UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA



Understanding Synchronous vs. Asynchronous Federated Learning in FADO

Pedro Alexandre Batista Bastos

Mestrado em Segurança Informática

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2025

To my family and friends for their support throughout these years.

Agradecimentos

Deixo o meu agradecimento a todos aqueles que me apoiaram nesta longa jornada e que de alguma forma contribuiram para a realização da mesma. Agradeço ao meu orientador, Prof. Nuno, por ter disponbilizado esta investigação e me ter acompanhado ao longo da mesma. Agradeço à unidade de investigação LASIGE, pela disponbilização dos recursos necessários para o desenvolvimento da investigação. Agradeço aos meus amigos e colegas de curso pelos momentos que partilharam comigo durante este percurso. Por fim, agradeço aos meus pais, pelo apoio e disponibilidade que mostraram ao longo dos últimos anos para que eu pudesse concluir a minha licenciatura e mestrado.

Resumo

Nos últimos anos, a aprendizagem automática tem surgido como uma das áreas relevantes de investigação e desenvolvimento principais para o futuro, pela sua capacidade de gerar impacto em vários setores da sociedade, desde o campo da saúde à área financeira, da mobilidade ao entretenimento digital. O crescimento exponencial da quantidade de dados produzidos diariamente, em conjunto com a necessidade de processar esta informação de forma rápida, eficiente e segura, tem feito com que estas áreas de estudo recebam cada vez mais investimento e foco. Entre estas, Federated Learning (FL) [24] surge como uma metodologia descentralizada que permite treinar modelos através da colaboração de múltiplos clientes, sem que os dados individuais de cada cliente tenham de ser transferidos para um servidor central. Este paradigma é bastante relevante em contextos onde a privacidade é uma das principais prioridades, como acontece em aplicações de saúde ou serviços financeiros. Uma vez que os dados são utilizados localmente, apenas os resultados dos treinos são enviados para o servidor central de forma a atualizar o modelo global.

Apesar de apresentar vantagens claras, FL enfrenta vários desafios. Entre estes, destaca-se a heterogeneidade dos dados, que são frequentemente distribuídos de forma não independente e idêntica (non-IID), a heterogeneidade dos dispositivos, que apresentam diferentes capacidades de processamento e de ligação à rede, causando atrasos na participação dos dispositivos nas rondas de treino. Os primeiros desenvolvimentos de FL centraram-se em metodologias síncronas, onde a agregação só ocorre quando o servidor recebe os resultados de todos os clientes selecionados. Embora esta estratégia seja consistente, sofre de um problema crítico, os stragglers, clientes com um comportamento mais lento, seja este de carácter computacional ou ao nível da rede, que acabam por atrasar ou causar a invalidação de rondas inteiras de treino desperdiçando os recursos de todos os clientes da ronda. Estudos recentes têm, por isso, explorado variantes assíncronas ou híbridas, onde as atualizações podem ser enviadas a ritmos diferentes e agregadas de forma contínua e individual ou em grupo, por exemplo definidos por um buffer. Estas abordagens permitem uma maior flexibilidade e tornam o processo mais robusto e resiliente face a dispositivos lentos computacionalmente, a redes instáveis, ou até mesmo a atrasos intencionais.

É neste contexto que surge esta dissertação, cujo objetivo principal consistiu em reestruturar e expandir a framework, Federated Learning Attack and Defense Orchestrator (FADO) [29], de modo a que esta suportasse algoritmos síncronos, assíncronos e híbridos, permitindo a emulação de cenários com condições realistas de heterogeneidade de dados e dispositivos. Com estes objetivos em mente, o trabalho desenvolvido incluiu a reestruturação da framework FADO, bem como a

implementação de diferentes algoritmos de FL. Com o intuito de mostrar a relevância da assincronidade no âmbito de FL avaliámos os algoritmos implementados em várias experiências onde se variou a heterogeneidade da distribuição dos dados e se incluiram stragglers nas rondas de treino.

A reestruturação do FADO passou a adotar uma arquitetura diferente, composta por componentes independentes. Este novo design inclui um orquestrador, que após ingerir o ficheiro de configuração, onde constam que algoritmos devem ser usados em cada um dos módulos, coordena todo o processo do inicio ao fim. Cada cliente é lançado como um processo individual, permitindo atingir o realismo e a escalabilidade pretendida. A componente de rede foi também um dos focos, passando a incluir a funcionalidade de emulação de atrasos de comunicação através das medições de latência em tempo real, utilizando a HORTA como servidor central e vários servidores em localizações geográficas distintas. Por fim, a simulação de clientes straggler com a possibilidade de configurar a sua probabilidade e o seu impacto. Este conjunto de funcionalidades dão ao FADO a capacidade de produzir condições próximas das reais, o que é um requisito fundamental para conseguir avaliar de forma relevante a resiliência dos algoritmos.

Os resultados demonstram que os algoritmos assíncronos são alternativas viáveis aos métodos síncronos, já que estes conseguem alcançar níveis de desempenho comparáveis à baseline síncrona e, em certos casos, revelam maior robustez em contextos de heterogeneidade de dispositivos. Em particular, o FedBuff [27] superou os restantes algoritmos no cenário IID e manteve uma performance semelhante nos cenários de stragglers. Os algoritmos síncronos mantiveram-se competitivos, com a adaptação do baseline síncrono para o FedAvg+30%, que mitigou de forma relevante o impacto dos stragglers, com o trade-off de recursos adicionais.

Como contributo, esta dissertação disponibiliza uma versão reestruturada da framework FADO, agora capaz de suportar algoritmos síncronos, assíncronos e híbridos, possibilitando a emulação realista de ligações cliente para servidor em diferentes condições de rede. O FADO passou a incluir múltiplas opções de configuração, como latência nas comunicações, comportamentos de stragglers com diferentes níveis de impacto, tornando-se assim uma ferramenta relevante para investigações futuras de FL. Por fim, a avaliação realizada permitiu fazer a análise e comparação dos trade offs entre as abordagens síncronas e assíncronas, mostrando o potencial das metodologias assíncronas para a resiliência do processo de FL.

Palavras-chave: Aprendizagem Automática, Assíncrona, Privacidade, Framework, FADO

Abstract

Federated Learning (FL), a decentralized machine learning paradigm, has received increasing attention in recent years. While early FL methodologies focused on synchronous communication, recent research has shown that hybrid and asynchronous algorithms can mitigate key limitations such as straggler effects and synchronization delays. This thesis explores asynchronous and hybrid FL algorithms by evaluating their performance in realistic scenarios and extending the Federated Learning Attack and Defense Orchestrator (FADO) to support these communication methodologies.

The FADO framework, originally designed for synchronous FL, was restructured to emulate client-server interactions under asynchronous conditions. This modification expanded the framework capabilities to support diverse scenarios, aggregation algorithms, attack vectors, and defense mechanisms. Key implementation efforts included a new client-server emulation, capable of multi-system deployment, a modular orchestration workflow for selecting aggregation, participation, and communication strategies, and the integration of real-time network latency based on measured round-trip delays between server and client locations.

Evaluation focused on comparing algorithm performance baselines and exploring how various FL algorithms behave under conditions of data and device heterogeneity. For data heterogeneity, experiments varied the class distribution of training data among clients with the CIFAR-10 dataset to simulate non-IID conditions. For device heterogeneity, measured network latency and straggler behavior was introduced across clients. Results showed that asynchronous algorithms achieved faster convergence and more efficient resource utilization in device heterogeneity scenarios, though they introduced challenges such as delayed updates. These results demonstrate the practical relevance of the framework and the capabilities of its modularity structure when creating different, realistic and variable conditions.

This thesis contributes a restructured version of FADO, a modular, extensible framework for evaluating any type of FL algorithms, along with insights into their performance metrics and trade-offs. This framework can serve as a basis for future research into new types of FL and their defenses.

Keywords: Federated Learning, Asynchronous FL, Privacy, FL Framework, FADO

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Goals . . . 3
1.3 Contributions . . . 3
1.4 Document Structure . . . 4

2 Context and Related Work **5**

2.1 Machine Learning . . . 5
2.1.1 Supervised Learning vs Unsupervised Learning . . . 6
2.1.2 Centralized Learning vs Decentralized Learning . . . 7
2.2 Synchronous vs Asynchronous . . . 8
2.3 Deep Learning . . . 8
2.3.1 Stochastic Gradient Descent . . . 9
2.3.2 Neural Networks . . . 9
2.3.3 Convolution Neural Networks . . . 9
2.4 Federated Learning . . . 10
2.4.1 Challenges and Problems . . . 13
2.4.2 Secure Aggregation . . . 15
2.4.3 Differential Privacy . . . 16
2.5 Federated Optimization Algorithms . . . 17
2.5.1 Federated Averaging (FedAvg) . . . 18
2.5.2 Asynchronous Federated Optimization (FedAsync) . . . 19
2.5.3 Federated Learning with Buffered Asynchronous Aggregation (FedBuff) 21

3 FADO Requirements and Architecture **23**

3.1 System Requirements . . . 23
3.2 FADO Architecture . . . 24
3.2.1 Overview . . . 24

3.2.2 Builder . . . 25
3.2.3 Runner . . . 26

4 FADO Implementation **39**
4.1 FADO Orchestrator Workflow . . . 39
4.2 Technologies and Tools Used . . . 41
4.3 Client and Server Emulation . . . 42
4.3.1 Client Overview . . . 42
4.3.2 Server Overview . . . 44
4.3.3 Client Management and Protocols . . . 53
4.4 Data Handling and Distribution . . . 57
4.5 Challenges during Implementation . . . 59

5 Evaluation **61**
5.1 Dataset and Model . . . 61
5.2 Dataset Distribution Experiments . . . 62
5.2.1 Compared Algorithms . . . 62
5.2.2 Evaluation Metrics . . . 62
5.3 Dataset Distribution non-IID (α = 0.1) . . . 63
5.4 Dataset Distribution Average (α = 0.9) . . . 65
5.5 Dataset Distribution IID (α = 10) . . . 67
5.6 Device Heterogeneity with Network Delays and Stragglers . . . 68
5.6.1 Setup and Context . . . 68
5.6.2 Different percentages of stragglers . . . 70
5.6.3 Different straggler delays . . . 71

6 Conclusion **73**
6.1 Future Work . . . 73

Bibliography **75**

List of Figures

2.1 Overview of a Machine Learning process. 6
2.2 On the blue box, we have a classification example of data. On the red box we have an association of data based on the location of data feature. 7
2.3 Deep Learning structure overview. 10
2.4 This is an overview of a Federated Learning algorithm. 11
2.5 Dataset IID vs Dataset non-IID with examples from MNIST. 14
2.6 This is an overview of Secure Aggregation protocol. 15

3.1 FADO framework main components overview. 24
3.2 FADO class interactions. 28

4.1 FADO Orchestrator Workflow. 40
4.2 Clients connecting to the server diagram. 54
4.3 Exchanged messages between a client and the server during the client life cycle. . 55

5.1 Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a non-IID setting ($\alpha = 0.1$). 63
5.2 Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 0.1$. 64
5.3 Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a avg-IID setting ($\alpha = 0.9$). 65
5.4 Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 0.9$. 66
5.5 Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a IID setting ($\alpha = 10$). 67
5.6 Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 10$. 68
5.7 Comparison of different straggler probabilities. 70
5.8 Comparison of different delays. 71

List of Tables

2.1 Federated Learning Algorithms. 18

4.1 Comparison of Federated Learning Algorithms in FADO 53
4.2 Latency Statistics for each server location 57

5.1 Data Dirichlet distribution statistics for the different values of α 62
5.2 Average communication delays and standard deviations for each client location . 69

Chapter 1

Introduction

1.1 Motivation

Over the last decade, we have witnessed an unprecedented growth of the digital space, with multiple new and innovative developments impacting almost all areas of society. One of these examples is the engagement on social media platforms boosted by the fact that most people have access to internet and smartphones. Smartphones also make the access to digital payment methods easier, as well as a safer option. The recent pandemic sped up the digital integration in some of the processes used in companies, creating a huge migration to remote working, making people rely even more on these new tools. These new technologies are now considered crucial for things like communication, such as video conferencing, shopping, ordering food, or healthcare apps managed by the government. Accompanied by a fast growth of digital usage is the associated amounts of data it generates, which creates new challenges in how we and the companies that hold our data manage that information.

The need to be able to process and interpret this data at a speed that matches its growth requires the usage of more advanced methods, requiring the assistance of digital tools and software. This is where Machine Learning (ML) comes up as an essential and important technology. ML facilitates this by enabling systems to learn from the existing data and improving their performance over time, without requiring humans to explicitly program it. The scope of ML is vast and is used on multiple areas. In healthcare, for example, ML models have proven to be effective in diagnosing based on medical imagery by analyzing them and outputting useful diagnosis comparable to those of professionals in the field. One specific examples of this is a study conducted in the field of dermatology [11]. Another example, expands to the financial institutions where ML models are trained and used based on financial fraud patterns in order to prevent other fraud attempts when using digital payments, as seen in [9]. In addition to these examples, other positively impacted areas are the development of autonomous self-driving vehicles [2], improving the recommendation systems in streaming platforms, such as Netflix [16], and innovating on the recent application of natural language processing technologies [22], such as chat bots.

Despite the advancements that ML provides, the massive amount of data that companies, governments and institutions collect and analyze create significant challenges, specifically in terms of

privacy regulations and security. This supports the argument that, concerns of what data is stored, where it is stored and how it is stored are relevant to discuss. It is required that these entities manage data responsibly by following the data protection regulations, such as the European Union's General Data Protection Regulation (GDPR) [12]. The GDPR serves as a guideline on how to handle individuals privacy rights. So, unless new approaches appear, there has to be a trade off because ML models require and improve significantly when more data is available. Furthermore, there are challenges on how all this data is handled and if companies and bodies of government act accordingly to the EU GDPR [12]. Liu2021 [23] underlines this by discussing some of the major privacy challenges and risks associated with the usage of ML by defining attack and threat models, such as attack by inference of client contributions. So that data breaches such as [32] are mitigated in the future. Other potential problems include database and data center breaches, such as the one reported in [32], where the mismanagement of sensitive information is a critical need for secure practices.

One of the new approaches to address this concern is Federated Learning [24], a decentralized method for training ML models. FL enables multiple entities or users to collaborate without requiring the sharing of their data with a central server. This is the big underlying motivation behind FL, the ability to leverage diverse and private data sources in order to improve the models of features everyone uses. The basic idea is, the server maintains the learning models and never receives any data transmissions during or before the training process. This characteristic is extremely crucial in certain situations and areas of research. Additionally, FL offers several other motivating factors, such as facilitating compliance with the EU GDPR regulation, the reduced cost on resources used, since in ML there is a step where all the data has to be transferred to the server for processing, in FL this step doesn't occur.

FL is a relatively new field of research, having emerged only in the past decade. As this technology continues to be researched and developed, new challenges and opportunities are constantly being created. Each year brings with it the development of new algorithms, and training methods aimed at improving the performance and scalability of FL systems. New possibilities for FL appear, such as allowing participants to train models asynchronously [38]. Clients can send their contributions at their own pace, meaning that they contribute in later rounds instead of discarding their update. The server never stops the FL process waiting for a client reply, it receives the contributions in parallel and manages the aggregations based on how late the updates are. These innovations not only improve the robustness of FL systems but also create new work to be researched and tested to the limit. This ongoing evolution presents another motivating factor for the framework we developed and present in this thesis, as FL evolves into an important tool when in need of collaborative options with privacy requirements in mind. A big example is Google Mobile Keyboard Prediction [18] that provides various functionalities such as, auto-correction, word completion, and next-word prediction.

1.2 Goals

Given the recency of FL, this work expects to provide a valuable framework that helps researchers evaluate their FL algorithms in terms of performance metrics and behavior when faced with a variety of customizations and environment scenarios, such as real network delays based on server locations, different intentionally induced network delays, the addition of a controlled percentage of straggler clients, as well as, extensive FL training settings. To achieve this we expect to have a framework design and implementation that easily allows FL algorithms to be implemented into the framework. These algorithms can be fully new ones or sub components of an existing one. For example, implementations of client selection algorithms, aggregation algorithms, or training algorithms. Thus, allowing the experiment of these sub components and algorithms in much more detail.

A future goal would be to deploy the framework in a multi-system in order to allow the deployment of clients across multiple machines. Currently the focus and work done was on a one machine setup, where client processes are launched and communicate with the server independently, leveraging python capabilities the experiments run up to 1000 clients, allowing us to reach scalability levels of real-world scenarios while using minimal resources.

In order to understand the details of the algorithms this work provides an extensive and very detailed number of customization settings, so that every specification of the training process in the clients, to the aggregation settings of the server are controlled by the user, as well as, the communication settings and workflow process. This allows the goal of studying and experimenting with different scenarios in regards to network behavior and load.

1.3 Contributions

The key contributions of this work are as follows:

1. A comprehensive study of both Synchronous and Asynchronous algorithms currently used in FL.

2. The restructuring of the FADO framework to support a variety of training methodologies and algorithmic options.

3. The re-implementation of the FederatedAveraging algorithm [24] and a variant FedAvg+30% [3].

4. The implementation of two Asynchronous approaches, FedAsync [38] and FedBuff [27].

5. The introduction of extensive customization settings for experimental configurations, covering client training, server behavior, and network communication parameters, including the ability to emulate delay scenarios.

6. Complete tracking and recording of the entire training process, with a variety of performance metrics, as well as the final trained model parameters.

7. An evaluation of these algorithms under different training and environment settings based on the tracked performance metrics.

1.4 Document Structure

The structure of the document is as follows:

- Chapter 2 - Provide the necessary context on Machine Learning and Deep Learning in order to introduce the Federated Learning concept and the related work associated with FL and the developed work.
- Chapter 3 - Presents the design and architecture decisions for the implementation of the framework and its components.
- Chapter 4 - Gives a detailed overview on the implementation of the framework and crucial components.
- Chapter 5 - Evaluates the implemented training processes and algorithms in the framework based on data and device heterogeneity, including straggler emulation, real time network latency between server locations, and intentionally induced delays.
- Chapter 6 - Draws final conclusions on the developed work and future directions.

Chapter 2

Context and Related Work

In this chapter we will provide some context on key concepts necessary to understand the presented work. We will present key concepts on Machine Learning as well as some of the concepts applied during the implementation of the framework. Even though some of these topics are not the main focus of the work, they provide the necessary background knowledge of the current state of FL and some of its possibilities. We will also present some of the studied literature and associated algorithms by explaining them and comparing the current literature.

2.1 Machine Learning

Flach2012 [13] states:

> *Machine learning is the systematic study of algorithms and systems that improve their knowledge or performance with experience.*

Figure 2.1 shows how the various components of a ML process interact, the component learning problem interacts with the component task by using learning algorithms that models. By comparison, the component task uses the model to infer from new data.

- **Learning Problem** - We call this component learning problem because its the part of the ML process where different learning algorithms are tested or chosen to train a model. Because it is relevant for the learning problem at hand that the chosen learning algorithm is compatible with the problem it is trying to solve. This is accomplished by feeding training data to these algorithms, generating in the end a trained model, capable of performing the necessary task.

- **Task** - The task part of the process comes after the Learning Problem is solved and uses the output of that component, the trained model to solve the expected task. This is done by feeding the trained model new data, because the model now has the capability of classifying or predicting the outcome based on the selected features.

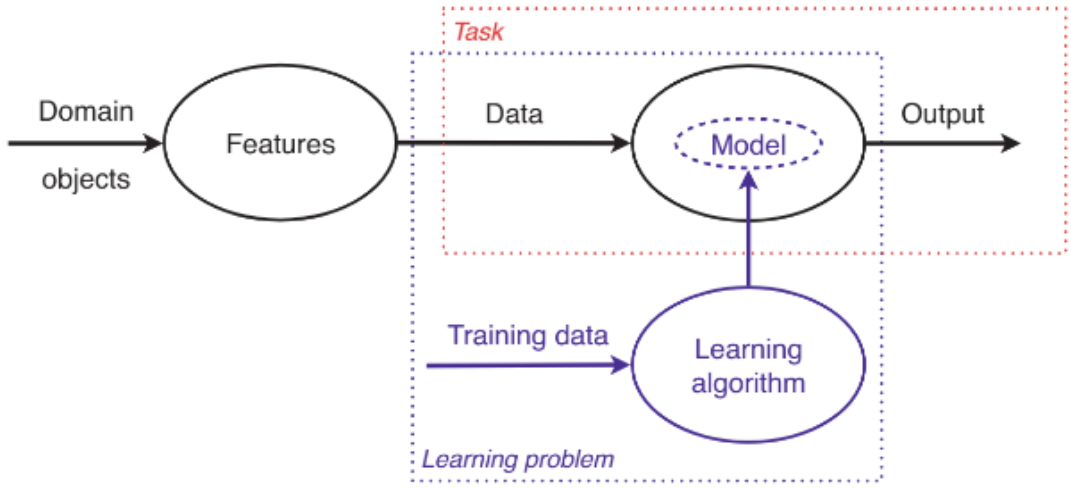


Figure 2.1: Overview of a Machine Learning process.

2.1.1 Supervised Learning vs Unsupervised Learning

Supervised Learning is a type of Machine Learning where the learning problem requires the data to be associated with a label so it can learn how to classify new data based on certain selected features. So the task of classifying or predicting the outputs can be done accurately. For this to be accomplished the dataset must be rich and include values for all features. Useful examples of this include Image Classification and Spam Email Detection.

We consider the Spam Email Detection a classification problem, because based on the known features the model will decide if it is considered spam or not. On the other hand, we also have regression problems, when the values are continuous, this includes labeling a price for a house based on certain features.

Unsupervised Learning on the other hand is where the learning problem does not require data to be associated with a label. The learning problem identifies hidden patterns or structures from the data, this is accomplished by analyzing and clustering the data points based on their features. The task is to cluster or associate the new data points based on the underlying discovered patterns or structures. Useful examples of this include Anomaly Detection, where the model receives data from normal usage, so it can accurately flag misuse.

