POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Accelerating Federating Learning via In-Network Processing

Supervisors

Prof. Guido MARCHETTO

Prof. Alessio SACCO

Candidate

Vera ALTAMORE

April 2022

Abstract

The unceasing development of Machine Learning (ML) and the evolution of Deep Learning have revolutionized many application domains, ranging from natural language processing, to video analytics, to biology and medical predictions. The most common approach for ML models training is cloud-centric, so data owners transmit the training data to a public cloud server for processing, where resides more powerful resources. However, this approach is often unfeasible due to privacy laws and restrictions, as well as the burdening of network communications because of the massive quantities of data that need to be transmitted to a distant cloud server. To solve these problems, Google introduced in 2016 the concept of Federated Learning (FL) with the objective of building machine learning models that takes into account security and privacy of data. In FL, instead of transferring the data to the central servers, the ML model itself is deployed to the individual devices to train on the data, and only the parameters of the trained models are sent to the central ML/DL model for global training. Thanks to this principle, FL is widely used today in sales, financial, medical, and Internet of Things (IoT) fields, where the privacy of data is essential. In particular, the underlying architecture can include many devices, each one with its own dataset, and a central server, which is responsible for the aggregation of data in order to build a global model. However, despite the privacy and security benefits, this approach can lead to synchronization issues, and the network and the server turn in bottlenecks and the load may become unsustainable.

To this aim, this thesis proposes a novel FL model that uses programmable P4 switches to compute intermediate aggregations and reduce the traffic on the network. The use of edge nodes for in-network model caching and gradient aggregating alleviates the bottleneck effect of the central FL server and further accelerates the entire training progress. In detail, we modified a traditional FL framework, as Flower, to communicate with P4 switches using a custom protocol to carry the model parameters. We also adapted the P4 switch behavior to support gradient aggregation. In addition, in this work we compare the execution time of the proposed model against current state-of-the-art models and verify the speedup of the global training phase.

Acknowledgements

At the end of this path, I feel I have to thank many people, who even with their simple presence have made me what I am today.

First of all, a special thanks to Prof. Alessio Sacco, who in these months has followed my work with constancy and patience, and to Prof. Marchetto for giving me the opportunity to do this thesis work and for stimulating my curiosity for Networking, which then became my Master Degree path.

To my parents and my sister, who always supported me in my choices, who allowed me to be here today and showed me that it's never too late for life changing. To my father, who has always been close to me, and who has supported me in moments of discouragement. This thesis is dedicated to you.

Thanks to Ada with whom I lived thousand adventures even if within four walls.

And thank You, friend, companion, past, present and future. Thank you for always being by my side, and for the love you show me every day. Thank you for everything. Always.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI
: Artificial Intelligence

FL
: Federated Learning

ML
: Machine Learning

EC
: Edge Computing

ED
: Edge Device

PS
: Parameter Server

Chapter 1

Introduction

In recent years, the number of connected IoT devices has grown exponentially, and it is estimated that by 2025 it will reach the amount of 31 billion, which is two and half times the amount of data produced in 2020 [1]. Moreover, these devices present heterogeneous architectures and applications, ranging between smart industry, healthcare, smart homes, and wearables, all of which require quality and speed. The current cloud infrastructures are not able to provide services at the required level of performance while managing massive, heterogeneous, and distributed IoT data, therefore a new architecture is necessary.
Edge Computing (EC) is proposed as a solution to these problems: it is a new architecture that leads network services closer to data sources, reducing latency and bandwidth costs while improving network resilience and availability. IoT data proliferation and the abundance of heterogeneous computing resources have made EC a useful computing paradigm for handling IoT data. EC together with Deep Learning (DL) and Machine Learning (ML) is a promising technology and is widely applied in several areas. To mention a few examples, currently, existing smart speakers use ML on Edges nodes to perform the training of speech recognition and pattern recognition models. There are companies in the energy and industrial sectors that are using machine learning systems to monitor their components and alert technicians when maintenance is required, or monitor for emergencies like machine malfunctions and meltdowns. There are studies and applications of Edge ML-based systems in hospitals and assisted living facilities to monitor things like patient heart rate, glucose levels, and falls by using cameras and motion sensors. In this situation, a quick response is essential and could save lives: if the data is processed locally and at the edge, the staff is notified in real-time and act promptly. The most common approach for ML models training is cloud-centric, so data storage and model training are performed on high-performance cloud servers. But with the growth of model size and dataset, as well as the number of connected nodes, the centralized approach becomes unfeasible. The main problems encountered

concerns communication cost, reliability, data privacy, security, and administrative policies. When large amounts of data are sent from EC nodes or edge devices to a remote server, an appropriate network traffic encoding and transmission time are required, and insufficient bandwidth can negatively affect the efficiency of data transmission. Furthermore, cloud servers are often far from end-users, so in a network with thousands of edge devices is difficult to obtain real-time, low-latency, and high Quality of Service (QoS) requirements.

In the age of Big data, interest in privacy and security of data is increasing, to such an extent that governments and organizations have responded by enacting data privacy legislation. For example, the General Data Protection Regulation (GDPR) in the European Union, California Consumer Privacy Act (CCPA) in the USA, and the Personal Data Protection Act (PDPA) in Singapore have the aim to restrict the collection of data to only those that are needed for processing and consented to by consumers. Moreover, other domain-specific regulations, like the Health Insurance Portability and Accountability Act (HIPAA) for medical data, the American Gramm–Leach–Bliley Act (GLBA), and the Payment Card Industry Data Security Standard (PCI DSS) for financial data have similar objectives.

Since raw data are used to train the model in the central server, to guarantee

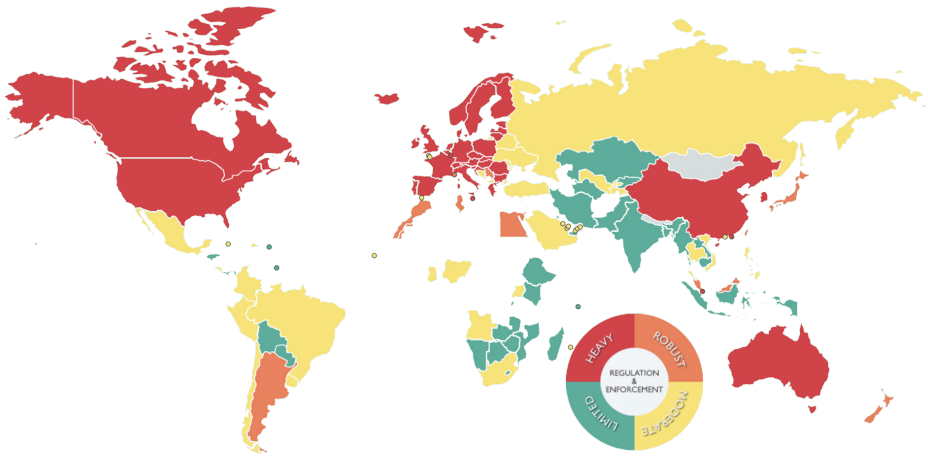


Figure 1.1: Global regulation and enforcement in privacy.

the confidentiality and integrity of these resources, specific controls and techniques must be implemented; traditional centralized training, however, is vulnerable to sensitive data privacy breaches, intruders, hackers, and sniffers and cannot provide the required security. Taking into account these requirements, Google researchers presented in 2016 the concept of Federated Learning (FL), as a solution to cloud-centric limitations. The FL approach is a distributed ML approach where models are trained on end devices without sharing their local datasets to ensure privacy

requirements.

Federated learning applications

Originally intended for mobile devices, FL has expanded rapidly into many other applications, such as the Healthcare industry, the FinTech and insurance sector, Industry 4.0, and blockchain fields, including the collaboration of many organizations to train a model. In *Healthcare industry*, for example, a prediction algorithm for clinical purposes could be trained using vast and varied datasets, as in the Pima diabetes and Covid 19 studies. In *FinTech* sector, ML models could be trained to search for data breaches and Account Take Over (ATO) Fraud or to prevent fraudulent activities. Also in *Insurance* sector, FL could be used to detect individuals or businesses accountable for fraud and illicit activity, for example by training a model that uses varied data ranging from health insurance to car to mobile to business assets. Another important application domain is the IoT field, in which federated learning can help to achieve the personalization of user experience and increase the performance of devices. As an example, the first application of Federated Learning was texts prediction in Google's Android Keyboard and today Apple utilizes federated learning to improve Siri's voice recognition. [2] Because data never leave client devices, in all these applications, where privacy and security of data are essential, Federated Learning represents an excellent solution.

Federated Learning Process

A federated learning process can be summarised in five steps:

1. The FL server first determines an ML model to be trained on the clients' local database.

2. A subset of clients is chosen at random or using client selection algorithms and the server sends the initial or updated global model to the selected clients.

3. The clients receive the global parameters and locally train the model.

4. Results of the local training are sent to the server.

5. The server receives the updated parameters and aggregates them using an aggregation algorithm; then sends the aggregated parameters to the clients. Since the server works only on model parameters, privacy and security requirements could be achieved.

The entire process can be repeated until the desired level of accuracy is reached.

Compared to traditional centralized ML training, FL has several distinct advantages, first of all, training and inference require much less time and bandwidth

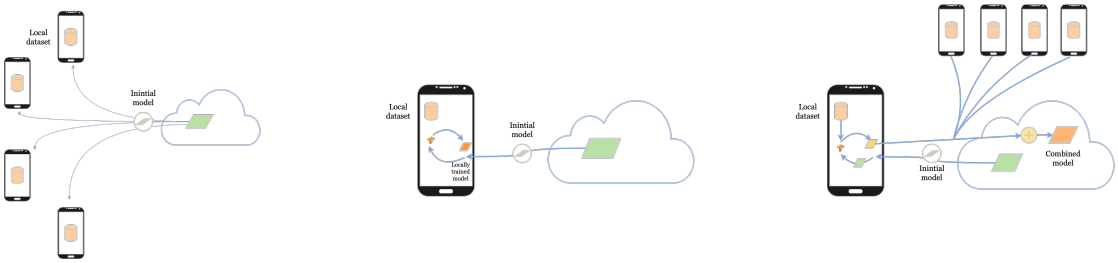


Figure 1.2: Federated Learning process steps: initial Model, local training and global update.

because the data remains local instead of being frequently sent over the network. Moreover, distributed learning using FL is easy and consumes less power on the central server as the models are trained on edge devices. There are three types of FL structures based on how global learning is implemented and where the aggregation is performed: cloud-enabled, edge-enabled, and hierarchical. *Edge-enabled FL* presents the aggregation at the edge, so after the model is locally trained on end devices, it is aggregated and updated on the edge server, and it is then broadcasted to the end devices. In contrast, a *cloud-enabled FL* suits well for model training on systems that are geographically distributed over vast areas, in this case, the parameter server resides on the cloud. The main difference between these two approaches is the proximity to clients that influences the communication latency; it is possible to notice a significant reduction in latency with edge-enabled FL compared to cloud-enabled FL. Despite this, edge servers often have limited resources, which limits their efficiency, especially if we consider the huge number of clients participating in FL and the size of used datasets. Another aspect to assess is the network congestion: the connection with the cloud server is slow and unpredictable, resulting in an inefficient training process. A hybrid approach that seeks to achieve a tradeoff between communication efficiency and the aggregation convergence rate, is *hierarchical FL*. This model makes use of a cloud server to access the enormous training samples and use its local clients to update the model quickly. By employing hierarchical FL, cloud communications will be significantly reduced, combined with efficient updates on the client-side.

1.0.1 Edge Computing in Federated Learning

Over the past few years, computing has become more consistent, and cloud services have moved to the edge. The next generation of Big Data will be based on dispersed data sources equipped with advanced computing capabilities, which results in an increase in edge computing devices. For different reasons, the current cloud-based computing paradigm is becoming incapable of managing and analyzing the huge

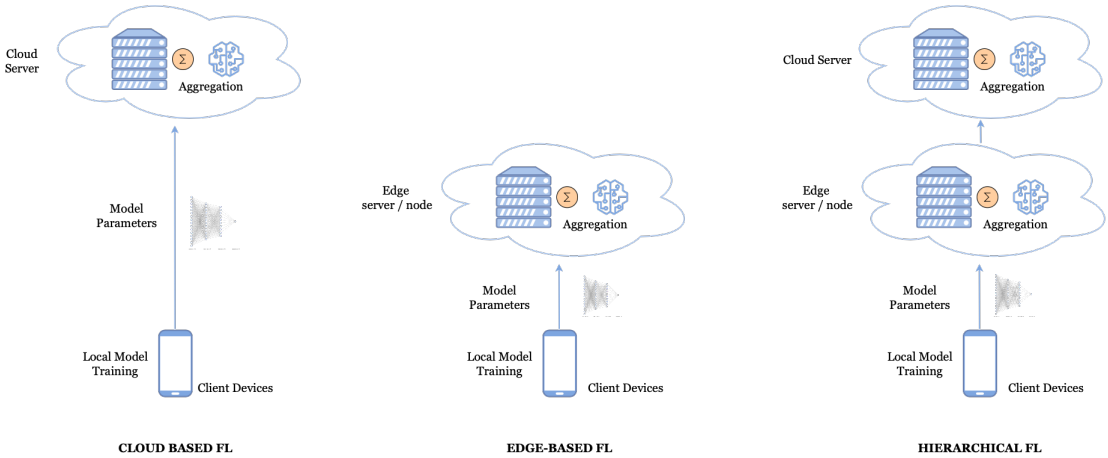


Figure 1.3: Federated Learning architectures, client-based, edge-based and hierarchical FL.

amount of data collected and produced at the edge. With the increase in the size of data and with the advent of new time-sensitive applications, such as augmented reality, virtual reality, and autonomous vehicle network systems, traditional cloud-based servers, typically located far from end nodes, are not able anymore to be performing. Furthermore, with the explosive increase of participants, the central FL server, which acts as the manager and aggregator of cross-device model training, as well as network links, is getting overloaded. Besides the computing and communication costs, IoT devices need to send raw data to the cloud for processing; data transmissions can involve sensitive information, such as patient information, which not only violates privacy but also poses a security risk due to frequent data transmissions. For these reasons, it would be more efficient to process the data at the edge of the network.

Edge computing (EC) makes it possible to bring compute power and storage closer to the edge devices by utilizing distributed computing. This technology is a solution to alleviate the bottlenecks of emerging technologies: data transmission is reduced, services are provided quickly, cloud computing pressure is eased by leveraging distributed computing, and security and privacy are enhanced. The concept of EC does not exclude cloud computing but is a supplement and extension of it; the edge is small, resource-constrained, and heterogeneous and exploits the proximity to end devices to achieve faster results. The application of FL techniques on edge networks has the following advantages over the traditional centralized ML model. By sending update parameters instead of raw data to the FL server, the number and size of communication data are reduced, which leads to a higher

network bandwidth utilization. Moreover, advantages in latency are achieved since edge devices are closer to the end devices, and the performance of real-time applications, such as event detection, augmented reality, and medical applications can be improved by processing them locally at the end user's device.

