters can have big dimensions and they waste time and network resources to be transmitted.

The idea of introducing mediators in the FL architecture is not completely new, as it was already proposed by Astraea [1]. However, in their architecture, the mediators are

[2]K is the number of total selected clients.

[3]N is the number of mediators. In the worst case, $N \leq K$, but we can usually suppose that $N \ll K$.

deputed to reschedule and balance training based on clients' data distributions to achieve a partial equilibrium in the aggregated model. Also, their mediators are virtual entities that can be destroyed and recreated at each training round by the server. Instead, we propose mediators that are physically located elsewhere with respect to the server (e.g., they can be edge servers), and they control the behaviour of the connected clients: there is no possibility of client exchange between mediators, as they may belong to different part of the network.

The request-acknowledge procedure with early-exiting

In order to directly tackle the network delay problem, we set up a request-acknowledge procedure similar to a handshake between the mediators and the clients. At the beginning of each round, the mediator randomly selects the new participants; before sending the fit instructions, it sends an empty message (the "request") and the clients immediately replies with another empty message (the "acknowledge"). On the basis of the measured RTT (the time that elapses between sending the request and receiving the ack), the mediator computes an ad hoc timeout for each client; the clients then receive the fit instructions complete with their ad hoc timeout. They apply *early-exiting*: they can decide whether to complete the total number of epochs or to finish early and send ahead of time so to compensate for their network delay and be compliant with the mediator timeout.

Figure 3.5 shows a schematic of the ack procedure. Clients reply almost immediately with the ack, as soon as they receive the request message (the light blue processing time is only symbolic). Client 1 and client 3 are compliant with the round timeout: however, since their network delay conditions are different, their processing time varies as well. Client 2, instead, decides to become a straggler and ignores the timeout.

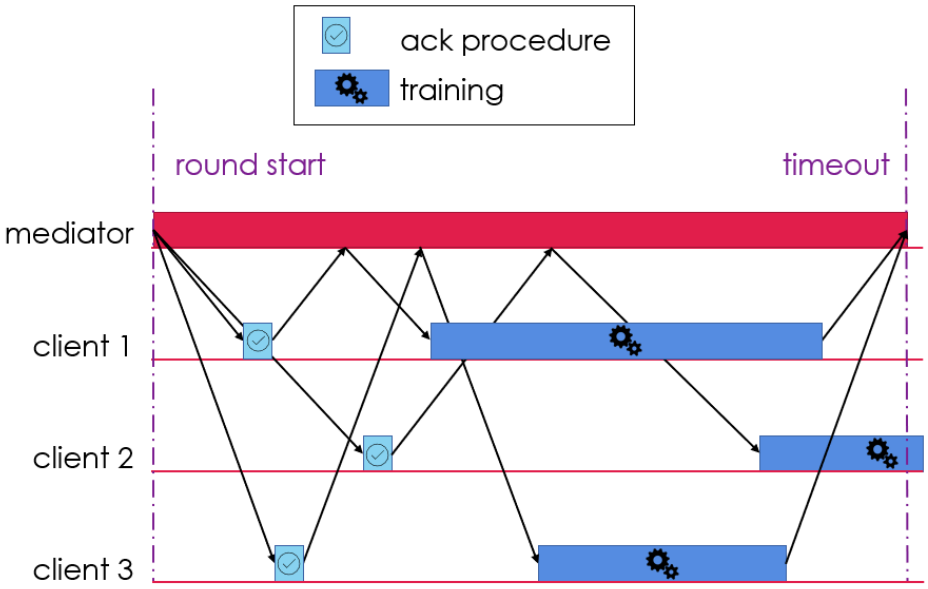

Figure 3.5. Schematic of the request-ack procedure between mediator and clients.

The ack procedure involves a macro parameter called γ which indicates a threshold on the difference between the current client model accuracy and the last epoch model accuracy. If the difference is greater than the threshold, the client considers its training worthwhile and conscientiously decides to become a straggler. In our framework, γ is a fixed value equal for all clients; however, this parameter could be controlled by the server/mediators and it could also be customised for each client.

3.2.4 Final model architecture

We have built a new system architecture composed by server, mediators and clients; we also have an asynchronous strategy that properly takes into account stragglers. Contrary to [1], we choose to make our model asynchronous in all its communicating parts. The asyncFedAvg strategy is used by server and mediators alike: the server becomes tolerant towards straggler mediators and, at the same time, the mediators become tolerant towards straggler clients. Together with the asynchronous strategy asyncFedAvg, we choose to apply the random client selection algorithm, not to bias the model by additionally requiring or measuring clients' characteristics.

In this asynchronous scenario, a FL round is governed by a timeout set by the server. The mediators are not autonomous in the choice of their round timeout: instead, they receive a customised timeout from the server. Because of our previous assumption that the mediator-server delay and bandwidth are known and fixed, the server can take into account the time needed by mediators to send their intermediate results.

To additionally counteract the network variability, we can turn on the ack procedure; based on their custom timeout, clients can decide to stop training earlier and send the

results back without completing all the epochs, being in this way compliant with the mediator round timeout.

The mediators: future directions

In our architecture, the mediator is almost transparent and does not include any intelligent feature. However, this is a versatile structure easy to be upgraded to future innovations, for example:

- a more elaborate dialogue between mediators and server can be put in place; for example, an ad hoc procedure can be established between them to estimate the link delay and bandwidth.
- the server can delegate more than one round to the mediators: in this scenario, the mediators will become responsible of the management of these rounds, taking on and partially substituting the role of the server. In this case, the mediator could also choose when to send an update on the basis of some accuracy metric to be checked at each round, to reduce even more the consumed network resources.
- the mediators could collect statistics about their pool of clients (e.g., data distribution and volume). This could be used to perform an informed client selection algorithm and the privacy risk would be reduced: clients' private information would only reach an edge server where the mediator is and not traverse an entire network to reach the central server.

Chapter 4

System components

This chapter focuses on the description of the platforms utilised to implement our framework. The three main components, Flower, Docker and OMNeT++/INET, are described in section 4.1, section 4.2 and section 4.3, respectively. Section 4.4 shows how all these pieces are eventually connected to one another to assemble the experimental platform of this work.

4.1 Flower framework

Flower [3] is an open-source platform-independent framework and was chosen in this project for its attribute of easy customisation: thanks to Flower, the whole system architecture, including the implementation of client/server behaviour, the exchanged messages and the learning strategies, were modified according to our theoretical system model described in chapter 3.

Between its many qualities, Flower is client-agnostic, meaning that the server does not care about the nature of its workers, and this allows to simulate a heterogeneous environment. Also, the aggregation strategy (the federated averaging algorithm used to aggregate the model parameters from the clients in a FL round) can be fully customised.

Flower provides the underlying structure to build a federated learning architecture (in particular, the message passing mechanism between workers, which uses gRPC and Google protocol buffers, is particularly interesting); workers behaviour (both clients and server), instead, can be defined by specifying the dataset, the learning model and other custom functionalities to be incorporated in the FL process.

Two additional tools are used on top of Flower on the client side in order to train models:

- *TensorFlow*: TensorFlow [22] is an open-source end-to-end machine learning platform; it offers multiple deployment and scaling options, along with differentiated levels of abstraction, for building and training models. It also supplies high performance and speed.

- *Keras*: Keras [23] is a deep learning API Python library which runs on top of TensorFlow. It is high-level and flexible, therefore it provides the right level of

abstraction from the complexity of deep learning needed for a simple deploy of a complex neural network.

In the original Flower implementation, the FL process is governed by the server, which is in charge of selecting and sending orders to the clients. It uses three main instruction messages:

- *get parameters*: initial message sent to a random client to obtain the current local model parameters.
- *fit instructions*: this message contains the current model parameters and the instructions to let the selected client carry out the training of the model on the local data; the updated model will then be answered back.
- *evaluate instructions*: this message contains the current model parameters and the instructions to let the selected client carry out the evaluation of the model on the local data; the results of the evaluation, i.e. loss and other accuracy metrics, will then be answered back.

Figure 4.1 shows the dynamics of the message exchange between the server S and its pool of available clients: in step 1, the server asks for the initial model parameters (answered back by a randomly chosen client in step 2). Then the FL rounds start: in step 3, the server sends the fit instructions to the participants, which send back the results in step 4. Step 5 represents the central aggregation of all the received results to build the federated model. In step 6, the server transmits the instructions for the evaluating phase, to validate the global model accuracy; the evaluation metrics are sent back in step 7. Steps 3 to 7 are repeated until the model reaches the target evaluation metric.

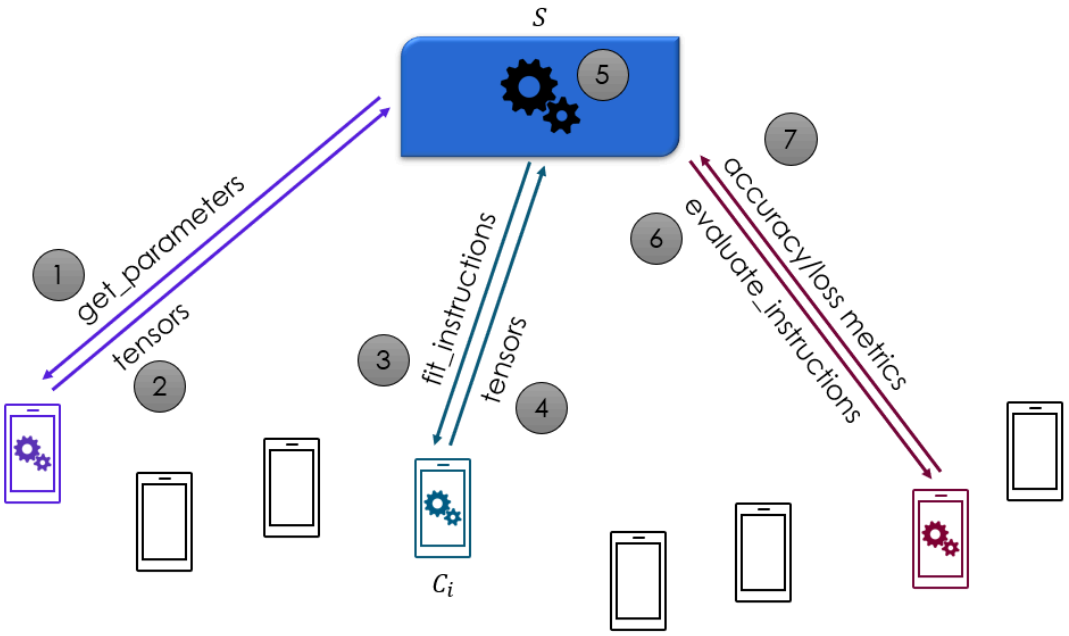

Figure 4.1. Schematic of the message exchange between the server S and its pool of clients.