On the Figure 2.2 we have short of examples of visual interpretations on how the data is labeled or associated.

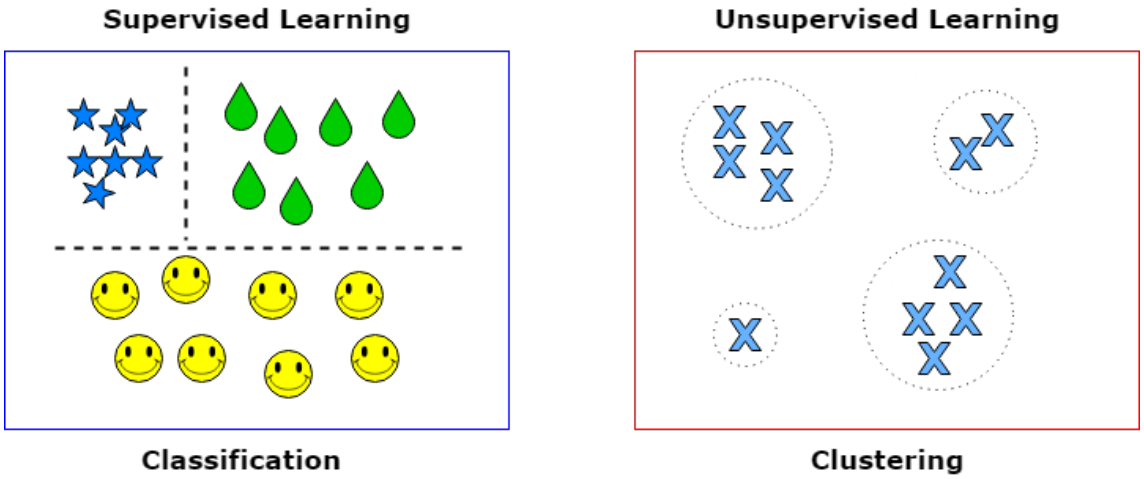


Figure 2.2: On the blue box, we have a classification example of data. On the red box we have an association of data based on the location of data feature.

2.1.2 Centralized Learning vs Decentralized Learning

Centralized Learning refers to a ML infrastructure that utilizes a central server or data center for both gathering and processing all training data. In this setup, the server is responsible for all the ML steps, validating the dataset, preparing the dataset for processing, training the model with the dataset, as well as validating the trained models and deploying them. Communication steps between the server and the data sources typically involves the transferring of the sources datasets. Most applications of Centralized Learning are seen in scenarios where there is less focus on problems such as privacy regulations and where there is a necessity for control over the dataset validity, process and resource management during the whole process.

Decentralized Learning involves the training of a model across multiple devices ideally without centralizing data in a server. The learning problem is distributed, because each device contributes with its own dataset, and trains a version of the global model locally, the communication step includes not the transferring of the device dataset but the parameters for the model that results from the global model trained locally. Since the model is improved through the collaboration of multiple devices we call this a decentralized learning process. This infrastructure, as techniques like Federated Learning [24] as example. These techniques have gained traction in scenarios where the communication of private datasets is not viable, this means that there can be either privacy regulations impeding the process, sensitive data that puts business cores and processes at risk or even lack of protection against data breaches. Thus, allowing multiple entities to cooperate in the training of a model without exposing their data to each other.

2.2 Synchronous vs Asynchronous

It is crucial to consider how tasks and processes work and coordinate with each other. These considerations extend to a fundamental concept in system design and communication, Synchronous and Asynchronous operations and communications.

In a synchronous environment, tasks and processes occur in a synchronized manner, following a specific order where each task waits for the previous one to finish before starting. In synchronous communication, interactions occur in real-time. For example, if a server is in a blocking operation waiting to receive a message it will only proceed to the next task when it receives the message, if there is a timeout option defined. Synchronous computing requires a progressive step by step evolution, where each step must be completed or discarded before moving to the next one. Examples of synchronous concepts in real world applications include video calls, real-Time systems, and live streaming.

Asynchronous environments allow tasks and processes to operate independently, i.e., not bound by a strict and defined timeline. Finishing a task may not indicate the start of a new one. Thus, allowing for the overlap of tasks and processes to work concurrently. In asynchronous communication, clients can receive messages from the server, process them at their own pace and respond when they are ready, without holding back the server process or task. Examples of this concept applied in real world scenarios include but are not limited to, Email communication and Asynchronous programming, where tasks are allowed to execute independently, and results are handled when they are received.

In terms of communication, the choice between these concept approaches varies based on factors such as what kind of tasks are required, if there is a need for immediate feedback, or if there is a desired level of flexibility for slower client processes. We can consider that both concepts have their own advantages and are suitable in different contexts. These concepts can also be used together to create systems that allow for both types to coexist, taking advantage of the pros of both approaches while mitigating the cons with each other.

2.3 Deep Learning

Deep Learning [17] comes from the necessity of solving tasks that are intuitive for people to classify but harder to properly describe them in rulesets, like recognizing incomplete words and objects in images. DL allows machines to understand past experiences by understanding them through simpler and smaller concepts. This avoids some of the traditional Machine Learning need for specific rules to be defined regarding to knowledge that is required. So a machine can learn complex problems by solving multiple simpler ones. The term "deep" comes from the comparison of DL with a graph, if we build a graph representing this, it has multiple layers that connect with each other. We can look at a simple example in Figure 2.3.

Deep Learning applications have increased over the years due to the growing available amounts of training data and computer infrastructures allowing for more complex models to be used.

2.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent also known as SGD, is one of the backbone algorithms used in Deep Learning. A continuous problem in ML is the need for large and rich datasets to be able to extract a good generalization from it, but this comes at a cost of computer resources. As the training data size grows to millions of examples, the time to take a gradient step also increases significantly. In order to minimize the time it takes for this step, SGD allows the sampling of the dataset using a mini-batch of examples, usually up to a hundred of examples only.

Flach2012 [13] states that it is common knowledge that SGD in fact works well in association with Machine Learning problems, due to the computational cost preservation. This is due to the fact that the computational cost per update doesn't increase with the increase of the dataset size. What increases is the number of updates to reach convergence.

2.3.2 Neural Networks

Neural Networks are the foundation of Deep Learning models, and they consist of at least three layers.

- **Input Layer** - This layer is usually the first one in the network and is responsible for receiving the data. If we take an image as an example each neuron in the layer might be referenced to a pixel in the image.

- **Hidden Layers** - These layers are responsible for transforming the input data into features, that can be recognized, this is accomplished with the introduction of weights and bias values to each associated neuron containing data.

- **Output Layer** - This layer is considered the last layer of the network and is responsible for producing the results based on the transformations performed in the previous layers.

2.3.3 Convolution Neural Networks

Convolution Neural Networks are a particular kind of neural networks for processing data, specifically used for data that has a grid topology. This includes image recognition for example.

Flach2012 [13] quotes:

> *Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*

Using an example to explain the operation called convolution, if we are tracking the location of an object with a laser sensor, and we get the following output from the laser sensor.

$$x(t) \tag{2.1}$$

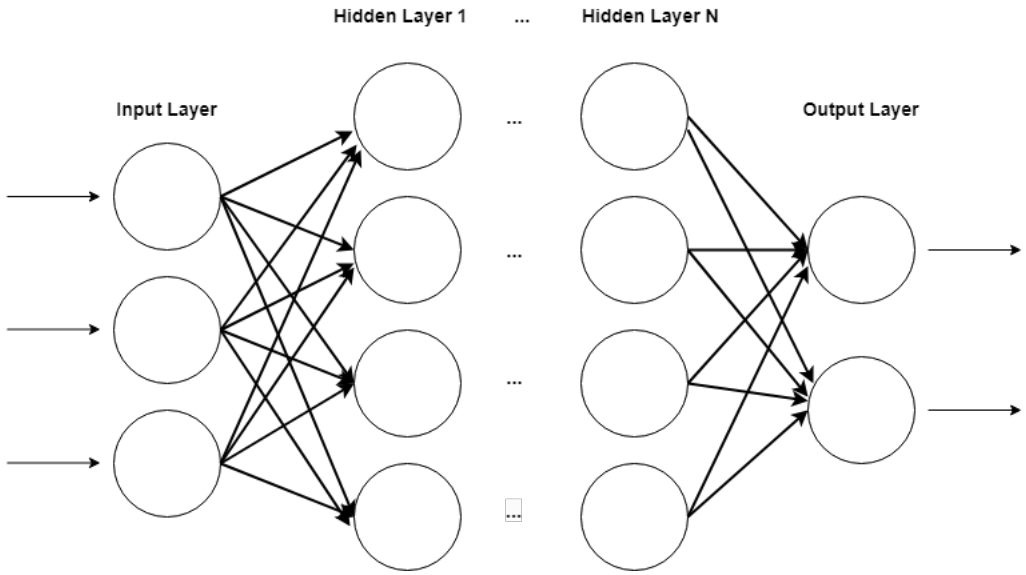


Figure 2.3: Deep Learning structure overview.

Given any time t, we receive the position back. If we consider the laser sensor to contain some associated noise from the environment, we would want to mitigate this factor. This is accomplished by using average values of the data we have. Simply put, if we had a weighting function designated:

$$w(a) \tag{2.2}$$

Where we have a, as the age of the data, i.e., if we consider that older data is less relevant to newer data. If we apply this weight function to the first equation we obtain an improved function for estimating the object position, this is what is referred as convolution.

This is relevant for image processing for example because it allows the processing of the data in a more efficient way. While traditional implementations of neural network layers use matrix multiplication to interact between nodes, this requires every node to interact with every other node. CNNs allow for less complex combinations, i.e., using sparse connectivity it allows the nodes to interact with fewer nodes to extract relevant features.

$$s(t) = (x * w)(t). \tag{2.3}$$

2.4 Federated Learning

Federated Learning is a decentralized machine learning approach where the model is trained across multiple devices or servers. Introduced in 2016 [24], FL enables collaborative training while preserving data privacy and leveraging distributed computation, which is particularly relevant in the era of edge computing. Individual users or entities can participate in training global models,

benefiting from a shared model while providing during training a new source of rich and untapped data. This data remains on the sources, sending only the training model update back to the server.

Federated Learning follows a collaborative structure where a centralized server coordinates distributed client devices. Figure 2.4 provides an overview of how FL works.

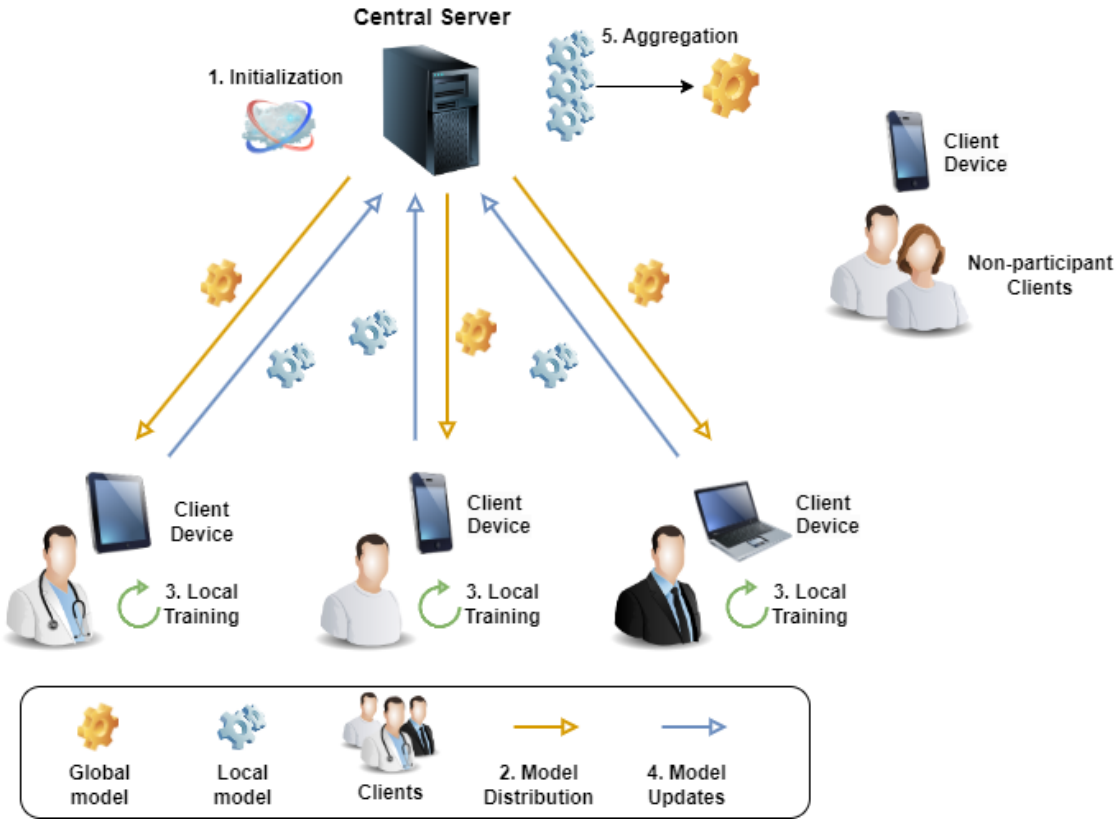


Figure 2.4: This is an overview of a Federated Learning algorithm.

1. **Initialization** - A global model is initiated on the central server.
2. **Model Distribution** - A selection algorithm chooses the devices that will participate in the next round, and the server shares the latest global model with them.
3. **Local Training** - Each selected device trains the model locally with its private data.
4. **Model Updates** - After training, each device sends its model update back to the server.
5. **Aggregation** - The central server aggregates the updates to create a new version of the global model.
6. **Iteration** - The steps 2-5 are repeated to refine the global model, with the subset of devices changing based on the selection algorithm.

With this structure for Federated Learning, there is room for alternative implementations and improvements on many stages, from different selection mechanisms to local training algorithms,

methods of aggregation, as well as types of communication used, such as using synchronous, asynchronous or hybrid approaches.

Synchronous Federated Learning is a variant of Federated Learning where a central server selects participating devices to collectively train the global model in a specific round [24]. The server waits for all devices to finish their local training before aggregating the updates for that round. This coordination ensures that devices contribute to and benefit from a consistent global model, promoting convergence across the system. It is suitable for scenarios where a uniform and coherent model is essential due to the coordination of training updates. Over several rounds, it is expected that the model will converge globally, reflecting the collective work of all participating devices.

However, challenges must be considered. Synchronous FL faces a significant hurdle known as the straggler problem, where the time taken to conclude a training round is influenced by the feedback of the slowest device. This bottleneck can potentially affect the overall efficiency of the model. Additionally, Synchronous FL lacks robustness in real-world environments where assumptions about stable network connections and computational power cannot be made. This lack of robustness may induce bias, especially when a poor selection algorithm is employed. Weaker edge devices may not be chosen as often due to their limited availability, further complicating the issue. The real-world implications of these challenges create the need for the consideration of alternative solutions in federating learning scenarios, opening the door to solutions based on asynchronous approaches.

Asynchronous Federated Learning is another variant of Federated Learning where participating devices have the capability to operate independently, sending their local model updates when they are finished, without adhering to strict round timelines or waiting for other clients to finish [38, 7, 27]. This independence allows devices to join subsequent rounds if they are slower, contributing to the global model at their own pace. One of the main advantages of Asynchronous Federated Learning is its flexibility, enabling the server to aggregate updates as they arrive or when a pre-defined minimum amount is reached. This inclusivity allows slower devices to actively participate in training rounds, reducing selection bias, and enhancing robustness to stragglers, thereby minimizing the impact on the overall efficiency of the model. In more detail, this flexibility allows devices to contribute to the global model whenever they complete local training rounds, not wasting the devices resources, as seen in some synchronous approaches.

A survey done on Asynchronous Federated Learning [23] states that this field has been gaining a lot of attention due to the advantages it brings. However challenges emerge, most notably in terms of consistency. Since updates arrive independently, considerations for regularization parameters maybe be necessary to alleviate the impact of older model updates on newer versions. Additionally, the continuous flow of asynchronous updates may lead to an increase in communication overhead at the server. Consequently, causing the overall convergence of the model may be

delayed, other areas include device heterogeneity, data heterogeneity, and privacy concerns.

2.4.1 Challenges and Problems

Federated Learning, with its decentralized approach, encounters several challenges [19] that impact its scalability, data distribution, privacy assurances, and communication efficiency. When considering a real-world scenario, we mostly encounter devices with unreliable connections, different computational resources, and with unbalanced non-IID datasets.

- **Scalability** - The scalability of Federated Learning poses a central challenge, particularly in scenarios involving a large number of devices. As the participant count increases, the communication overhead escalates, influencing bandwidth requirements and introducing potential bottlenecks. Strategies for efficient scalability are essential for applications such as IoT networks and large-scale distributed systems, making system design an important step [3].

- **Non-IID Data** - In real-world scenarios the data distributions on the client devices is not homogeneous and identical, it is rather independently and non-identically distributed. This can lead to challenges in achieving a globally optimal model, because the updates provided by the clients are biased towards their data. Handling non-IID data is critical to ensuring the model robustness and generalization across diverse user contexts [26].

- **Privacy Concerns** - Privacy preservation of client data remains a primary focus in Federated Learning, yet it introduces challenges. Balancing collaborative model training with established privacy requirements [34, 1, 15, 12] demands innovative techniques. As data stays localized on devices, addressing privacy concerns becomes vital, especially in applications involving sensitive information such as healthcare and finance. Despite considering FL an approach to mitigate data exposure several literature shows that new attack vectors surge, such as, membership inference attacks [30], property inference attacks [25], model inversion attacks [14], and leakage from gradients attacks [39].

- **Communication Overhead** - Effective communication is crucial in Federated Learning, and the associated overhead can be a limiting factor. Transmitting model updates between the central server and client devices demands efficient strategies to minimize bandwidth usage and optimize communication protocols [20]. Overcoming communication challenges is what enables the system responsiveness and overall performance.

With the evolving scenarios of Federated Learning, addressing the challenges that arise from the heterogeneity of devices is paramount. Achieving robust model convergence amidst variations in device capabilities, data distributions, and computational power is a key focus. This is effectively tackled through the implementation of robust-oriented aggregation algorithms and personalized Federated Learning approaches. Striking a balance between optimization and privacy

is crucial, with secure aggregation and Federated Learning employing homomorphic encryption playing vital roles in preserving individual client device privacy while optimizing models.

As Federated Learning extends its range into other domains such as edge computing, healthcare, and IoT, the significance of optimized federated learning algorithms becomes relevant. This approach facilitates collaborative learning, minimizing the impact and resource waste of individual devices. It ensures efficient and privacy-aware model updates, reinforcing the need to continually refine the global model for seamless collaboration across heterogeneous client devices. The ongoing pursuit of solutions to these challenges will not only enhance the effectiveness of Federated Learning but also fortify its adherence to privacy and its ability to accommodate diverse data distributions, ultimately contributing to its continued success across various applications.

Non-IID vs IID

In most Machine Learning scenarios IID datasets are considered due to facility they provide when applying algorithms and experiment result inferences, however in Federated Learning there isn't one big dataset, the number of datasets is associated with the number of clients participating in the training, and consequently the data on these devices is non-IID. Thus, making it a relevant point to test these algorithms and systems with both types on distributions. On the Figure 2.5 we can see an example for the dataset MNIST [10].

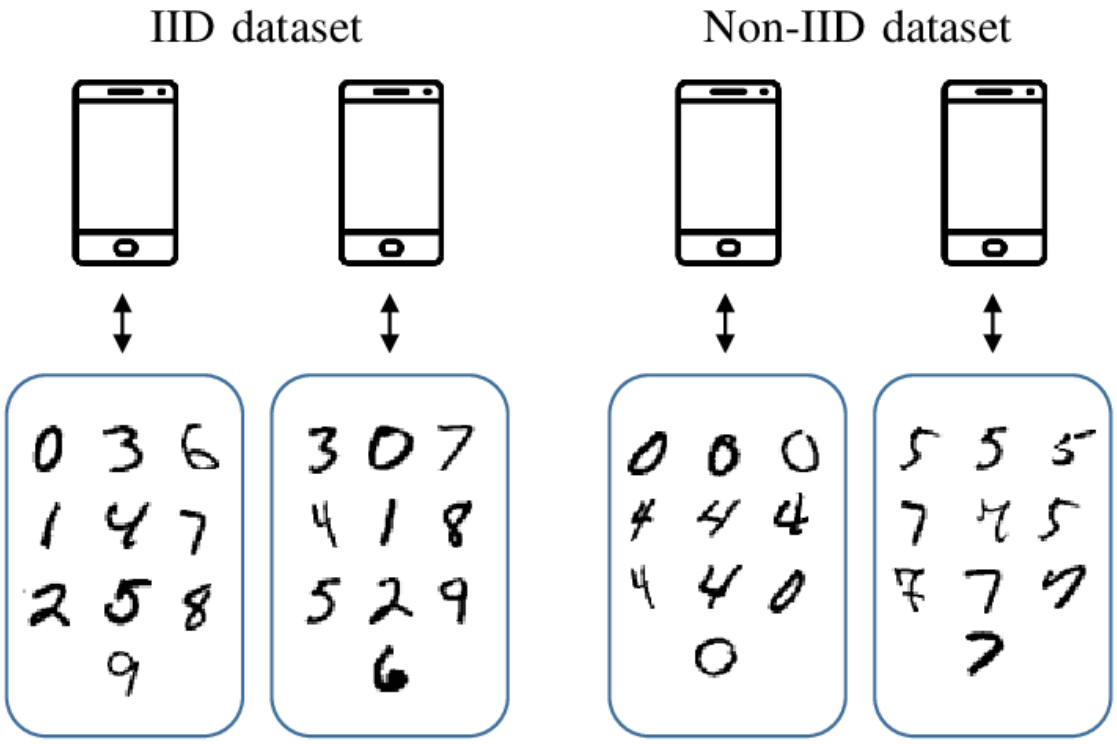


Figure 2.5: Dataset IID vs Dataset non-IID with examples from MNIST.