Despite the great advantages of using edge computing in FL, there may still be issues mainly caused by the limited resources of edge nodes that have to manage an increasing amount of data. So both the cloud-based model and the edge-based model can lead to various issues, in particular, the network and the server may turn into bottlenecks and the load may become unsustainable. A solution to these problems consists of the combination of cloud and edge computing in a hierarchical approach. By this method, both the central server and edge nodes perform aggregation and update of parameters reducing the traffic on the backbone. Edge computing nodes can perform many computations without requiring data to be exchanged with the cloud, furthermore, hosting services at the edge can reduce data transmission delays and improve response times. Another important aspect is privacy: using FL, client devices can collectively train a global model using their combined data without revealing any personal information to the central server.

Until today, intermediate operations on edge nodes, have been performed by computational devices, like smart objects, mobile phones, or servers placed at the edge of the network, but with the advent of programmable switches, moving these computations inside network devices is becoming a possibility. Following this, the thesis proposes a novel FL model that uses programmable *P4 switches* to compute intermediate aggregations and reduce the traffic in the network, hence alleviating the bottleneck effect on the central FL server and further accelerating the entire training progress. In detail, we modified a traditional FL framework, as Flower, to communicate with P4 switches using a custom protocol to carry the model parameters, moreover, we adapted the P4 switch behavior to support gradient aggregation. The main advantage in using this kind of *in-network computation*, compared to the traditional model, in which the model update is performed on a central server, is first of all the ability to execute computation at line rate, hence providing faster results to the clients.

In order to speed up the aggregation on switches, the model parameters were encapsulated into BPP packets. BPP (NewIP/Big Packet Protocol) is a new protocol and framework that allows defining the behavior of packets and flows through information encoded in the packets themselves. The basic idea is to insert a BPP header, between the traditional packet header and user payload, which contains commands and metadata that inform network devices about how to process the packet. In our thesis work, we used BPP commands to indicate the type of operation to be performed, as well as metadata to store model parameters.

After the local training, the end-host creates and sends the packet containing

the parameters, which is received by the intermediate switches performing the aggregation. The behaviour of these switches has been defined using P4, a high-level programming language for packet processing. P4 defines a device-independent way of expressing how packets should be managed by programmable forwarding elements, such as a programmable switch, but it can also refer to NICs, routers, and many other devices. This introduces a lot of flexibility in the system compared with a traditional one because the data plane can be changed in a programmatically way, particularly it is possible to parse the packet header, extract information and use them to process the packet. There are, however, some limitations of P4 that have been addressed in this thesis: P4 language does not support the arithmetical division operation and floating-point values. We used a modified version of BMV2, the P4 standard architecture model, in order to support external functions, written in C++, to overcome the division limitation, while we scaled parameters on end nodes, by a factor of 10^8 to handle integer values.
On top runs the FL framework. Our implementation is a modified instance of Flower, a framework for Federated Learning that supports experimentation with both algorithmic and systems-related challenges in FL. The entire Flower training process relies on the exchange of request-response messages between the central server and the Clients involved in the training. We changed the communication pattern without changing the behavior of the system.

Since the P4 switches need to extract model parameters to perform aggregation, the original Protocol Buffer messages have been replaced with BPP packets containing both the action to be performed and the model parameters in metadata. Also, the transport protocol has been replaced with UDP socket connections to make the packet parsing smoother and enable the Multicast forwarding of the packet directed to all Clients. The final change in the framework concerns the number of connections with the Clients: in the traditional version, the server establishes connections with each Client that joins the training process, in our version the server, since receive just aggregated values from the switch, establishes connections only with special elected hosts representing of the entire subnetwork.

To summarise, all clients participate in the Federated Learning process, but the server will obtain an opaque vision of the topology as it sees just an aggregation of them: this allows for better training process performance in terms of accuracy and loss, with the same number of server connections.

We conducted experiments using a network topology composed of a central server, three intermediate P4 switches performing aggregations, and eight clients participating in the FL process. A significant result is the one about network traffic: having eight clients registered to the server, it is possible to see that the traffic registered in the network is much higher. Another notable result is the one about accuracy that has been measured using a number of registered clients equal to 3. When the aggregation occurs, the accuracy is slightly higher than in the

case of no aggregation; this can be explained by the fact that, even if the number of clients seen by the server is equal in both cases, using aggregation the real number of participating clients, hidden by the elected, is higher, so it is possible to better train the model. Lastly, considering the training time for both solutions, we observed how the aggregation introduces some overhead brought by the P4 additional computation. However, this result is influenced by the virtualized nature of the experimentation system, the specific implementation of BMv2 that we used, the limited amount of parameters in the model, and the absence of challenging network conditions. Despite these factors, the difference between the two solutions is quite negligible, and, along with other metrics, demonstrates the validity of our solution.

The thesis work is organized as follows. In Chapter 2, related works are described with their pros and cons highlighting the innovation they bring. Chapter 3 introduces some necessary background about In-Network computation and P4 switches, in addition, it provides a brief introduction of used protocols and frameworks like BPP and Flower. In Chapter 4, are analyzed the challenges of the thesis work, the architecture, and the technologies used. The implementation details are presented in chapter 5, followed by simulations results and a comparison with other solutions in chapter 6. Finally, in chapter 7, are provided conclusions about the work and points for reflection for future implementations.

Chapter 2

Related work

This chapter presents previous related studies conducted in the field of FL and, in particular, examines various solutions proposed to improve FL performance using distributed architectures or the edge computing paradigm (EC). Variouse papers in the field of programmable networks are presented to explore the possibilities and applications of this technology. This introduction is intended to help the reader understand the value of the solutions proposed in this work. In addition, the presentation of other works is intended to point out possible similarities between the different solutions and to suggest new challenging ideas to improve the current work.

2.1 Federated Learning and Edge Computing

In the current literature, there are many studies on FL, which differ in terms of application domain, technologies and protocols used, but all of them have the common goal of reducing the *bottleneck effect* in the network, reducing the *computational overhead* on the central server and *speeding up the training process*. To achieve these goals, some of them try to use different aggregation algorithms on the central server [3] [4], use other architectures, or move some computations to the edge of the network.

Edge Computing refers to a processing performed at the location closest to the data source in a system, or to the end user. Edge architecture enables faster processing by reducing latency and bandwidth costs and improving network resilience and availability. In particular, it is shown that moving computations to edge nodes to process data closer to the point of origin can reduce model training time and end-devices energy consumption compared to cloud-based Federated Learning. The cloud-based and edge-based FL systems differ in communication and number of participating clients, so it is possible to reduce the computational overhead. In

addition, edge servers enable fast model updates with their local clients, which accelerates the training process.
In some proposed solutions for edge-based FL, the central cloud server is replaced by different edge servers [5] [6] [7], that collaborate to obtain the final global model; in other cases, the edge servers or, in general, edge nodes, only perform intermediate aggregation of model parameters before sending them to the central server for global aggregation.
Many solutions rely on D2D (device-to-device) communication to speed up the training process. For example, in [8], two timescales hybrid federated learning (TT-HF) is proposed, a semi-decentralised learning architecture that combines the traditional paradigm of device-to-server communication for federated learning with device-to-device communication for model training. In this solution, clients are divided into clusters and in each round, the central server elects one device from each cluster. Once the clients finish the local training, a consensus procedure starts, where an aggregation of the local training parameters is performed, and at the end, the sampled devices send the final results to the server.
In [9] is proposed the introduction of intermediate edge servers to perform partial aggregation of the model. This architecture is called *hierarchical FL system*, which has one cloud server, L edge servers indexed by l, to which clients with distributed datasets are connected. After each k_1 local update on each client, each edge server aggregates its clients' models. Then after every k_2 edge model aggregations, the cloud server aggregates all the edge servers' models; in this way, the communication with the cloud happens every $k_1 k_2$ local update, much smaller than the traditional cloud FL. An application of this architecture is proposed in [10].
A similar strategy is proposed in [11]. The authors present a system model where data owners, called workers, participate in FL model training under different cluster heads, for example, base stations that support the intermediate aggregation of model parameters and efficient relaying to the central server (model owner). It is considered a two-level resource allocation and incentive design problem: a first level, a lower level between workers and cluster heads, and a second level between the cluster heads and the central server. At the first level, each worker can freely choose which cluster to join. To encourage worker participation, cluster heads offer reward pools that are shared among workers based on their data contribution in the cluster. There can be multiple model owners in the network who want to train a model, but at any point in time, each worker and cluster head can only participate in the training process with a single model owner. A distinctive feature of the work is the use of the evolutionary game theory to derive the equilibrium solution for the cluster selection phase.

2.2 In-Network computation

An interesting work on edge-based FL is discussed in [12]. The focus of this work is on using edge nodes for In-Network model caching and gradient aggregation. Specifically, the authors propose two new protocols, Model Download Protocol (MDP) and Model Upload Protocol (MUP), which are based on UDP and break the conventional end-to-end principles. These new protocols allow edge nodes to download a single instance of the model from the FL server and pre-aggregate the associated gradient upload requests. In this way, it is demonstrated that it is possible to mitigate the bottleneck effect of the central FL server and further significantly accelerate the overall training progress.
The proposed system consists of three elements: the FL server (FLS), end devices (ED), and the edge box (EB) that provides cache function. To start a round of training, the FLS selects a group of EDs to pull the model using the cache service provided by the EBs. When the EDs complete the training, they send their local gradients back to the FLS, which are aggregated by the EBs. Then the FLS generates the new global model and moves on to the next round of training.
Figure 2.1 shows the workflow of the two proposed protocols with and without the presence of an intermediate EB.

It should be noted that without the EB, any request from ED has a corresponding response from the FL server, while with an intermediate EB it is possible to exploit the cache service to obtain a response and perform intermediate aggregation. In model download, the first time a ED sends a request, a cache miss occurs, so the request is forwarded to the FLS. The response is cached in the EB so that the next time some ED sends a new request, a cache hit occurs and the ED receives the cached response without contacting the FLS. The same is true for upload: the EB caches all model gradients received from the EDs and, once all has finished, performs aggregation and sends the aggregated model to the FLS. Performing these operations through the EB, which is closer to ED, reduces both the traffic to the FLS and the average response time.
The great value of this work lies in the proposed innovative approach, which is different from *Hierarchical Federated Learning*. Typically, edge servers run inside VMs and require complicated synchronization protocols. As an alternative, the authors propose *In-Network processing*, which can be implemented as an optional edge service.
The authors of [13] follow a similar approach for generic data, performing aggregation along network paths instead of edge servers. In particular, they describe the functioning of NETAGG, a software platform that supports on-path aggregation for network-bound partition/aggregation applications. NETAGG uses software middleboxes, *agg boxes*, to perform aggregation along the network path. This allows us to minimize edge server bandwidth usage and congestion in the core of

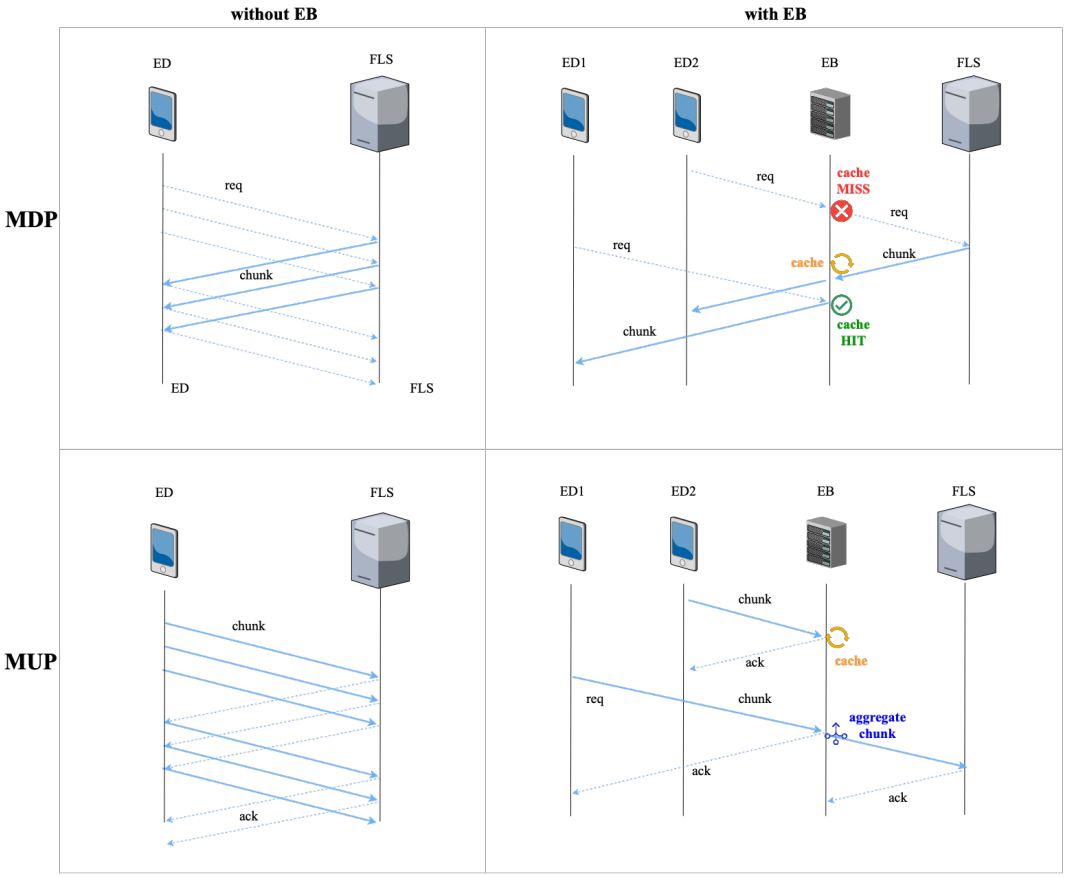


Figure 2.1: Workflow of MDP and MUP with and without EB.

the network, since the forwarded aggregated data is typically smaller than the original one. NETAGG uses shim layers on edge servers to transparently redirect application traffic to agg boxes connected to network switches via high-bandwidth links. A spanning tree, called aggregation tree, is created in which the root is the final master node that collects/consumes the final data, the leaves are the workers that produce data, and the internal nodes are the agg boxes. Each agg box aggregates the data coming from its children and forwards it to the parent.

2.3 Programmable switches and In-Network computation

The concepts of Federated Learning and In-Network aggregation are summarized in [14] and [15]. These two studies are, as far as known, the only ones that contain

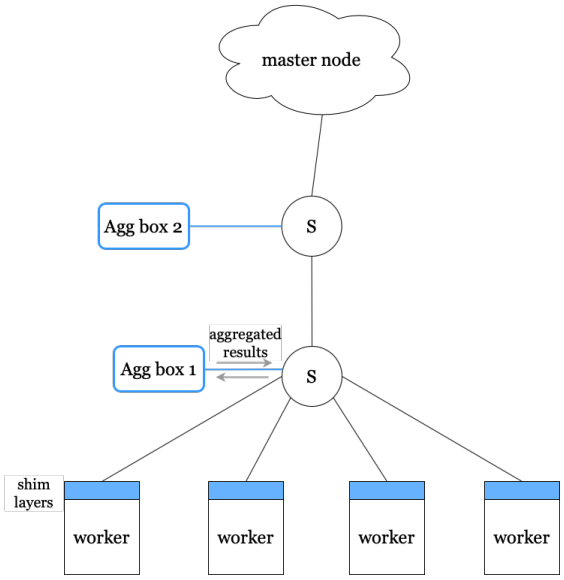


Figure 2.2: NetAgg system architecture.

the same initial idea, overall architecture, and functioning workflow as the proposed thesis; the difference lies in the implementation details.