4.1.1 Framework modifications to the Flower implementation

In the original Flower synchronous implementation [24] [25], the client process is essentially an infinite event loop, which listens for instructions messages from the server, executes them and sends back the proper answer, and then starts listening again for new instructions; the server, instead, sends instruction messages, then awaits until it receives results or failures from clients.

The first step of our implementation was to make the server asynchronous to obtain an asynchronous FL structure: this was done by modifying the thread generation in the server code and by adding proper data structures and functions able to handle on-time and stale results and straggler threads. This also implied the introduction of the timeout, not present in the original synchronous implementation.

The FedAvg strategy was then modified to work in this asynchronous system, and the DynSGD rule was added to the aggregation phase: in this way, using the Flower library we implemented our asyncFedAvg strategy presented in section 3.2.2.

The mediator, whose behaviour is described in section 3.2.3, was built on top of the pre-existing components: as it needed to behave in part as a client (in relationship with the server) and in part as a server (in relationship with the clients), it shares elements from both the implementations. Like the original Flower client, the mediator too is an infinite event loop: when it receives instructions from the server, it switches from *client mode* to *server mode*, and forwards the fit instructions to client and starts waiting for feedbacks from them. Once the timeout has expired, the mediator sends the aggregated results and switches back to client mode, awaiting for the next round instructions.

To implement the ack procedure, new messages are introduced in the Flower protocol: the *request* message, to which the client answers with an *acknowledge* message. To be conservative and not waste time and resources, the messages are empty messages which only contain a client ID; therefore, their size is tiny with respect to the parameters and the RTT becomes small as well.

From packet captures, we estimated the ack/request messages size to 1.0381 kbit and the parameters/instructions messages to 600 kbit each. The following code shows how the custom timeout is computed.

```
RTT = AckRxTime - RequestTxTime # [s]
# bitrate of the channel between mediator and client: estimatedBitRate = packet size / RTT,
# where packet size is the measured size of the ack procedure messages (1.0381 kbit)
estimatedBitRate = 1.0381 / RTT # [kbps]
# 600 kbit is the measured instructions size
instructionsTxTime = 600 / estimatedBitRate # [s]
# 600 kbit is the measured parameters size
parametersTxTime = 600 / estimatedBitRate # [s]

# compute custom timeout to cope with transmission delays
custom_timeout = nominal_timeout - instructionsTxTime - parametersTxTime # [s]
```

The parameter γ (the threshold to decide whether to accomplish the training or to perform early-exiting) is checked at the end of every batch. The snippet below is the code of the callback function that is called at the batch end:

```
def on_batch_end(self, batch):
    if time() >= start_time + timeout and current_accuracy - last_accuracy < gamma:
        self.model.stop_training = True
```

The client compares the current execution time: if it is greater than the timeout, it may be time to send, and a further check on γ is performed. If the current value of accuracy is greater than the previous epoch accuracy by a factor of γ, the training can be considered worthwhile and the client decides to conscientiously become a straggler and ignore the timeout.

4.2 Integration (Docker)

Docker is a platform used to create, organise and run applications [2]. The running applications are called *containers*: these isolated containers are independent from each other and also from their environment and they are built starting from a Docker *image*. The image is built from a file called Dockerfile, which invokes all the command line instructions needed to set it up. The image contains a description of the container

characteristics and single containers may be customised starting from the same image by passing different parameters to it.

Containers are more lightweight and efficient with respect to virtual machines because they share the OS kernel belonging to the underlying machine that hosts them. This allows a quick development and deployment of many instances on top of the same host.

In order to provide the federated learning workers with properly simulated networking connections, in this work we chose to exploit Docker isolation by encapsulating every participant inside a Docker container. Each worker type (client, server, mediator, see chapter 3) is built using a specific Docker image, which also includes the Flower library that was customised for this work. Once started, all the containers that host the workers are completely isolated. The networking is added in a later step and it includes only two routes: the default gateway towards the Internet (to download necessary models or datasets, used only before the true FL process starts) and one connection towards the network emulator provided by OMNeT++/INET (explained in detail in section 4.3). All the routing decisions happen inside the network emulator.

In addition to providing isolation, a Docker container also has other advantages, one of them being the possibility to manage the amount of available CPU that can be assigned to it. This means that a container can be limited in accessing the host machine's CPU clock cycles, providing us with the control on the container computing speed. This is especially useful in our framework to control the computational capacity of each worker to explore different scenarios. Another advantage of using Docker is data persistency: it is possible to exchange data between the host filesystem and the containers to preserve the results once the container is terminated. In this work we used volumes, dedicated storage spaces that can only be created and managed by Docker, to achieve data persistency.

4.3 OMNeT++/INET frameworks

The software used to simulate the network is described in section 4.3.1 and section 4.3.2. Section 4.3.3 presents the experimental setup of the network.

4.3.1 OMNeT++

OMNeT++ is the discrete event simulator chosen to emulate the network and its effect. It is a public-source C++ simulation library and framework [4], whose main characteristics are:

- *component-based architecture*: base modules are the main building blocks, providing reusability; they are programmed in C++.
- *modularity*: modules can be recombined and assembled using the NED high-level topology description language, to make the model flexible.
- *extensibility*: base modules can be extended and customised to obtain the desired behaviour.

OMNeT++ modules communicate thanks to a message-passing mechanism, which makes the framework suitable to build network simulators. In this work, on top of OMNeT++, we chose the INET framework as the standard protocol model library, described in the next section.

4.3.2 INET

INET is an open-source OMNeT++ model suite for wired, wireless and mobile networks [5]. This library allows to combine common networking devices to develop and simulate a communication network with the protocols from the Internet stack and the link layer protocols.

In order to carry out experiments in different environmental conditions, the simulated network has been designed to have configurable parameters, so that the bandwidth and delay of the channel between each worker and its end-point (either the server or a mediator) can be controlled. To achieve this, two routers are inserted between each worker: OMNeT++/INET makes it possible to design the point-to-point link that connects them, specifying datarate, propagation delay, bit error rate, packet error rate, etc.

The most important OMNeT++/INET feature used in this work is the network emulation support: this allows to perform real-time tests integrating real-world hosts (the Docker containers described in section 4.2) with a simulated network environment.

4.3.3 Network setup

In our framework, each Docker container is associated with a *veth* device that provides the connection towards OMNeT++/INET, as described later on in the next section. Inside the network emulator, there is a router for each container: these routers are necessary to connect the clients with the mediators, and the mediators with the server. The simulated links between routers can be customised with the desired bandwidth and delay values[1].

Figure 4.2 shows the network setup in the most simplified case. The clients, the mediator and the server each have their own veth pair towards the network emulator; inside it, they are connected to their router and the links between routers are customised point-to-point connections. Every router has only one Ethernet connection towards the outside; in addition to this connection, the server router has an extra link towards every mediator; the mediator routers have one connection towards the server and also one towards each client connected to them. Client routers only have two links, the Ethernet one and the point-to-point link towards their mediator.

[1]The veth pairs cannot be controlled in terms of bandwidth and delay, therefore they are not adequate to connect the containers by themselves alone - hence the need for the network emulator.

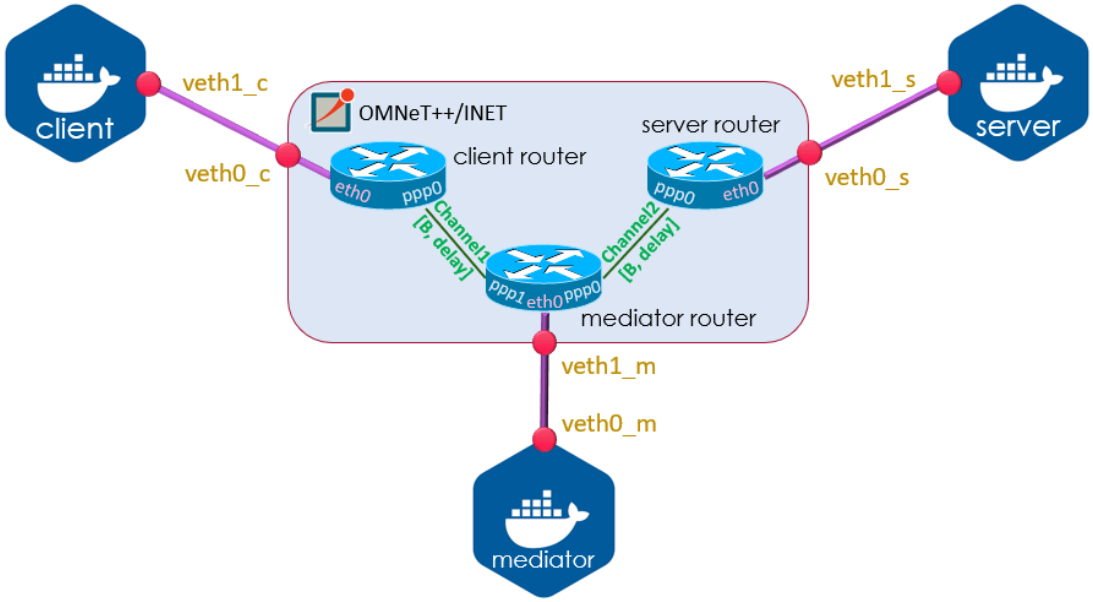

Figure 4.2. Schematic of the OMNeT++/INET network setup in the simplest case of one client, one mediator and the server, complete with the Docker containers and the veth pairs.

Veth pairs

Veth devices are virtual Ethernet Linux devices [26], created as interconnected pairs so to connect isolated namespaces or, in this case, Docker containers. In a veth pair, each end of the virtual cable has a virtual NIC, a network adapter used to increase the overall performance by relieve the operating system from some computing tasks.

From the experimental evaluation of the platform, we discovered that some parameters need to be adjusted in order to have the veth pairs properly working in our framework. In particular:

- **TCP checksum offload**: the TCP checksum computation is cumbersome and requires many CPU cycles, therefore it is often offloaded to the NICs. The TCP checksum offload happens before the transmission onto the network and upon the reception from the network for validation purposes (that is, at either end of a veth pair). However, it may happen that a virtual NIC will omit the checksum computation as an optimisation, with the assumption that it will be calculated later on by the host kernel: in the case of packets never leaving the same virtual machine, this optimisation causes a known bug due to the veth source code implementation and the packets become corrupted.

  Analysing packet captures, we experimentally verified the bug in the *ip_summed* field of the packet header. To solve this issue, the TX TCP checksum offloading needs to be disabled only on the container-side interfaces to make the OMNeT++/INET emulation work correctly.