2.4.2 Secure Aggregation

Secure Aggregation [4, 5, 31] is a cryptography technique that helps mitigate some of the mentioned challenges within Federated Learning, addressing the problem some of the architectures used in collaborative model training. Its primary objective is to protect the leakage of data from client contributions to the global model while allowing the process to remain distributed. The central server is then capable of aggregating updates securely and improving the global model. Secure Aggregation is split into different stages of the FL process. (see Figure 2.6):

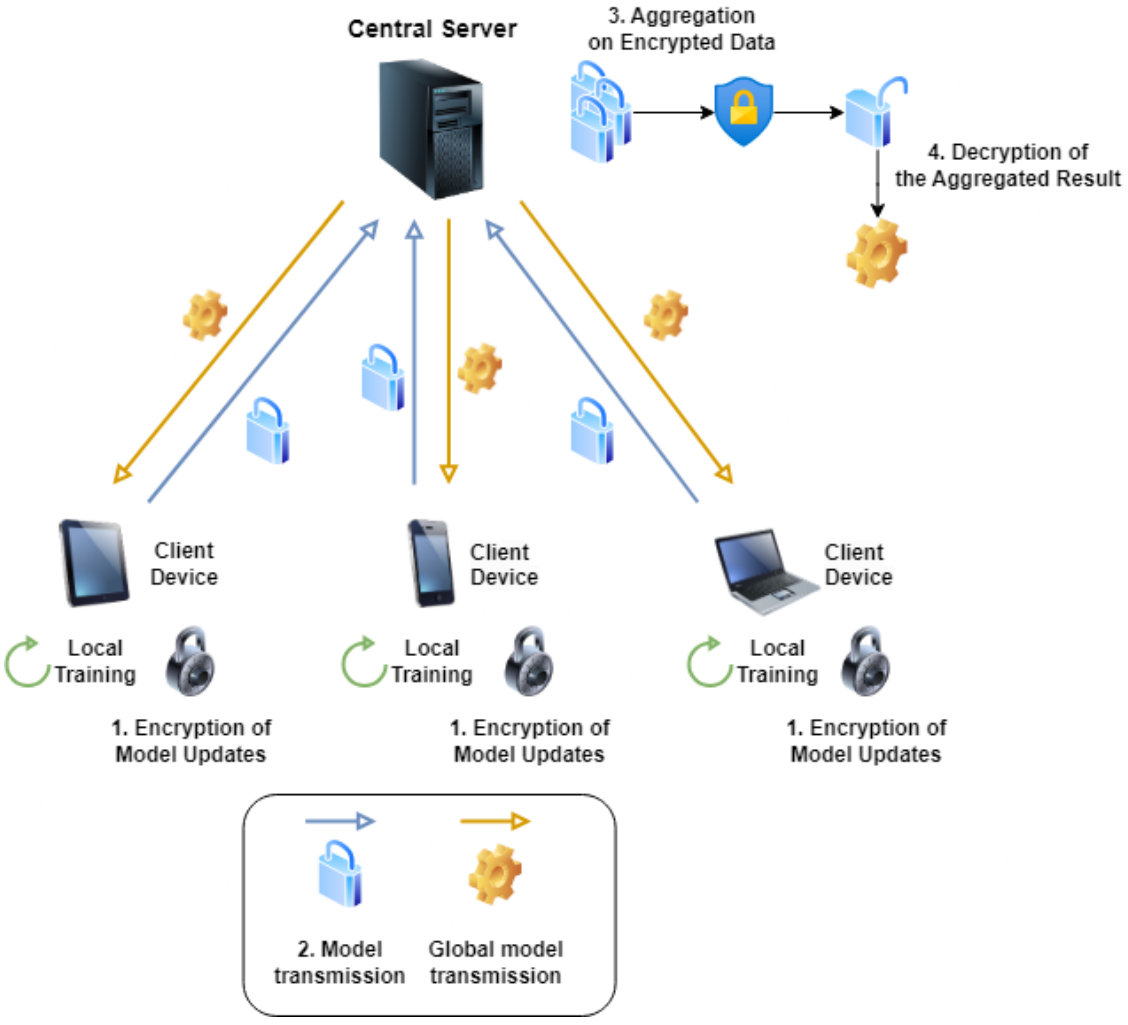


Figure 2.6: This is an overview of Secure Aggregation protocol.

1. **Encryption of model updates** - After every client finishes on device local training, each device then encrypts the resulting model parameters before proceeding to the communication step with the server maintaining the global model. Cryptography techniques, ranging from the use of homomorphic encryption techniques to the deployment of secure multi-party computation environments are employed to further enhance privacy safeguards.

2. **Model transmission to the Central Server** - The communication of the encrypted model

update depends on the system architecture. SecAgg seamlessly integrates with various algorithms, encompassing synchronous, asynchronous or hybrid approaches.

3. **Aggregation on Encrypted Data** - The server aggregates encrypted updates, without the need to decrypt each one individually, thereby preserving the confidentiality of individual contributions.

4. **Decryption of the Aggregated Result** - The server decrypts the aggregated result, revealing the updated model while ensuring the continued privacy of individual contributions.

So2021 [31] states that it is able to implement Secure Aggregation in FedBuff without the use of TEEs, allowing for any buffer size to be used. The experiments done by them also show that the comparison between FedBuff and FedBuff with Secure Aggregation on convergence time is similar.

Although very interesting from a privacy perspective, challenges may arise. Depending on the cryptography operations performed, computational overhead could impact the overall model performance. The task of communicating encrypted updates with numerous participating devices introduces additional challenges.

Choi2020 [8] shows that further improvements can be made in terms of resource usage without compromising data privacy.

Applications of Secure Aggregation are particularly indispensable in sectors where data privacy holds crucial relevance, including healthcare [11], and finance.

2.4.3 Differential Privacy

Differential Privacy (DP) is a concept that provides a robusteness to FL frameworks by mitigating privacy concerns that exist in Federated Learning [33, 35]. DP is a privacy-preserving paradigm designed to protect the individual contributions, the client updates, during the training process of the model. The main assumption made is that the inclusion or exclusion of an update does not provide a significant impact in the data leakage of updates. Even though users use their own local data, the exchange of model updates between the server and the devices introduces potential privacy vulnerabilities, such as model inversion attacks, inference attacks, and unintentional information leakage. DP comes to help establish a balance between utilizing the richness from data and safeguarding individuals privacy.

In the Federated Learning setting, where devices collaboratively improve a global model without centralizing data, DP plays an important role. By incorporating DP mechanisms, FL models can learn from the user's data while minimizing the risk of exposure for the data. This integration allows the contribution of user's data to the model training while minimizing the exposure of personal details, greatly increasing the trust in scenarios that have privacy as a requirement.

DP achieves this by introducing noise during the model training process before aggregating. The introduced noise creates a privacy layer by default, making the extraction and inference of

specific details about any individual contribution or data. In the FL model this is usually accomplished by the user adding some noise into the model update after the local training before the communication step where the user sends it to the server.

Despite the positive results we can achieve by using DP, some challenges emerge. A critical challenge is balancing between level and quantity of noise added to an update and convergence performance. The introduction of noise has a clear impact on model performance and convergence time making the trade-off between privacy protection levels and convergence performance an ongoing area of research, with new and innovative techniques such as adaptive noise injection and advanced privacy-preserving algorithms improving this trade off.

Wei2020 [35] states that the tradeoff is inevitable, with higher convergence performance leading to lower protection levels of privacy, but also showing that a higher number of participating clients with a fixed privacy protection level helps improving the model convergence.

In a real-world scenario, Differential Privacy finds applications in sectors where both data collaboration and user privacy are crucial, such as healthcare, financial, and personalized services. It ensures that FL models can learn from the rich private data of clients without compromising their individual contributions. With the development of Federated Learning, the integration of Differential Privacy emerges as a key enabler, to ensure the preservation of privacy concerns and regulations.

2.5 Federated Optimization Algorithms

FL operates on the principle of collaborative model training across decentralized client devices, presenting unique optimization challenges. Federation Optimization emerges as a crucial way to tackle these challenges, focusing on enhancing the communication efficiency, convergence rate, and overall performance of FL models [38, 6, 7, 36, 27, 21].

FO addresses the complexities involved in coordinating model updates from decentralized client devices to achieve a convergence for the global model. In FL scenarios, client devices may vary in different aspects such as computational power, network availability, and data distribution. Solving or optimizing the effective use of these resources becomes essential in achieving the potential of collaborative learning.

One of the key challenges is balancing local and global learning objectives. Individual client devices perform local training only using their data. Therefore, the aggregation process becomes rather crucial to optimize, ensuring that the global model effectively captures the various insights of the every client. Some of the techniques used can be federated averaging [24] or weighted aggregation [6], as these techniques aim to reach a good balance between the local and the global objectives.

In order to efficiently communicate the updates, the server has to take into account the way they are handled and communicated, therefore communication efficiency becomes a crucial area research to improve FL processes. Seeking strategies to minimize the communication overhead that exists between the devices and the server maintaining the model can be achieved by tech-

niques such as optimization of the transmission of model updates through the use of compression algorithms, reducing the required bandwidth usage, achieving faster global model convergence, as well as using optimization algorithms to reduce the number of communication round trips done overall. Several possibilities exist with already several viable approaches existing that use either synchronous or asynchronous approaches, or even hybrid ones, where pieces of both methodologies coexist [21, 36, 6].

FO also explores other methods like adjusting the learning rates across devices [7], using adaptive learning rate parameters to ensure that devices with faster convergence contribute more significantly, preventing slower devices from impeding overall model progress. However this might introduce bias in the model, where the participation of slower devices is less common, making the model biased towards clients that have more computational resources.

A summary of the studied literature on these algorithms can be seen on Table 2.1.

Name	**Ref**	**Type**	**Description**
FedAvg	[24]	Synchronous	Uses the weighted average of local computations over the client's data to improve results.
FedAsync	[38]	Asynchronous	Similar to FedAvg local approach, but introduces asynchronous updates to increase slower client's participation.
ASO-Fed	[7]	Asynchronous	Introduces asynchronous updates, local decay coefficient and a dynamic learning step size to tackle slower client contributions.
SAFA	[36]	Hybrid	Introduces an hybrid approach to model distribution and update, as well as an after training client selection.
FedAT	[6]	Hybrid	Hybrid approach that splits clients into tiers based on their training speed and performance.
HFL	[21]	Hybrid	Hybrid approach that uses an approximation function to adapt results from stragglers.
FedBuff	[27]	Asynchronous	Asynchronous approach that uses a buffer to store client updates on the server until a threshold of updates is reached, making it compatible with Secure Aggregation.

Table 2.1: Federated Learning Algorithms.

2.5.1 Federated Averaging (FedAvg)

The FedAvg algorithm, short for *FederatedAveraging* [24], is an optimization algorithm for the federated optimization problem that leverages concepts from stochastic gradient descent (SGD). The main idea behind SGD is to update the model parameters in the direction that reduces the loss function the most. SGD selects a subset of the dataset on each iteration and computes the gradient. Inspired by the success of SGD in deep learning applications, FedAvg adapts this approach to FL, following the main structure present in Figure 2.4.

1. **Initialization** The central server generates the first iteration of the global model randomly, or based or previously pre-trained data.

2. **Model Distribution** In each round of training a fraction of clients, C, is selected to participate and is sent the current global model.

3. **Local Training** Each selected client locally takes a one step gradient descent on the received global model using their local private data. This process is repeated multiple times in a round in order to achieve a weighted average of the local results.

4. **Model Updates** After finishing the local training round, a weighted average of the gradients is done and only the resulting model update is sent to the server.

5. **Aggregation** The central receives individual model updates from the participating clients, calculates their average, and updates the global model accordingly.

The algorithm begins with a baseline similar to Federated Learning + SGD, where the initial global model is trained using large-batch synchronous SGD, i.e., the batch size are the size of the clients data. In each round, a subset of clients is selected, and their local gradients are aggregated to update the global model.

FedAvg introduces an improvement to this baseline by incorporating additional computations on each client. In this enhanced approach, clients iterate over their local datasets before contributing to the global model. The algorithm introduces three key parameters for customization: a) the subset of clients involved in each round; b) the number of training passes each client does over its local data; c) the local mini-batch size. These parameters allow more control over the amount of computation used and adapt the algorithm to specific FL scenarios. By iteratively using this process over multiple rounds, the model is collaboratively improved by decentralized devices without centralizing sensitive data.

FedAvg follows the synchronous approach helping maintain consistency and coherence in the global model across all participating clients. While this proves to be beneficial in terms of achieving a uniform and consistent global model that is usable for the generic client, it also introduces challenges common in synchronous systems. An example is the potential delays caused by slower clients, commonly known as the straggler problem, since the server has to wait for every client to send their update in each round.

2.5.2 Asynchronous Federated Optimization (FedAsync)

The FedAsync algorithm [38] is named after the combination of Asynchronous methodology and Federation Optimization. The key ideas of FedAsync are solving local problems to ensure model convergence and using a weighted average of local results to update the global model, where the weight is adaptively modified based on the staleness of the received updates. It follows the same

Algorithm 1: FederatedAveraging, total number of participating clients K, number of clients per round k, selected clients for the round N, minibatch size B, learning rate η

Process `Server`**:**
- Initialize w_0
- **for** *each round r = 1,2,...* **do**
  - $N \leftarrow$ (random set of k clients $\in K$)
  - **for** *each client* $c \in N$ **do**
    - $w_{r+1}^c \leftarrow$ CollectClientUpdate(w_r)
  - $w_{r+1} \leftarrow$ WeightedAggregate($\sum w_{r+1}^c$)

Process `Client`**:**
- $\beta \leftarrow$ (split client dataset into batches of size B)
- **for** *each local epoch i from 1 to E* **do**
  - **for** *each batch* $b \in \beta$ **do**
    - $w_{new} \leftarrow w_{old} - \eta \Delta \ell(w, b)$
    - $w_{update} \leftarrow w_{new} - w_{old}$
- return w_{update} to the server

basic protocol structure as other FL algorithms (see Figure 2.4).

1. **Initialization** The central server generates the first iteration of the global model and runs the Scheduler and Updater threads.

2. **Model Distribution** In each round of training triggered by the Scheduler thread, a fraction of available training clients, workers, is selected to participate receiving the current global model with a timestamp.

3. **Local Training** Each worker locally trains the global model with their private data, following a similiar structure to FedAvg training.

4. **Model Updates** After finishing the local training round, a weighted average of the local results is pushed to the Updater thread via the Coordinator, that queues the model update.

5. **Aggregation** The central server Updater thread receives individual model updates from workers via Coordinator and updates the global model accordingly.

The server starts by creating a global model and running the Scheduler and Updater threads, both threads run in parallel asynchronously. The Scheduler thread is in charge of periodically triggering training tasks on some of the workers, the available training clients. The Updater thread job is to wait for workers responses and updating the global model.

When a training task is triggered, the Scheduler thread sends the global model with the current timestamp via Coordinator to the selected workers. Then, each worker solves locally an optimization problem using SGD for multiple iterations. The resulting model update is the weighted average of the local results. This version of the model is then sent back to the Updater thread via

Coordinator, i.e., the Coordinator acts as a middle man communicator between the Server and the Devices. The Scheduler, Updater threads and worker devices operate asynchronously, with non-blocking communication, allowing the Updater thread to immediately update the global model upon receiving a new client model version via Coordinator sent by a worker.

This approach helps mitigate the straggler issue, because in a real-world scenario where devices have heterogeneous conditions slower devices can still contribute to the global model, in order to mitigate the staleness introduced by updates from older rounds, FedAsync introduces a mixing hyperparameter in the aggregation function.

Algorithm 2: Asynchronous Federated Optimization (FedAsync)

Process `Server`**:**
- Initialize x_0
- Run Scheduler() thread and Updater() thread asynchronously in parallel

Thread `Scheduler()`**:**
- Periodically starts a training round on the selected subset N of clients K, and send the global model with the round stamp

Thread `Updater()`**:**
- **for** *round each* r **do**
  - Receive the pair (x_{new}^c, r^c) from any client
  - Optional: $\alpha_t \leftarrow \alpha \times s(r-\tau)$, $s(\cdot)$ is a function of the staleness
  - $x_r \leftarrow (1-\alpha_r)x_{r-1} + \alpha_r x_{new}$

Process `Client`**:**
- Receive the pair of the global model and its time stamp (x_t, r) from the server
- $\beta \leftarrow$ (split client dataset into batches of size B)
- **for** *each local epoch i from 1 to E* **do**
  - **for** *each batch* $b \in \beta$ **do**
    - $x_{new} \leftarrow x_{old} - \eta\Delta\ell(x, b)$
- Push (x_{new}^c, τ) to the server

2.5.3 Federated Learning with Buffered Asynchronous Aggregation (FedBuff)

The FedBuff algorithm, short for Federated Learning with Buffered Asynchronous Aggregation, as outlined in the research paper [27], ingeniously combines some synchronous benefits with asynchronous elements. By employing a buffer on the server to store client updates before aggregating them into the global model, FedBuff introduces a synchronous advantage to an asynchronous framework. This unique approach not only enhances the overall efficiency of the federated learning process but also ensures compatibility with Secure Aggregation. Notably, the use of a buffer prevents individual updates, contributing to the algorithm's robustness and security measures. This approach follows the generic FL algorithm overview in Figure 2.4.

1. **Initialization** The central server initializes a global model.

2. **Model Distribution** When clients start training asynchronously they receive the current version of the global model, and the number of rounds for local training.

3. **Local Training** Similar to other FL algorithms, the local training is done using SGD over a number of pre-defined rounds.

4. **Model Updates** When a client finishes local training, the results are sent asynchronously to the server.

5. **Aggregation** Even though FedBuff presents an asynchronous communication approach, the updates to the global model do not occur after every client update, but only when a defined number of updates is achieved.

After initializing the global model, the server selects a sample of clients from the available clients pool, asynchronously initiating local training tasks on these devices.

Each device starts local training upon receiving the global model, the designated learning rate for training, and the specified number of Stochastic Gradient Descent (SGD) rounds to be iterated.

At the end of local training each client sends its resulting update back to the server asynchronously, to participate in the round aggregation. The advantage of asynchronously sending the update is being able to participate in subsequent rounds even if the local training time exceeds the alocated time for the round.

The distinguishing feature of the FedBuff algorithm lies in its aggregation technique for handling updates. Unlike conventional asynchronous approaches that immediatly update the global model upon receiving a local update, FedBuff introduces a crucial shift. A buffer is employed to store incoming asynchronous model updates from clients. The update to the global model occurs instead when the pre-defined threshold for the buffer is reached. This approach makes it compatible with Secure Aggregation because it does not make individual updates to the global model, but an aggregation of them.

Algorithm 3: FedBuff, number of client updates per aggregation n

Process `Server`**:**
- Initialize w_0;
- **for** *each round* $r = 1, 2, \dots$ **do**
  - $N \leftarrow$ (random set of k clients $\in K$);
  - **for** *each client* $c \in N$ **do**
    - $w_{r+1}^c \leftarrow$ CollectClientUpdate(w_r);
  - $w_{r+1} \leftarrow$ WeightedAggregate($\sum w_{r+1}^c$);
  - This aggregation occurs multiple times based on the selected n *async*;

Process `Client`**:**
- $\beta \leftarrow$ (split client dataset into batches of size B);
- **for** *each local epoch* i *from 1 to* E **do**
  - **for** *each batch* $b \in \beta$ **do**
    - $w_{new} \leftarrow w_{old} - \eta \Delta \ell(w, b)$;
    - $w_{update} \leftarrow w_{new} - w_{old}$;
- **return** w_{update} to the server;

Chapter 3

FADO Requirements and Architecture

In this chapter, we go over the system requirements and the key architectural decisions that make the FADO framework. We start by outlining the system requirements in Section 3.1, followed by a closer look at the FADO architecture in Section 3.2. In this section, we also point out the most important improvements made compared to the previous version.

3.1 System Requirements

FL presents unique challenges and in order to tackle them, the system requirements for FADO must align with the framework goals. These requirements are essential to ensure that the system functions properly as intended and offers ease of use without requiring extensive back-end knowledge of the framework from users. FADO is designed to tackle these challenges by providing a scalable architecture capable of emulating various FL scenarios while ensuring reproducible results.

- **Realism -** Realism is the core objective of the FADO framework. It aims to emulate a wide range of FL scenarios that produce results resembling those found in real world environments. This allows users to test both new and existing FL approaches, ensuring that the results are comparable and applicable in real world applications.
- **Comparable and Repeatable Results -** Research requires results that are both comparable and replicable across different environments. FADO ensures that experiments produce consistent metrics for various algorithms and scenarios, enabling researchers to easily compare their results and reproduce experiments when needed.
- **Scalability -** Given the focus on realism, scalability is a critical requirement. FADO is designed to handle large scale experiments involving large quantities of clients, ensuring that the framework can emulate diverse FL experiments.
- **Ease of Use and Customizability -** FADO flexibility is balanced with ease of use. New users should be able to quickly run experiments using a documented configuration `.yaml` file, making the framework accessible to both beginners and experienced users.

- **Modularity -** The FADO framework follows a modular approach, dividing the FL process into distinct components. This design allows users to work with specific parts of the framework independently, creating opportunities for easier development of new components, such as algorithms that detail training processes or those that implement different aggregation methodologies.

- **Versatility -** Thanks to its modular design, FADO facilitates the integration of new algorithms for aggregation, client selection, attacks, and defenses, among other aspects of FL. This versatility allows users to implement new features with minimal changes to the FADO framework components.

3.2 FADO Architecture

3.2.1 Overview

The FADO framework architecture is shown in the Figure 3.1, with the 2 main components, *Builder* and *Runner* and their associated sub components.