The work of Chen *et al.*, aims to reduce the bottleneck in the network during model synchronization, with special attention to the security of the transmitted data; in particular, is performed In-Network aggregation of ciphered parameters in *programmable switches.*.

The solution in [14] focuses on cross-silo FL and emphasizes the need for privacy not only in the training set but also in the training results. For this reason, the training results leaving the clients are enciphered using homomorphic encryption.

The entire process can be divided into three main steps:

1. *Local training.* Each client, after being synchronized with the initial model, performs a local training with its local data, then encodes the model parameters into multiple batches, encrypts them, and sends encrypted packets to the parameter server.

2. *Packet forwarding and In-Network aggregation.* When a packet containing model parameters arrives at a switch, In-network aggregation of the parameters is performed. The result of the aggregation is stored in the switch's memory and the packet is dropped. Thanks to the presence of the controller, the switch knows which packets should be aggregated. After aggregating all packets, the switch sends the aggregated result to the next-hop device to reach the parameter server and send back a response to the clients.

3. *Global aggregation.* The aggregated and unaggregated parameters reach the parameter server for global aggregation. The global, updated model is sent back to the clients so they can proceed to the next round of training.

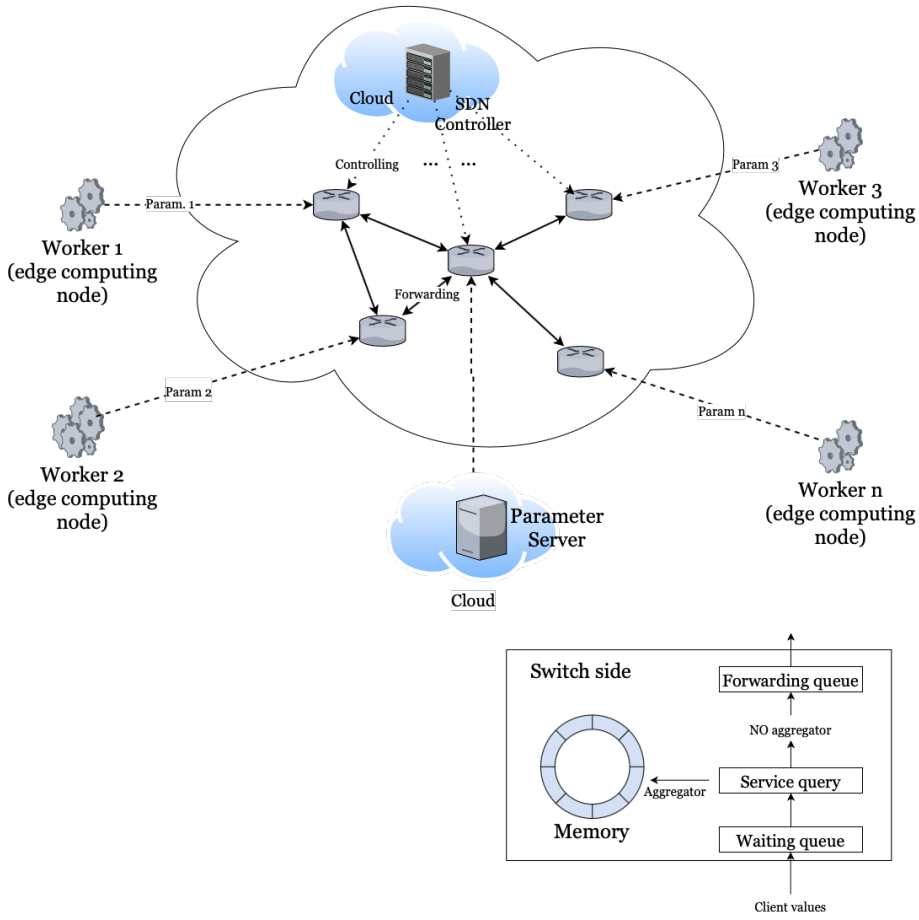


Figure 2.3: System architecture of cross-silo FL with privacy-protecting.

This article highlights the advantages of using programmable switches for In-Network processing and especially for aggregation. Other studies, such as. [15], use this approach and implement In-Network processing with P4 programmable switches. P4 language allows the implementation of custom packet processing logic in programmable switches and well suits for data aggregation.

Sapio *et al.*, propose SwitchML as an approach to reduce the volume of exchanged data during ML training by performing aggregation of model updates from multiple workers in the network. In this way, it is possible to achieve the minimum possible latency and communication cost, measured by the amount of data that each worker

sends and receives. The main motivation for this idea is that the aggregation operation is computationally cheap, taking about 100 ms, but is communication intensive, since hundreds of megabytes can be transmitted at each iteration. For this reason, SwitchML uses computations on the switch to aggregate model updates on the network. This involves minimal communication so that each worker sends its update model vector and receives the aggregated updates back. To achieve this goal, they use programmable P4 switches to aggregate data and prove that a programmable network device can perform in-network aggregation at line rate. The training is an iterative process where each iteration consists of the following steps:

1. Training on workers using local dataset. At the end of training, a gradient vector is produced and sent to the switch.

2. Updating the model with aggregated values. The switch performs aggregation by computing the mean of all received gradient vectors. Then it sends the updated model back to the workers.

In their paper, they present the challenges and solutions of using P4 programmable switches for in-network aggregation and provide a complete implementation and evaluation on a hardware switch. First, per-packet processing capabilities are limited, and so is on-chip memory. For this reason, they limit resource usage and process data grouped into vectors separately using a streaming approach. Specifically, workers send groups of data to the switch, which computes the aggregation and sends the result back to the workers to update the model. Second, the computational units in a programmable switch operate on integer values, while the frameworks and models of ML work with floating-point values. To solve this problem, the workers scale and convert floating-point values to fixed-point numbers using an adaptive scaling factor with negligible approximation loss. Moreover, operations can only be simple integer arithmetic/logic operations, so neither floating-point nor integer division operations are possible. For this reason, the P4 switch performs integer aggregation and only calculates the sum of values, while the end hosts are responsible for managing reliability and performing more complex calculations such as averaging. Third, in-network aggregation requires mechanisms for worker synchronization and packet loss detection and recovery.

This thesis presents the same architecture and goals as [15], but additionally aims to solve the computation problem of averages by attempting to compute divisions within switches as well.

2.3.1 P4 programmable switches

In [16] the authors use P4 switches to implement DAIET, a system for In-Netowrk data aggregation that allows reducing network traffic and computation overhead.

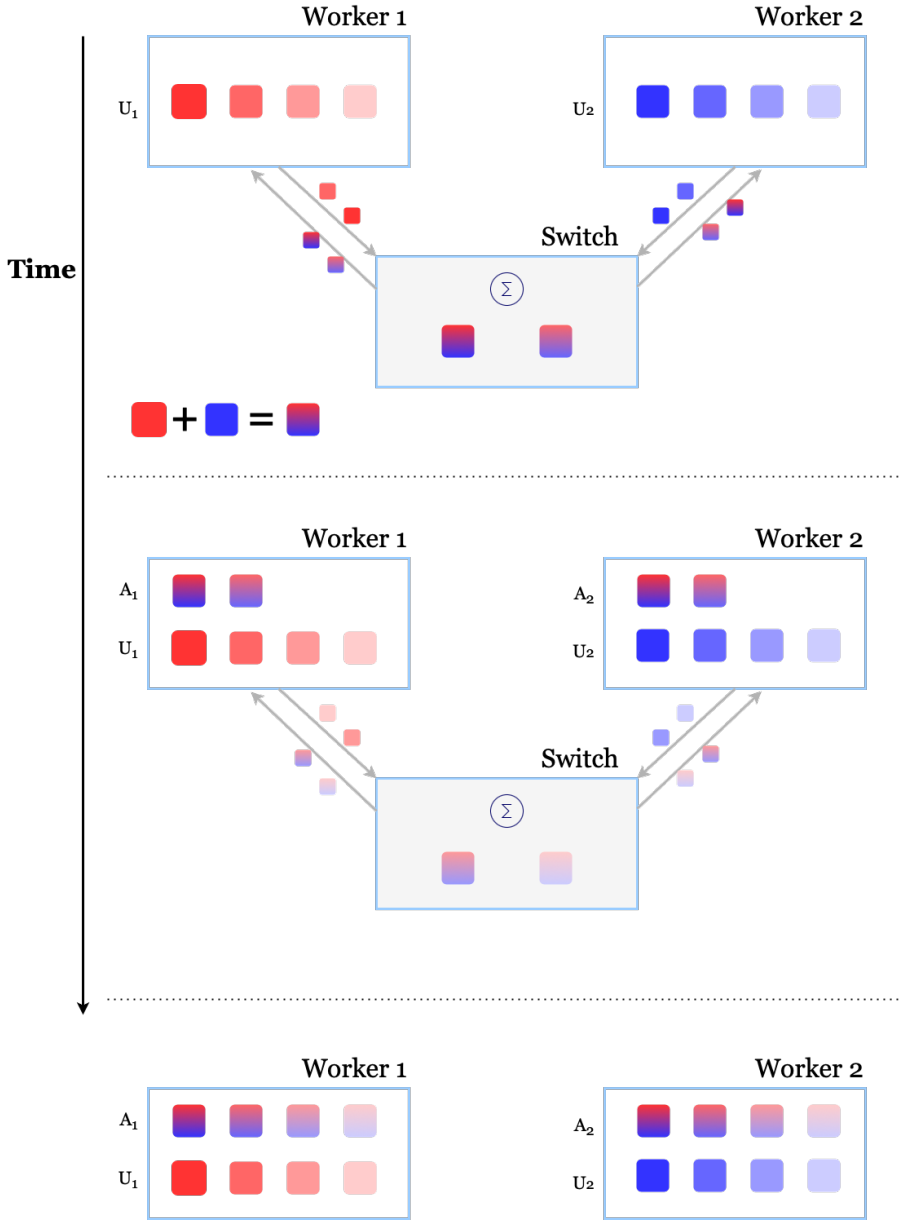


Figure 2.4: SwitchML. Example of model updates in-network aggregation.

Similarly, Yang *et al.* [17] proposed SwitchAgg, a system that performs similar functions but does not require a change in network architecture compared to DAIET and provides better processing capabilities with significant data reduction rate. Moreover, SwitchAgg is implemented on an FPGA, and experimental results show that the entire job can be completed in less than 50% of the time.
Since IoT devices are constrained in terms of size and processing capabilities, they

usually generate packets with small payloads but large headers. When a large number of IoT devices transmit packets, a significant amount of network bandwidth is wasted on transmitting these headers. To solve this problem, packet aggregation can be performed to combine the payloads of small packets into a single larger packet to reduce the bandwidth consumption by headers. An implementation of this mechanism is contained in [18]: to reduce the number of messages transmitted from sensor devices to the IoT server, P4 switches are used to aggregate several small IoT messages into one large packet before transmitting them over the network. The study shows that packet aggregation in a P4 switch can be achieved at its line rate (without any additional packet processing cost). On the other hand, the processing time to disaggregate a packet that combines N IoT messages, the processing time is about the same as processing N individual IoT messages. The same authors have extended this work to solve some constraints related to the payload size and the number of aggregated packets [19]. When the P4 switch receives a packet, it parses the headers and determines if it is an IoT packet. If it is, parses and extracts the payload data. Then, the payload is stored in the switch's registers along with some other metadata, and the packet is dropped. Once the packets are aggregated, the resulting packet is sent across the network to reach the remote server. An important observation is that the aggregation/disaggregation processes are transparent to both the IoT devices and the servers; therefore, no changes are required on either side.
Finally, in [20],the authors use P4 switches used in Federated Learning not to perform intermediate aggregations of model parameters, but to perform scheduling of packets. In the article, they show that the scheduling of packets originating from workers in a FL system, can reduce network blocking time and speed up the overall training time.

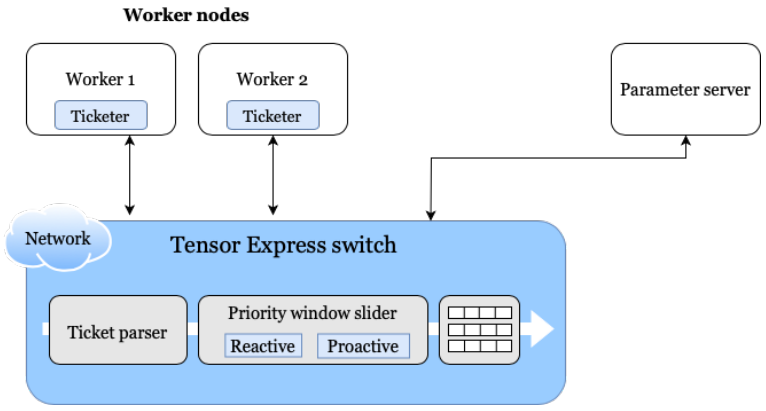


Figure 2.5: Tensor express architecture.

Chapter 3

Background

3.1 Overview

Before delving into the details of the thesis, it is important to provide a brief introduction of the tools and frameworks used in this thesis work. In this chapter, it is introduced the concept of programmable switch and it is given an overview of P4 language, then the Big Packet Protocol (BPP) is briefly introduced. The chapter concludes with an overview of the Flower Framework and how it works.

3.2 Programmable switches

With traditional devices, networks are connected using protocols such as OSPF(Open Shortest Path First) and BGP (Border Gateway Protocol), running in each device and both control and data planes are under full control of vendors. With the introduction of SDNs (Software Defined Network), a first step has been taken toward network programmability: these kinds of networks mark a clear separation between the control plane, implemented in software under the control of the network owner, and the data plane, and consolidates the control plane so that a single centralized controller can control multiple remote data planes.

While the introduction of SDNs (Software Defined Network) reduced network complexity and brought the definition of the control plane to the speed of software development, the packet processing functions, which make up the data plane, have not been affected and remained in the hands of network vendors. Traditionally, the data plane is designed with fixed functions to forward packets using a small set of protocols (e.g., IP, Ethernet) and implemented with ASICs, the design and production of which is lengthy, costly, and inflexible compared to the agility of the software industry. For this reason, the introduction of *programmable switches* led to a new era of innovation and experimentation by reducing the time for designing

and testing, adopting new protocols, providing granular visibility of packet events defined by the programmer, and much more. Specifically, programmable switches enable customization to quickly innovate and differentiate, allow to scale of networks for next-generation workloads, and give more visibility into the paths of the network traffic [21].
Software-defined networking is now becoming a reality, and it is catching up with other areas of information technology infrastructure. Applications of programmable switches ranging from customizing the switch table size for efficient scalability to enhancing existing networking functions and adding new features such as telemetry, security and load balancing. Other use cases may concern DNS caching, firewalling, network packet broker and tunnel gateway.