- **MTU values**: the MTU (Maximum Transmission Unit, the maximum size of the protocol data unit) value is set to the standard value of 1500 bytes on the simulated network veth interface, while it is set to 1496 bytes on the container interface (see fig. 4.3). The 4 bytes difference between the two interfaces is needed to prevent datagram fragmentation or packet discards within the network emulation: as a matter of fact, by analysing packet captures inside and outside the simulated network we observed that 4 bytes were added by OMNeT++/INET to each packet. Nevertheless, inside the network emulator all interfaces have the MTU value set to 1500 bytes.

Figure 4.3 shows a diagram of the veth interfaces, also depicting their MTU value and the TCP checksum offloading: veth1 and veth0 represent the chosen nomenclature for the container-side interface and the network-side interface, respectively.

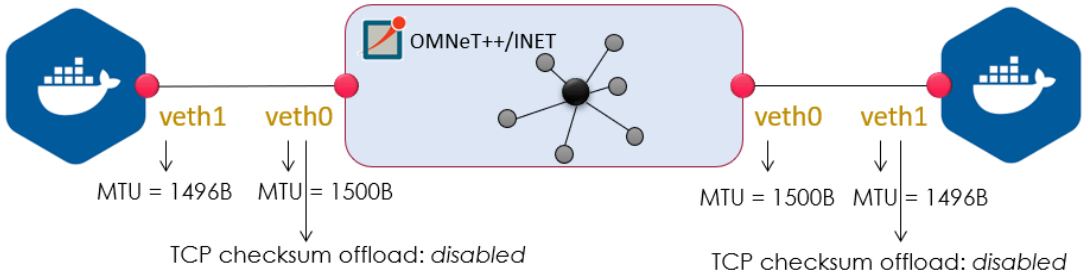

Figure 4.3. Schematic of the veth interfaces: details of the MTU values and the TCP checksum offloading.

4.4 Connecting the dots

The three key components of this work, Docker, Flower and OMNeT++/INET, all have one quality in common: since they are open-source platforms, they can be customised at will to fit the needs of those who used them. In particular, Flower offered the possibility of shaping the learning process; by combining the Docker container parameter of the CPU limit with the OMNeT++/INET configurable delays and bandwidths, we obtained a fully customisable emulation environment.

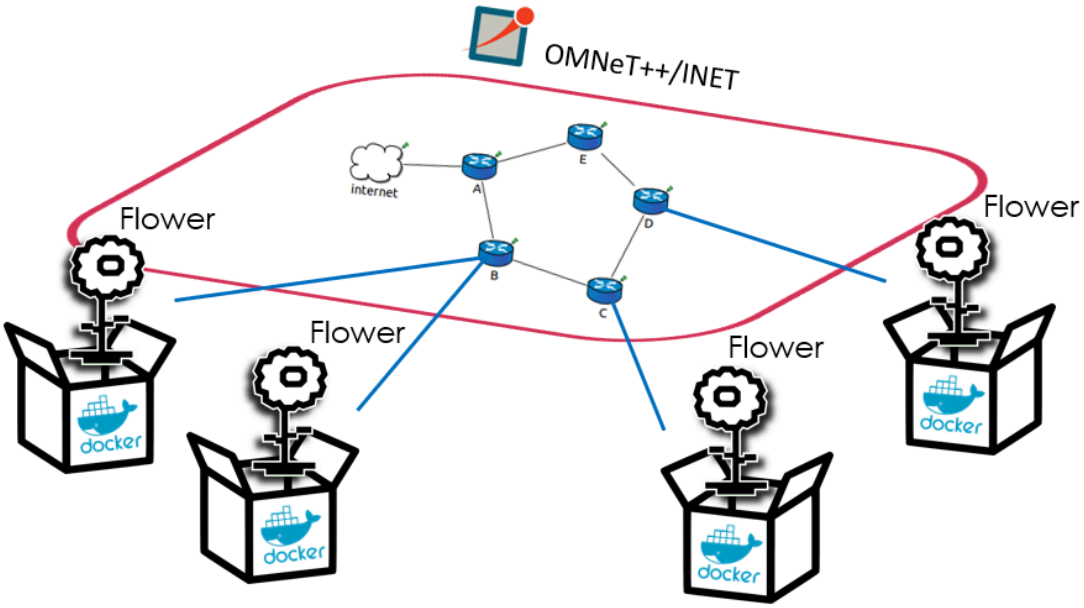

Figure 4.4. Schematic of the framework components.

To set up a simulation, we first build the workers' Docker images that encapsulate their custom behaviour, including the dataset, the neural network model and the modified version of the Flower library. Isolated Docker containers are set up from these images: the connections towards OMNeT++/INET and the internet are put in place later on by connecting veth pairs to them. Once the networking is ready, the last step is to start the OMNeT++/INET simulation, through which the workers will be able to communicate.

Below, the code to set up a simulation, taking as an example a generic worker. The worker can either be server, mediators or clients.

```
#!/bin/bash
sudo docker run -d --cpus=2 --name=worker --net=none --volume=/home/log:/log doc-worker

# create veth interfaces
# create veth pair and connect it to container
sudo ip link add veth0 type veth peer name veth1
sudo ip link set veth1 netns $(docker-pid worker)
# create veth pair for internet connectivity and connect it to container
sudo ip link add vethNs type veth peer name vethDefault
sudo ip link set vethNs netns $(docker-pid worker)

# bring up veth interfaces
sudo nsenter -t $(docker-pid worker) -n ip link set veth1 up
sudo nsenter -t $(docker-pid worker) -n ip link set vethNs up
sudo ip link set veth0 up
sudo ip link set vethDefault up

# assign IP addresses to interfaces
sudo nsenter -t $(docker-pid worker) -n ip addr add 192.168.0.1/24 dev veth1
sudo nsenter -t $(docker-pid worker) -n ip addr add 192.168.10.2/24 dev vethNs
sudo ip addr add 192.168.10.1/24 dev vethDefault

# disable TCP checksum offloading
sudo nsenter -t $(docker-pid worker) -n ethtool -K veth1 tx off

# set MTU value
sudo ifconfig veth0 mtu 1500
sudo nsenter -t $(docker-pid worker) -n ifconfig veth1 mtu 1496

# connect namespaces to the internet
sudo iptables -A FORWARD -o vethDefault -i ens3 -j ACCEPT
sudo iptables -A FORWARD -i vethDefault -o ens3 -j ACCEPT
sudo iptables -t nat -A POSTROUTING -s 192.168.10.2/24 -o ens3 -j MASQUERADE

# add default gateways (towards the internet-connected device)
sudo nsenter -t $(docker-pid worker) -n ip route add default via 192.168.10.1

# add routes (towards OMNeT++/INET)
sudo nsenter -t $(docker-pid worker) -n ip route add 192.168.1.0/24 dev veth1

# start the simulation
inet -u Cmdenv
```

In line 2, the worker container is created from the doc-worker image. The *docker run* command provides several useful options, in particular, the number of CPUs available to the container (*--cpus=2*) and the volume to be used to store files for data persistency (*--volume=/home/log:/log*). The *--net=none* option isolates the container from the outside: the connectivity will be provided later on with the veth pairs.

In lines 6-7 and 9-10, two veth pairs are created and connected to the container: they are used to connect the worker to the network emulator and to internet. The interfaces are then brought up in lines 13-16, and IP addresses (in this case, /24 private subnets) are assigned to them in lines 19-21. Lines 24, 27 and 28 set the transmission TCP checksum offloading off on the veth1 interface and the MTU values to 1500B and 1496B on veth0 and veth1 respectively, for reasons better explained in the subsection dedicated to veth pairs in section 4.3.3. The commands in lines 31-33 are used to provide the host's internet connectivity to the container, by associating it to the host interface ens3.

Finally, the internet-connected device (the vethDefault interface) is set as the container default gateway (line 36) and routes towards OMNeT++/INET to reach other workers are added by specifying their IP, as in line 39. The last step in line 42 is to start the OMNeT++/INET simulation[2].

[2]Once created, the containers are still isolated, as the networking takes some second to be set up; the FL workers are forced to be idle for a couple of seconds to stop them connecting to the network too early.

Chapter 5

Experimental evaluation

In this chapter, we measure the system emulator performances to obtain a clear picture of the simulation range in terms of delay and bandwidth. Next, we evaluate our federated learning framework, to verify the goodness of our design choices and to assess the system performances. As a benchmark for comparison, we take the original Flower synchronous FL implementation, combined with the FedAvg aggregation strategy and the random client selection algorithm.

5.1 System validation

In this section, we provide the evaluation of our federated learning emulator behaviour: in section 5.1.1, we validate the connectivity of the containers through OMNeT++/INET, while in section 5.1.2 we measure the network emulator performances.

5.1.1 Docker validation

We run a number of functional tests to verify the Docker containers setup, combined with the OMNeT++/INET framework (see section 4.3), and the obtained results are good: containers are able to successfully communicate through the network emulator. As an additional check, we verify the available CPUs with the *portainer* tool [27]: fig. 5.1 shows a single client's CPU utilisation during the training (the most demanding phase in terms of computations) when the Docker CPU limit is set to 2, 1 and 0.5 CPUs, respectively.

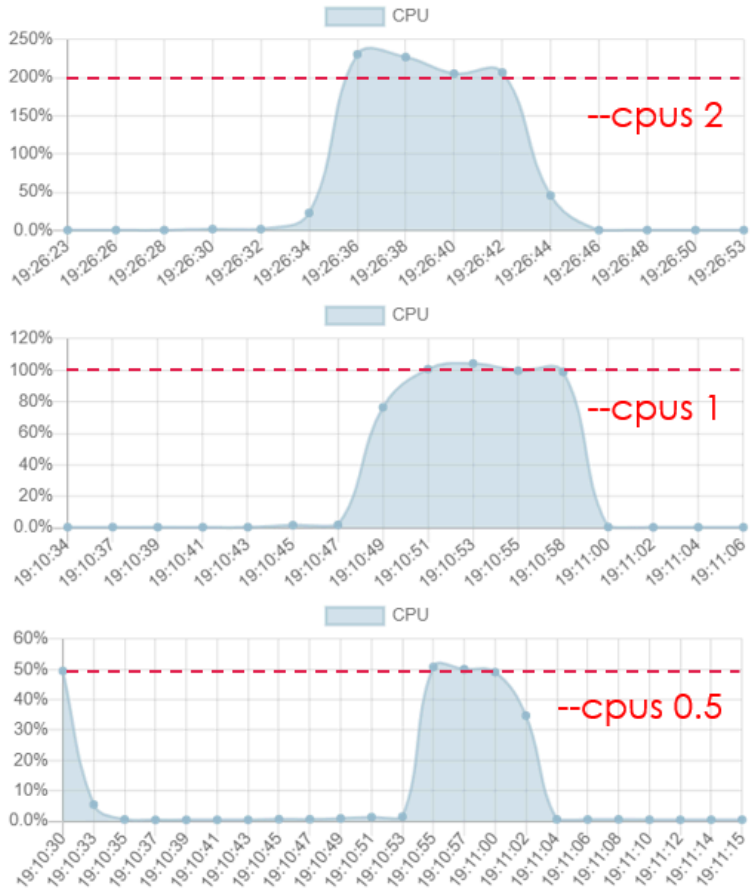

Figure 5.1. Example of container CPU usage during the client training phase with maximum CPU limit set to 2, 1 and 0.5 CPUs.