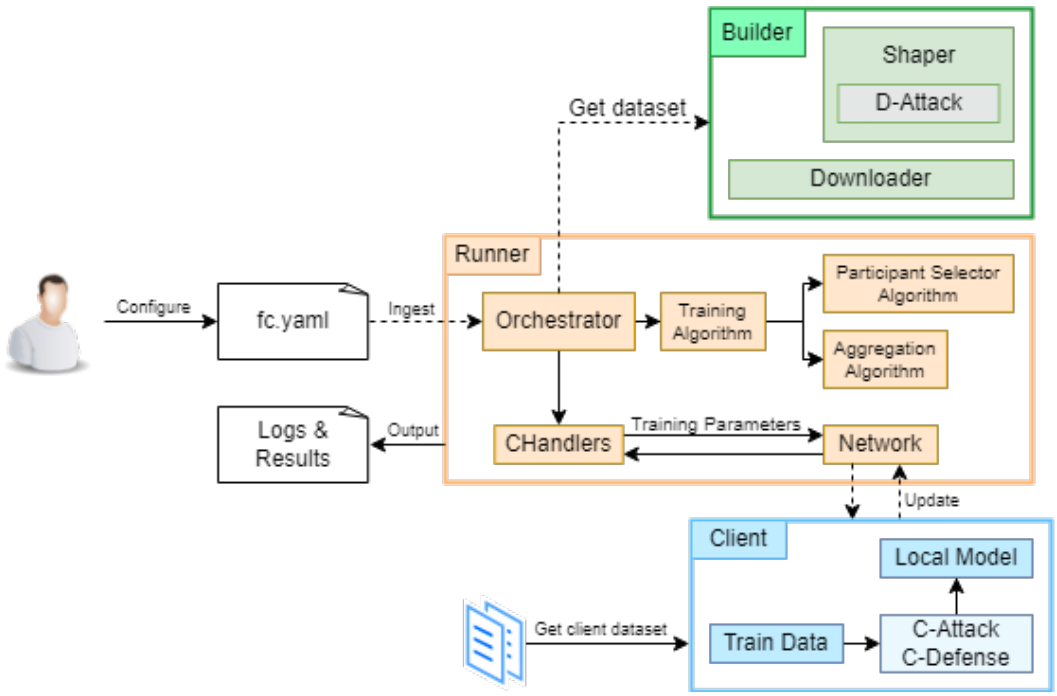


Figure 3.1: FADO framework main components overview.

The *Builder* handles dataset preparation, including tasks such as downloading and shaping the data for both clients and the server. Meanwhile, the *Runner* orchestrates the FL workflow, coordinating the training rounds and managing the global model along with its evaluation.

The FL workflow is divided into key sub components:

- **Training Algorithms (TAs)** control the initiation and conclusion of training rounds, determining if rounds are accepted or discarded.

- **Aggregation Algorithms (AAs)** update the global model with client contributions, triggered by the *Training Algorithm*.
- **Participation Selector Algorithms (PSAs)** select which clients participate in each round.
- Finally, the **Network** sub component manages the client-server connections during the training rounds.

In summary, the *Training Algorithm* serves as the central sub component, invoking the other sub components when required during each training round. These components will be discussed in further detail in the following sections.

The framework behaviour is defined and controlled using a configuration `.yaml` file. This file is manually edited and contains sections for each component and its sub components. Based on the chosen settings, users can run different experiments, emulating a wide variety of distinct scenarios, while ensuring the possibility to easily replicate them.

3.2.2 Builder

The **Builder** component is responsible for preparing the datasets utilized in the FL process. Its primary role is to divide the training examples among the defined number of clients and the testing examples for the server evaluation of the global model produced during the FL training.

To accomplish this, the Builder executes two steps, first, downloads the dataset from a repository, second, it shapes the data to make it compatible for training and testing. This shaping process considers various factors, such as: (i) the number of clients participating in the FL training process, (ii) the number of classes in the dataset, (iii) the data distribution.

Specifically, the Shaper allows for different distributions of training examples for each client, enabling scenarios that range from independent and identically distributed (IID) to non-IID (non-IID) data distributions. The distribution is achieved using the **Dirichlet** distribution, with the option to adjust the **alpha** parameter, influencing the disparity of examples split among the clients.

Users can configure the Builder by selecting from available options in the configuration `.yaml` file or by implementing their own version of a Shaper, as shown in the next Subsection 3.2.2. This flexibility allows users to customize the dataset according to their needs, whether integrating different datasets or applying specific operations to generate poisoned datasets.

Dataset and Shaper

The FADO framework is designed to be compatible with multiple dataset sources, provided they meet the next requirements:

- **Subclass of Shaper** - To integrate a new dataset into the FADO framework, the user must implement a `shape()` method. This method should return the following elements:
  - `train_loaders`: A list of DataLoaders, one for each client participating in the FL process, containing their training examples.

  - `test_loader`: A DataLoader that is responsible for providing test examples to the server to evaluate performance metrics during training.
  - `target_loader`: A DataLoader used for specific class testing, especially when targeted testing is needed.
  - `indices`: A list that tracks the number of examples allocated to each client, used for metrics collection.

- **Minimum number of examples** - Each client must be assigned at least one example, even in non-IID scenarios, ensuring every client can have a contributing role.
- **Compatibility with Deep Learning Models** - The dataset must be pre-processed or formatted to work with deep learning models. This includes compatibility with models like Resnet [37] for tasks such as image classification or complex visual recognition tasks.

Listing 3.1 illustrates the customizable options that control how the Builder component manages the dataset processing and shaping.

- **dataset -** Specifies the dataset to be used, as well as the associated Shaper that will preprocess and set up all the necessary data.
- **num_classes -** Defines the number of distinct classes within the dataset, required for the `Dirichlet` division.
- **data_distribution -** Also referred to as `alpha`, this parameter regulates the level of distribution between non-IID and IID scenarios.
- **resave -** This option indicates if the training examples for clients need to be stored in a file. This option is useful to save time when reusing the same dataset under identical settings.

Listing 3.1: Configuration file that manages how the FADO framework behaves (fado_config.yaml).

```
# Dataset/Shaper settings
dataset_spec:
    dataset: cifar10
    num_classes: 10
    data_distribution: 0.9     # values between 0.1 and 10,
       where 0.1 is the most non-IID possible, and 10 is the
       most iid possible
    resave: False
```

3.2.3 Runner

The **Runner** component includes several of the key contributions of this work. It enhances real world emulation by introducing a network communication layer between the server and clients, thus simulating the behaviour of distinct entities more realistically. Additionally, this architecture

improves modularity, allowing for the easy integration of new whole algorithms(TA+AA+PSA) or sub components of algorithms, such as TAs, AAs and PSAs.

The Runner is responsible for managing and executing the FL emulation. Its primary role is supervise the training rounds and update the global model with client contributions. To achieve this, the Runner includes several sub components, each in charge of a different phase in the FL process. The sub components are:

- **FL Process -** The main server-side component that orchestrates the emulation. Its sub components include:
  - **Training Algorithms (TA) -** Manage the initiation of training rounds, the triggering of other components during the round and its conclusion.
  - **Aggregation Algorithms (AA) -** Update the global model with client contributions when triggered by the TAs.
  - **Participation Selector Algorithms (PSA) -** Choose a subset of clients to participate in each round.
- **ML -** Responsible for constructing and maintaining the global model.
- **Network -** Manages client connections, creating handlers for each client.
- **Client -** Executes the client-side process, including obtaining the client dataset, local model training and communication with the Server.

FL Process

The **FL Process** is the core component responsible for orchestrating the entire FL workflow on the server side. It manages the execution of each training round and coordinates the communication between other sub components, such as the TAs, AAs, and PSAs.

The FL Process supports both synchronous and asynchronous training rounds. In synchronous rounds, the server waits for all clients to provide their updates before continuing, or it proceeds once a timeout threshold if reached. In contrast, asynchronous rounds allow the server to continue the workflow without waiting for all client contributions, processing updates as they arrive.

Figure 3.2 illustrates the interactions between the orchestrator of the FL Process and other key classes of sub components within the FADO framework, including the interaction with the Client entity.

Listing 3.2 illustrates some of the customizable options that control initial server settings and FL Process workflow. These settings allow users to run consistent and replicable experiments. Below is a brief explanation of the configuration options:

- **General Settings:**

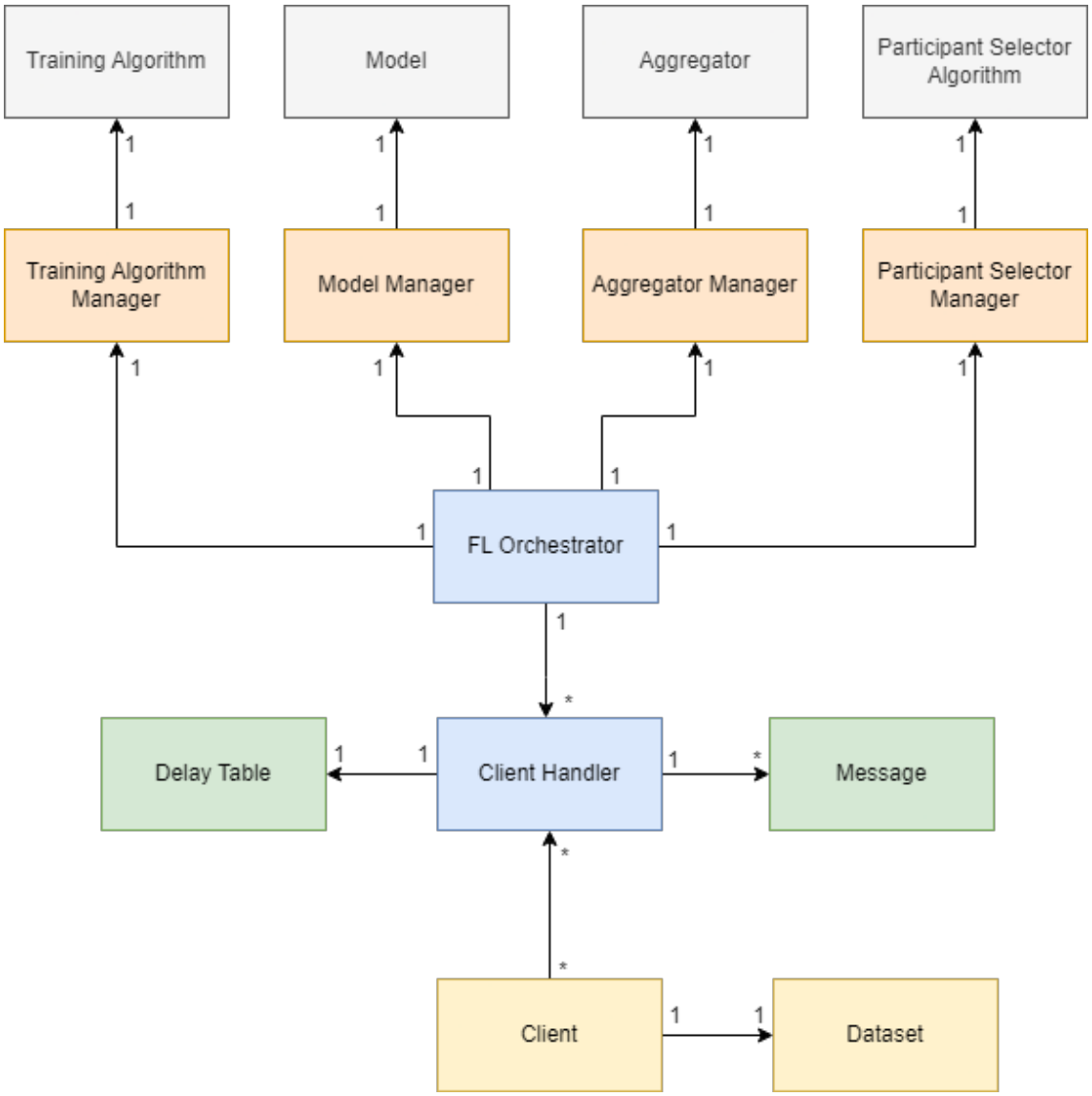


Figure 3.2: FADO class interactions.

- **random_seed -** Defines the seed used in pseudorandom number generators (PRNGs) and other Random implementations, ensuring experiments are consistent and replicable.
- **use_gpu -** Specifies whether the torch library should use the GPU, if supported, or default to the CPU for model training.
- **device -** Selects the computation device. `cuda:0` for the first GPU.
- **debug -** Activates verbose output and enables breakpoints during the FL workflow, allowing debugging at both the server and individual client levels.

- **FL Training Process:**
  - **rounds -** Defines the total number of rounds in the FL Process.

  - **agg_learning_rate -** Sets the learning rate for the aggregation algorithm used to update the global model.
  - **number_clients -** Specifies the total number of available clients. This affets the distribution of data across clients.
  - **clients_per_round -** Defines the number of clients that will be selected by the PSA for each round.

- **Output:**
  - **exp_name -** Names the experiment for logging and identification.
  - **logs_file_name -** Specifies the name of the log file containing verbose information about round by round progress and training duration.
  - **results_file_name -** Defines the filename where model evaluation results are stored, these include accuracy and loss per round.

- **Checkpoint Args:**
  - **checkpoint_file -** Specifies the path to the checkpoint file containing saved model parameters.
  - **checkpoint_round -** Indicates the round from which to resume an experiment.

Listing 3.2: FADO General and FL Process settings.

```
# General settings
# random_seed - [1, ...]
# use_gpu - [True, False]
# device - [cuda:0, cuda:1, cuda:2, cpu] # NOTE: this is for
   HORTA with 3 gpus. If your machine supports only one gpu,
   the available options will be [cuda:0, cpu], cuda does not
   support AMD
# debug - [True, False] # NOTE: This enables client side
   debugging if you support xterm process launching
general:
    random_seed: 1
    use_gpu: True
    device: "cuda:0"
    debug: False

# Federated Learning process settings
# rounds - [1, ...]
...
# number_clients - [1, ...]
# clients_per_round - [1, 20] # NOTE: Currently each process
   may take between 600mb and 870mb of gpu memory during peak
   training, so the maximum number of concurrent clients is
   somewwhat limited.
...
fl_training_process:
    rounds: 10
```

```
            agg_learning_rate: 1.0
            number_clients: 1000
            clients_per_round: 10

        # Log files
        # exp_name - Experiment name for logging files
        # logs_file_name # NOTE: contains experiment verbose, from
           training duration to round duration, to evaluation and
           aggregation results. Also, the initial config is printed
           into tables.
        # results_file_name # NOTE: .npy file containing, per client
           dataset examples, loss and accuracy per round, this file
           config.
        output:
            exp_name: "FL_TEST"
            logs_file_name: "FL_TEST.log"
            results_file_name: "FL_TEST"

        # Checkpointing settings
        # checkpoint_file: .pth, containing parameter models.
        # checkpoint_round: round of saved model, it also starts the
           experiment at this round
        checkpoint_args:
            checkpoint_file: "../../CKP/FedAsync_FLround2000.pth"
            checkpoint_round: 2000
```

Training Algorithms

The **Training Algorithms (TAs)** are the primary component of any FL algorithm implemented within the FL Process. Their purpose is to manage each training round, trigger client processes, and initiate interactions with other components like AAs and PSAs. By keeping TAs as a separate component, the framework allows for the easy integration and substitution of different AAs without modifying the core training logic.

The framework currently supports a variety of TAs, including both synchronous and asynchronous algorithms. This modular structure allows users to easily incorporate new algorithms by adding their class reference to the `AlgTypeManager class`, which manages available algorithms and selects the chosen option in the configuration file. The currently available algorithms are described in Section 2.5 and include:

- `FedAvg` - This synchronous algorithm initiates each round with a predefined number of clients and aggregates their updates once all contributions are received or a timeout occurs. If the timeout is reached, the round is discarded and no aggregation takes place.

- `FedAvg30` - Identical to FedAvg in most aspects, but it triggers an additional 30% of clients per round while maintaining the same required number of updates for aggregation. This ensures less rounds are discarded when some clients fail to provide updates.

- `FedAsync` - This asynchronous algorithm performs aggregations as soon as client updates reach the server. It does not discard delayed updates but instead applies a mitigation weight to stale the updates. The degree of this staleness and the associated weight are configured by the user.

- `FedBuff` - An asynchronous algorithm that stores client updates in a buffer and aggregates them when the set threshold is reached. It allows the contribution of delayed updates in later rounds, applying a mitigation weight if necessary.

Listing 3.3 presents the parameters and settings for each of these Training Algorithms, giving users the control over timeout thresholds and staleness function parameters. Below is an explanation of these settings:

- **FedAvg & Variants:**
  - **favg_round_timeout -** Specifies the timeout threshold in seconds for each training round. If the timeout is reached before receiving all client contributions, the round is labeled as discarded and client updates are deleted.
- **FedAsync:**
  - **fa_time_per_round -** Defines the time interval in seconds between the start of new rounds.
  - **fa_a -** Specifies the mitigation weight applied to updates that arrive after a set threshold.
  - **fa_b -** Sets the number of delayed rounds a client update is allowed before the mitigating weight is applied.
  - **fa_alpha -** Defines the weight of an individual client update during the aggregation process. For example, a setting of 0.1 means the global model weight is 90%, making the client contribution only 10%.
- **FedBuff:**
  - **fb_time_per_round -** Defines the fixed time interval in seconds between the start of new rounds.
  - **fb_updates_to_agg -** Specifies the number of client updates that must be collected and stored in the buffer before triggering the aggregation process. The buffer size must be compatible with the number of clients per round.
  - **fb_staleness_threshold -** Sets the number of delayed rounds a client update is allowed before it is punished with a mitigating weight.

Listing 3.3: Training Algorithm settings.

```
# favg_round_timeout: Timeout in seconds for the current round,
   discards all client updates if reached.
FedAvg:
    favg_round_timeout:  12

# fa_time_per_round: Time until next clients are triggered.
# fa_a: Mitigation weight parameter for delayed updates.
# fa_b: Number of delayed rounds an update is allowed to before
   mitigating weight is applied.
# fa_alpha: [0.1, 0.5, 0.9] # NOTE: Adaptive hyperparameter that
   impacts the weight of the update on the global model.
FedAsync:
    fa_time_per_round: 9   #seconds
    fa_a: 10
    fa_b: 4
    fa_alpha: 0.1

# fb_time_per_round: Time until next clients are triggered.
# fb_updates_to_agg: Buffer threshold until aggregation is triggered.
   Must be divisible by the number of clients per round.
FedBuff:
    fb_time_per_round: 12   #seconds
    fb_updates_to_agg: 10
    fb_staleness_threshold: 4
```

Aggregation Algorithms

The **Aggregation Algorithms (AAs)** are responsible for updating the global model based on the contributions received from participating clients. They are triggered by the TAs to perform this task. Each AA is designed to handle model updates differently, based on their corresponding TA.

The framework currently implements several aggregators such as, `FedAvg`, `FedAsync` and `FedBuff`. New aggregators can be easily integrated by following the basic requirement:

- **Subclass of class Base_aggregator** - Any new aggregator must inherit from the `Base_aggregator` class and implement at least the `aggregate()` method. This ensures compatibility with the input format required by the corresponding TA. For example, some TAs might return the delta (the difference between the initial global model parameters sent by the server and the parameters after the client training), while others might return the trained model parameters. The aggregator must be able to process that information to perform the update of the global model.

Each aggregator is designed to handle the input and the output required by the corresponding TA. The following aggregators are available:

- `FedAvgAggregator` - This aggregator is used in the **FedAvg** algorithm. It receives a reference to the latest version of the global model and the delta of each client participating

in the current round. Then it proceeds to calculate their weighted average before adding the result to the global model.

- `FedAsyncAggregator` - This aggregator is designed for the **FedAsync** algorithm. It receives a reference to the latest global model, the model parameters from the client trained model, the current round number and the round number when the client did the training. It verifies if a staleness function should be applied and proceeds to update the global model with the individual client contribution based on the weight previously set in the configuration file, `fa_alpha`, as seen on Listing 3.3.

- `FedBuffAggregator` - This aggregator is implemented for the **FedBuff** algorithm. It receives a reference to the latest global model and a weight accumulator. The weight accumulator is a list containing the client updates, where each update has already been weighted through a staleness function. The staleness function determines the weight based on the difference between the round the aggregation is triggered in and the round in which the client participated. Using the weighted updates, the aggregator performs a weighted sum, capturing the contribution of each update. The result is then added to the global model.

Participant Selector Algorithms

Currently there is only one option implemented in the framework for **Participant Selector Algorithms (PSAs)**, the random selector. However, it is designed to accommodate the integration of new PSAs. To add a new algorithm, it must meet the following requirements:

- **Subclass of ParticipantSelector class** - The new PSA must inherit from the `ParticipantSelector` class. It should implement the method `get_participants()`, which must return a list containing the indices of the selected clients.

- Additional methods can be added to support more complex PSAs, for example to save metrics such as number of contributions and time to finish training.

The settings for the participant selector in the FL Process can be defined in the configuration file. As seen in the Listing 3.4:

- **participant_selector -** Indicates the name of the PSA to be used for the selection of clients in each round.

Listing 3.4: Participant Selector settings.

```
# Federated Learning process settings
...
# participant_selector - [random] # NOTE: Currently there's no
    implementations for participant selectors besides the
    random selector
```

```
...
fl_training_process:
    ...
    participant_selector: random
    ...
```

Network

The **Network** component in the FADO framework provides extensive customization options via the configuration file, allowing users to simulate various real world network conditions. These settings offer possibilities of different scenarios, such as encrypted communications, IP and port server configuration to allow socket communication and simulation of network latency's delays based on different geographical locations. The available options are:

- **Encrypted Communication -** The framework support secure communication by wrapping socket connections in an SSL/TLS context when the encryption option is enabled. This ensures that all messages exchanged between the server and the clients are encrypted. This is done using an asymmetric key pair and a signed certificate.
  - The server provides a certificate file, typically signed by a trusted Certificate Authority (CA) or self-signed during development. This certificate contains the server's public key and helps clients verify the server's identity during the SSL handshake. Each client must have access to this file.
  - The private key corresponding to the public key in the certificate is stored in a separate key file. The server uses this private key to prove its identity and complete the SSL handshake with each client.
- **IP and Port configuration -** Users can specify the ip and port that the server socket will listen on. Clients will use this ip:port configuration to attempt to establish a connection in order to participate in the training rounds.
- **Delay emulations -** This option is used to emulate different network scenarios by introducing artificial delays to the client-server communications, copying real world latency conditions. These delays are applied to client messages and are based on the calculated delay table.