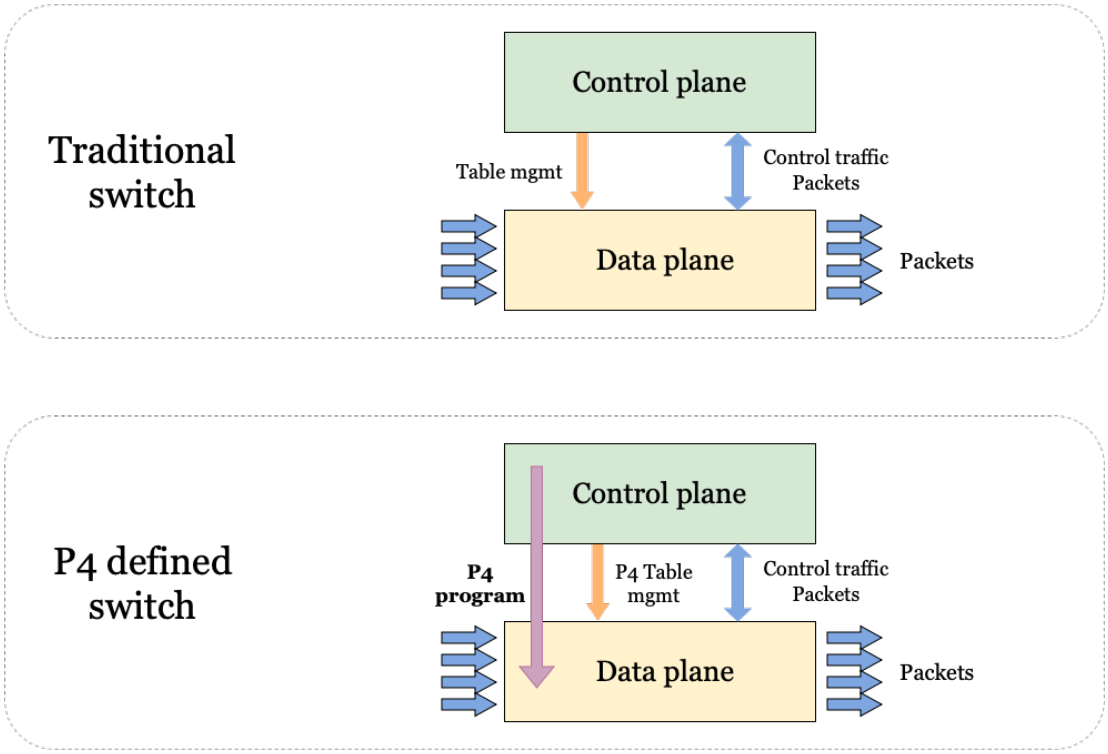


Figure 3.1: Traditional switch vs. P4 programmable switch.

The de-facto standard for defining the forwarding behavior in programmable switches is the P4 language (Programming Protocol-independent Packet Processor)[22]. To understand the significance and resonance of this new technology is possible to see how operators such as ATT [23], Comcast [24], NTT [25], KPN [26], Turk Telekom [27], Deutsche Telekom [28], and China Unicom [27], are now using P4-based platforms and applications to optimize their networks and how big companies with large data centers such as Facebook [29], Alibaba [30], and

Google [31] are operating on programmable platforms in contrast to the fully proprietary implementations of just a few years ago. Also switch manufacturers such as Edgecore [32], Stordis [33] and Cisco [34] have perceived the impact of programmable switches and have started their dedicated production.

Demonstrating the great potential of programmable switches and the resonance they are having in the market, we cite the *Intel Tofino* series, a P4 programmable Ethernet switch ASIC offering better performance at lower power. Intel, KAUST and Microsoft have developed a range of techniques to migrate the network-communications bottleneck and accelerate the performance of distributed training using P4-programmable Intel Tofino Intelligent Fabric Processors [21]. In addition to improving performance, these techniques reduce infrastructure and power costs for the network, because it is possible to turn off features that are not needed and reduce power or use smaller tables. Moreover, it has been proved that programmability does not involve a compromise on performance, for example, Intel® Tofino™ and Intel® Tofino™ 2 can be fully programmed by users using the P4 programming language and are capable of processing up to 12.8 Tb/s.
Table 3.1 shows the main characteristics of traditional, SDN, and P4 programmable devices [35]. In general, it is important to highlight the advantages of P4 programmable switches, the most important the user-defined forwarding behavior; other advantages include the presence of protocol-independent primitives to process packets, a more powerful computation model where the match-action stages can be executed in parallel and the infield reprogrammability at runtime. A P4 implementation is more general and provides greater flexibility than OpenFlow, usually used in SDN, as users can specify exactly how packets are processed in the forwarding plane.

In summary, the main features of programmable switches [35] are :

- Agility. The design and testing time are shorter, and is possible to quickly adopt new protocols and features.
- Visibility. Programmable switches provide greater visibility into the behavior of the network without intervention of the control plane.
- Reduced complexity. Compared to traditional switches that incorporate a large superset of protocols, programmable switches can integrate only those protocols that are needed, reducing resource consumption and complexity.
- Differentiation. The protocols implemented by the user needs not to be shared with the chip manufacturer.
- Enhanced performance. Programmable switches do not introduce performance penalty, on the contrary it has been demonstrated that in some cases they may produce better performance than fixed-function switches.

Feature	Traditional	SDN	P4switch
Control - data plane separation	No clear separation	Well defined separation	Well defined separation
Control - data plane interface	Proprietary	Standardized APIs (OpenFlow)	Standardized (OpenFlow, P4Runtime) and program-dependent APIs
Control and data plane program-dependent APIs	NA/Proprietary	NA/Proprietary	Target independent
Functionality separation at control plane	No modular separation of functions	Modular separation: functions to build topology view and algorithms to operate on network state	Same as SDN
Customization of control plane	NO	YES	YES
Visibility of events at data plane	Low	Low	High
Flexibility to define and parse new fields and protocols	No flexible, fixed	Subject to OpenFlow extensions	Easy,programmable by user
Customization of data plane	NO	NO	YES
ASICs packet processing complexity	High, hard coded	High, hard coded	Low, defined by user's source code
Data plane match-action stages	Proprietary	OpenFlow assumes in series match-action stages	In series and/or in parallel
Data plane actions	Protocol-dependent primitives	Protocol-dependent primitives	Protocol-INdependent primitives
Infield runtime reprogrammability	NO	NO	YES
Customer support	High	Medium	LOW
Technology maturity	High	Medium	LOW

Table 3.1: Comparison between traditional networking, SDN and P4 switches [35]

Despite programmable switches providing many benefits, they also present limitations. In particular, the field of programmable switches and in particular P4

switches is not mature and the customer support, as well as the documentation, is very sparse. Due to this, it is difficult to comprehend the language and users have to be trained for a very long period of time. This is the major restriction in approaching switch programming and has been the major restriction also in tackling this thesis work.

3.2.1 P4 systems

In recent years, interest in programmable switches has increased greatly, both in industry and in research; for this reason, it is possible to find in the literature many studies that try to exploit the characteristics of such devices and explore their application in various use cases. In 2 has been proposed some studies that use programmable and P4 switches to perform In-network computations and aggregate values but there are other fields in which programmable switches are used. [35].

- In-Band Network Telemetry (INT): Variations and collectors.
- Network performance: congestion control, measurements, AQMc(Active Queuing Management), QoS (quality of service) and TMc(Traffic Management), multicast.
- Middle-box functions: load balancing, caching, telecom services, content-centric networking.
- Accelerated computations: consensus, machine learning. IoT (Internet of Things): aggregation, service automation.
- Cybersecurity and attacks: heavy hitter, cryptography, access control, attacks and defenses.
- Network and P4 testing: troubleshoot and verification

Network-accelerated computations

Traditional network devices are not capable of performing computations, so these have to be performed at upper logical layers, but with the advent of programmable switches, performing computations inside network devices is becoming a possibility. Since switch ASICs are designed to process packets at terabits per second rates, the computation inside them can be faster compared to applications implemented in software, and for this reason network computations are becoming a trend in data centers and backbone networks.

One field of application in which network-accelerated computation can have a strong impact is Machine Learning (ML) and in particular Federated Learning

(FL). Previous studies focused on methods to accelerate the computation process, but it has been proven that the real bottleneck is in communication. Specifically, communication times between workers and a central server could be tens of orders of magnitude higher than computational times. Programmable switches emerge as a solution to accelerate the overall FL training process through the network and could be used in different modes; for instance, they could be used to perform aggregation of model updates or to classify new samples. The main advantage in using this kind of *in-network computation*, compared to the traditional model, in which the model update is performed on a central server, is first of all the ability to execute computation at line rate, hence providing faster results to the clients. A more detailed comparison between switch-based computation and server-based computation is reported in table 3.2.

Feature	**Training**	
	Switch-based	**Server-based**
Speed	Faster: computation at line rate	Slower: computation on server
Complex computation support	Lower	Higher
Communication overhead	Lower: switch is the centralized aggregator and is closer to workers. Only the aggregated values are sent to the server	Higher: updates are exchanged with workers
Storage	Lower: update is not stored entirely at once	Higher .
Encrypted traffic	Difficult	Easy

Table 3.2: Comparison between Switch based and Server based ML model training.

3.2.2 P4 language

P4 (Programming Protocol-independent Packet Processor)[22] is a high-level programming language for packet processing. It was first introduced in 2014 with the specific purpose of expressing how packets have to be managed by a programmable forwarding element, like a programmable switch but also a NIC, a router, and many

other varieties of devices. This introduces a lot of flexibility in the system, compared with a traditional one: in particular, in the former, the manufacturer defines all the data plane functionalities that remain fixed for the entire life of the device, with a P4 switch instead the data plane can be changed in a programmatically way. In particular, the data plane is configured at initialization time to implement the functionality described by the P4 program and has no built-in knowledge of existing network protocols. Furthermore, the control plane communicates with the data plane like in a fixed-function device, but the set of tables and other objects in the data plane, are no longer fixed since they are defined by a P4 program. Important to note that even if P4 partially define the interface between the control plane and the data plane, through the generation of API by the P4 compiler, it is specifically designed to manage just the data plane.

Other advantages in using P4 systems are:

- Expressiveness: P4 language can express sophisticated, hardware-independent packet processing algorithms using only general-purpose operations and table look-ups.
- Software engineering: P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- Component libraries: Manufacturers can supply component libraries to wrap hardware-specific functions into portable high-level P4 constructs.
- Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.

Forwarding model

The P4 standard architecture model, named BMv2 (Behavioral Model v2), is based on the following components: the Parser, the match-action pipeline containing the Ingress and Egress processing, the verification and updating steps of checksum, and the Deparser.

Following this model, a P4 program contains the key component:

- Headers: describe the sequence and structure of a series of fields. In general, each header is provided by declaring a list of field names and their widths in bit.
- Parser: specifies how to identify header sequences within packets.
- Tables: contains the fields to match in order to execute a specific action.

- Actions: specify the behaviour of the switch when a match in tables occurs.
- Control program: expresses an imperative program that describes how packets are processed.

When a packet arrives at the ingress port, the parser handles it: it recognizes and extracts all the fields in the headers and defines the protocols supported by the switch. The packet is then passed to the match-action pipeline, which is divided into ingress and egress. Ingress match-action tables determine the egress port to which the packet has to be forwarded, while the egress match-action tables perform per-instance modifications of the packet headers.

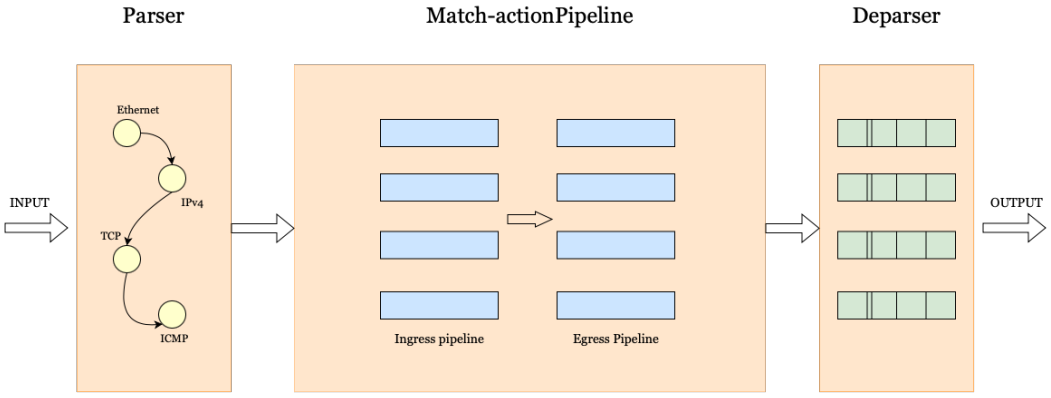


Figure 3.2: P4 internal functioning.

3.3 BPP: Big Packet Protocol

BPP (NewIP/Big Packet Protocol) is a new protocol and framework that allows defining the behavior of packets and flows through information encoded in the packets themselves. The basic concept is to insert a BPP block between the traditional packet header and the user payload, which contains commands and metadata that provide indications to network nodes on how to handle packets and flows, or what resources must be allocated. In this way the device will act just on those commands and metadata to handle the packet, overriding any "regular" packet processing logic that is deployed on the device. Commands can be used, for example, to determine conditions when to drop a packet, which queues to use, when to swap a label or to allocate a resource.

Because of these potentialities, BPP has been extended to a variety of application domains, including the use of metadata for collaborative vehicular information exchange [36], latency guarantees in multimedia streaming [37], semantic mashup in the Internet of Things (IoT) [38], and computation offloading in Mobile Edge Cloud (MEC) [39].

3.3.1 BPP Block

A BPP Block starts with a BPP Block Header that describes the overall structure of the BPP Block and includes several fields, such as the BPP version and block length. Then is present a Command Block with commands, their conditions and parameters, as well as a Metadata Block carrying additional metadata. A BPP

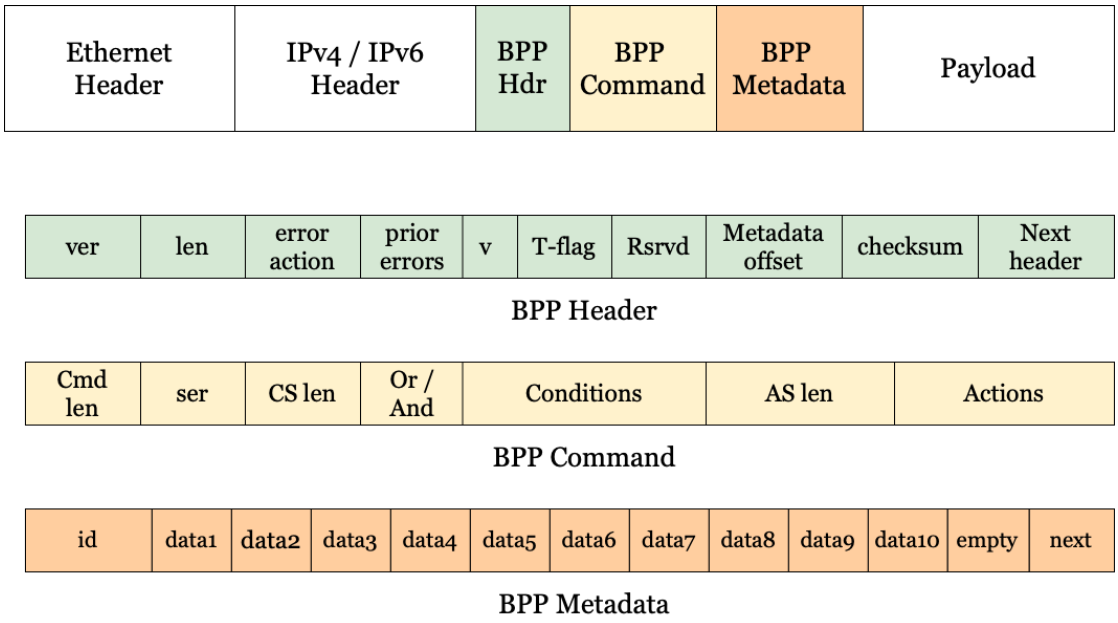


Figure 3.3: BPP Block structure.