5.1.2 OMNeT++/INET validation

In this section, the performances of the network emulator are evaluated by setting up two different network configurations and performing ping tests and iperf3 tests, to measure the actual system characteristics against the theoretical ones.

In the following experiments, we connect two hosts with a plain point-to-point link. We firstly set up a single-router network, which is the simplest network topology that allows the emulation, in order to evaluate the OMNeT++/INET performances without further restrictions. Then, we build a two-routers network to be able to configure and evaluate the network bandwidth and delay of the point-to-point connection.

Two containers and single-router OMNeT++/INET network setup

The first experiment, which involves two Docker containers and a network composed only by one router, is the first step to verify connectivity (id est, correct routing and

parameters setup) and to check the system bottleneck on the bandwidth and delay. The setup is shown in fig. 5.2. Ping and iperf3 tests are run in these conditions: the results are summarised in table 5.1.

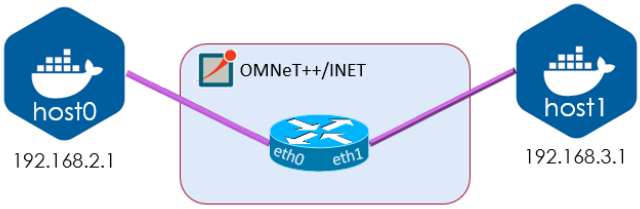

Figure 5.2. Single-router network setup.

Table 5.1. Results of ping and iperf3 between two containers (host0 and host1) with a single-router network. The bandwidth measure is the average of 10 tests run in the same conditions.

command	result
sudo nsenter -t $(docker-pid host1) -n ping 192.168.2.1 -c 10000 -i 0.0001	average RTT = (0.161 ± 0.034) ms
sudo nsenter -t $(docker-pid host0) -n iperf3 -c 192.168.3.1 -p 5001 -t 60	bandwidth = (61.2 ± 0.9) Mbps
sudo nsenter -t $(docker-pid host0) -n iperf3 -c 192.168.3.1 -p 5001 -t 60 -u -b 100M	bandwidth = (76.8 ± 1.5) Mbps

The results indicate that the maximum reachable bandwidth is around 75 Mbps. This limitation is due to the network emulator setup: OMNeT++/INET does not take advantage of hardware parallelism and only uses one CPU in this experimental environment. As a proof of this, with the *htop* tool provided by Linux [28] it is possible to verify that the utilisation of a single CPU (the one dedicated to the network emulator process) goes up to 100% during the iperf3 test under stress. Also, the average delay introduced by the OS is less than 1 ms, therefore it is possible to make this delay negligible by properly configuring higher delays inside OMNeT++/INET.

Two containers and two-routers OMNeT++/INET network setup

The second experiment involves two Docker containers and a network made up by two routers with a customisable point-to-point link between them, to verify the network emulator bottleneck on the bandwidth and delay (setup shown in fig. 5.3).

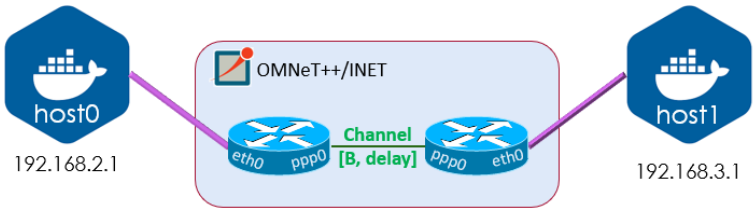

Figure 5.3. Two-routers network setup: details of the channel bandwidth B and delay.

Figure 5.4 shows the bandwidth measured with iperf3 with respect to the theoretical bandwidth set in OMNeT++/INET: the blue line in the graph represents the bisector. The emulator comfortably sustains bandwidths up to around 40 Mbps, as the points are very near to the blue line; however, when setting higher values, the measured bandwidth saturates at around 47 Mbps because the CPU used by the simulator goes up to 100% of usage.

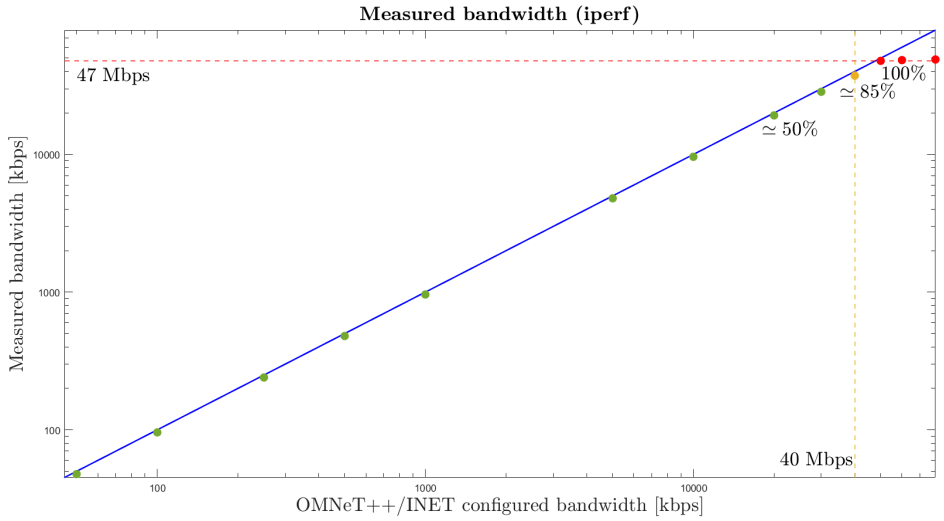

Figure 5.4. Measured vs. configured OMNeT++/INET link bandwidth between two routers. Details of the CPU usage when the emulator saturates; the blue line is the graph bisector.

Figure 5.5 shows the measured RTT with ping with respect to the theoretical delay value set in OMNeT++/INET. The red line is the bisector: since all measured values are very close to the bisector, we can conclude that the measured RTT is very accurate; also, considering that the ping test is not a very computationally demanding task, the simulator CPU does not saturate for any tested value.

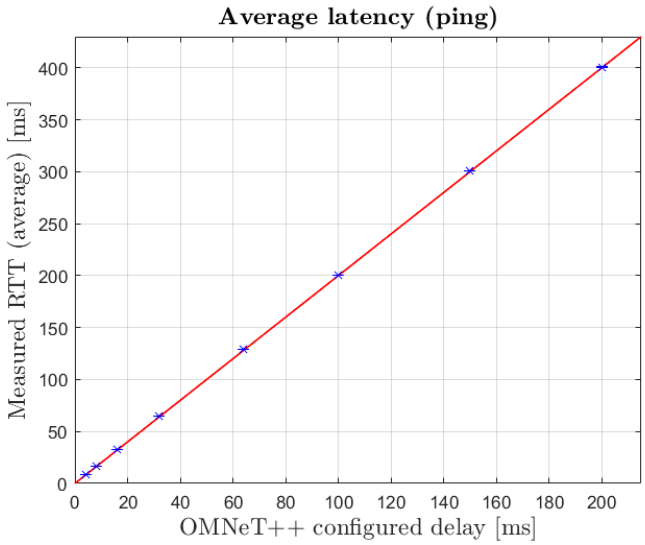

Figure 5.5. OMNeT++/INET configured delay vs. measured average RTT: the red line indicates the bisector, that is, the theoretical values if the simulator did not introduce additional delays.

From the last two experiments, we deduce that, while the configured delay does not affect the simulation, setting a high value of link bandwidth can saturate the CPU. Therefore, the next experiment is to evaluate the average delay introduced by the OMNeT++/INET for different bandwidths (setting a fixed delay of 500 ms). The results of the experiment are shown in fig. 5.6 and fig. 5.7; in particular, the second graph shows the RTT coefficient of variation, which indicates how much variance is present within the average delay measure. The network emulator does not introduce any significant delay for bandwidth higher than 50 kbps; however, the delay becomes quite large if the bandwidth decreases under that threshold. In conclusion, any bandwidth values between 50 kbps and 40 Mbps are acceptable in our network simulator.

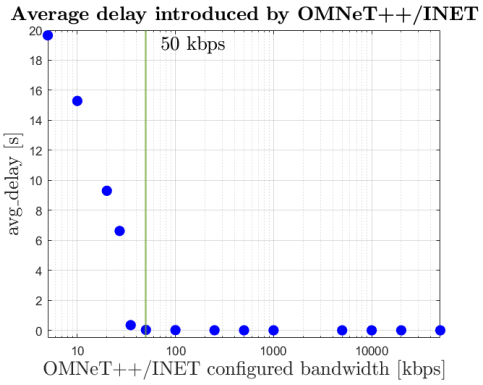

Figure 5.6. Average delay introduced by OMNeT++/INET, computed as the measured RTT subtracted by the nominal RTT.

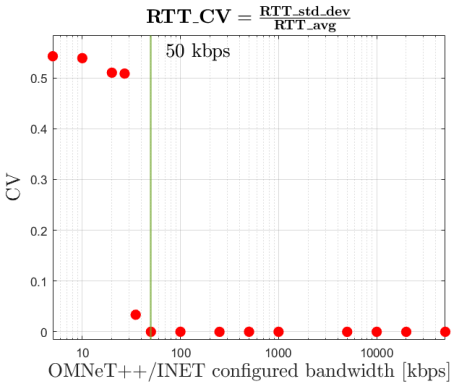

Figure 5.7. Coefficient of variation (CV) of the average delay introduced by OMNeT++/INET in fig. 5.6.

Figure 5.8. Mean and CV of the delay introduced by OMNeT++/INET.

5.2 Experimental evaluation

This section presents the main experimental results we obtained. Our comparison benchmark is synchronous FL with random selection and FedAvg aggregation strategy; the analysed network configurations include symmetric networks, with no delayed clients, and asymmetric networks, with clients which are penalised in terms of link bandwidth. We study both IID and non-IID scenarios.

To explore the different aspects of our framework, we perform three main simulations:

- section 5.2.1 shows a comparison of our asynchronous architecture with respect to the classical synchronous one under different network conditions;
- section 5.2.2 analyses the effect of the position of the mediator in the edge-cloud continuum on the FL rounds;
- section 5.2.3 evaluates the performances of the ack procedure in a heterogeneous scenario.