  The delay table is calculated once at the start of the experiment. It is generated by running a script with the ping command to measure the network latency between the server and several provided geographical server locations. The script results, which represent the communication delays, are stored in the delay table in `.json` format.

  Each client that participates in the FL Process is assigned to one of these geographical locations. When the delay emulation option is enabled, every message exchanged between

the client and the server has the additional latency applied from the delay table. This feature is relevant because it enables the framework to emulate real world network conditions more closely.

- **Stragglers -** This option allows for the simulation of straggler clients, which represent clients that have significantly slower response times when compared to the average time of other clients. Emulating stragglers becomes important because it reflects the real world conditions of some devices. In reality, not every client will perform at the same computation speed. By incorporating stragglers into the experiments, users can better understand how these slower clients impact the training of the global model based on the used algorithm.

  This is accomplished in the framework by introducing specific increase delays or random delay values. These delays are then applied to messages from clients that are tagged as stragglers.

Listing 3.5 shows the settings that allow the customization of the network behavior. Below there is a detailed explanation:

- **ssl -** Enables the use of encrypted communication.
- **certificate_file -** Indicates the path to the file containing the certificate file. This is used by the clients to obtain the server public key and connect to the server via socket when the SSL option is enabled. It helps clients authenticate the server and establish the encrypted connection.
- **key_file -** Sets the path to the key file containing the private key of the server used to prove ownership of the certificate when the SSL handshake is performed.
- **ip -** Specifies the IP address of the server.
- **port -** Defines the port where the server socket is listening.
- **delay -** Whether the Delay Table functionality is enabled. This will introduce network delay simulations when clients trade messages with the server.
- **recalc_delay -** Option to recalculate the delay table, used to generate new values. Usually done when using new locations.
- **delay_table_file -** Path for the delay table file in `.json` format.
- **straggler_percentage -** Probability that a client will act as a straggler.
- **increase -** Sets the number of seconds a straggler client message will be delayed.
- **different_delays -** Uses a random delay between 1 and 100 in seconds instead of increase.
- **num_locations -** Number of different locations/servers used to populate the delay table.

- **location_n -** IP or domains of pingable servers to obtain latency's used to simulate different geographical client positions.

Listing 3.5: Network settings.

```
# Network settings
# ssl - [True, False]
# certificate_file: location of certificate
# key_file: localtion of key
# ip - Server ip address
# port - Server port
# delay - Use the delay table to emulate client delays (
    Requires a .json file with format similar to delay_table.
    json)
# recalc_delay - Recalculate the delay_table.json file and
    resave it or generate a new one if it didn't exist.
# delay_table_file - file name for the delay table (.json
    format)
# straggler_percentage - Chance of a client delay to act as a
    straggler [1,2,...,100]
# increase - added delay is seconds if straggler is triggered
network:
    ssl: False
    certificate_file: "fado/runner/fl/network/server.crt"
    key_file: "fado/runner/fl/network/server.key"
    ip: "localhost"
    port: 6001
    delay: False
    recalc_delay: True
    delay_table_file: "fado/runner/fl/network/delay_table.json"
    straggler_percentage: 5
    increase: 10
    different_delays: False
    num_locations: 5
    location_1: "esmd-bluevps.metercdn.net" # Madrid, Spain
    location_2: "defr-bluevps.metercdn.net" # Frankfurt,
       Germany
    location_3: "ilta-bluevps.metercdn.net" # Tel Aviv, Israel
    location_4: "sgsg-bluevps.metercdn.net" # Singapore,
       Singapore
    location_5: "ausd-bluevps.metercdn.net" # Sydney, Australia
```

Client Process

The Client component is designed with realism and scalability in mind, enabling it to run independently even when executed on the same machine as the server. In this setup, each client is launched as a separate process, ensuring isolated execution and reserved memory space while taking advantage of the Network component to communicate with the server during training rounds. Each client is only using GPU memory when participating actively in a round, thus it allows for a more realistic emulation of multiple clients while addressing the GPU memory constraints.

In the Listing 3.6, the customizable settings for the client local training are shown. These allow users to configure different aspects of the client training process:

- **model -** Defines the type of model used by the server so the client can create a model of the same type with the received parameters during communication.
- **batch_size -** Specifies the number of training samples in each mini batch during training.
- **epochs -** Indicates the number of local training iterations each client must perform over their dataset.
- **learning_rate -** Specifies the step size at each iteration of the training process, controlling how much to adjust the model weights in response to the estimated error.
- **momentum -** Sets the momentum used, helps the model achieve convergence faster.
- **weight_decay -** Sets the weight decay used during training to help decreasing overfitting.

Listing 3.6: Client training settings.

```
# Client side training process settings
# model - [resnet, MNIST]
# batch_size - [64] # NOTE: This value can change.
# epochs - [1, ..., 10] # NOTE: Number of local iterations
   clients do.
# learning_rate: [0.001, ..., 1]
client_training_process:
    model: resnet
    batch_size: 64
    epochs: 2
    learning_rate: 0.07
    momentum: 0.9
    weight_decay: 0.0005
```

Chapter 4

FADO Implementation

This chapter details the technical implementation of the FADO framework, explaining how the design principles and architectural decisions presented in Chapter 3 were put into practice. It covers the tools and libraries utilized during the development process, along with the methods used for client-server interactions that reflect real world federated learning scenarios. The chapter also outlines the approach taken for distributing data across clients and provides a comprehensive overview of the federated learning workflow in FADO. Finally, we address the challenges encountered in ensuring scalability and the solutions implemented in the framework to overcome these obstacles.

4.1 FADO Orchestrator Workflow

Figure 4.1 illustrates the FADO workflow, beginning with the server initializing the Runner component. This component ingests the user configured file containing the experiment settings. Once loaded into FADO_Arguments, the Runner starts the Orchestrator, the sub component responsible for managing and coordinating the experiment's steps.

In the setup phase, the Orchestrator initializes the necessary subcomponents for FL and the Builder component to get the shaped dataset. Using the `get_model()` and `get_shaper()` methods to receive the global model from the Model Manager and the training and testing datasets from the Builder component. Before concluding this step, the Orchestrator stores each client's training dataset on disk, so clients can access them avoiding transferring datasets over socket, as it is not expected in Federated Learning.

Following the setup, the Orchestrator starts Step 2. During this phase, the training algorithm iterates over the number of specified rounds. For each round, the selected TA requests a subset of clients from the participation algorithm manager. These clients are launched as independent processes, isolated from the Runner control.

This initiates the third step in the process, where clients connect to the server, which uses client handlers to communicate with and manage each client connection. The client receives and sends feedback from and to the server via this setup.

Upon completing Step 3, the TA starts the aggregation process, step 4. If the client contri-

butions are received and within acceptable parameters the TA requests an aggregation from the Aggregation Manager. For FedAvg, this step occurs every round after receiving all the client contributions, and for the asynchronous algorithms it occurs after receiving every contribution, or in the case of FedBuff, after the buffer threshold is filled.

Steps 2 through 4 repeat until the loop condition is met, concluding the experiment.

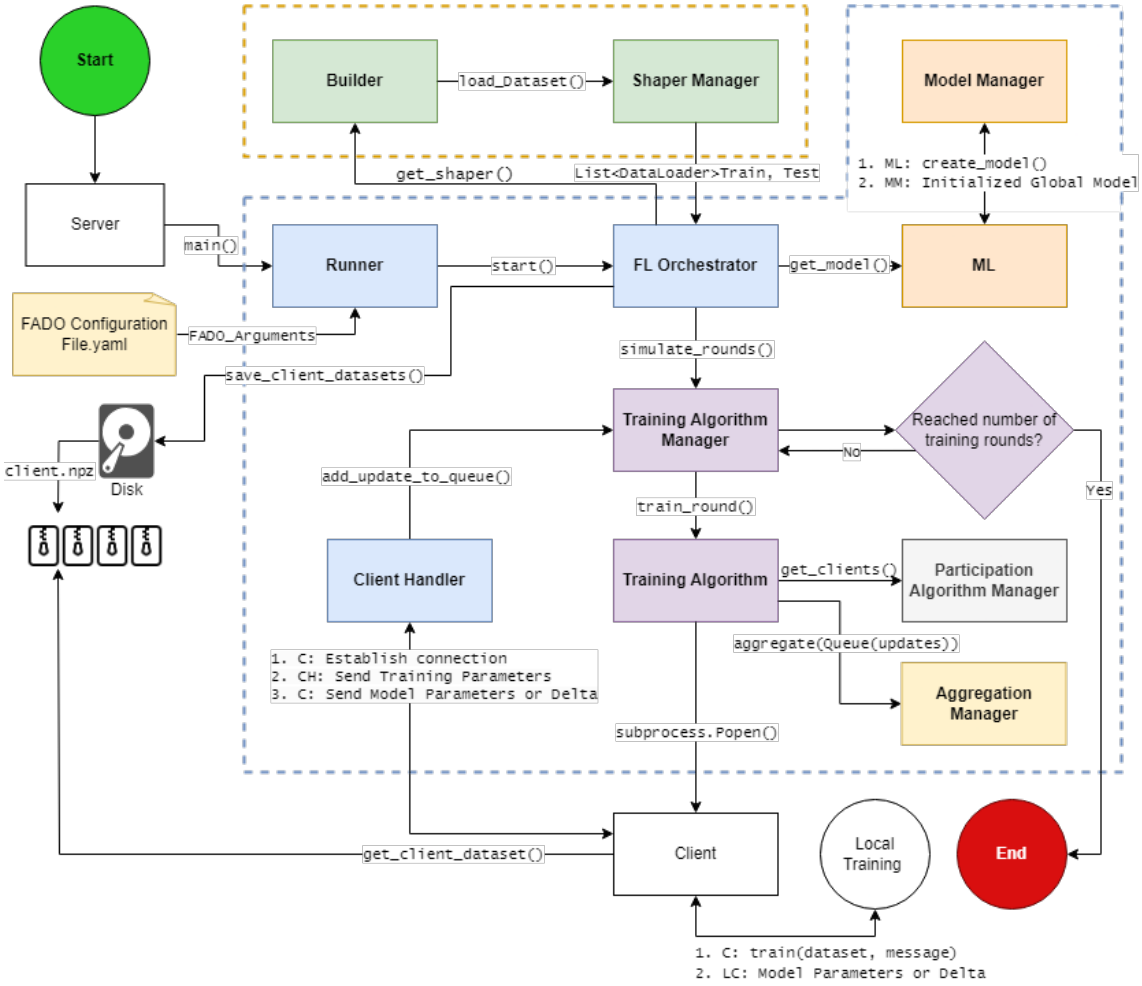


Figure 4.1: FADO Orchestrator Workflow.

4.2 Technologies and Tools Used

Development Language. The FADO framework is developed in **Python**, chosen for its extensive selection of libraries, compatibility with ML models, and the various algorithms that are part of the FL workflow. Python's simple syntax and active community support make it a good choice for the maintenance and further development of the framework.

Machine Learning Library. FADO relies on PyTorch [28] as its primary ML library, maintained from the previous version due to its extensive toolkit for model implementations, efficient handling of neural network, and common use within the FL research community. The implemented ResNet model, designed for efficient image classification tasks, uses the **torch.nn** library to define and create its layers and structure. This model shows to be a good choice for experiments using the CIFAR-10 dataset.

Client Process. To manage client processes, we opted to replace the previous use of Docker containers with independent processes using subprocess.Popen. This adjustment was considered due to the FADO's goals of scalability and ease of implementation for new algorithms and communication methods. Thus, client processes are launched via subprocess.Popen and optionally in a separated window(xterm) if debugging is enabled, giving direct access to each client. This approach reduces resources demands without sacrificing isolation and scalability, facilitating the potential deployment of clients on separate machines when higher scalability is required.

Data Management and Distribution. Data distribution among clients in FADO uses `torch.utils.data.DataLoader` for dataset handling and NumPy's **np.savez_compressed** for storage of individual client datasets in `.npy`. This structure enables each client to access its data independently, minimizing memory usage and allowing the framework to support larger datasets and multiple clients, without the need for the server to transmit to the clients their datasets.

Logging and Results. The training process within FADO is monitored and logged using Python's logging library. This aids both in debugging and in maintaining a record of each training experiment, which is critical for the metric evaluation of algorithms. Final results, including evaluation metrics, are saved in `.log` and `.npy` files, allowing for further statistical analysis after the experiment.

These choices were made with the framework's goals in mind, focusing on scalability, approximation to real world scenarios and ease of use. The specific implementations of each components are explored further in the following sections.

4.3 Client and Server Emulation

In a real world FL scenario, a client is any device that may contribute to training a global model by performing local training using its own data. In these settings, clients are not required to participate in every training round. Instead, they are selected each round using PSAs that may be random or consider multiple factors, such as availability, network conditions and computational powers.

To emulate a client in the FADO framework, two primary parameters are considered:

1. Operate as an independent process: Each client functions as a separate process, distinct from the server that hosts the global model.

2. Local dataset: The client holds its dataset locally, never receiving data transfers from the server.

This emulation is achieved by launching each client as an independent process from the main framework at the start of each round, therefore fulfilling the first parameter. This method to enforce isolation also creates a decentralized approach, as each client is independent from the others.

The following subsections describe the technical implementation of the framework components used for clients and the server.

4.3.1 Client Overview

In Listing 4.1 is the representation of the `fl_client.py` file, the `client_main.py` and a snippet of how the clients are launched as independent processes within FADO round life cycle.

```
subprocess.Popen(['python', client_path, str(client_id),
↪  str(fado_args.ssl), fado_args.ip, str(fado_args.port),
↪  fado_args.certificate_file])
```

Listing 4.1: Client class.

```
#fl_client.py
class FLClient:
    def __init__(self, client_id):
        ...
    def train(self, dataset, message) -> tuple:
        ...
    def _compute_delta(self, global_model, client_model):
        ...
    def get_dataset(self, batch_size):
        ...

#client_main.py
def main(args):
    validate_arguments(args)

    client_socket = socket.socket(socket.AF_INET, socket.
        SOCK_STREAM)
    if sslstate:
```

```
            ssl_context = ssl.create_default_context()
            ssl_context.load_verify_locations(cafile=certificate_file)
            client_socket = ssl_context.wrap_socket(client_socket,
                server_hostname=ip)
        client_socket.connect((ip, port))

        message = Message(client_id=client_id)
        Message.send(client_socket, message)
        message = Message.receive(client_socket)

        client = FLClient(client_id)
        dataset = client.get_dataset(message.batch_size)
        client_parameters, loss = client.train(
            dataset,
            message
        )

        if message.modelp_or_update:
            after_train_message = Message(model_parameters=
                client_parameters, client_id=client_id, round_number=
                message.round_number)
        else:
            after_train_message = Message(update_params=
                client_parameters, client_id=client_id, round_number=
                message.round_number)
        Message.send(client_socket, after_train_message)
```

Launching a Client

At the beginning of Listing 4.1, we see how FADO framework launches each client using the `subprocess.Popen`. By creating clients as independent processes, each instance has its own dedicated memory space, aligning with the framework's design objectives outlined in the previous Chapter 3. The use of independent processes enables client isolation, which is essential for emulating real world FL scenarios.

Each client receives through command line arguments its communication settings and its identifier, allowing it to connect to the server and participate in a training round.

Client Workflow

The `client_main.py` definition on the Listing 4.1 defines the workflow of a client within each training round. This process consists of the following steps:

1. Connection Establishment: The client initiates a connection to the server listener socket, establishing a connection to receive the training round parameters and later send the training results.

2. Client Identification: After the connection is established, the client identifies itself, requesting the training parameters.

3. Local Training: Using its dataset, the client trains the received global model locally. The dataset is obtained by decompressing the `.npz` file corresponding to the client id.

4. Model Update: Once local training is complete, the client, based on the requested output by the server, either calculates the model update or sends the model parameters. This adaptation is used to ensure that the implemented algorithms on the server receive the expected data structure.

This process allows the client component to manage its own data, perform local training, and contribute with its updates to the optimization of the global model, without compromising the integrity or privacy of its data.

4.3.2 Server Overview

The server within the FADO framework acts as the orchestrator of the FL process. It is responsible for initializing the global model, communicating with clients to collect their updates, aggregating these updates to the global model, and maintaining the global model. The following subsections detail the server's initialization process and its interactions with clients.

Server Initialization

To begin a FADO emulation, a configuration file (`fado_config.yaml`) must be created and populated with the necessary parameters. Once the configuration file is set up, the server can be launched with the following command:

```
fado --f fado_config.yaml run
```

Executing this command starts the `fl_orchestrator` process, which initializes the server and begins the FL workflow according to the specifications defined in the ingested configuration file.

Listing 4.2: FLOrchestrator class overview.

```
class FLOrchestrator:

    fado_args = ingest_config_file()

    def __init__(self):
        self.global_model = ModelManager.get_model(fado_args)
        ...

    def start(self):
        ...
        listen = threading.Thread( target=listener_thread, args=(self,
            logger, fado_args))
        listen.start()

        self.alg_type.simulate_rounds(self, logger, fado_args)
```

As shown in the Listing 4.2, the server initiates the global model first iteration using the `ModelManager.get_model()` function, which creates a model initialized with random values based on the selected options in the configuration file. After initializing the model, the server launches the listener thread, which is responsible for handling incoming client connections during training rounds. This multi threading approach allows the server to efficiently manage communications with multiple clients at the same time.

Once the server is configured and initialized, it begins the experiment by running the selected algorithm. This algorithm is composed by multiple sub components, TAs, AAs and PSAs.

Training Algorithms

The choice of training algorithm significantly impacts the behavior and scalability of the FL process. Each algorithm is designed to tackle specific challenges, such as handling straggler clients and reducing communication overhead. Within the FADO framework, we have implemented several training algorithms to emulate different scenarios. The selected implementations represent some of the most well known and established algorithms in the field, for synchronous and asynchronous approaches.

FedAvg: The FederatedAveraging algorithm, seen in the Listing 4.3, follows a traditional synchronous methodology for communication and training. In this approach, only the clients selected to participate in a given round contribute to the model aggregation of that round.

The FedAvg implementation operates with these steps:

1. `simulate_rounds()`: For loop iterates over the number of training rounds, invoking the `train_round()` function each iteration.

2. Training Round: The `train_round()` function is called and returns a boolean value, indicating if the round was successful or not. This will influence if the round contributions are discarded or added to the global model.

3. Aggregation Process: If the training round concludes as expected, the updates from participating clients are aggregated using the `aggregate()` function. Otherwise the `Queue` holding the client updates is cleared.

4. Round Completion: The current training round is considered finished if either the specified timeout is reached or the server receives the expected number of updates from the clients.

Listing 4.3: Implementation of the FedAvg training algorithm.

```
class FedAvg():
    def simulate_rounds(cls, flo, logger, fado_args):
        for flo.current_round in range(flo.current_round, fado_args.
            rounds + 1):
            discard_round = cls.train_round(flo, logger, fado_args)
            if not discard_round:
```

```
                cls.aggregate(flo)
                loss, accuracy = flo.global_model.evaluate(flo.
                    server_test)
                log_results()
            else:
                with flo.messages.mutex:
                    flo.messages.queue.clear()
                logger.info(f"Discarding round {flo.current_round} due
                    to timeout.")
        FedAvg.saveModelCKP(fado_args.exp_name, flo, flo.current_round)

    def train_round(flo, logger, fado_args):
        round_clients = flo.participant_selector.get_participants()

        for client_id in round_clients:
            subprocess.Popen(['python', client_path, str(client_id),
                str(fado_args.ssl), fado_args.ip, str(fado_args.port),
                fado_args.certificate_file])

        while (time.time() - tround) < tround_timeout and flo.messages.
            qsize() < fado_args.clients_per_round:
            pass

        return flo.messages.qsize() == fado_args.clients_per_round
```

FedAvg+30%: The FederatedAveraging algorithm, while effective in its synchronous approach, faced significant challenges due to the presence of *straggler clients*. In many scenarios, rounds were discarded because the established timeout for a round was reached by one or more clients. To address this issue, we implemented a variant of the FedAvg algorithm, FedAvg+30% on Listing 4.4, that retains the same approach but introduces an adjustment. Using a trade off between used resources by clients and wasted resources by the server, it launches an additional 30% of clients on top of the defined number of participants per round. This adjustment aims to reduce the number of discarded rounds.

A discarded round directly correlates with wasted computational work performed by the clients and the server. By increasing the number of clients per round by 30% as shown in [3], we establish the following formula 4.1 for acceptable percentage of *straggler clients* per round.

- N as number of clients per round
- N' as number of clients per round + 30%

$$DelayedPercentage = \frac{N' - N}{N'} \times 100 \tag{4.1}$$

$$DelayedPercentage \approx 23.08\%$$

The FedAvg+30% variant proves to be very impactful in scenarios where straggler clients are more prevalent, making **FedAvg** much more robust during the training process.

Listing 4.4: FedAvg variation with +30% clients per round.

```
class FedAvg30():
    def train_round(flo, logger, fado_args):
        ...
        round_clients = flo.participant_selector.get_participants(
            list(range(1, fado_args.number_clients + 1)),
            (int(fado_args.clients_per_round * 1.3)),
            flo.current_round
        )
        ...
```

FedAsync: The FedAsync algorithm on Listing 4.5 implements an asynchronous approach to FL, following the methodology in [38]. The key differences compared with the FedAvg approach are:

- There are no timeouts for discarding a round, i.e., when the `Scheduler`, `simulate_rounds()` reaches one of the defined parameters it starts a new round. This is done by using a while loop based on the time left.
- The `train_round()` function is non-blocking to the process, it iterates over the number of clients per round and launches them using the `subprocess.Popen()`.
- There is an extra thread called `updater()`, and it is responsible for aggregating new client updates to the server. While in **FedAvg** the aggregation occurs when all the client updates are collected, in **FedAsync** each client update is a single aggregation to the global model.
- The updater thread continuously checks for queued updates to perform aggregations.