Command Block can carry one or more BPP Commands. Each BPP Command consists of three parts: a Command Header, a set of conditions, and a set of actions to be applied when the conditions are met. Conditions and actions parameters might be a reference to a node's built-in data item, a reference to a metadata item included in the BPP Block, or a data value. Action primitives in BPP Commands can be classified into different types based on the command's target: the packet, the device, or the flow. Primitives that act on a packet include primitives to drop, mark, buffer a packet, select a queue, direct the packet to a particular interface, or interact with metadata. Primitives that act on the device include actions to allocate and reserve resources, while primitives that act on a flow, for example to reduplicate or reorder packets, are possible but require additional investigation. BPP does not include primitives to interact with payload data: to avoid interfering with application-level protocols and to maintain the confidentiality, privacy, and integrity of user data, the packet content is fully opaque to BPP. A Metadata Block can be used to transport extra metadata as part of BPP Blocks; these can be accessed by conditions and instructions, or nodes along the route can act on

them in various ways.

3.4 Federated Learning frameworks

The potential and applications of Federated Learning have led to its establishment and many frameworks, both proprietary and open-source, are currently being used to implement it.

- *IBM Federated Learning Framework*: a Python licensed framework for FL in an enterprise environment. The main characteristic of this framework is the variety and the number of contained ML algorithms: i.e. NN, linear classification, decision trees (ID3 algorithm), K-means, naive Bayes, and reinforcement learning algorithms.
- *NVIDIA Federated Learning Framework*: a licensed framework that uses NVIDIA Clara Train SDK working with CUDA 6.0 or later. It supports TensorFlow, TResNet, and AutoML.
- *TensorFlow Federated (TFF)*: an open-source framework for decentralized ML and other computations. TFF was developed by Google in order to foster open research and experimentation with FL.
- *PySyft*: a MIT-licensed open-source Python library to perform encrypted, secure, and private deep learning. The key techniques that are used in this framework are Secured Multi-Party Computations (sMPC), differential privacy and FL.
- *Federated Learning and Differential Privacy framework (FL DP)*: a FL and DP open-source framework that has been released under the Apache 2.0 license. This framework uses TensorFlow Version 2.2 and the SciKit-Learn library to train linear models and clusters.
- *FATE*: is another open-source FL framework developed by Webank's AI department. The main goal of this framework is to support big data collaboration according to the regulations by integrating multiple secure computation protocols like homomorphic encryption and multi-party computation. Fate platform uses several features such as a flexible scheduling system, a modular, scalable modeling pipeline, and clear visual interfaces to keep the scalability, user-friendliness, and improved operational performance.
- *Flower: A Friendly Federated Learning Research Framework*: Flower is an open-source platform-independent FL framework. It is specifically designed to provide high scalability and support for heterogeneous clients.

3.4.1 Flower: A Friendly Federated Learning Framework

Flower is a novel end-to-end federated learning framework that supports experimentation with both algorithmic and systems-related challenges in FL[40]. The main design goals of this framework are:

- Scalability: because real-world FL involves a large number of clients, Flower should scale to a large number of concurrent clients to reflect a realistic scenario. Thanks to its ability to scale even with large numbers of clients, more than 10 000, it is used for many real-world applications.
- Client-agnosticism: given the heterogeneous environment on mobile clients, Flower should be interoperable with different programming languages, operating systems, and hardware.
- Communication-agnosticism: because of heterogeneous connectivity settings, Flower should allow different serialization and communication approaches.
- Privacy-agnosticism: different FL settings (cross-device, cross-silo) have different privacy requirements hence Flower should support common approaches.
- Flexibility: it should be able to accommodate both experimental research and rapid adoption of the recently proposed approaches with low engineering overhead due to the rate of change in FL and the general ML ecosystem.

Federated Learning mechanism is based on the combination of global and local computations: in Flower framework, global computations are executed on the server-side that orchestrates the learning process over a set of available clients. Local computations are executed on individual clients and consist of training, using local data, or the evaluation of model parameters. All the global logic for client selection, configuration, parameter update aggregation, and federated or centralized model evaluation can be expressed through the Strategy abstraction; local logic instead, focuses more on model training and evaluation on local data partitions.
In summarizing, on the server-side, there are three major components: the *ClientManager*, the *FL loop*, and a *Strategy*. Server samples clients from the ClientManager, which manages a set of *ClientProxy* objects, each representing a single client connected to the server: they are responsible for sending and receiving Flower Protocol messages to and from the actual client.
The main actor of the entire process is the FL loop: inside each round of the loop the server communicates to the clients and aggregates the results: it requests the Strategy to configure the round of FL, sends those configurations to the affected clients, receives the resulting client updates (or failures) from the clients, and engages the Strategy to aggregate the results. It takes the same approach for both federated training and federated evaluation. The client-side is simpler: it only waits

for messages from the server and reacts to them accordingly by calling training and evaluation functions provided by the user. Figure 3.4 summarises the framework behaviour.

Flower is designed to be open-source, extendable and, framework and device agnostic. These features have been exploited in this thesis work to realize a new FL model; in particular, starting from the original Flower framework, we replaced the communication implementation to make the switch aggregation possible.

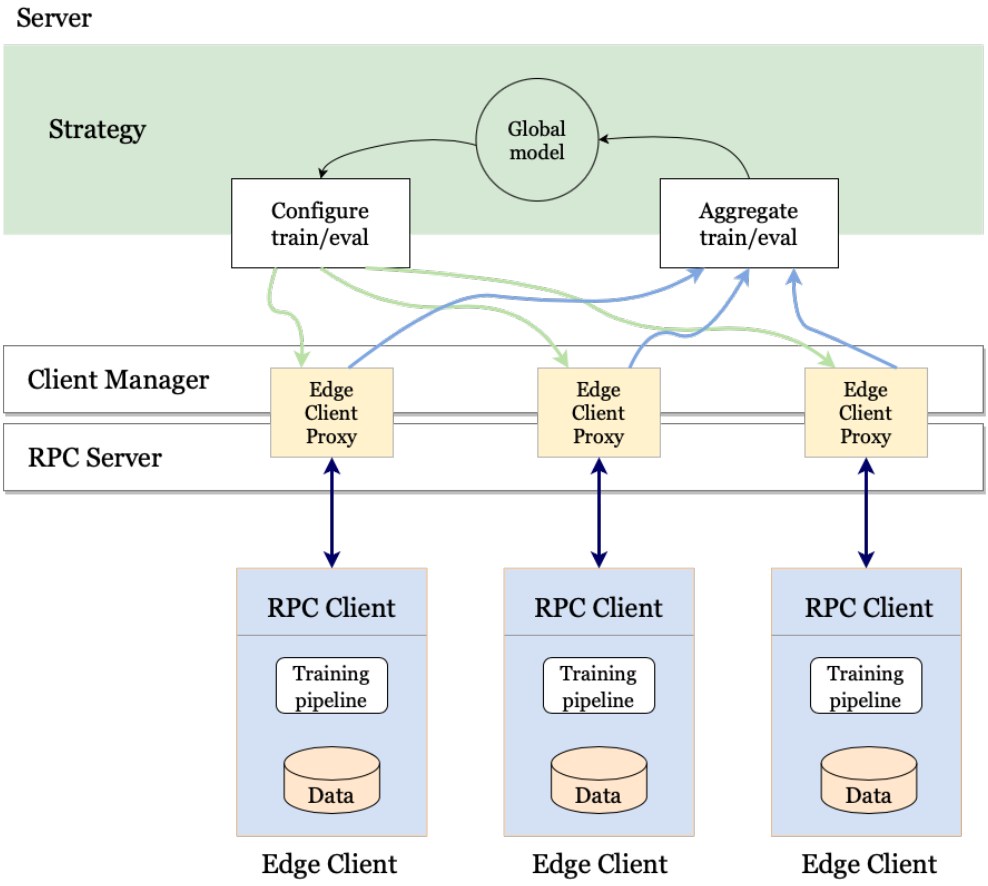


Figure 3.4: Flower core framework architecture.

Chapter 4

System model

4.1 Architecture

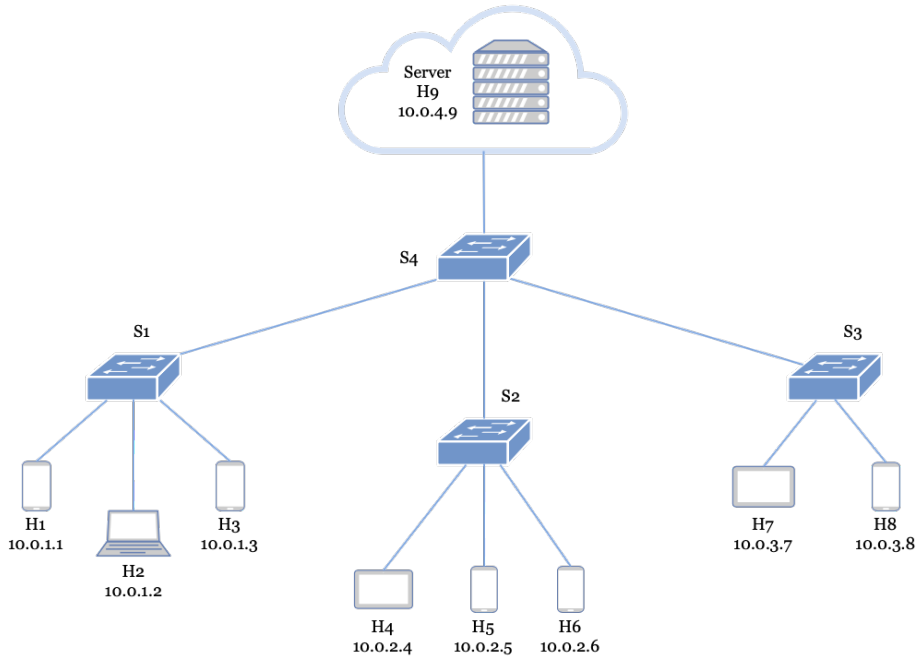


Figure 4.1: Network topology

The proposed system has been developed and tested using an emulated system in *Mininet*, a network emulation orchestration system that uses lightweight virtualization to run a collection of end-hosts, switches, routers, and links on a single Linux kernel. Following this approach, it was possible to recreate a network with a

behaviour resembling a real one, with certain link speeds and delays, and it was also possible to send and process packets through what seems like a real Ethernet interface.
Figure 4.1 shows the network topology in question. It consists of a server, four P4 switches, which rely on BMv2 software switch, and eight clients belonging to different networks accordingly to the switch they are connected to. The central server is responsible for the aggregation of the parameters that are generated by clients, who locally perform the model training. The P4 switches located at the edge of the network perform intermediate aggregations of the parameters received from the connected clients, while the switch S4 has just to forward packets from and to the server.

4.2 Flower functioning and communication protocol

Flower framework is based on a message exchange between the central server and the clients (workers) to train a Federated Learning model. For its design and implementation, the framework is ML model agnostic so it is possible to use whatever model for training. In this thesis work, we used the "Pima Indians Diabet Dataset" [41] as data source and a custom Neural Network (NN), shown in Figure 4.2, composed of 2 hidden layers. The objective of this NN is to use certain diagnostic measurements contained in the dataset, originally compiled by the National Institute of Diabetes and Digestive and Kidney Diseases, to predict whether or not a patient has diabetes.
The choice of using this dataset and network was driven by the limited number of entries and model weights that made the implementation, and especially the verification and the analysis of the results, easier. It is certainly possible to enhance the NN and obtain better results in terms of accuracy and loss and to use different dataset in order to apply the proposed solution to various use cases.
In Figure 4.9 is shown an example of the message exchange between server and clients in the standard version of Flower framework.

All the communications are made upon gRPC connections and messages types are defined using protocol buffers. As the first operation, the server establishes a gRPC server connection and waits until some client connects; then randomly selects one of these and sends to it the first message requesting the initial parameters. Once it obtains the initial parameters, it puts itself on hold until the minimum number of clients for training has been reached. Each client is registered and managed by a Client Proxy and all of them are orchestrated by a Client Manager. If the number of *registered clients* reaches the minimum number, the FL loop starts and the Server sends the first Fit request (FIT_INS) message. Each registered client, on receiving

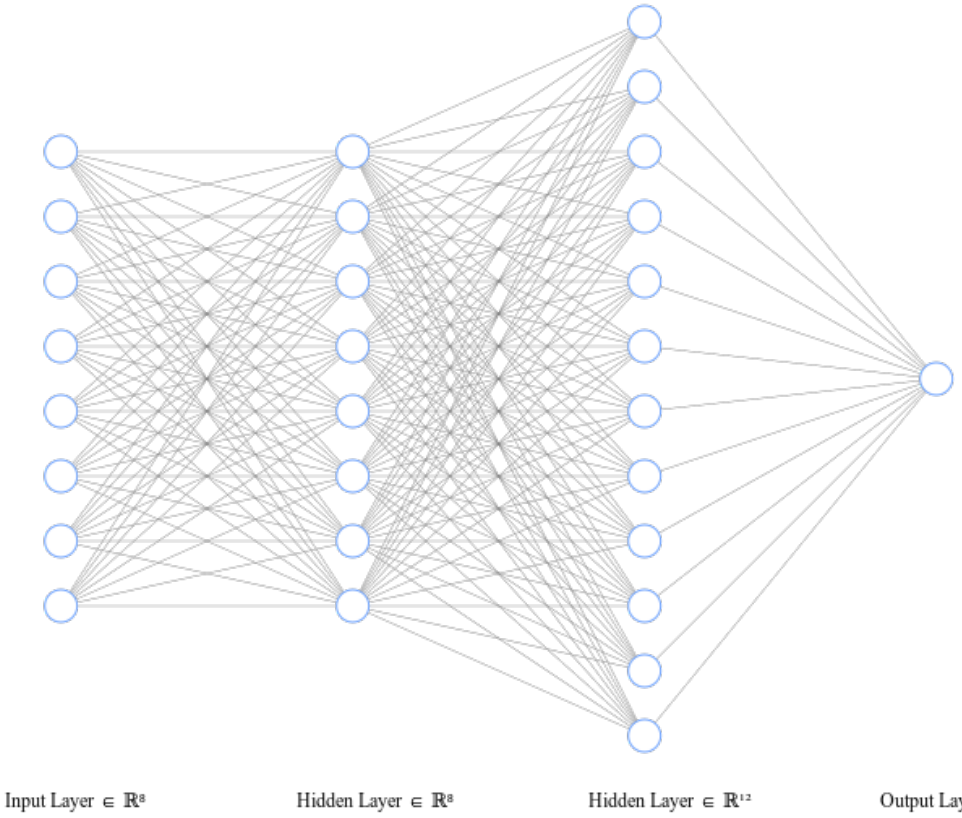


Figure 4.2: Neural network.

this message, starts the local training and at the end sends back to the Server a new message (FIT_RES) containing the new model parameters. The server, once has collected all the results, aggregates them using the selected Strategy; in our case, we used the FedAvg strategy, which aggregates the parameters by computing their average value. After each round of fit, the server sends also a message requesting the evaluation of the model (EVALUATE_INS), to which each Client will react by evaluating locally the model and sending back the results; the server will average also these results. After the fit and the evaluation step, a new round of FL could start and the new Fit request message will contain the aggregated parameters of the previous round. For our study, we used a number of rounds equal to three, as we saw that with this value the accuracy and loss of the model settle down.