Dataset and neural network model

In our experiments, we use the MNIST [29] dataset: it is made of 60000 training images and 10000 validation images that represent each a digit between 0 and 9 (10 classes). The images are gray-scale, composed by 28×28 pixels. We also employ a sequential neural network model with 4 layers, characterised as follows:

- layer 1: first layer to flatten the input, input dimensions equal to the images dimensions (28×28)
- layer 2: dense NN layer, ReLu activation function, output space dimension 128
- layer 3: dense NN layer, ReLu activation function, output space dimension 256
- layer 4: dense NN layer, SoftMax activation function, output space dimension 10 (equal to the number of classes of the MNIST dataset)

5.2.1 Synchronous vs. asynchronous FL architectures

In the first experiment, we compare our asynchronous double-layer framework with the classical synchronous FL implementation in two different network settings: one with no delays and symmetrical client conditions, the other one with two delayed clients (in terms of bandwidth). We also verify the behaviour in case of IID and non-IID data distributions among clients, with the same network setups. In this first simulation, the ack procedure is turned off to analyse only the asynchronous structure.

Experimental setup 1: IID data distribution

The experiment involves 6 clients, with equal computational capabilities between them. The whole MNIST dataset is provided to all clients. The server and clients setups for the synchronous FL simulation are shown in table 5.2, while the asynchronous framework setup for server, mediators and clients is shown in table 5.3.

Table 5.2. Server and clients settings for the synchronous FL simulation with the MNIST dataset.

Server settings		**Clients settings**	
n.° clients	6	dataset	MNIST
rounds	20	epochs	3
CPUs	1	CPUs	0.5
n.° selected clients per round	4		

Table 5.3. Server, mediators and clients settings for the asynchronous FL simulation with the MNIST dataset. The total number of clients is 6 (three clients for each mediator, two mediators in total).

Server settings		**Mediators settings**		**Clients settings**	
n.° mediators	2	n.° clients	3	dataset	MNIST
rounds	20	n.° selected clients per round	2	epochs	3
CPUs	1	CPUs	1	CPUs	0.5
timeout [s]	90				

In the first stage of the experiment, the network links are configured to be symmetric for all clients; also, to be fair with respect to the double-layer framework, an additional 20 Mbps link is added between clients and server in the synchronous setup. Figure 5.9 and fig. 5.10 show the network configurations in the synchronous and asynchronous cases, respectively.

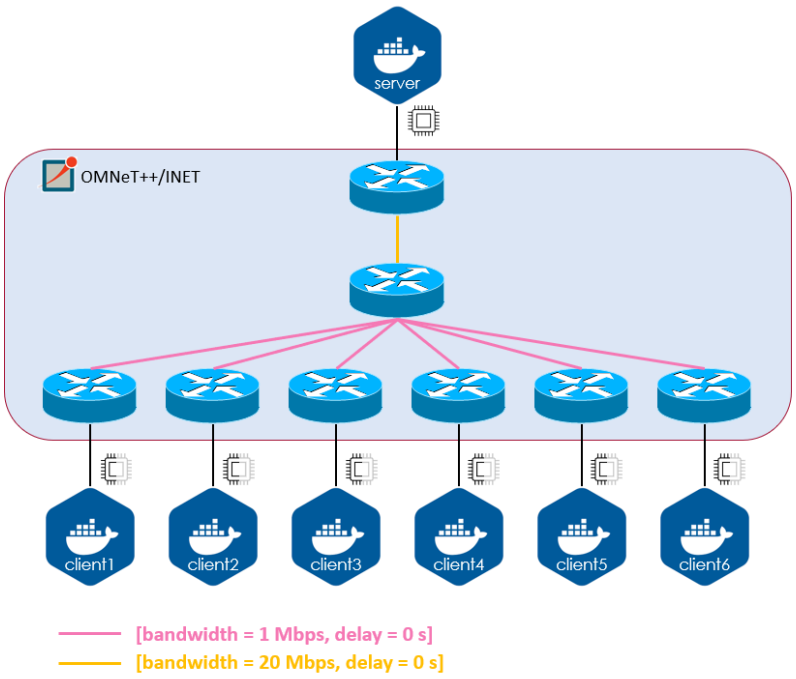

Figure 5.9. Network and computational conditions of the 6-client synchronous simulation setup: case of symmetric network.

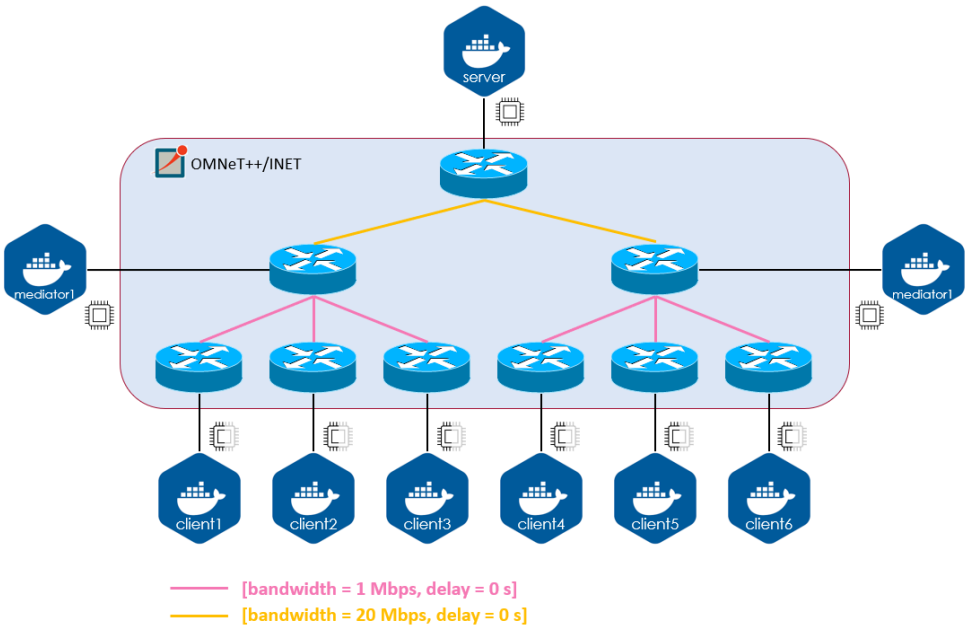

Figure 5.10. Network and computational conditions of the 6-client asynchronous simulation setup: case of symmetric network.

In the second setup, two clients become slower with respect to the others since their configured link bandwidth is reduced by a factor of 20 (50 kbps with respect to 1 Mbps). Figure 5.11 and fig. 5.12 show the network configurations in the synchronous and asynchronous cases, respectively: in the asynchronous framework, the two slow clients are divided between the two mediators.

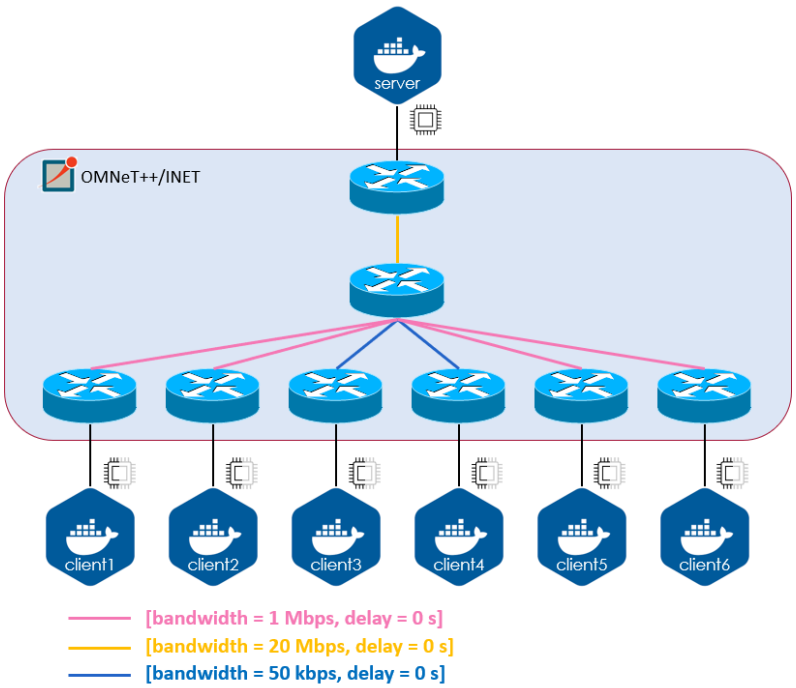

Figure 5.11. Network and computational conditions of the 6-client synchronous simulation setup: case of asymmetric network. Two clients have links with a reduced bandwidth, 20 times smaller than the other clients.

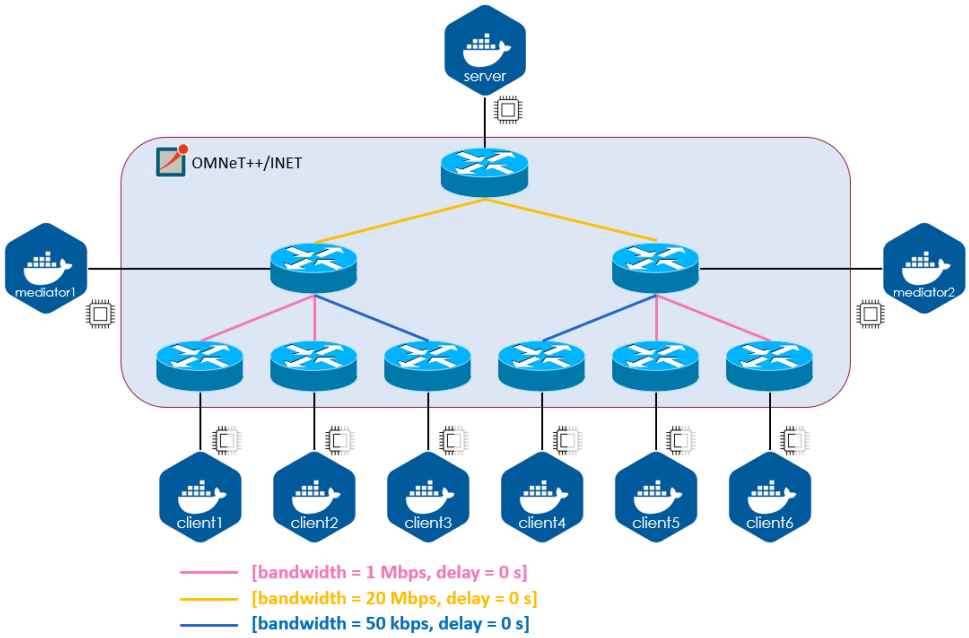

Figure 5.12. Network and computational conditions of the 6-client asynchronous simulation setup: case of asymmetric network. Two clients have links with a reduced bandwidth, 20 times smaller than the other clients.