This asynchronous approach enables the framework to handle updates from *straggler clients*, without discarding their contributions. If necessary, late updates are mitigated using a staleness function, ensuring that all client contributions are considered.

Listing 4.5: Implementation of FedAsync training algorithm.

```
class FedAsync():
    def simulate_rounds(cls, flo, logger, fado_args):
        updt = threading.Thread(target=FedAsync.updater, args=(flo,
            fado_args.rounds, logger, fado_args, gpu_lock))
        updt.start()
        for loop:
            train_round()
            num_updates = 0
            while tleft > 0 and num_updates < fado_args.
                clients_per_round:
                tleft -= time.time() - looptimer

            loss, accuracy = flo.global_model.evaluate(flo.server_test)
            log_results()
        FedAsync.saveModelCKP(fado_args.exp_name, flo, flo.
            current_round)
```

```
    def train_round(flo, logger, fado_args):
        round_clients = flo.participant_selector.get_participants()

        for client_id in round_clients:
            subprocess.Popen(['python', client_path, str(client_id),
               str(fado_args.ssl), fado_args.ip, str(fado_args.port),
               fado_args.certificate_file])

    def updater(flo, rounds, logger, fado_args, gpu_lock):
        while (flo.current_round <= rounds) and not last_round_check:
            if flo.messages.qsize() > 0:
                up = flo.messages.get()
                with gpu_lock:
                    flo.aggregator.aggregate(flo.global_model, up.
                       model_parameters, flo.current_round, up.
                       round_number)
```

FedBuff: The FedBuff algorithm implementation can be seen in Listing 4.6. It implements a buffered asynchronous approach as described in [27], which addresses the challenges of straggler clients. This algorithm utilizes a buffer to collect updates from multiple clients before performing aggregation. By aggregating updates in groups, FedBuff controls the frequency of updates to the global model and makes itself compatible to other privacy implementations. This offers multiple advantages comparing with other approaches:

- Compatibility with Secure Aggregation: Since updates are not processed individually, this method can introduce **Secure Aggregation** protocols [31], enhancing security during the aggregation phase.
- Contribution of Straggler Clients: Straggler clients can still contribute to the training process, as they can participate in group aggregations even if they arrive later in the training rounds.

A key distinction between FedBuff and other asynchronous approaches is the parameter:

- `fado_args.fb_updates_to_agg`: This threshold defines the number of asynchronous updates from clients that the buffer will hold before performing an aggregation.

This approach gives robustness to the asynchronous approach, because it makes it compatible with mitigating tools for relevant challenges in FL.

Listing 4.6: Implementation of FedBuff training algorithm.

```
class FedBuff():
    def simulate_rounds(cls, flo, logger, fado_args):
        for loop:
            train_round()

            num_updates = fado_args.clients_per_round / fado_args.
               fb_updates_to_agg
```

```
            curr_updates = 0
            while tleft > 0 and curr_updates < num_updates:
                looptimer = time.time()
                if flo.messages.qsize() >= fado_args.fb_updates_to_agg:
                    aggregate()
                    curr_updates += 1
                tleft -= time.time() - looptimer

            loss, accuracy = flo.global_model.evaluate(flo.server_test)
            log_results()

            FedBuff.saveModelCKP(fado_args.exp_name, flo, flo.
               current_round)
```

Aggregation Algorithms

FedAvgAggregator: The FedAvgAggregator listed on Listing 4.7 implements the FederatedAveraging algorithm, which groups client updates that were trained on the current training round locally on each client with their own dataset. The key points of this aggregator are:

- **Requirement of number of updates.** The aggregation in this algorithm only occurs if the number of updates is equal to the number of clients per round. This ensures that the contribution of a round properly weights the contribution of all the clients participating in it.
- **Weighted Average.** This aggregation uses the average weight of each update to calculate the contribution of each client for the impact it has on the global model. This means every client update is weighted equally. This is done using:

```
update_per_layer: torch.Tensor =
↪  weight_accumulator[name] * self.lr *
    ( 1 / fado_args.clients_per_round )
```

- **Client deltas.** This approach saves server side computation by requesting the client to calculate its delta locally, i.e., the client calculates difference between the global model it received on the round start, with the resulting trained model.

Listing 4.7: FedAvgAggregator class.

```
class FedAvgAggregator(Aggregator):
    def aggregate(self, global_model, received_deltas):
        weight_accumulator = dict()

        for delta in received_deltas:
            for key, value in delta.items():
                if key not in weight_accumulator:
                    weight_accumulator[key] = torch.zeros_like(value)
                weight_accumulator[key].add_(value.to(fado_args.device)
                    )
```

```
        for name, data in global_model.model.state_dict().items():
            update_per_layer: torch.Tensor = weight_accumulator[name] *
                self.lr * (
                1 / fado_args.clients_per_round
            )
            #update_per_layer = update_per_layer.type(data.dtype).clone
                ().detach()
            update_per_layer = update_per_layer.type(data.dtype)

            data.add_(update_per_layer.to(fado_args.device))
```

The aggregation process first creates a dictionary `weight_accumulator` with the client deltas, then it creates an `update_per_layer`, this is a tensor that will reflect the weight of each update equally. Finally, the `data.add_` function adds the `update_per_layer` to the global model in place.

FedAsyncAggregator: The FedAsyncAggregator implementation on Listing 4.8 enables an asynchronous aggregation of individual updates, which is critical for scenarios where client updates may arrive at different times. Key points include:

- **Individual Updates.** The aggregation occurs whenever the server receives a client update. This means that during a round there may be multiple independent aggregations, where the server does not need to wait for each client to finish their training and communicate the results.
- **Staleness function.** With the introduction of asynchronous approaches, the aggregator must be prepared to receive updates from older rounds. In order to combat the negative impact these updates may introduce and while still giving the possibility for the slower clients that provide these updates to contribute a staleness function is provided, with the factors `alpha, a, b, delay`. The function calculates an `alpha_factor` that adjusts the weight of incoming updates based on how old they are. Specifically, `delay` represents the difference between the current round and the round from which the client was launched. If the `delay` is within an acceptable range, the updates maintain the full weight. For older updates, the `alpha_factor` decreases proportionally to the delay, ensuring that the influence of updates is reduced over time. The final weight, `alpha_scaled`, is calculated by multiplying the `alpha` by the calculated `alpha_factor`. This scaled weight is then applied to the updates, ensuring that the contribution of slower clients is mitigated while still allowing their participation in the training process.
- **Global Model and Client Model Weights.** Finally, the `float_data. mul_((1 - alpha_ scaled))` is used to reduce the current weight of the global model parameters, so `float_data. add_(update_per_layer)`, the weighted tensor containing the update of the client can be added.

Listing 4.8: FedAsyncAggregator class.

```
class FedAsyncAggregator(Aggregator):
    def aggregate(self, global_model, received_parameters,
        current_round, update_round):
        alpha = fado_args.fa_alpha
        a = fado_args.fa_a
        b = fado_args.fa_b

        delay = current_round - update_round

        if delay <= b:
            alpha_factor = 1
        else:
            alpha_factor = 1 / ((delay - b) * a + 1)

        alpha_scaled = alpha * alpha_factor

        weight_accumulator = dict()

        for key, value in received_parameters.items():
            if key not in weight_accumulator:
                weight_accumulator[key] = torch.zeros_like(value)
            weight_accumulator[key].add_(value.to(fado_args.device))

        for name, data in global_model.model.state_dict().items():
            data = data.to(fado_args.device)

            float_data = data.float()

            update_per_layer: torch.Tensor = weight_accumulator[name] *
                alpha_scaled
            update_per_layer = update_per_layer.type(float_data.dtype)

            float_data.mul_((1 - alpha_scaled))
            float_data.add_(update_per_layer.to(fado_args.device))

            data.copy_(float_data.to(data.dtype))
```

FedBuffAggregator: Listing 4.9 represents the implementation the FedBuffAggregator, which utilizes a asynchronous buffer approach to manage asynchronous updates from clients. Its key points are:

- **Buffer threshold for group aggregation.** The number of client deltas used for each aggregation is based on the buffer threshold. Thus, when this threshold is reached an aggregation is triggered. This process can occur at anytime during the experiment, because it is done asynchronously.
- **Weighted Sum.** Unlike other aggregation implementations that usually use a weighted average of the client contributions, the FedBuffAggregator uses a weighted sum. This method allows for more control over each contribution because there may be a need to be mitigate updates with different weights applied to each. To accomplish this, we have a auxiliary

method `_aggregate()` that prepares the dictionary `weight_accumulator` with the weighted updates, thus giving the proper contribution for each update based on the rounds delay.

- **Staleness function.** The staleness function activates if an update is older than the defined threshold. The formula 4.2 represents how the weight is calculated:

$$weight = \frac{1}{(fado_args.fb_updates_to_agg \times delay)} \tag{4.2}$$

Listing 4.9: FedBuffAggregator class.

```
class FedBuffAggregator(Aggregator):
    def aggregate(self, global_model, weight_accumulator):

        for name, data in global_model.model.state_dict().items():
            update_per_layer: torch.Tensor = weight_accumulator[name] *
                self.lr
            update_per_layer = update_per_layer.type(data.dtype)

            data.add_(update_per_layer.to(fado_args.device))

    def _aggregate(flo, logger, fado_args):
        client_ids = []
        weight_accumulator = dict()

        for _ in range(fado_args.fb_updates_to_agg):
            up = flo.messages.get()

            delay = (flo.current_round - up.round_number)
            max_staleness = fado_args.fb_staleness_threshold

            if delay <= max_staleness:
                if delay > 0:
                    weight = 1 / ( fado_args.fb_updates_to_agg * delay)
                else:
                    weight = 1 / fado_args.fb_updates_to_agg

                FedBuff._calculate_update_with_weight(up.update_params,
                    weight, fado_args, weight_accumulator)
                client_ids.append(up.client_id)

        flo.aggregator.aggregate(flo.global_model, weight_accumulator)

        return client_ids

    def _calculate_update_with_weight(update, weight, fado_args,
        weight_accumulator):
        for key, value in update.items():
            if key not in weight_accumulator:
                weight_accumulator[key] = torch.zeros_like(value)
            update_per_layer: torch.Tensor = value.to(fado_args.device)
                * weight
            update_per_layer = update_per_layer.type(weight_accumulator
                [key].dtype)
```

```
            weight_accumulator[key].add_(update_per_layer.to(fado_args.
                device))
```

The Table 4.1 represents a summary of the various algorithm implementations presented in the sections above.

Table 4.1: Comparison of Federated Learning Algorithms in FADO

Algorithm	Communication	Handling Stragglers	Aggregation Method
FedAvg	Synchronous	Rounds discarded due to timeout, all clients must respond	Aggregates once the number of client updates per round is received
FedAvg+30%	Synchronous	30% additional clients launched to reduce discarded rounds	Equal to FedAvg, discards excessive updates
FedAsync	Asynchronous	No discarded rounds, late updates are aggregated with staleness function applied	Individual updates upon receiving
FedBuff	Buffered Asynchronous	Allows updates from stragglers by applying a staleness function to each update individually and using weighted sum of the updates	Aggregates updates in groups based on the buffer size

4.3.3 Client Management and Protocols

Connection Management: The server network is configured to handle multiple simultaneous client connections, aligning with the FL round structure. For each client connection, a separate thread, `fl_client_handler.py`, is created to manage communication independently, enabling the framework to support asynchronous approaches by maintaining active connections or terminating them based on synchronous timeouts.

```
client_thread = threading.Thread(target=client_handler,
 ↪  args=(client_socket,
flo, logger, fado_args))
client_thread.start()
```

With concurrent connections, each client has a timeout threshold during the connection establishment phase to prevent bottlenecks, thus avoiding denial of service for other clients. This is done using `listener_socket.settimeout(5.0)`, a five second timeout.

This approach optimizes client emulation capacity by preventing client processes from consuming continuous resource like GPU memory. Instead, once a client completes training, its process terminates, closing the handler thread on the server. This enables high client emulation while minimizing the utilization of server resources, as well as the machine running the experiment.

Protocols: Secure connections are supported through SSL encryption, an option that is configurable in the configuration file. Enabling SSL requires the clients to have the server certificate file, and the server to have the corresponding key to authenticate itself with the client.

```
if fado_args.ssl:
    ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    ssl_context.load_cert_chain(fado_args.certificate_file,
    ↪  fado_args.key_file)
```

On the client side:

```
if sslstate:
    ssl_context = ssl.create_default_context()

    ↪  ssl_context.load_verify_locations(cafile=certificate_file)
    client_socket = ssl_context.wrap_socket(client_socket,
    ↪  server_hostname=ip)
```

Figure 4.2 represents a diagram of the connections between the clients and the server.

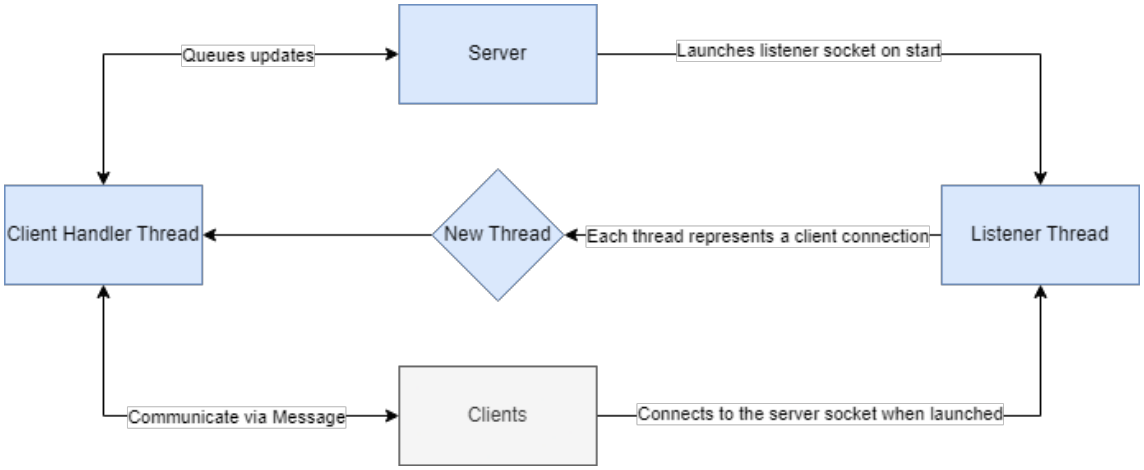


Figure 4.2: Clients connecting to the server diagram.

The communication between the client and the server utilizes a structured message format defined within the `Message` class, which includes all parameters required for training and feedback exchanges, as detailed in the Figure 4.3.

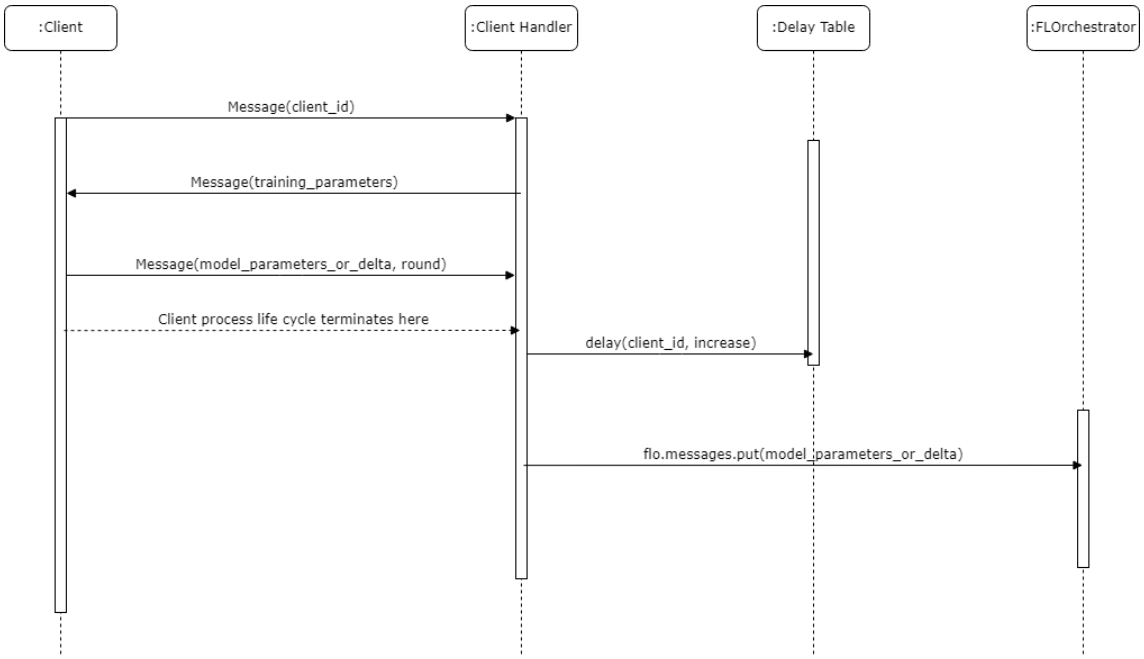


Figure 4.3: Exchanged messages between a client and the server during the client life cycle.

Client Handler: The `client_handler` function is called for each client connection accepted by the server. This function is called in an individual thread, making it responsible for managing the connection with the client, handling the communication, and receiving the training feedback. The implementation of this function below shows that a structured Message is sent to the client with the training parameters for the client local training. The client responds in a similar way using a structured Message response where only some of the fields are used sending the model parameters or the calculated delta depending on the algorithm.

Another impacting role of the `client_handler` is its individual contact with each client. It has the ability to emulate network delays, or simulate straggler devices, which proves crucial for the goals of the framework. Thus, allowing the emulation of different scenarios for different network conditions, or various levels of stragglers in the FL process.

In contrast with the client workflow previously presented in Listing 4.1, the Listing 4.10 below illustrates the client handler thread of the server workflow for each client.

- The handler starts by receiving the client identifier, and preparing the Message containing the training parameters to send to the client.

- After receiving the feedback from the client, the handler, based on the chosen configurations applies network delays to the messages.

- At the end of this process, if no errors occurred the handler adds the message update to the `Queue`.

Listing 4.10: Client Handler class.

```
def client_handler(client_socket, flo, logger, fado_args):

    client_id = Message.receive(client_socket).client_id

    if fado_args.algorithm_type == "fedasync":
        message = Message(model_parameters=flo.global_model.
            get_parameters(), modelp_or_update=True, model=fado_args.
            model, batch_size=fado_args.batch_size,
                      test_batch_size=fado_args.test_batch_size, epochs
                          =fado_args.epochs, learning_rate=fado_args.
                          learning_rate,
                      momentum=fado_args.momentum, weight_decay=
                          fado_args.weight_decay, round_number=flo.
                          current_round, use_gpu=fado_args.use_gpu,
                          device=fado_args.device)
    else:
        message = Message(model_parameters=flo.global_model.
            get_parameters(), modelp_or_update=False, model=fado_args.
            model, batch_size=fado_args.batch_size,
                      test_batch_size=fado_args.test_batch_size, epochs
                          =fado_args.epochs, learning_rate=fado_args.
                          learning_rate,
                      momentum=fado_args.momentum, weight_decay=
                          fado_args.weight_decay, round_number=flo.
                          current_round, use_gpu=fado_args.use_gpu,
                          device=fado_args.device)

    Message.send(client_socket, message)

    after_train_message = Message.receive(client_socket)

    # EMULATE NETWORK DELAYS
    if fado_args.delay:
        time.sleep(flo.delay_table.delay(after_train_message.client_id,
            fado_args.increase))

    # DISCARD LATE UPDATES FROM DISCARDED ROUNDS FOR SYNC ALGS
    if type(algorithm_type) == Sync():
        if after_train_message.round_number == flo.current_round:
            flo.messages.put(after_train_message)
    else:
        flo.messages.put(after_train_message)
```

Delay Table: In order to emulate the different network latencies that are close to real world scenarios, we implement a delay table that gets real world values when created. This table calculates the latency for any number of provided servers and splits the clients equally amongst these locations. The purpose of this implementation is to fulfill the goal of real world proximity.

To achieve this we use the ping statistics function by doing:

```
subprocess.run(['ping', '-c', '10', location_url],
↪  stdout=subprocess.PIPE,
stderr=subprocess.PIPE, text=True)
```

This process is repeated for each location and the result of this function is processed to create the following Table 4.2.

Table 4.2: Latency Statistics for each server location

Location	**Avg. Latency**	**Std. Deviation**
Madrid	73.933	1.012
Frankfurt	41.809	2.101
Tel Aviv	100.089	0.271
Sydney	284.961	0.819

Every time a `client_handler` calls the delay function it checks various parameters, such as:

- `random.normalvariate()`, using a normal distribution it gets the network latency delay to be applied based on the client assigned location.
- If the straggler option is available, and the client is defined as a straggler for that round, then there are two options:
  - `inc` use the defined fixed increase for a straggler
  - `delay_random` that is calculated using a number between one and one hundred seconds.
- Otherwise, if no additional delay is applied, the network latency is still simulated for each communication between the server and the clients, whenever the delay option is enabled. In such cases, the latency value is determined by a normal distribution based on the table values:

```
random.normalvariate(delay_values["rtt_avg"],
delay_values["rtt_stdev"]) * 0.001
```

The delay table is prepared to handle misconfiguration cases where the server locations may not be available during the table creation. In this case, the table will recalculate the number of available locations. The table latency results is saved into a `.json` file to allow experimentation replicability as well as using the same table for different training parameters, such as different percentage of stragglers.

4.4 Data Handling and Distribution

Federated Learning can lead to different scenarios related to Data Heterogeneity. Therefore it is relevant for the framework to have the ability to support different dataset distributions for the client.