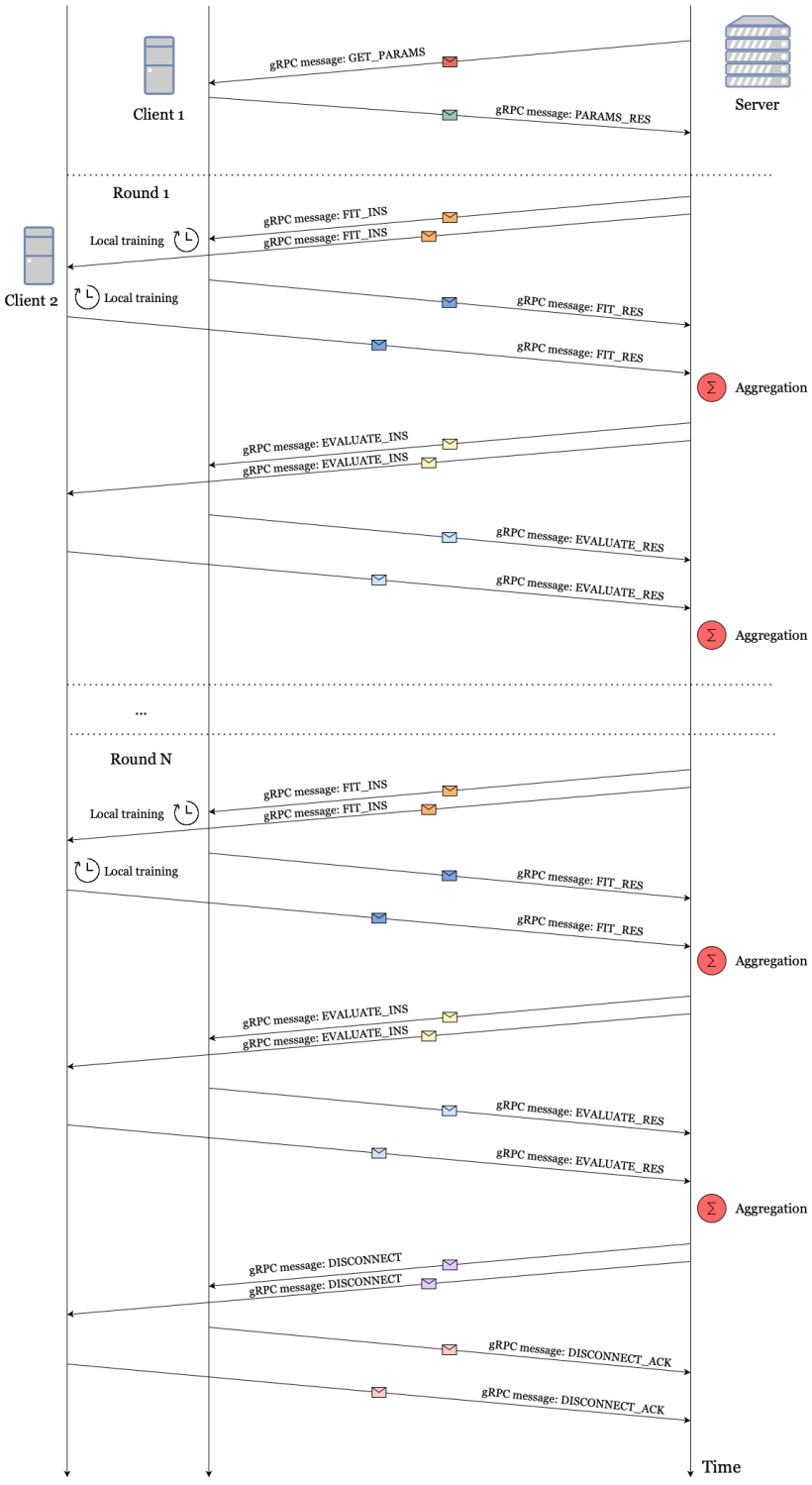


Figure 4.3: Flower communication protocol. Min_num_Clients = 2

This thesis wanted to maintain the standard behavior of Flower, but changing the protocol of communication. Even if in the BMv2 implementation is proposed a switch version with gRPC capabilities, it is not complete and continuously updated; for this reason, we used the stable simple_switch implementation and we replaced the gRPC connections with UDP sockets that make easier the elaboration of packets by the programmable switches. In connection with this, to make this elaboration possible, the protobuf messages have been replaced with UDP/BPP packets: specifically, end devices, both server and clients, creates BPP packets containing in metadata the model parameters; then these packets are encapsulated in UDP packets and transmitted through the socket. In conclusion, both on the client and server-side we made changes in the code to support the new communication protocol, the creation, and management of the UDP socket, and the creation of the novel packet format.

4.3 Client

The main activities performed by the client are the communication with the server and the reaction to the received messages performing local training and evaluation of the model. The received BPP packet contains, in the Command header, the type of action to be performed, hence the Client will react accordingly to this value. The fit of the model and the evaluation are performed using the local dataset: in our study, we assigned 80% of the data from the "Pima Indian Diabetes Dataset" to perform the training and 20% to perform the testing. Moreover, the clients built these sets randomly, to make sure that different values could be present.
The training and evaluation operations internally calls the Tensowrflow functions *train()* and *evaluate()*. The choice of Tensorflow is just an implementational choice: Flower is ML framework agnostic so it would be possible to use either Keras, PyTorch or Numpy.

Algorithm 1 Flower algorithm - Client side

```
while round ≠ n_rounds do
    Fit_Ins = receive(Server)
    LocalTraining()
    send_FIT_RES_packet(Server)
    Eval_Ins = receive(Server)
    LocalEvaluation()
    send_EVAL_RES_packet(Server)
end while
```

4.3.1 Elected Clients

The main focus of this thesis is the parameter aggregation in the intermediate switch; following this concept each P4 switch will receive N messages from the N-connected clients, perform the parameters aggregation and forward just one message containing the computation results. Please note that with the term message in this part, we mean the collection of parameters sent by a client, that in the real implementation corresponds to multiple UDP/BPP packets.

The server, from its point of view, does not have any knowledge of the network topology and the number of switches, but it bases the aggregation only on the number of registered clients. If each client registers to the server, it will expect a number of messages as the number of registered clients to perform the aggregation; but since the intermediate switch performs the aggregation and forward just the aggregated message, the server will receive fewer messages than expected, in particular, it will receive one message from each switch, so from each subnetwork. To solve this problem, the concept of *Elected Client* was introduced. A client, named the elected, is selected from each subnetwork, and it will be the only one that sends, at the beginning of the process, a registration message, a packet with `BPP_Command.a1Type` equal to `BPP_PUT`, to the server. The switch will perform on this message, and on all the following packets directed to the server, an IP address translation (a sort of NAT). In this way, the server will obtain an opaque vision of the topology as is depicted in Figure 4.4. To summarise, all clients participate in the Federated Learning process, but the server will see just an aggregation of them: this allows for a greater training process performance in terms of accuracy and loss even with the same number of registered clients.
In our implementation the choice of elected clients was arbitrary: we choose as elected the host with the lower hostname, belonging to a specific subnetwork, but in a more general scenario an election process could be implemented. This process should include all hosts involved in the training process and should restart each time the current elected dies or disconnects.

4.4 Server

The central server is in charge of starting the FL process, initializing the Strategy, configuring the round, sending requests to the clients and aggregating the responses. To perform these tasks, it relies on three major components: the *Client-Manager*, the *FL loop*, and a *Strategy*. As soon as the server is started, it creates N threads, where N is the maximum number of workers and waits on them for the connection to be established before registering clients. The framework's standard version makes use of gRPC connections, so the server registers the client once it is notified that the RPC has been invoked; in our version, the server is waiting for a special

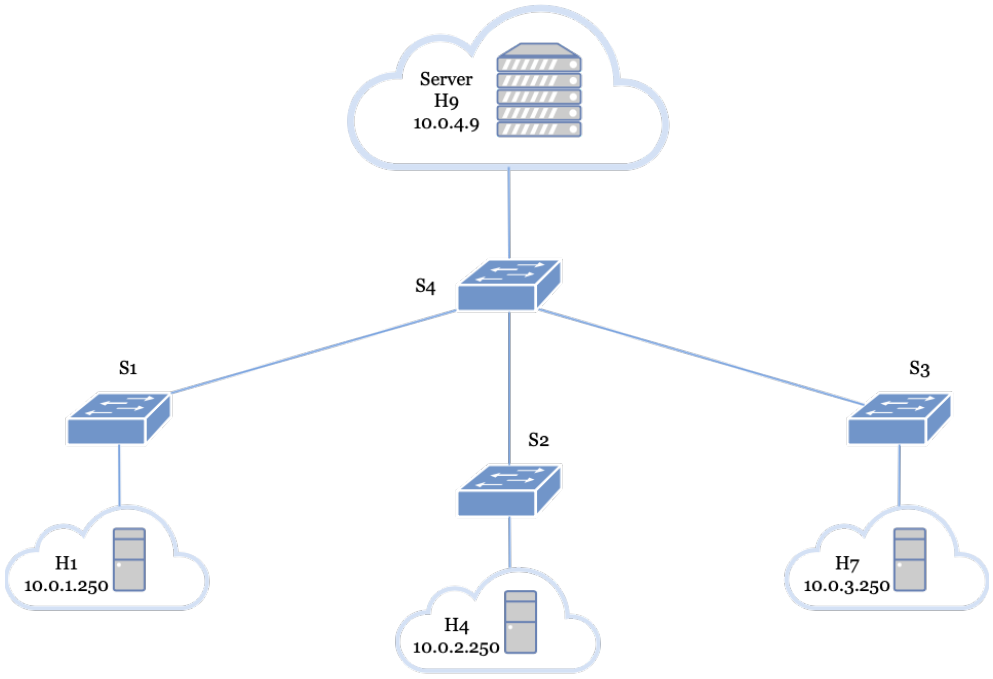


Figure 4.4: Network topology seen by Server. NAT translation address is in the form 10.0.x.250 where x represent the subnetwork number and correspond to the id of the connected switch.

packet to register the client that consists in a BPP packet with command.a1type set to `BPP_PUT`. Note that these packets are sent only by the elected clients. The registration consists on the creation of a Client Proxy instance, representing the client, and of a Server Bridge instance to manage the communication with the specific worker. After at least one client has been registered, the server randomly selects one from the Client Manager, who manages Client Proxies. FL will then be started and the initial parameters are requested if they have not already been specified. Message exchange with clients relies on a Server Bridge: this is in charge of creating the message request, in the form of BPP packets, sending them through the connection, that in our case is a UDP socket, and waits for responses. Server messages include: initial parameters request (GET_PARAMS), Fit request messages (FIT_INS), and Evaluate request messages (EVALUATE_INS). When packet responses arrive, the Bridge parse the packets, extracts metadata from BPP block, and collects the parameters in the proper form, then passes them to the main process for the aggregation.

Algorithm 2 Flower algorithm - Server side

```
while len(registered_Clients) <= 0 do
    wait()
end while
randomClient = choose_random_Client()
send_GET_PARAMS_packet(randomClient)
receive(randomClient)
while round ≠ n_rounds do
    send_FIT_INS_packet(allClients)
    Fit_Res = receive(allClients)
    aggregate(results)
    send_EVAL_INS_packet(allClients)
    Eval_Res = receive(allClients)
    aggregate(results)
end while
```

4.4.1 Strategy

Flower allows full customization of the learning process using the Strategy abstraction: methods for sampling clients, configuring clients for training, aggregating updates, and evaluating models depend on the strategy used. In our thesis work, we used the simplest strategy, the FedAvg strategy, that performs the average of collected parameters, but Flower framework provides the implementation of other Strategies and optimization methods, like FedAdaGrad, FedYogi, and FedAdam.

In FedAvg [43] a subset of clients are selected, typically at random, and the

Algorithm 3 Simplified FEDAVG [42]

```
Input: x_0
for t = 0, ..., T − 1 do
    Sample subset S of Clients
    x_i^t = x_t
    for each client i ∈ S in parallel do
        x_i^t = SGD_k(x_t, η_l, f_i) for i ∈ S (in parallel)
    end for
    x_{t+1} = (1/|S|) Σ_{i∈S} x_i^t
end for
```

server broadcasts its global model to all of them, while the clients perform SGD on their loss functions, and then transmit their model to the server. The server then

Algorithm 4 FEDADAGRAD, FEDYOGI and FEDADAM [42]

Initialization: $x_0, v_{-1} \geq \tau^2$, decay parameters $\beta_1, \beta_2 \in [0, 1)$
for $t = 0, ..., T-1$ **do**
  Sample subset S of Clients
  $x_{i,0}^t = x_t$
  for each client $i \in S$ **in parallel do**
    for $k = 0, ..., K-1$ **do**
      Compute an unbiased estimate $g_{i,k}^t of \nabla F_i(x_{i,k}^t)$
      $x_{i,k+1}^t = x_{i,k}^t - \eta_l g_{i,k}^t$
    end for
    $\Delta_i^t = x_{i,K}^t - x_t$
  end for
  $\Delta_t = \beta_1 \Delta_{t-1} + (1-\beta_1)(\frac{1}{|S|}\sum_{i \in S} \Delta_i^t)$
  $v_t = v_{t-1} + \Delta_t^2$ **(FEDADAGRAD)**
  $v_t = v_{t-1} - (1-\beta_2)\Delta_t^2 sign(v_{t-1} - \Delta_t^2)$ **(FEDYOGI)**
  $v_t = \beta_2 v_{t-1} + (1-\beta_2)\Delta_t^2$
  $x_{t+1} = x_t + \eta \frac{\Delta_t}{\sqrt{v_t} + \tau}$
end for

updates its global model as the average of these local models. For the FedAdaGrad Algorithm, there are three steps: initialization, sampling subsets, compute estimates. Other algorithms are based on the same structure, but have different parameters; FedYogi and FedAdam rely on the degree of adaptivity, which refers to how well the algorithms can respond to changes: having smaller values for their parameters indicate high adaptivity. In [44], it has been proven that the use of these other algorithms could lead to obtaining higher accuracy than the standard FedAvg algorithm.