Results

Figure 5.13 shows the accuracy and loss of the simulation run in the symmetric network conditions: these metrics are centrally computed by the server after each aggregation phase, and the whole MNIST dataset is used for the evaluation. The two frameworks have comparable execution times, and the time-to-accuracy metric of the asynchronous double-layer architecture is slightly superior than the synchronous FL since higher accuracies are reached in less time. Figure 5.14, however, shows the same metrics in the asymmetric network case with two slow clients: here, it is possible to see that our framework is able to react better to the presence of stragglers. Not only it outperforms the synchronous FL in terms of execution time, but is also reaches a higher value of accuracy earlier.

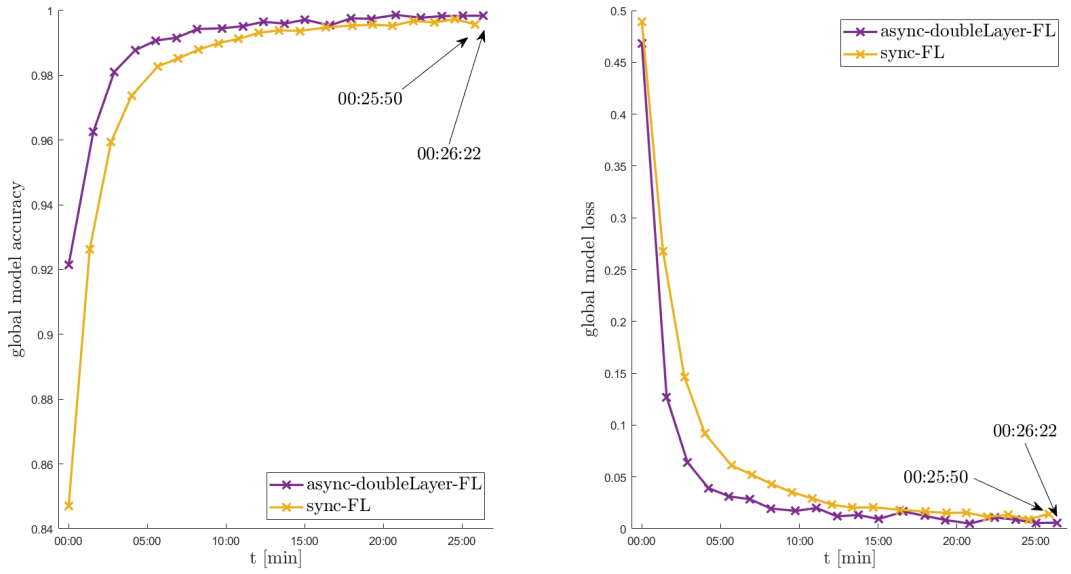

Figure 5.13. Accuracy and loss for the 6-client setup with symmetric network conditions and IID data distribution.

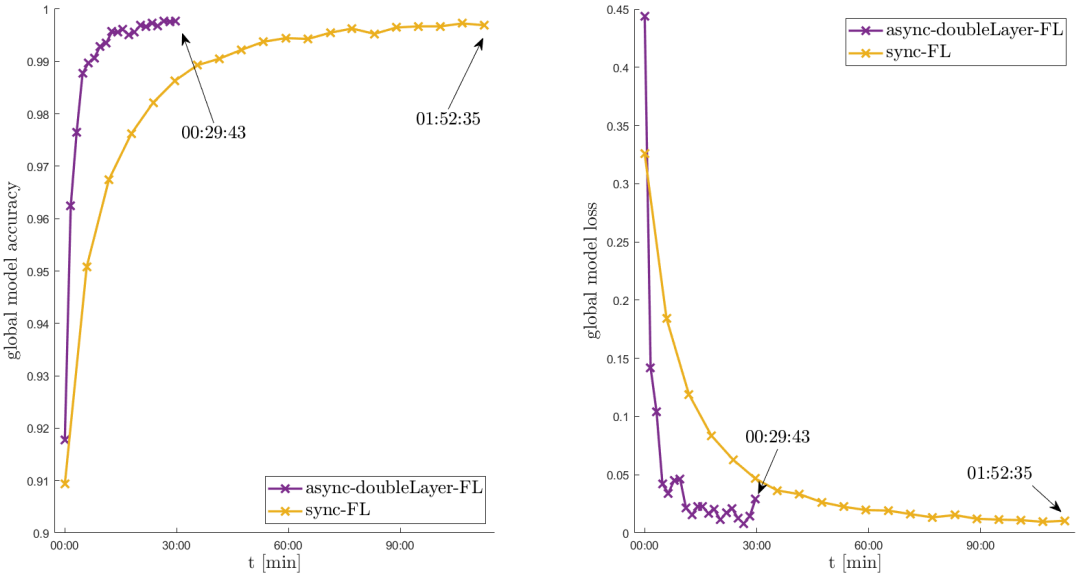

Figure 5.14. Accuracy and loss for the 6-client setup with asymmetric network conditions and IID data distribution.

Experimental setup 2: non-IID data distribution

In the second experiment, we change the distribution of the dataset among clients to obtain a non-IID scenario. The MNIST dataset is divided into 6 parts with equal size and divided among the 6 clients; in this way, classes "0", "2", "4", "5", "7", "9" are uniquely assigned to clients 1, 2, 3, 4, 5, 6 respectively, while the remaining classes are shared among two clients. The symmetric/asymmetric network setup and computational capabilities are left unchanged with respect to the last experiment (refer to fig. 5.11, fig. 5.12). The workers settings are shown in table 5.4 and table 5.5: since the dataset size is smaller and the computational capability is the same, the client training time becomes smaller as well and, to adjust to this, in the asynchronous case we set the server timeout to 30 s instead of 90 s.

Table 5.4. Server and clients settings for the synchronous FL simulation with the MNIST dataset and non-IID data distribution.

Server settings		**Clients settings**	
n.° clients	6	dataset	1/6 MNIST
rounds	20	epochs	3
CPUs	1	CPUs	0.5
n.° selected clients per round	4		

Table 5.5. Server, mediators and clients settings for the asynchronous FL simulation with the MNIST dataset and non-IID data distribution. The total number of clients is 6 (three clients for each mediator, two mediators in total).

Server settings		**Mediators settings**		**Clients settings**	
n.° mediators	2	n.° clients	3	dataset	1/6 MNIST
rounds	20	n.° selected clients per round	2	epochs	3
CPUs	1	CPUs	1	CPUs	0.5
timeout [s]	30				

Results

In the non-IID scenario, accuracy and loss are evaluated by the server on the whole MNIST dataset: fig. 5.15 and fig. 5.16 show the accuracy and loss metrics of the simulation run in symmetric and asymmetric network conditions, respectively. In these simulations, the overall execution time is smaller than in the IID data distribution case: this is caused by the smaller size of the dataset, which reduces the clients' computational time at each round. Due to the non-IID data distribution, the accuracy curve grows slower than the IID case for the same number of epochs; however, the comments

of the previous experiment hold in this situation too, and the asynchronous double-layer framework performances exceed the synchronous FL ones.

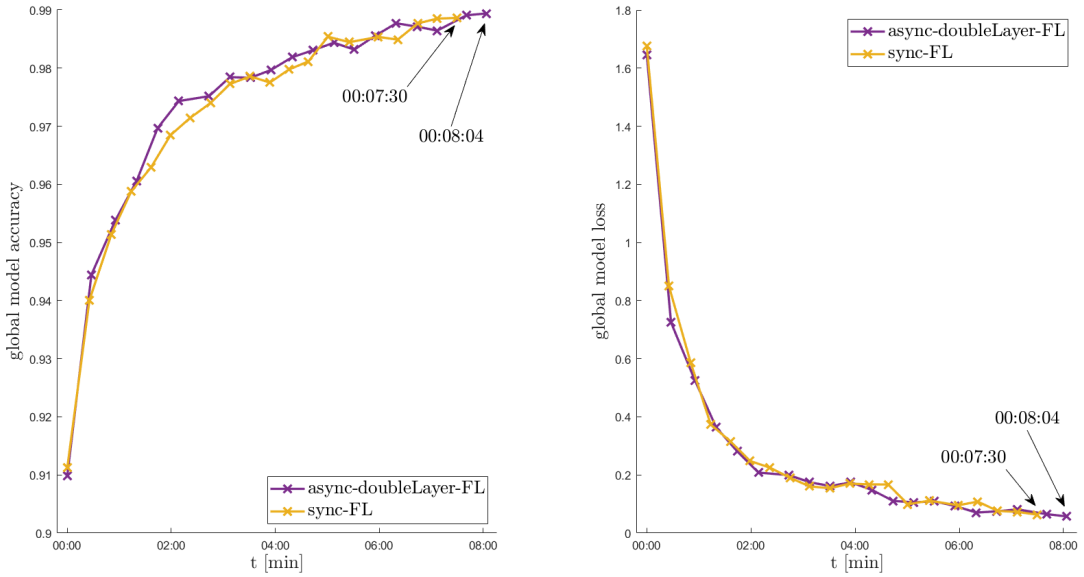

Figure 5.15. Accuracy and loss for the 6-client setup with symmetric network conditions and non-IID data distribution.

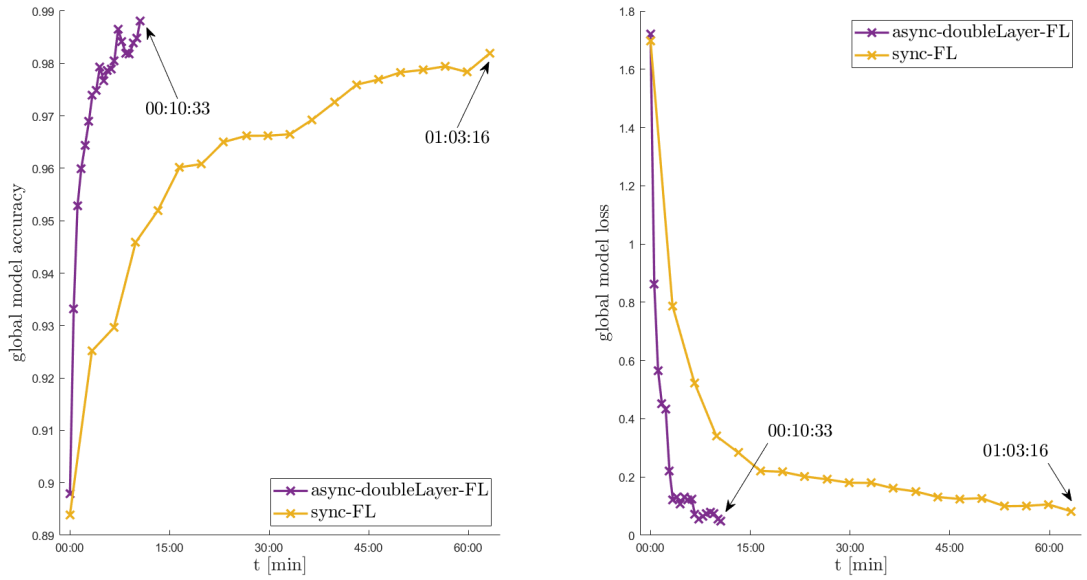

Figure 5.16. Accuracy and loss for the 6-client setup with asymmetric network conditions and non-IID data distribution.