To achieve this, we have a component that fully prepares and processes the dataset for each client. We utilize the CIFAR-10 dataset as an example, where we distribute 50.000 examples amongst the number of clients participating in the training of the global model, and 10.000 for the evaluation of the global model.

To be able to emulate different scenarios were clients have different distributions of data we employ a `Sampler` using a *Dirichlet's* distribution.

```
np.random.dirichlet(np.array(no_participants * [alpha])) *
↪  class_size
```

The framework is prepared to have scenarios with both IID and non-IID data distributions. The sampling with the *Dirichlet's* distribution allows the control of the allocation of examples to the clients, with the `alpha` variable being that adaptive factor. Lower values of alpha will provide non-IID distributions, some of these giving clients zero examples. To counter this fact and to ensure each client has at least one example for the correct functioning of the FL process we add examples of the remaining train dataset to the clients without examples. This process is seen in Listing 4.11.

Listing 4.11: Shaper class.

```
    ...
    unallocated_participants = [p for p in range(no_participants) if
        len(per_participant_list[p]) == 0]

    for class_indices in classes.values():
        remaining_data.extend(class_indices)

        if not remaining_data:
            for participant, data in per_participant_list.items():
                if len(data) > 1:  # If a participant has more than one
                    data point
                    remaining_data.append(data.pop())  # Take one and
                        put it into remaining_data

        for p in unallocated_participants:
            if remaining_data:
                per_participant_list[p].append(remaining_data.pop())
```

4.5 Challenges during Implementation

During the development of this framework, we encountered several challenges related to scalability and the realistic emulation of FL communication scenarios. Here, we outline some of the obstacles and the solutions implemented to address them.

- **FL with large number of clients.**
  - **Challenge -** Emulating a FL scenario with potentially millions of clients, each requiring a separate CUDA context, posed a significant challenge on setups using only one machine due to the limited GPU memory. The ideal setup would be able to closely achieve real world conditions where each client is treated as an independent entity without shared computational contexts and resources. However, a single machine emulation of this scale would quickly use all the available GPU memory.
  - **Solution -** To tackle this limitation, we implement a system that manages client processes dynamically, maintaining alive only those actively participating in the current round or clients still completing local training from prior rounds(*straggler clients*). This approach reduces memory usage and allows the framework to handle large client scenarios more efficiently.
- **Simultaneous client communication with the server.**
  - **Challenge -** During the start of each FL round, the server has to handle multiple connections and communications with the clients so they can receive the global model and the parameters for local training, while also having to handle both synchronous and asynchronous approaches to these rounds.
  - **Solution -** To solve this challenge, we ensure that the server creates a thread for each client connection, thus handling multiple clients at the same time, as required for every round start. We also ensure we do not maintain needless connection threads active by having timeouts, which cause the termination of the thread to free up resources.

This system structure and resource management implementation enables the framework to support large scale FL scenarios and experiments, while minimizing the used resources.

Chapter 5

Evaluation

The goal of this chapter is to test and evaluate the capabilities of the FADO framework and the implemented algorithms, FedAvg, FedAvg+30%, FedAsync and FedBuff. FADO framework calculates a number of metrics during the FL workflow, mainly time per round and accuracy.

The experiments were done on a single machine, the Navigators HORTA GPU server, running Ubuntu 22.04.4 LTS. The experiments utilized one of three GPUs, NVIDIA L40S, NVIDIA RTX A6000, or NVIDIA A30. The server CPU was an AMD EPYC 7443 24-Core Processor. Since the GPUs were shared and used by multiple users, GPU utilization may have varied during each experiment, directly affecting the response time in round duration.

To guide the experiments, the following questions were defined:

1. Can asynchronous algorithms achieve comparable results to synchronous baselines?

2. Under what conditions can asynchronous algorithms outperform synchronous baselines in terms of accuracy?

3. How resilient are synchronous algorithms in the presence of straggler clients?

To address these questions, two categories of experiments were defined: data heterogeneity and device heterogeneity. Each described in detail in the next sections.

5.1 Dataset and Model

FADO relies on PyTorch as its primary ML library. The implemented ResNet model, designed for efficient image classification tasks, uses the torch.nn library to define and create its layers and structure. This model shows to be a good choice for experiments using the CIFAR-10 dataset.

The CIFAR-10 dataset consists of 60000 32x32 colour images, distributed across 10 distinct classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is publicly available at `https://www.cs.toronto.edu/~kriz/cifar.html`.

5.2 Dataset Distribution Experiments

As shown in previous sections, non-IID data is one of the major challenges in FL. Different clients may hold highly imbalanced data distributions, making it challenging to train a global model out of diverse local datasets. To emulate varying degrees of non-IID distributions, the data distribution step makes use of a Dirichlet distribution to partition the dataset examples through the various clients. This variation is controlled through different values for the α parameter. Lower values of α create a non-IID distribution for the training data among the 1000 clients. Whereas higher values will create scenarios closer to IID conditions. We can observe the resulting distributions in the Table 5.1.

Table 5.1: Data Dirichlet distribution statistics for the different values of α

Data Distribution			
Alpha α	**α = 0.1**	**α = 0.9**	**α = 10**
Mean nº of samples	49	50	50
Standard Deviation	48.5	16.5	5.3
Min. nº of samples	1	14	14
Max. nº of samples	332	115	67
Median	35	48	50
Total	49826	49853	49986

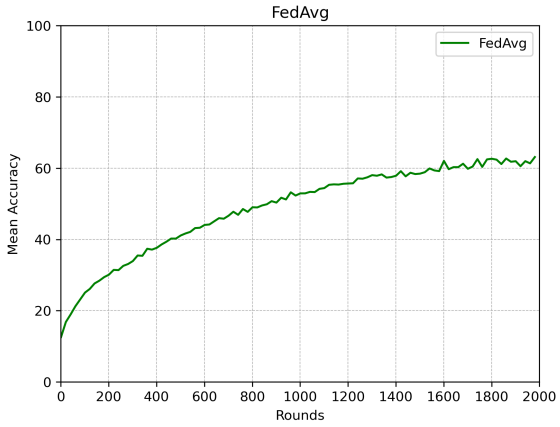
5.2.1 Compared Algorithms

Three algorithms were selected for comparison in these experiments. **FedAvg** represents the synchronous baseline where all client updates are aggregated at the end of the round. **FedAsync** is an asynchronous algorithm that allows clients to contribute their updates when they finish training, with the parameter fa_α controlling the weight assigned to each client update during aggregation. In these experiments, fa_α weights of 0.1, 0.5, and 0.9 were compared. **FedBuff** is another asynchronous algorithm which waits for a buffer of n clients before aggregating. The chosen values for n were 5 and 10.

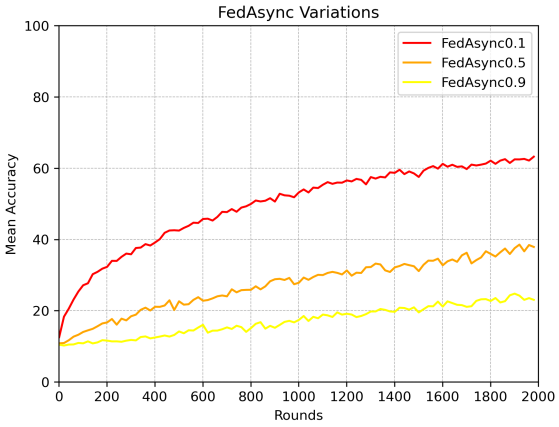
5.2.2 Evaluation Metrics

The primary metric for evaluation is accuracy over rounds. This reflects how well each algorithm trains the model across aggregations. Model convergence speed, defined as the rate at which the accuracy increases, is also observed through the plotted curves. In addition, the stability of each algorithm can be visually assessed by observing the smoothness and variability of the plotted curves.

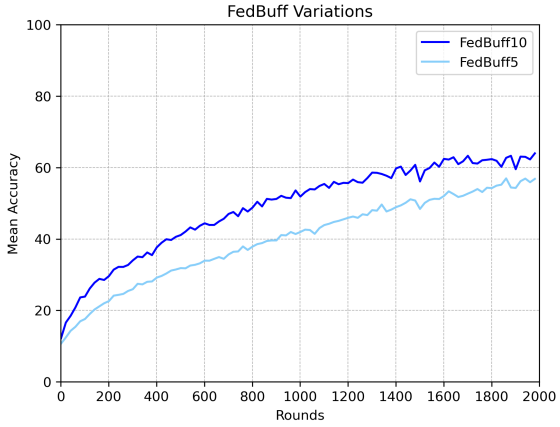
5.3 Dataset Distribution non-IID (α = 0.1)



(a) FedAvg



(b) FedAsync



(c) FedBuff

Figure 5.1: Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a non-IID setting ($\alpha = 0.1$).

This experiment evaluates the performance of the selected algorithms under conditions of extreme data heterogeneity. The goal is to assess how non-IID dataset distributions affect the stability and convergence speed of synchronous, asynchronous, and buffered asynchronous aggregation algorithms. The training data was distributed across 1000 clients using a Dirichlet distribution with $\alpha = 0.1$, resulting in a highly unbalanced number of training examples per client. As shown earlier in Table 5.1, the number of examples per client ranged between 1 and 332, with a standard deviation of 48.5 and a median of 35.

In this scenario, **FedAsync0.1** consistently achieved higher accuracy than the other two FedAsync configurations, showing both a faster initial rise and stronger final convergence. FedBuff with $n = 10$ outperformed the $n = 5$ configuration over 2000 rounds, suggesting that increasing the buffer size is a better option to grasp the data because it avoids smaller aggregations with disruptive contributions, even if it does not necessarily accelerate convergence in highly heterogeneous environments.

Figure 5.2 shows that both **FedAsync0.1** and **FedBuff10** performed close to the **FedAvg Baseline**. This indicates that asynchronous methods can remain competitive with the synchronous baseline, even under severe non-IID conditions.

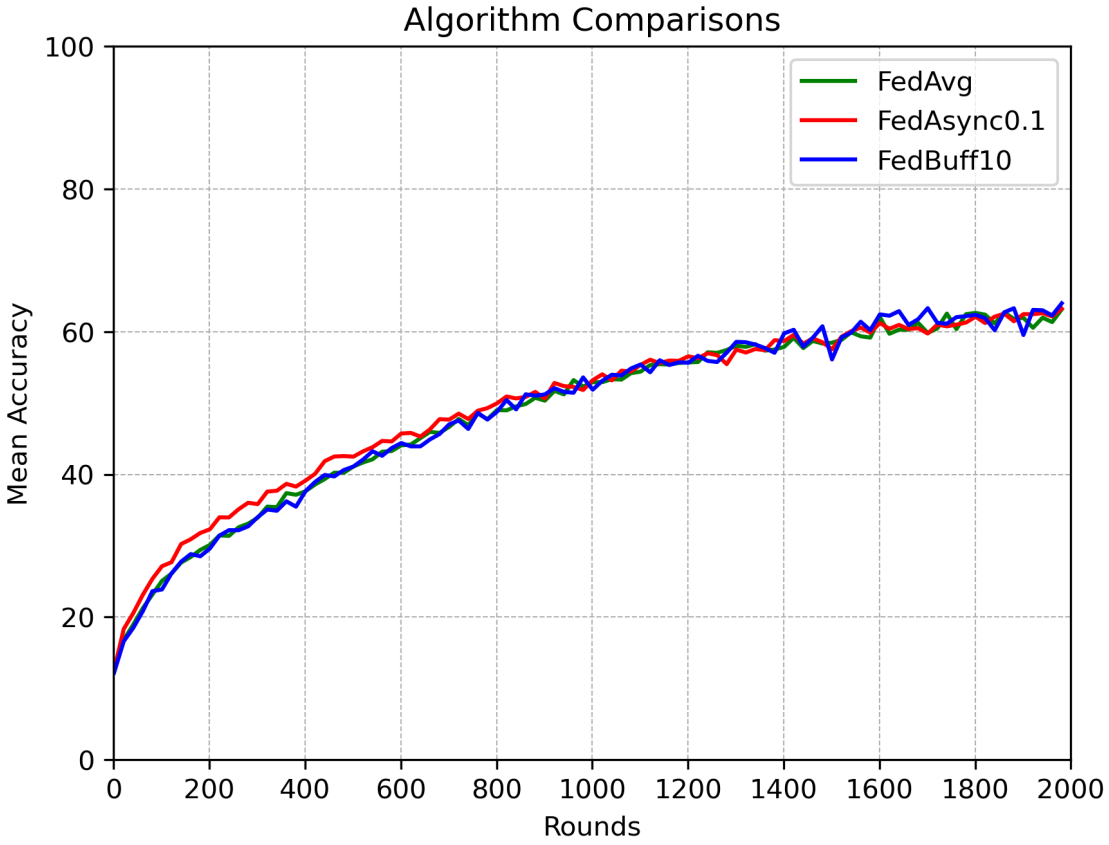
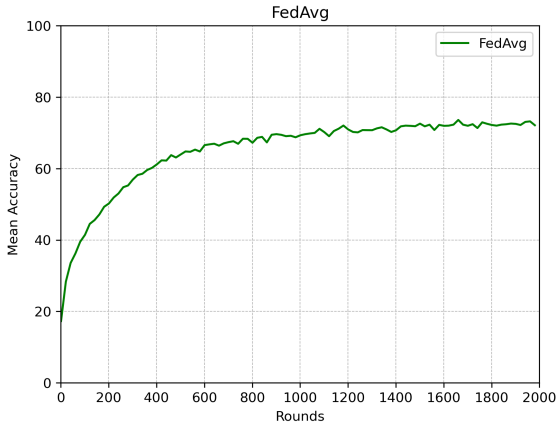


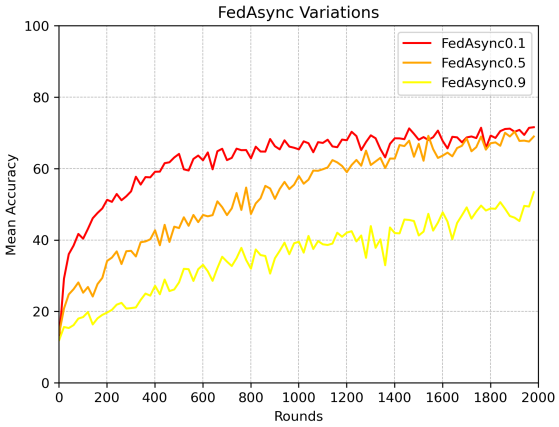
Figure 5.2: Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 0.1$.

Overall, the results indicate that under conditions of extreme data heterogeneity, the three algorithms achieve comparable performance, with only marginal differences between their best configurations. While **FedAsync0.1** showed slightly faster convergence in early rounds, all methods eventually approached similar accuracy levels, suggesting that in non-IID dataset distribution environments, there is no clear winner. The next experiment increases the Dirichlet parameter to $\alpha = 0.9$, to explore whether these trends persist in moderately heterogeneous client data distributions.

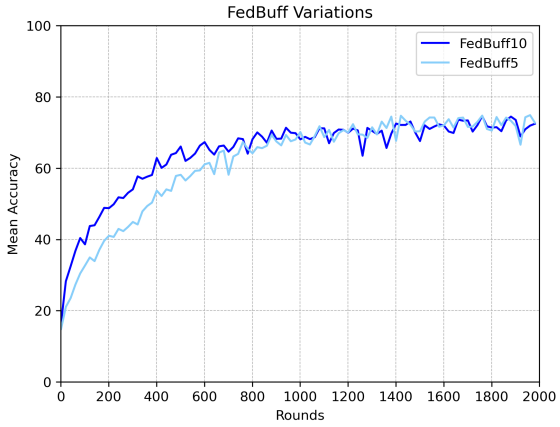
5.4 Dataset Distribution Average (α = 0.9)



(a) FedAvg



(b) FedAsync



(c) FedBuff

Figure 5.3: Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a avg-IID setting ($\alpha = 0.9$).

This experiment evaluates the performance of the selected algorithms under conditions of average data heterogeneity. The goal is to assess whether the performance trends observed in the previous experiment persist when the training data is distributed more evenly among clients. With $\alpha = 0.9$, the Dirichlet distribution still introduces some disparity in the data distribution, but with significantly lower skew compared to $\alpha = 0.1$. As shown in Table 5.1, the standard deviation of client data samples drops to 16.5, with most clients holding between 14 and 115 examples.

In this scenario, **FedAvg** showed improved performance in the early rounds compared to the previous experiment, benefiting from the more balanced data distribution among clients. In the FedAsync configurations, $fa_\alpha = 0.1$ outperformed the other variants in early convergence, while only **FedAsync0.5** managed to catch up in final convergence. **FedAsync0.9** averaged around 51% final accuracy, significantly less than the other two. For FedBuff, the gap between $n = 5$ and $n = 10$ was reduced compared to the previous experiment, suggesting that the impact of buffer size is reduced when client data distributions are more uniform.

Figure 5.4 presents the best performing configurations for each algorithm. As seen, all three methods converge closely, with only minor variations.

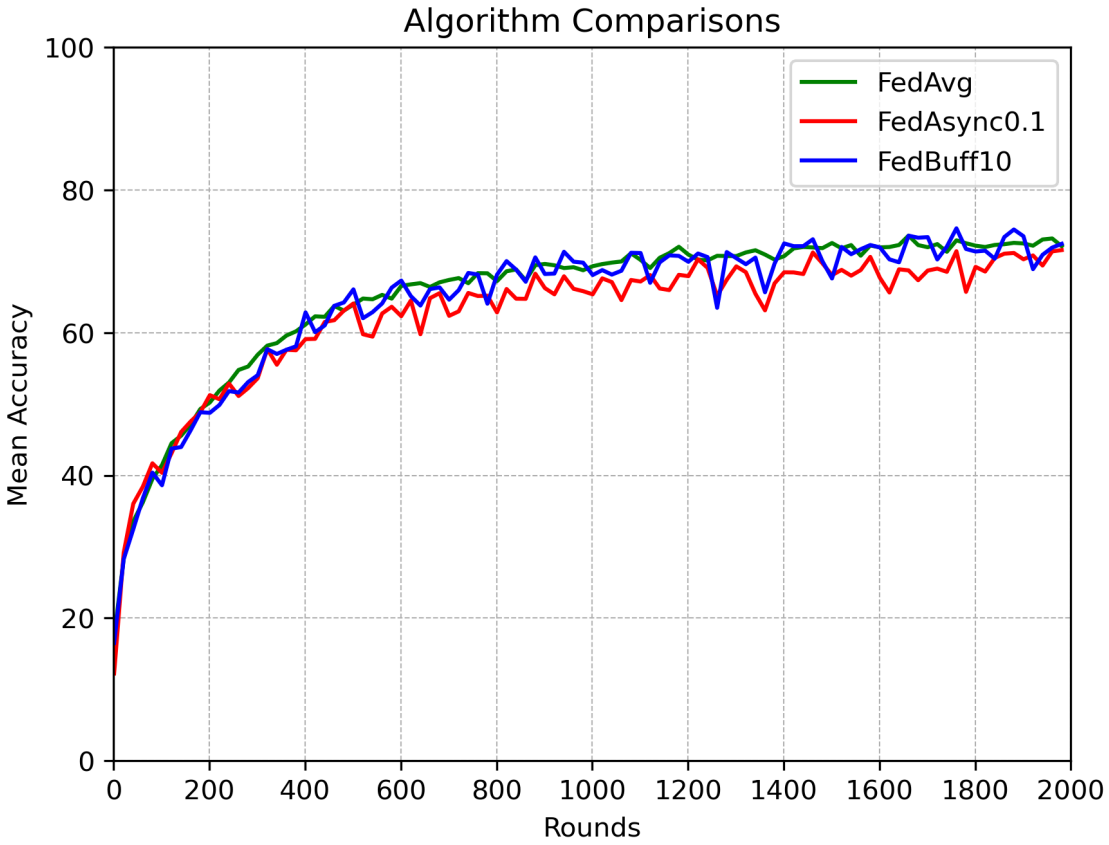
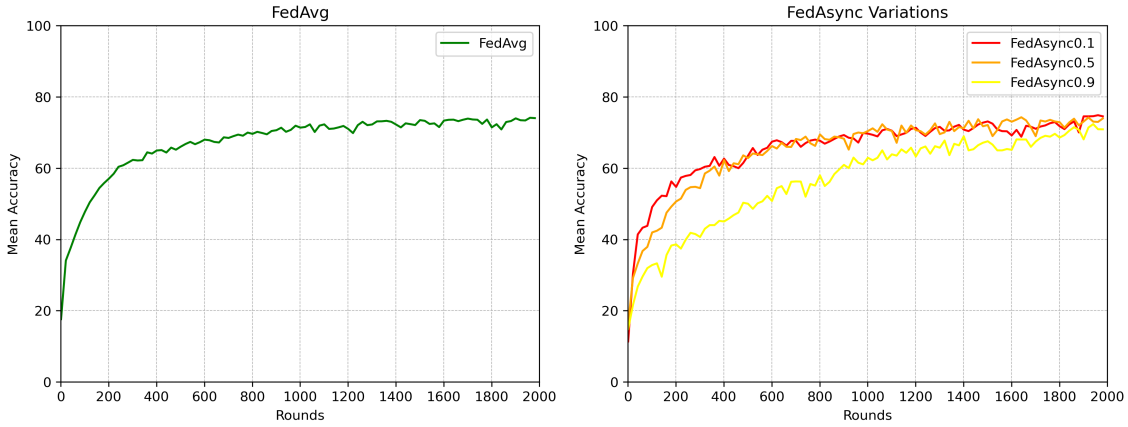


Figure 5.4: Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 0.9$.