4.5 Programmable switches

This thesis investigates the use of P4 programmable switches for aggregating model parameters, but they also used to perform other important functions: they perform NAT translation on packets directed to the server, multicast forwarding on packets directed to the clients, BPP Metadata extraction and manipulation, and checksum updating.

UDP sockets and port

Since communications are based on UDP sockets, BPP packets created on end nodes are encapsulated into UDP packets. While clients use just one UDP socket connection, bind on port 10000, both for sending and receiving packets, different socket instances are used on the Server. For client registration, packets from elected are all received on server port 6500, while a per-client dedicated port is used to receive model parameters. The Server Bridge uses the UDP socket instances for managing the communications. When the switch receives the packets, encapsulated in UDP, parses them considering the internal and external headers: the internal one is composed of the internal Ethernet and IPv4 headers, considered in the switch as a unique header called "internal", and the BPP block, containing the Commands and Metadata. Figure 4.5 shows the structure of UDP/BPP packets. Consideration

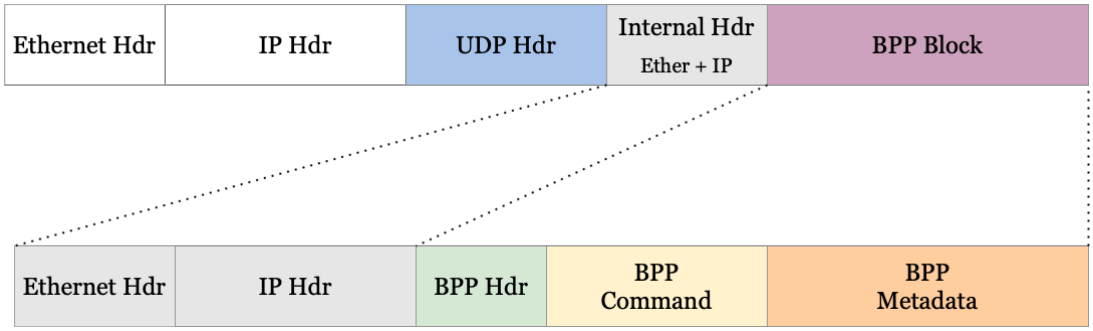


Figure 4.5: Encapsulation of BPP packet into UDP packet.

should be given to the choice of UDP sockets over TCP, which brings pros and cons since it is a packet-based, connectionless, best-effort service. As the server sees just one client per subnetwork, it sends just one request message, so the switch must replicate this message to all connected clients. Using TCP connections would make this impossible because it requires a one-to-one handshake to synchronize counters (sequence numbers) to ensure reliable transport. In other words, the use of TCP sockets requires the establishment of connections with all participating clients, losing all the advantages of aggregation, but avoids handling the first packet for registration, since the server registers the client when the TCP connection is established. The only way to perform packet multicast (or broadcast) forwarding is using UDP, compromising between the transmission reliability and aggregation benefits.

Multicast forwarding

Differently from packets coming from clients, that have a single destination, packets from Server need to be replicated and sent to all clients. In this operation, the role

of the switch is fundamental because the server only knows the registered client which, moreover has been registered with the fictional address. In fact, multicast was chosen over broadcast as the technique to spread packets to all clients because of its future applications; broadcasting is actually a technique that will disappear in future applications like IPv6, replaced by multicast. In order that messages could be received, a client needs to register to a specific multicast group: for this reason, each client, as it starts, registers to the multicast group with IP address `224.0.0.191`. Then, when the switch receives packets coming from the server, changes the fictional destination IP address with the one of the multicast group and replicates them on ports connected to clients.

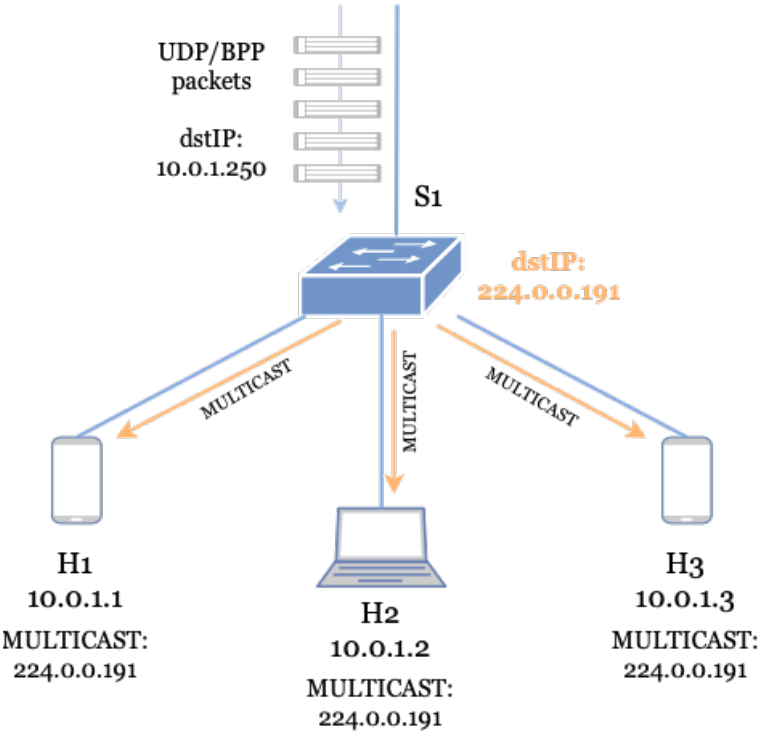


Figure 4.6: Multicast forwarding in switch S1.

NAT translation

One of the most important tasks performed by programmable switches is NAT translation: the switch changes the source IP address of all packets coming from the connected clients and directed to the server. This operation is necessary because in the FedAvg strategy, the global aggregation is based on the number of registered clients, so in the number of server connections. As the server only receives the aggregated results of training and evaluation, it has an opaque vision of the real

network topology; the internal structure is hidden behind the switch that changes the source IP address with a fictional one, of type `10.0.x.250/24` where x represents the number of the subnetwork.

Packets coming from clients can belong to two groups, registration or FL packets, and according to this, they are manipulated differently by the switch. The former are sent by elected clients at the beginning of the entire process to register to the server and, after the NAT translation, are always forwarded. The latter instead, are sent during FL rounds for the transmission of model parameters and could be dropped by the switch after the aggregation.

Parameter aggregation

Parameters aggregation inside the programmable switches is performed using data contained in the BPP Command block and BPP Metadata. A specific field in BPP Command, `BPP_Command.c2p1Value`, is used to instruct the switch on how to handle the packet. Packets created by the cclients have this field set to 1, indicating that an aggregation needs to be performed, the same field in Server packets is set to 0xbb, indicating that the packets need to be forwarded in multicast.

In order to perform aggregation, several limitations have been addressed, first of all, the presence of floating-point values. P4 language support only integer and bit operations, so it is not possible to directly perform aggregation on model parameters, which are usually floating-point values. To solve this limitation a scaling by a factor 10^8 was performed both on client and server, obtaining an integer result with an accuracy of 8 digits on one side and restoring the floating-point value on the other.

The second limitation we faced concerns the presence of parameters with a negative value. Although this might not be a problem, since such values might be represented in a2 complement within the Metadata, it may be inconvenient during aggregation in the switch. To overcome this problem, a specific value in the BPP block, `BPP_Command.c1p1Value`, was used to contain a mask. Each bit in the mask, corresponds to a specific metadata value, takes its value depending on the positivity of the parameter. Using this solution, BPP Metadata contains the parameters' absolute values and then on end nodes, based on the bit value in the mask, are eventually multiplied by -1.

Model parameters are organized in an N-dimensional array that in our specific case is a 3D-array that contains 221 total values. Representation of this array is given in Figure 4.7. The presence of BPP Metadata has been exploited to transport these values, but since each BPP Metadata block could contain at least ten values, the parameters have to be manipulated. In particular, on sending side, the array is flattened to have a 1D-array, then parameters are grouped in blocks of

10 elements, eventually padded with 0, and saved in BPP Metadata. On receiving side, parameters are extracted from BPP block, and once all have been collected, the initial 3D-array is rebuilt thanks to reshaping functions. It is important to note that in this way both sides need to have knowledge of the model structure since from this one depends the structure of the array.

Another reflection to be done is about metadata size: we used in our implementation the BPP standard definition that expects ten values of 64bit. It would be possible to change this definition, using little values in bit and increasing their number, for example using twenty values of 32bit. It is also possible to use multiple Metadata blocks in a single packet, thereby reducing the number of packets while increasing the number of parameters sent. Choosing the size of packets over the number of transported metadata values, we decided to use one Metadata block containing ten values. The last, but most important limitation that we addressed

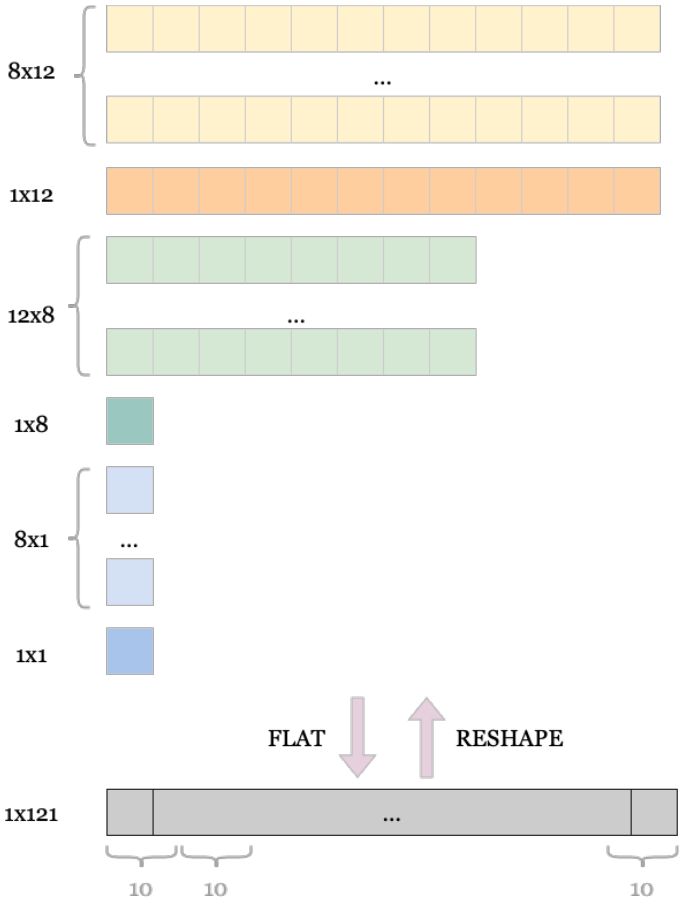


Figure 4.7: Parameters array structure of implemented model.

concerns the actual aggregation of parameters, and in particular the computation

of average.

P4 language does not support arithmetical division operations, so the average calculation becomes challenging. To solve this problem we tried two different approaches: since parameters in metadata are expressed in bit, we used registers to sum and save values in switches and we implemented a binary division function in order to compute average values. The algorithm used for binary division function is reported in A. Limitations of this solution are imposed by the language itself because of loops are not supported, so loop unrolling has been performed with consequent limitations in scalability. Handling a huge amount of data, as in FL applications, the solution is unfeasible so another solution has been proposed. P4 systems support *extern* functions as a set of methods that are not already implemented, but the programmer could provide custom implementations. We used a modified version of BMv2 in order to use extern functions [45], and we implemented a function for the collection and computation of parameters. In particular, we used three support vectors to collect the sum of values (sumValues), to keep track of received and processed packets (endedSum), and to index the next averaging operation (indexSum).

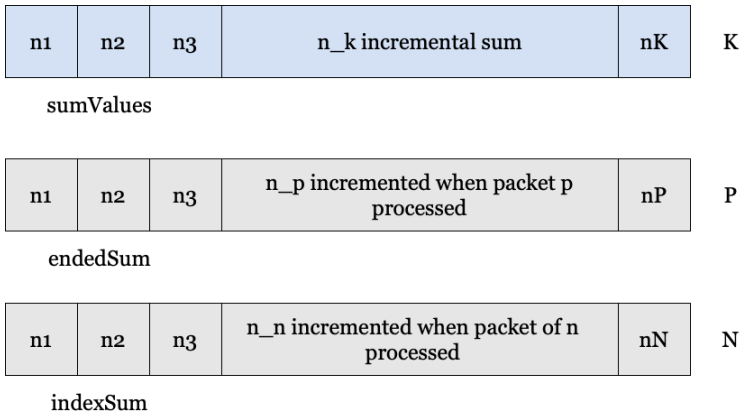


Figure 4.8: Support vectors in extern function used for aggregation. K is the number of model parameters, P is the number of packets and N is the number of hosts.

The mode of operation consists of packet reception and parsing by the switch; then metadata values are passed to the extern function where are summed and stored in the vector. The sum occurs in parallel in the sense that all packets with index p coming from different clients are summed together; each time a packet p is processed, the correspondent value in `endedSum` vector is incremented, so it is possible to know how many clients sent that specific packet. Once the value of `endedSum[p]` is equal to the number of connected hosts, a division operation is

performed and metadata in the packet are updated with the resulting values. It is important to underline that only the last packet will be forwarded with updated values, while others will be dropped after the sum operation.

Checksum updating

The version of BMv2 used in this thesis work, provides a function to update IPv4 checksum, but not the UDP checksum; to solve this lack, a custom implementation for checksum calculation and updating has been implemented. Since the UDP checksum takes into consideration a Pseudo-Header, the UDP header and the payload, that in our case is the encapsulated BPP packet, the proposed implementation adapts to the specific UDP/BPP format. Internal Ethernet and IPv4 header have been considered as a single header called *internal*, while BPP blocks are considered individually. The process, completely written in P4, performs additions and shifting of packet fields and at the end updates the UDP checksum field.