Discussion on network delays

Considering the emulator validation performed in section 5.1, 40 Mbps is the maximum link bandwidth limit which cannot be exceeded, not to saturate the CPU dedicated to the emulator; in the clients links, we never exceed the reasonable threshold of 1 Mbps. From packet captures, we evaluated the time needed to transmit the model parameters from client to mediator; with a 1 Mbps link, the transmission time is around 40 s for the MNIST dataset (the data size depends on the dataset and NN model).

Realistic network delays for a fiber link are below 200 ms; the RTT delay due to a satellite connection with a geostationary satellite is around 600 ms and this is one of the slowest physical connections nowadays. Even considering this higher value, in our simulation a delay below 1 s would not have a major impact on the transmission time of 40 s. Inserting a delay greater than 1 s would therefore be unrealistic: we then choose to vary the link bandwidth instead, since this has a far greater effect on the transmission time.

5.2.2 Effect of the position of the mediator in the network

In the next experiment, we analyse how the position of the mediator in the edge-cloud continuum influences the federated learning process.

Experimental setup: symmetric network, non-IID data distribution

In this test, the workers settings are the same as the last experiment in the non-IID case (refer to table 5.4, table 5.5). Figure 5.17 shows the network setup: the mediator-server link bandwidths are varied at every run of the experiment (while they are also kept equal to each other). Since we are interested in the effect of the mediator position with respect to the server, we set the client-mediator links to the same value of 1 Mbps.

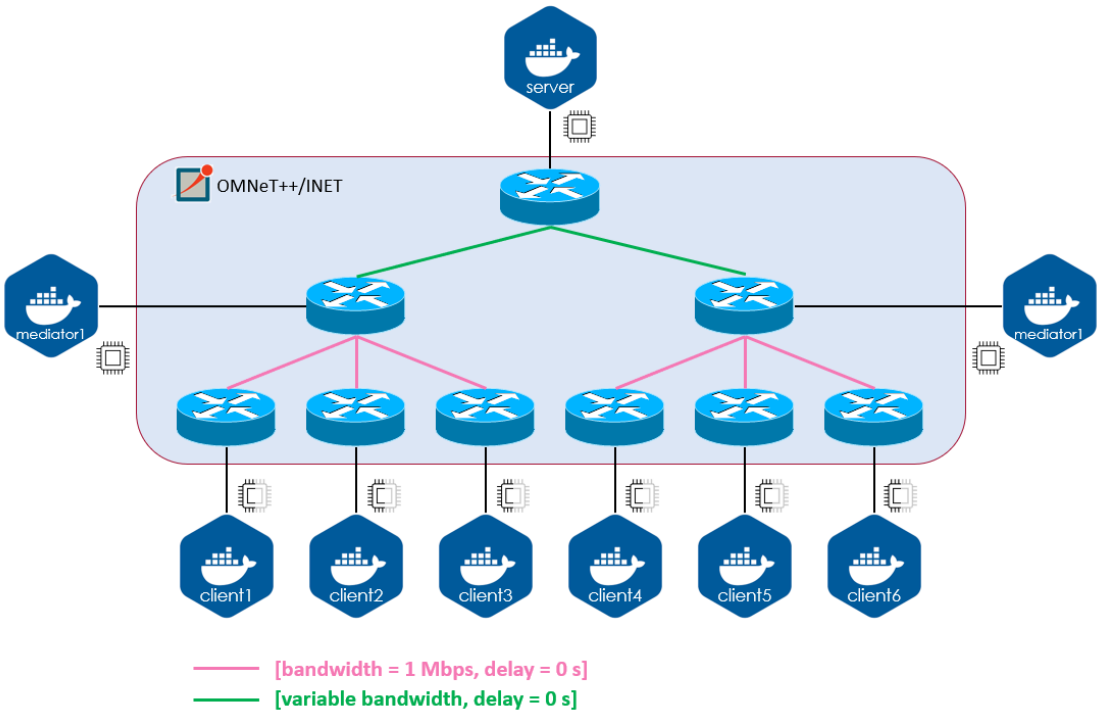

Figure 5.17. Network and computational conditions of a 6-client asynchronous non-IID simulation setup: symmetric network, adjustable link-server link bandwidth.

Results

Figure 5.18 shows the accuracy and loss of the simulations run with the previously described settings, with 6 different mediator-server link bandwidth values (ranging from 200 kbps to 40 Mbps). The result is unexpected: when the mediator-server link is high with respect to the client-mediator links, the performances in terms of time-to-convergence are very similar; however, the farther the mediator is from the server (when the mediator-server link bandwidth becomes comparable with the client-mediator links), the longer the model needs to converge. Also, the final accuracy of the low-bandwidth models is lower than the other cases.

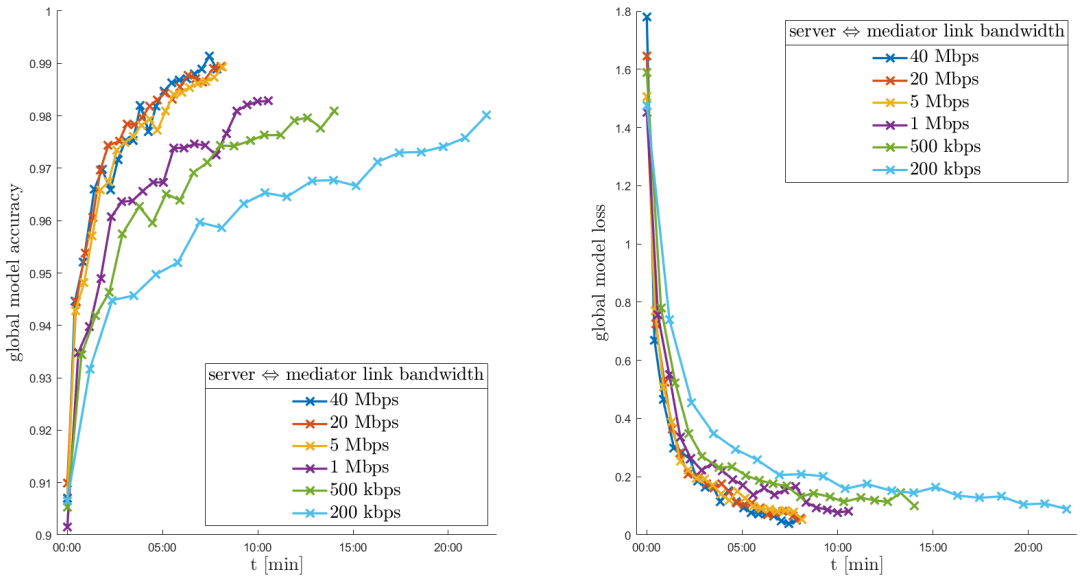

Figure 5.18. Accuracy and loss for the 10-client setup with symmetric network conditions and non-IID data distribution.

5.2.3 Ack procedure and complete heterogeneous scenario

The last experiment involves the ack procedure described in section 3.2.3. Two different tests are run: the first one with asymmetric client network conditions, the second one with a complete scenario with both asymmetric network conditions and diverse client computational capabilities.

Experimental setup 1: simulation with network delays

In the first stage of the experiment, we focus on the ack procedure effectiveness with respect to the synchronous FL: in particular, we show the effect of the γ macro-parameter on our framework. We run simulations with two different γ values (0.01 and 0.005) on 10 clients: table 5.6 and table 5.7 show the synchronous and asynchronous simulation settings; clients have either 1 Mbps or 200 kbps of link bandwidth towards the first router and 0.5 CPUs each (the setup is similar to the one described for previous experiments).

Table 5.6. Server and clients settings for the complete scenario synchronous FL simulation with network delays.

Server settings		**Clients settings**	
n.° clients	10	dataset	MNIST
rounds	20	epochs	5
CPUs	1	CPUs	0.5
n.° selected clients per round	4		

Table 5.7. Server, mediators and clients settings for the complete scenario asynchronous FL simulation with network delays. The total number of clients is 10 (five clients for each mediator, two mediators in total).

Server settings		**Mediators settings**		**Clients settings**	
n.° mediators	2	n.° clients	5	dataset	MNIST
rounds	20	n.° selected clients per round	2	epochs	5
CPUs	1	CPUs	1	CPUs	0.5
timeout [s]	90				

Results

Figure 5.19 shows the accuracy and loss metrics for the asymmetric network conditions setup: our framework was tested with two different γ. The setup where $\gamma = 0.01$ converges to higher accuracy values in less time with respect to synchronous FL, while it has similar performances in terms of loss; the peaks in the loss graph are due to the asynchronous nature of the framework, which causes more or less updates to be aggregated within the same round. Instead, the simulation where $\gamma = 0.005$ has similar accuracy behaviour to the synchronous case, but far worse performances for what regards the loss: not only the loss curve decreases slowly, but also the peaks due to the asymmetric rounds are more noticeable than the case with $\gamma = 0.01$.

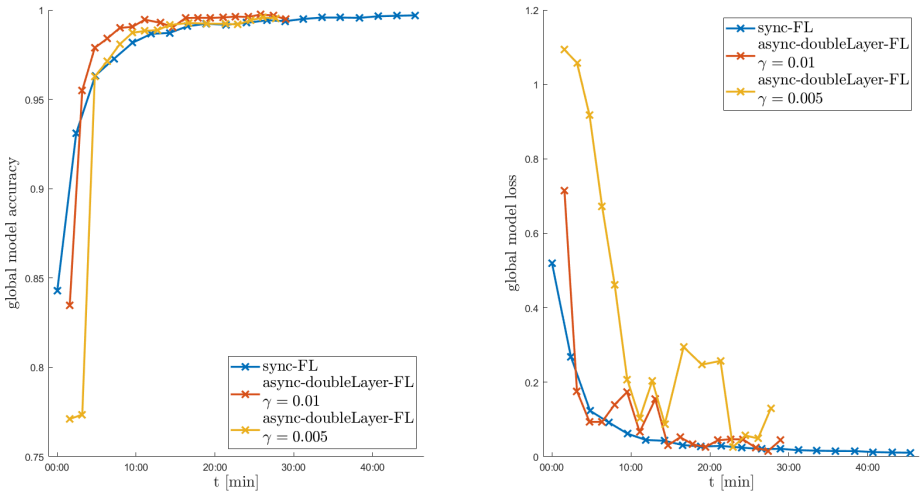

Figure 5.19. Accuracy and loss for the 10-client setup with symmetric network conditions and non-IID data distribution.