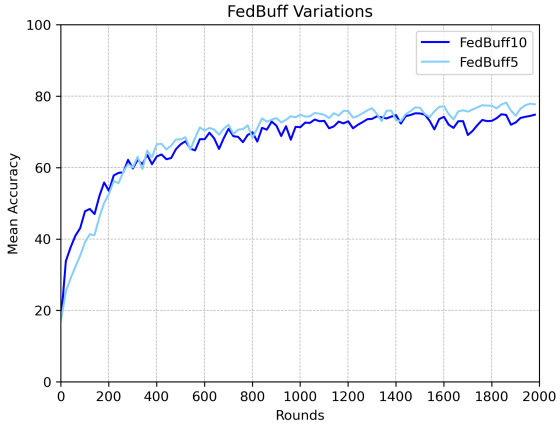
Overall, these results show that as client data becomes more balanced, the initial performance of all algorithms converges faster, and in the end reaches higher final accuracy values. The difference between algorithms in convergence speed is similar to the previous experiment as the three best performing algorithms follow a similar curve, showing there is no clear advantage or disadvantage for asynchronous or buffered methods as they stay close to the synchronous baseline. The next experiment further increases the Dirichlet parameter, to $\alpha = 10$, approaching a fully IID scenario.

5.5 Dataset Distribution IID (α = 10)



(a) FedAvg

(b) FedAsync



(c) FedBuff

Figure 5.5: Comparison of FedAvg, FedAsync ($fa_\alpha \in \{0.1, 0.5, 0.9\}$), and FedBuff ($n = 5, 10$) in a IID setting ($\alpha = 10$).

This experiment evaluates the selected algorithms under conditions of close to a fully IID data distribution. The goal is to assess how each algorithm performs when all clients receive a uniform subset of the training data. With $\alpha = 10$, the Dirichlet distribution produces nearly identical data partitions across the 1000 clients. As shown in Table 5.1, the standard deviation drops to just 5.3, and both the median and mean number of samples per client are approximately equal, indicating a near perfect IID scenario.

In this scenario, we can see a much faster initial convergence than what we saw in the previous two experiments. Among the FedAsync variants, **FedAsync0.1** stood out once more as the best performing one in the initial rounds. **FedAsync0.5** improved a lot and managed to keep a similar pace with the top performers, though **FedAsync0.1** showed a slightly more stable curve throughout. **FedAsync0.9**, which gives greater weight to the latest client update, started slower but achieved similar final accuracy values. FedBuff results revealed minimal differences between $n = 5$ and $n = 10$, supporting the idea that in uniformly distributed environments, the choice of

buffer size becomes less impactful on overall accuracy and convergence behavior.

Figure 5.6 presents the best performing configurations of each algorithm. Surprisingly, **FedBuff5**, despite its slower start, surpassed both the synchronous baseline and FedAsync, maintaining higher accuracy from approximately round 400.

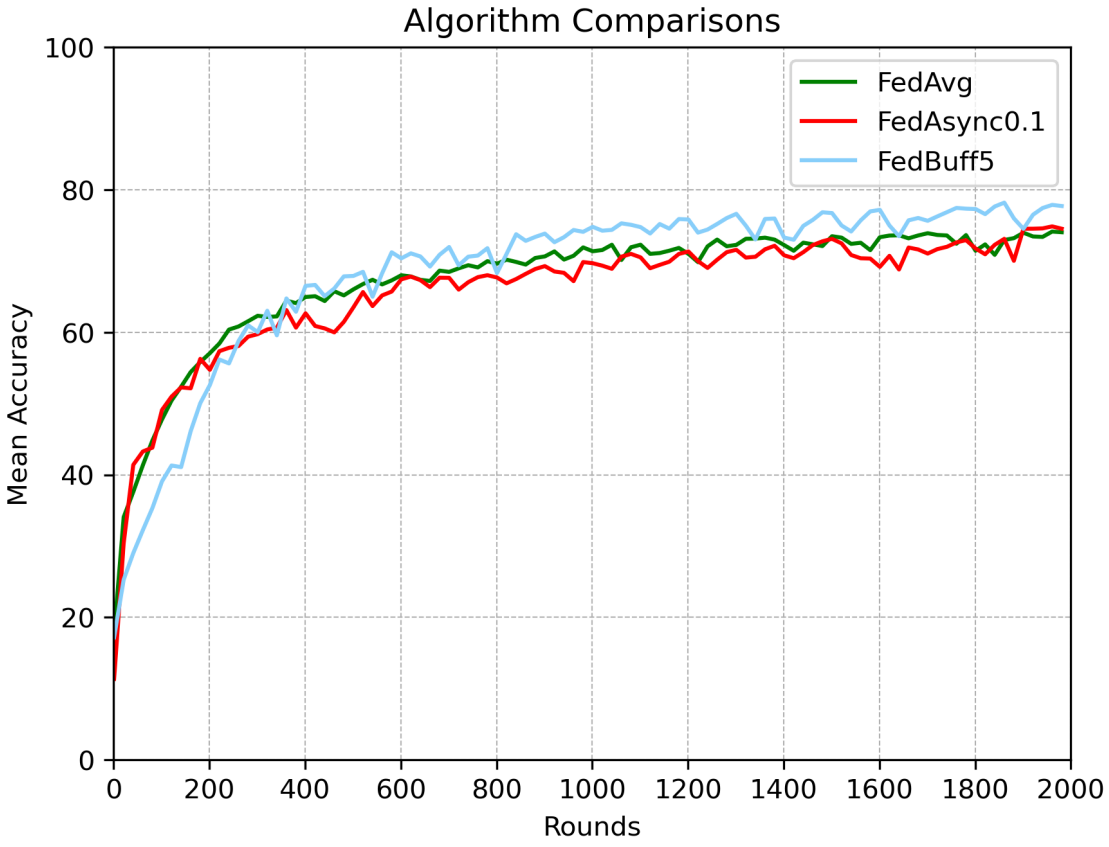


Figure 5.6: Best performing configurations of FedAvg, FedAsync, and FedBuff under $\alpha = 10$.

Overall, the results in this experiment confirm that under IID conditions, all algorithms perform similarly in terms of accuracy and convergence, with **FedBuff5** averaging two aggregations per round being the top performer. However, the advantage of asynchronous buffered strategies is not yet decisive, as the balance of data reduces the issues observed in earlier experiments. This experiment serves as a reference to show that understanding performance in ideal data scenarios is not enough. We have to introduce device heterogeneity factors such as delays and straggler behavior, as we show in the next experiment.

5.6 Device Heterogeneity with Network Delays and Stragglers

5.6.1 Setup and Context

This set of experiments introduces network level delays and emulated straggler behavior to assess the resilience and performance of asynchronous algorithms in contrast with the synchronous

baseline, and how they handle device heterogeneity in a more realistic FL environment. While the previous experiments focused solely on data heterogeneity, here we explore how algorithms behave when faced with varying communication latencies and delayed client updates.

All experiments in this section use a configuration of ten clients per round using the dataset distribution described in Table 5.1 with $\alpha = 0.9$. Each client is assigned a randomly selected from up to five possible server locations. Communication delays were measured in real time at the start of the experiment between each client location and the central server, in this case our machine HORTA, located in Lisbon. Table 5.2 shows the average latency and standard deviation per available location.

Table 5.2: Average communication delays and standard deviations for each client location

Location	**Avg. Latency**	**Std. Deviation**
Madrid	73.933	1.012
Frankfurt	41.809	2.101
Tel Aviv	100.089	0.271
Sydney	284.961	0.819

In addition to communication delays, straggler behavior was emulated to add delayed client responses. In any given round, each participating client had a defined probability of behaving as a straggler. When selected as a straggler, the client processing time was increased by a fixed or variable time value.

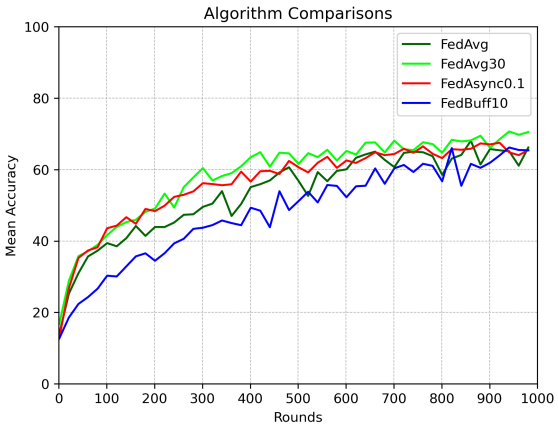
The following configurations were tested:

- 5% chance of being a straggler, with a fixed induced delay of 20s, see Figure 5.7a.
- 15% chance of being a straggler, with a fixed induced delay of 20s, see Figure 5.7b
- 30% chance of being a straggler, with a fixed induced delay of 20s, see Figure 5.7c
- 15% chance of being a straggler, with a fixed induced delay of 100s, see Figure 5.8a
- 15% chance of being a straggler, with a *variable* delay between 1s and 100s using a uniform distribution, see Figure 5.8b

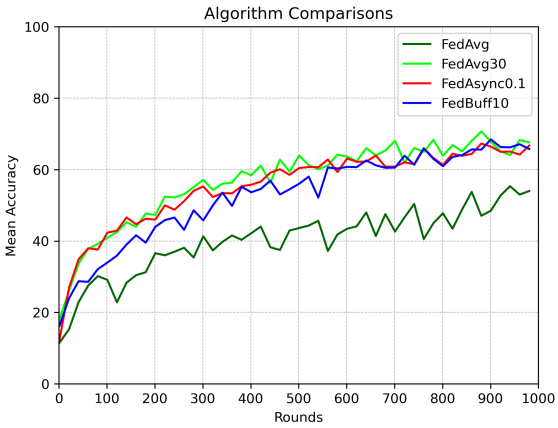
The algorithms compared in all experiments were:

- **FedAvg** – The standard synchronous baseline algorithm.
- **FedAvg30** – FedAvg with an additional 30% tolerance window, requiring 10 out of 13 clients per round.
- **FedAsync0.1** – Asynchronous individual aggregations with the weight factor at $fa_\alpha = 0.1$.
- **FedBuff10** – Buffered asynchronous aggregation with a buffer size of $n = 10$.

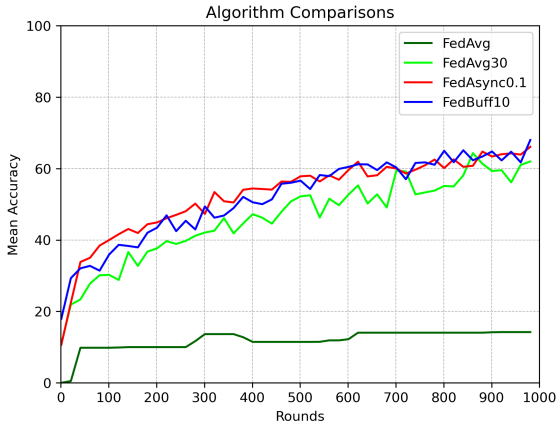
5.6.2 Different percentages of stragglers



(a) Straggler probability = 5%, Delay = 20s



(b) Straggler probability = 15%, Delay = 20s



(c) Straggler probability = 30%, Delay = 20s

Figure 5.7: Comparison of different straggler probabilities.

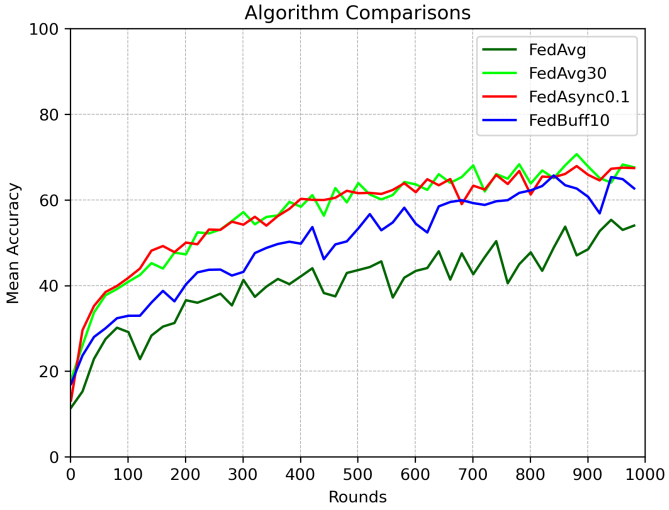
Figure 5.7 shows the evolution of the accuracy levels for increasing percentages of straggler behavior while maintaining a constant 20s delay.

In Figure 5.7a, corresponding to a 5% straggler rate, all the algorithms perform closely, with FedAvg30 coming out as a top performer due to the low straggler rates and additional computational resources used, since there are 13 clients per round competing for the 10 aggregation slots. The synchronous baseline also shows a slight decrease in curve consistency when compared to previous experiments. In the end, both asynchronous and synchronous algorithms manage to achieve a similar final accuracy.

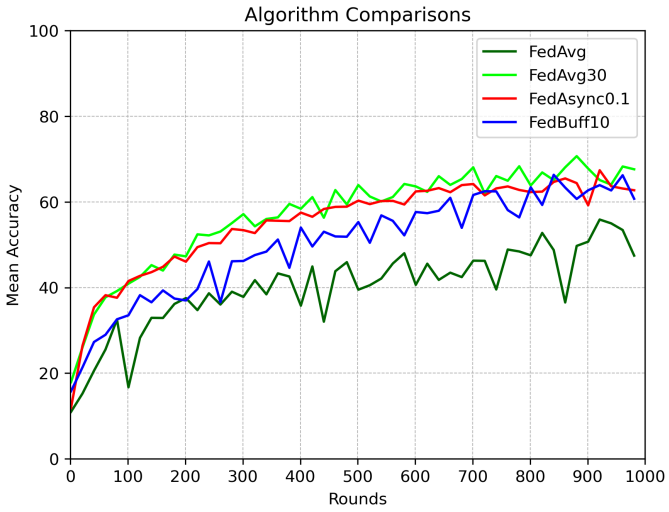
In Figure 5.7b, where the rate increases to 15%, the synchronous baseline, **FedAvg**, starts to experience a more apparent dropoff caused by discarded rounds due to its strict requirement to wait for all clients, which amplifies the straggler impact. In contrast, **FedAvg30** and the asynchronous options show better tolerance, maintaining somewhat stable convergence. **FedBuff10** remains competitive and benefits slightly from its buffered mechanism, though it does not outperform the other algorithms.

In Figure 5.7c, at a 30% straggler probability, the performance gap widens drastically. **FedAvg** is not able to achieve any progression due to the constant rate of discard rounds, showing how much adaptability FedAvg30 brings in comparison. Although it does not maintain the level of performance the asynchronous alternatives have, **FedAvg30** presents itself as a much simpler alternative, showing good levels of resistance of stragglers. **FedAsync0.1** and **FedBuff10** are the clear winners of this experiment, showing the resilience possible with asynchronous approaches, having much less discarded rounds and wasted resources.

5.6.3 Different straggler delays



(a) Straggler probability = 15%, Delay = 100s



(b) Straggler probability = 15%, Delay = 1–100s (uniform)

Figure 5.8: Comparison of different delays.

Figures 5.8a and 5.8b maintain a fixed 15% straggler rate while varying the delay magnitude—one fixed at 100 seconds, the other uniformly distributed between 1s and 100s.

In this experiment, both Graphs 5.8 look similar. In synchronous algorithms, if a straggler exceeds the round timeout, the entire round is discarded, leading to wasted computation resources. In the two asynchronous approaches, severely delayed updates, such as those delayed by up to 100 seconds, are received several rounds later and must pass through a mitigation function. This ensures that outdated updates do not poison the model while still allowing the client training effort to not be wasted. As in the previous experiments, **FedAvg30** was the top performer in achieving better convergence results.

These results demonstrate that device heterogeneity, in the form of latency and straggler behavior, can severely impact the performance of synchronous algorithms. Increasing the presence of stragglers reduces the effectiveness of synchronous baseline models such as FedAvg. Introducing the use of additional resources to mitigate this impact, like **FedAvg30** or adopting asynchronous paradigms, **FedAsync0.1** and **FedBuff10**, results in more robust convergence in most straggler scenarios.

Overall, the results validate the practical importance of frameworks such as FADO. It is important for us to study new and existing algorithms under different network conditions that may be relevant for real world scenarios.

Chapter 6

Conclusion

The thesis explored the challenges of federated learning under diverse dataset conditions and presented a modular framework to study and evaluate both synchronous and asynchronous aggregation algorithms. Based on the FADO framework, this work restructured its architecture to support asynchronous communication and realistic emulation of client server interactions, allowing experimentation with diverse local datasets and communication settings.

The implemented framework integrates both synchronous and asynchronous standards, as well as new emerging options. On the synchronous side, FedAvg was re implemented with the new architecture and served as the baseline. In order to compare the asynchronous additions with a synchronous option that aims to tackle similar issues, such as the occurrence of stragglers, FedAvg+30% was also implemented. On the asynchronous side, two approaches were added, FedAsync, which continuously updates the global model with individual contributions, and FedBuff, which uses buffered aggregation. The resulting framework supports a wide range of configurations, including network delays, straggler behaviors, and varying levels of data heterogeneity, offering a realistic and reproducible platform for future research.

The evaluation demonstrated that asynchronous algorithms achieve performance comparable to the synchronous baseline, while offering increased robustness under device heterogeneity experiments. In non-IID scenarios, the best configuration of each algorithm achieved very similar performance levels. Under IID conditions, FedBuff with a buffer size equal to half of the participating clients in a round consistently surpassed FedAvg and FedAsync, highlighting the benefits of buffered asynchronous aggregation. When we introduced experiments with network delays and straggler activity, FedAvg managed to remain competitive under balanced conditions, but was strongly impacted by stragglers, discarding entire rounds when one client would exceed the limit of a round. FedAvg+30% partially mitigated this issue at the cost of higher resource usage. This suggests that while simple adaptations can improve resilience, asynchronous alternatives provide strong long term adaptability to these issues.

6.1 Future Work

For future work, there are several research directions for FADO framework:

- **Distributed Deployment -** Extending the framework to multi machine clients would provide additional robustness to experiments and allow scalability.
- **Additional Dataset and Models -** Evaluating with diverse datasets and state of the art models that emulate possible real world scenarios.
- **Additional Privacy mechanisms -** Integrating security protocols such as secure aggregation and differential privacy techniques to reduce attack vectors and surfaces.
- **New FL methodologies -** Not excluding new implementations of algorithms, it would be very interesting to focuses on more areas of the FL process not just aggregation and communication methods. For example, client training or client selection process.

Bibliography

[1] Jane Andrew and Max Baker. The general data protection regulation in the age of surveillance capitalism. *Journal of Business Ethics*, 168:565–578, 2021.

[2] Mariusz Bojarski. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[3] Keith Bonawitz. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[4] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482*, 2016.

[5] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[6] Zheng Chai, Yujing Chen, Ali Anwar, Liang Zhao, Yue Cheng, and Huzefa Rangwala. Fedat: A high-performance and communication-efficient federated learning system with asynchronous tiers. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–16, 2021.

[7] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. Asynchronous online federated learning for edge devices with non-iid data. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 15–24. IEEE, 2020.

[8] Beongjun Choi, Jy-yong Sohn, Dong-Jun Han, and Jaekyun Moon. Communication-computation efficient secure aggregation for federated learning. *arXiv preprint arXiv:2012.05433*, 2020.

[9] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen, Cesare Alippi, and Gianluca Bontempi. Credit card fraud detection: a realistic modeling and a novel learning strategy. *IEEE transactions on neural networks and learning systems*, 29(8):3784–3797, 2017.

[10] 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.

[11] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[12] EU. GDPR Archives. https://gdpr.eu/tag/gdpr/, 2024.

[13] P. A. Flach. *Machine Learning - The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

[14] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.

[15] EU GDPR. General data protection regulation (gdpr), 2018.

[16] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[19] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and trends® in machine learning*, 14(1–2):1–210, 2021.

[20] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[21] Xingyu Li, Zhe Qu, Bo Tang, and Zhuo Lu. Stragglers are not disaster: A hybrid federated learning algorithm with delayed gradients. *arXiv preprint arXiv:2102.06329*, 2021.

[22] Bing Liu. *Sentiment analysis and opinion mining*. Springer Nature, 2022.

[23] Bo Liu, Ming Ding, Sina Shaham, Wenny Rahayu, Farhad Farokhi, and Zihuai Lin. When machine learning meets privacy: A survey and outlook. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.

[24] 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.

[25] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE symposium on security and privacy (SP)*, pages 691–706. IEEE, 2019.

[26] Jose G Moreno-Torres, Troy Raeder, Rocío Alaiz-Rodríguez, Nitesh V Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern recognition*, 45(1):521–530, 2012.

[27] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. Federated learning with buffered asynchronous aggregation. In *International Conference on Artificial Intelligence and Statistics*, pages 3581–3607. PMLR, 2022.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[29] Filipe Rodrigues, Rodrigo Simões, and Nuno Neves. Fado: A federated learning attack and defense orchestrator. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 141–148, 2023.

[30] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[31] Jinhyun So, Ramy E Ali, Başak Güler, and A Salman Avestimehr. Secure aggregation for buffered asynchronous federated learning. *arXiv preprint arXiv:2110.02177*, 2021.

[32] Financial Times. Facebook data breach. https://www.ft.com/content/87184c40-2cfe-11e8-9b4b-bc4b9f08f381, Mar 2018.

[33] Marten van Dijk, Nhuong V Nguyen, Toan N Nguyen, Lam M Nguyen, Quoc Tran-Dinh, and Phuong Ha Nguyen. Asynchronous federated learning with reduced number of rounds and with differential privacy from less aggregated gaussian noise. *arXiv preprint arXiv:2007.09208*, 2020.

[34] Sandra Wachter and Brent Mittelstadt. A right to reasonable inferences: re-thinking data protection law in the age of big data and ai. *Colum. Bus. L. Rev.*, page 494, 2019.

[35] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. Federated learning with differential privacy: Algorithms and

performance analysis. *IEEE transactions on information forensics and security*, 15:3454–3469, 2020.

[36] 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.

[37] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern recognition*, 90:119–133, 2019.

[38] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934*, 2019.

[39] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. *Advances in neural information processing systems*, 32, 2019.