Experimental setup 2: complete simulation with network delays and asymmetrical computational capabilities

To evaluate our framework in a more realistic environment, we set up a complete scenario with heterogeneity both in network conditions and in client computational capabilities. We run training on 10 clients with either 1 Mbps or 200 kbps link bandwidth and with a number of CPUs between 0.1 and 2, combining the two variables to obtain the most diverse scenario. Also, considering the results of the previous test, for the ack procedure we choose $\gamma = 0.01$ in this experiment. Table 5.8 and table 5.9 show the synchronous and asynchronous settings; table 5.10 provides the chosen network and computational variability conditions, valid for both frameworks.

Table 5.8. Server and clients settings for the complete scenario synchronous FL simulation with network delays and uneven clients computational capabilities.

Server settings		**Clients settings**	
n.° clients	10	dataset	MNIST
rounds	20	epochs	5
CPUs	1	CPUs	between 0.1 and 2
n.° selected clients per round	4		

Table 5.9. Server, mediators and clients settings for the complete scenario asynchronous FL simulation with network delays and uneven clients computational capabilities. The total number of clients is 10 (five clients for each mediator, two mediators in total).

Server settings		**Mediators settings**		**Clients settings**	
n.° mediators	2	n.° clients	5	dataset	MNIST
rounds	20	n.° selected clients per round	2	epochs	5
CPUs	1	CPUs	1	CPUs	between 0.1 and 2
timeout [s]	120				

Table 5.10. Network and computational conditions of the 10 simulated clients in the complete scenario: these settings provide the environmental heterogeneity. The setup is valid both for the synchronous and the asynchronous simulations; in our double-layer framework, clients 1 to 5 are assigned to mediator 1 and clients 6 to 10 are assigned to mediator 2.

Client n.°	**Link bandwidth**	**CPUs**
1	1 Mbps	2
2	1 Mbps	0.5
3	200 kbps	0.5
4	1 Mbps	1
5	200 kbps	0.1
6	200 kbps	0.1
7	200 kbps	2
8	200 kbps	1
9	1 Mbps	0.5
10	1 Mbps	0.1

Results

In fig. 5.20, the usual metrics of accuracy and loss are shown for the complete simulation scenario. Our asynchronous framework needs less than 40 minutes to complete the 20-epochs training (synchronous FL requires 130 minutes); it also converges to a higher accuracy with respect to synchronous FL. Peaks in the loss function are due to the asynchronous nature of the framework, as explained before. In synthesis, we can conclude that our framework performs substantially better than synchronous FL in terms of time-to-accuracy metric, provided that the macro-parameter γ is properly tuned.

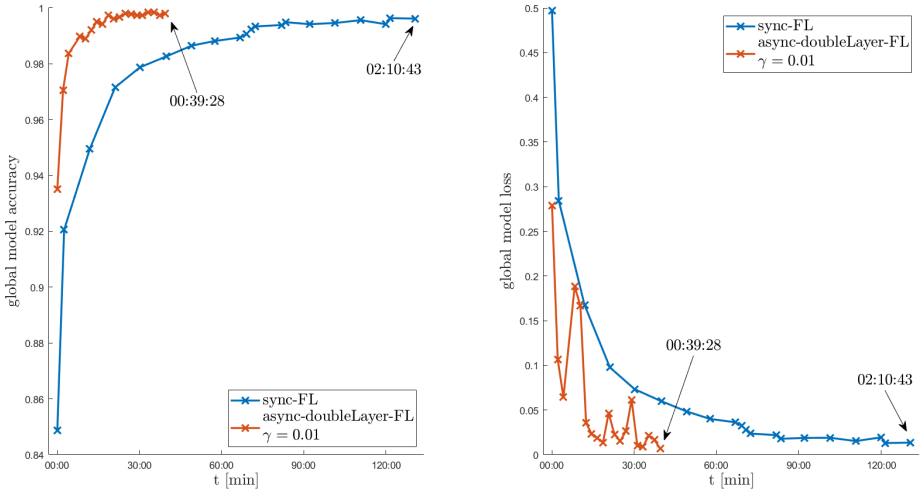

Figure 5.20. Accuracy and loss for the 10-client setup with symmetric network conditions and non-IID data distribution.

Chapter 6

Conclusion

Today need for efficiently performing machine learning tasks on multiple devices in a heterogeneous scenario motivates the foundation of this thesis.

Chapter 1 introduced the distributed machine learning problem and the target scenario, the IoT world, to provide a context for the real-world applications of this work.

In chapter 2, we reviewed the main solutions that have been proposed in literature to move towards distributed machine learning, focusing in greater detail on federated learning. We also describe the main FL papers that have been our starting point.

The design choices made in chapter 3 and implementation solutions devised in chapter 4 can be summarised in three main point: an asynchronous aggregation strategy to counteract the presence of straggler clients; a double-layer structure with an additional intermediate aggregator (the mediator) to provide an extra layer of intelligence to the process and to reduce the amount of data crossing the network; the ack procedure between clients and mediators with early-exiting to directly tackle the network delay impact on a round.

In chapter 4, the Flower [3] library was modified to adapt the classic client-server structure to our design and to obtain the mediator component. The customisable network conditions were achieved through the OMNeT++/INET [4] [5] network emulator and the control on the clients computational capabilities through Docker [2] containers.

We experimentally verified the quality of the proposed architecture step by step in chapter 5: we firstly tested the asynchronous double-layer structure against the classical synchronous FL and we discovered that the two frameworks have similar performances in case of identical network conditions and computational capabilities; when network delays come int play, however, our framework behaves better to stragglers than the asynchronous one. Next, we verified the influence of the position of the mediator on the performances of the framework. When the server-mediator link bandwidth is comparable with the client-mediator bandwidth, the time-to-accuracy metric converges slowly, while when it is higher the framework is more performant. Eventually, we tested the ack procedure in a heterogeneous scenario adding both network and computational capability variability: provided that the γ value is properly tuned, our complete framework outperforms the synchronous one.

6.1 Future directions

Although the proposed framework modifies the FL structure by adding mediators between clients and server, it is very easily adapted to existing techniques; with this design, the framework is able to accommodate other aggregation strategies or client selection algorithms; also, more sophisticated methods based on affinity and divergence from the current model can be applied locally by the mediators or centrally by the server, to discriminate between updates.

Thanks to the huge range of simulation tools offered by OMNeT++, the simulated network could be improved or substituted by a more sophisticated one (e.g., a 4G network could be simulated with SimuLTE, which runs on top of the OMNeT++/INET framework); mobility can be added to the clients to add mobility to the environment.

Considering the modular nature of our system components, the testing of this framework could be moved into a real environment by replacing the simulated network with the broad internet and by substituting the Docker containers with IoT devices. Since it is written in Python, the Flower platform also offers the possibility to develop code for real-world devices; even if for simulation purposes we used a single machine to host all the workers, the code could be adapted to be run on the Android operating system. This operation would substitute the simulated network with a real cellular network.

Bibliography

[1] Moming Duan, Duo Liu, Xianzhang Chen, Yujuan Tan, Jinting Ren, Lei Qiao, and Liang Liang. Astraea: Self-balancing federated learning for improving classification accuracy of mobile deep learning applications. In *2019 IEEE 37th international conference on computer design (ICCD)*, pages 246–254. IEEE, 2019.

[2] Docker overview. `https://docs.docker.com/get-started/overview/`. Accessed: 2023-01-09.

[3] Flower documentation. `https://flower.dev/docs/`. Accessed: 2023-01-20.

[4] What is omnet++? `https://omnetpp.org/intro/`. Accessed: 2023-01-09.

[5] What is inet framework? `https://inet.omnetpp.org/Introduction.html`. Accessed: 2023-01-09.

[6] Ibm machine learning definition and main concepts. `https://www.ibm.com/topics/machine-learning`. Accessed: 2023-01-21.

[7] Sawsan Abdulrahman, Hanine Tout, Hakima Ould-Slimane, Azzam Mourad, Chamseddine Talhi, and Mohsen Guizani. A survey on federated learning: The journey from centralized to distributed on-site learning and beyond. *IEEE Internet of Things Journal*, 8(7):5476–5497, 2021.

[8] Yoshitomo Matsubara and Marco Levorato. Split computing for complex object detectors: Challenges and preliminary results. *arXiv preprint arXiv:2007.13312*, 2020.

[9] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*, 55(5):1–30, 2022.

[10] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.

[11] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, and H Vincent Poor. Federated learning for internet of things: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 23(3):1622–1658, 2021.

[12] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[13] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 463–478, 2017.

[14] Georgios Damaskinos, Rachid Guerraoui, Anne-Marie Kermarrec, Vlad Nitu, Rhicheek Patra, and Francois Taiani. Fleet: Online federated learning via staleness awareness and performance prediction. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 13(5):1–30, 2022.

[15] Ahmed M Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A Fahmy. Resource-efficient federated learning. *arXiv preprint arXiv:2111.01108*, 2021.

[16] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. Ibm federated learning: an enterprise framework white paper v0. 1. *arXiv preprint arXiv:2007.10987*, 2020.

[17] Facebook federated learning simulator (flsim). `https://github.com/facebookresearch/FLSim/blob/main/README.md`. Accessed: 2023-02-12.

[18] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, pages 19–35, 2021.

[19] Wentai Wu, Ligang He, Weiwei Lin, Rui Mao, Carsten Maple, and Stephen Jarvis. Safa: A semi-asynchronous protocol for fast federated learning with low overhead. *IEEE Transactions on Computers*, 70(5):655–668, 2020.

[20] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2019.

[21] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[22] Tensorflow documentation. `https://www.tensorflow.org/about`. Accessed: 2023-01-21.

[23] Keras documentation. `https://keras.io/about/`. Accessed: 2023-01-21.

[24] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[25] Flower github repository. `https://github.com/adap/flower`. Accessed: 2023-03-20.

[26] veth(4) — linux manual page. `https://man7.org/linux/man-pages/man4/veth.4.html`. Accessed: 2023-01-09.

[27] Portainer documentation. `https://www.portainer.io/`. Accessed: 2023-02-18.

[28] htop(1) — linux manual page. `https://man7.org/linux/man-pages/man1/htop.1.html`. Accessed: 2023-02-10.

[29] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.