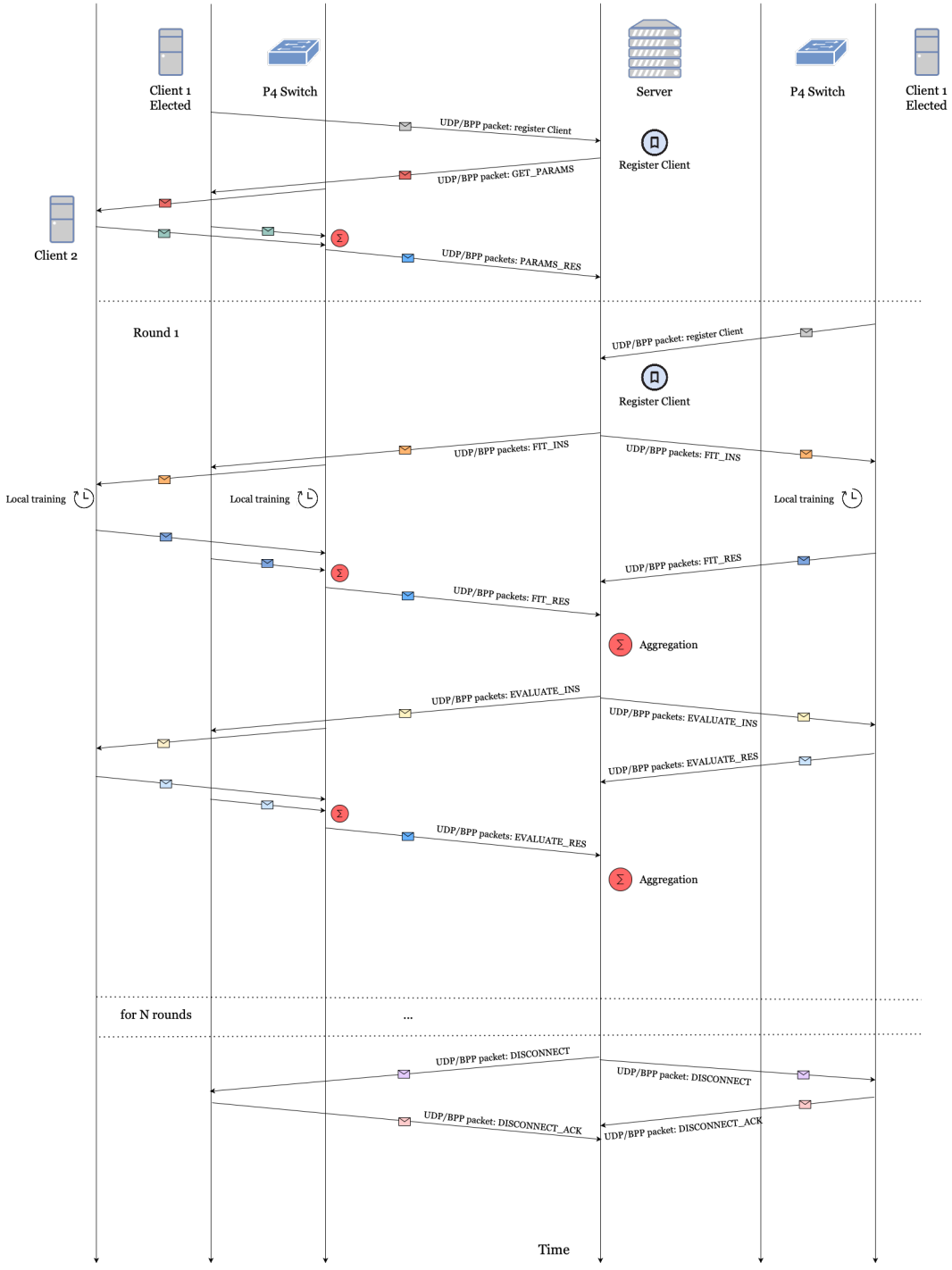


Figure 4.9: Flower communication protocol with UDP sockets. Registered Clients = 2, Client participating in FL process = 3

Chapter 5

Implementation details

In the chapter some implementation details are analyzed; particularly some code extracts and details on packet format are provided to give a more comprehensive treatment.

To achieve the goal of performing aggregation of model parameters in programmable switches, the main technologies used are: NewIP/BPP protocol, P4, a technology for the implementation of networking functioning, and Flower, a Federated learning framework. The proposed objective encountered a number of P4 limitations that proved very challenging to overcome so this work tries also to be an example for future implementations as well as for P4 evolution.

5.1 P4 Switch

The greatest challenge to overcome consisted of division operations, since P4 does not support them. To solve this limitation, we used a BMv2 version [45] that supports extern functions. Extern in P4 are objects, written in C++, with methods and attributes that allow increasing the functionality of the switch.

In `control BPPEgress`, that is the P4 program part that manages egress packets, the initialization of the extern is invoked.

Listing 5.1: P4 control Egress

```
control BPPEgress(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata) {
    [...]
    @userextern @name("custom_extern_instance")
    CustomExtern<bit<32>>(0x00) custom_extern_instance;

    // CLIENT REGISTRATION PACKET
    if (hdr.bpp_cmd.isValid() && hdr.bpp_cmd.a1Type == 0x2){
                topology_ed.apply();
```

```
9
10                  n_ed.read(nHosts, 0);
11                  custom_extern_instance.initNhosts(nHosts);
12
13                  bit<32> maskAddrMost = 0xffffff00;
14                  bit<32> nat = 0xFA;
15                  // takes the subnet address
16                  bit<32> addr = hdr.ipv4.srcAddr & maskAddrMost;
17                  addr = addr | nat;
18                  hdr.ipv4.srcAddr = addr;
19              }
20
21      [...]
22  }
```

Specifically at `line 4` and `5`, there is the construction of an extern object called "custom_extern_instance" and the initialization of one of its private attribute with the bit value 0x00. At `line x` instead, it is possible to see the initialization of the nHosts attribute: in order to compute the average value of model parameters, and perform this operation only once all the clients have sent their packet, the number of hosts is required. So when the elected client sends the registration packet, the switch initializes the nHost attribute of the extern object, which will be used both as a divisor in the average calculation and as a control to start the division. In the reported code is possible to see also, how the NAT translation is performed: a mask is applied to the source IP address to extract the network address, then the 8 least significant bit are replaced with the value 0xFA.

Listing 5.2: P4 control Egress

```
1  control BPPEgress(inout headers hdr, inout metadata meta, inout
       standard_metadata_t standard_metadata) {
2       [...]
3
4       if(hdr.bpp_cmd.a1Type == BPP_EVALUATE_RES || hdr.bpp_cmd.a1Type==
        BPP_FIT_RES || hdr.bpp_cmd.a1Type == BPP_PARAMS_RES){
5           [...]
6
7           bit<1> drop_bit;
8
9           // read and sum metadata in packet
10
11          custom_extern_instance.add10(hdr.bpp_md.data1, hdr.bpp_md.
       data2, hdr.bpp_md.data3, hdr.bpp_md.data4, hdr.bpp_md.data5, hdr.
       bpp_md.data6, hdr.bpp_md.data7, hdr.bpp_md.data8, hdr.bpp_md.data9
       , hdr.bpp_md.data10, id_host, drop_bit, hdr.bpp_cmd.c1p1Value );
12
13              if(drop_bit == 0){ // not drop
14                  // update address with NAT
```

```
15                  hdr.ipv4.srcAddr = addr;
16
17                  // NO AGGREGATION (in next hop)
18                  hdr.bpp_cmd.c2p1Value = 0;
19              }
20              if(drop_bit == 1){ // DROP
21                  _drop();
22              }
23          }
24  }
```

Listing 5.2 shows the invocation of extern method `add10` that represents the core of the parameters aggregation. This method takes as input various parameters: first of all the metadata values, then the host id, that corresponds to the least 8 significant bit of the source IP address, the drop_bit, used to discriminate if the packet has to be dropped or not, and the c1p1Value, that is used as a mask to identify negative numbers.

In listing 5.3 is reported the implementation of method `add10` written in C++.

Listing 5.3: Extern function

```
1  void add10(Data& num1, Data& num2, Data& num3, Data& num4, Data& num5
       , Data& num6, Data& num7, Data& num8, Data& num9, Data& num10,
       Data& hostId, Data& drop, Data& mask) {
2
3      int host_id = hostId.get<std::int64_t>();
4      int starter = indexSum[host_id-1];
5
6      if(sumValues.size() <= (uint)starter){
7        std::cout << "Resizing sumValues vector " << std::endl;
8        sumValues.resize((sumValues.size()*2), 0);
9      }
10
11     uint m = mask.get<std::int64_t>();
12     int64_t n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
13
14     n1 = num1.get<std::int64_t>();
15     // get other N values
16     [...]
17
18     if(((m >> 0) & 1) == 1){
19       n1 *= -1;
20     }
21     [...] // convert other N values
22
23     sumValues[starter] += n1;
24     sumValues[starter+1] += n2;
25     sumValues[starter+2] += n3;
```

```
26      sumValues[starter+3] += n4;
27      sumValues[starter+4] += n5;
28      sumValues[starter+5] += n6;
29      sumValues[starter+6] += n7;
30      sumValues[starter+7] += n8;
31      sumValues[starter+8] += n9;
32      sumValues[starter+9] += n10;
33
34      indexSum[hostId.get<std::int64_t>()-1] += 10;
35      endedSum[(starter/10)]++;
36
37      if(endedSum[(starter/10)] == n_hosts){
38        int count_mask = 0;
39        uint new_mask = 0;
40        for(int j = starter; j<starter+10; j++) {
41          sumValues[j] /= n_hosts;
42          if(sumValues[j] < 0){ // update mask
43            sumValues[j] *= -1;
44            new_mask = new_mask | (1 << count_mask);
45          }
46          count_mask ++;
47        }
48
49      // OVERWRITE METADATA
50        num1 = Data{sumValues[starter]};
51        num2 = Data{sumValues[starter+1]};
52        num3 = Data{sumValues[starter+2]};
53        num4 = Data{sumValues[starter+3]};
54        num5 = Data{sumValues[starter+4]};
55        num6 = Data{sumValues[starter+5]};
56        num7 = Data{sumValues[starter+6]};
57        num8 = Data{sumValues[starter+7]};
58        num9 = Data{sumValues[starter+8]};
59        num10 = Data{sumValues[starter+9]};
60
61        drop = Data{0}; // FORWARD
62        mask = Data{new_mask};
63      }
64      else{
65        drop = Data{1}; // YES DROP
66      }
67    }
```

To perform the aggregation operation, three support vectors, with overestimated sizes, are used. Considering the flattened vector of model parameters, vector `sumValues` contains at each position the sum of correspondent elements. Vector `endedSum` of size P equal to the number of total packets, contains at each position the number of hosts that sent the specific packet p. Finally, vector `indexSum`

of size equal to the number of clients N, contains at each position, the index in sumValues vector in which the operation of sum, and eventually division, will be performed for the host n.

At line 4, variable `starter` is initialized with the index in sum vector where the aggregation starts from. Then metadata values are extracted, eventually multiplied by -1 consistently to the mask passed in input and added to the vector. Values in `indexSum` and `endedSum` are updated, subsequently check on processed packets is executed. If the number of received packets is equal to the number of hosts, so all hosts have sent that specific packet, the division has to be computed. Final operations consist of metadata values update, and setting of drop bit: only packets containing aggregation results must be forwarded, the others instead will be dropped.

5.2 Client

Code on Client is quite simple and consists mainly on the definition of the model and the functions for the training and evaluation.

```
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')

# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2)

# define the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(8, input_dim=8, activation='relu'))
model.add(tf.keras.layers.Dense(12, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

# compile the  model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=
    ['accuracy'])
```

The dataset we used for the system implementation is the "Pima Indians Diabetes" [41]; the choice of this dataset is motivated only by its small dimensions that make easier the monitoring and the analysis of the implementation. The proposed solution is not constrained by either the model used or the dataset and can be used in any ML application.

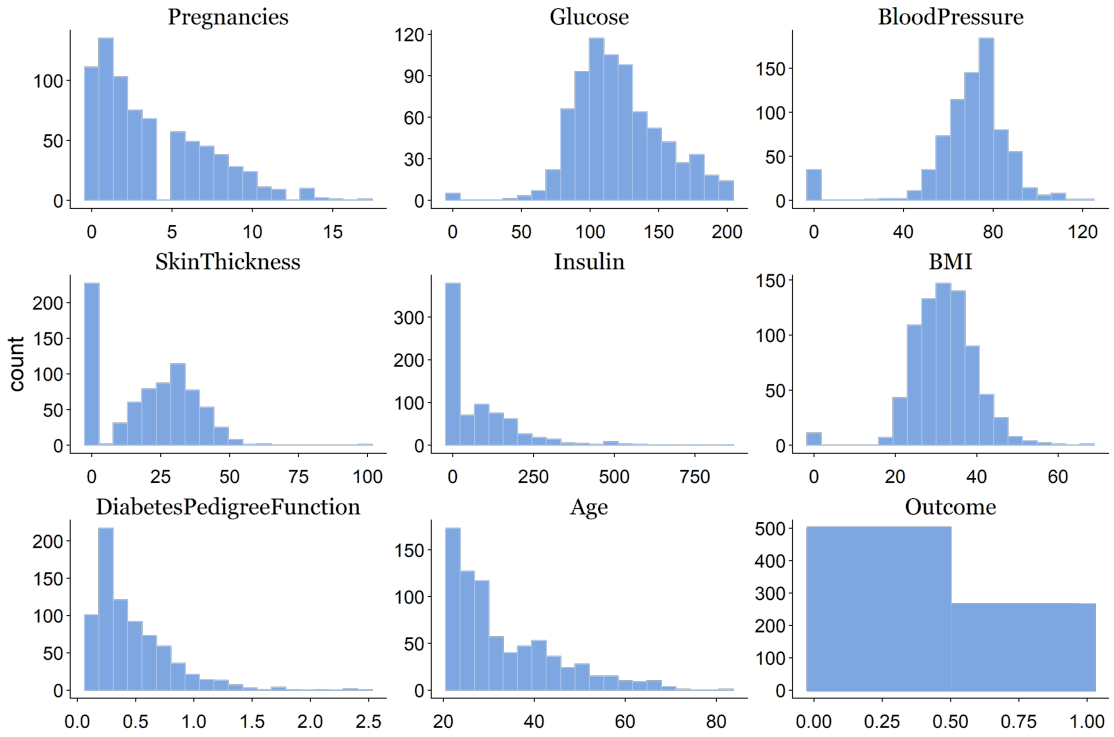


Figure 5.1: Pima Indians Diabet Dataset feature distribution.

Each client locally defines the ML model and prepares the dataset; we decided to use 80% of the dataset as training values, while the remaining 20% compose the evaluation set. To have clients working on different values, the construction of the sets was done randomly, but the model used is assumed to be the same for everyone.

Listing 5.4: Model fit and evaluation functions on client

```
def fit(self, parameters, config):  # type: ignore
    model.set_weights(parameters)
    # fit the model on the training set
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=
    batch_size)
    return model.get_weights(), len(x_train), {}

def evaluate(self, parameters, config):  # type: ignore
    model.set_weights(parameters)
    # evaluate the model on the test set
    loss, accuracy = model.evaluate(x_test, y_test)
    return loss, len(x_test), {"accuracy": accuracy}
```

5.3 Server

Implementations on server, mainly concern modification in client handling and communication protocol; in particular, the standard communication protocol of Flower framework was changed in order to support packet parsing and aggregation on P4 switches.

As it starts, the server creates N thread, one for each client, and waits until it receives a BPP packet for client registration. In this preparatory phase, also the Strategy and other values are initialized; we used in our implementation the FedAvg strategy, but others are possible. The strategy defines how the clients are managed and how parameters are aggregated, in our case the aggregation consists of the calculation of the mean.

Listing 5.5, 5.6 and 5.7 show the code in Flower server to perform parameters collection and aggregation in a round of fit.

Different clients are managed separately using different threads and the concept of future is exploited to gain results of computations. Each server thread takes care of sending the request through the Socket Bridge, waiting for packet response, extracting values from packet and reshaping parameters into the original vector. Once all threads return, the union of results is computed through the aggregate function defined in the strategy.
In FedAvg strategy the aggregate function calculates the number of examples used during training, create a list of weighted weights and then computes the average weights of each layer.

Listing 5.5: fit_clients function

```
def fit_clients(client_instructions: List[Tuple[ClientProxy, FitIns
    ]]) -> FitResultsAndFailures:
    """Refine parameters concurrently on all selected clients."""
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [
            executor.submit(fit_client, c, ins) for c, ins in
    client_instructions
        ]
        concurrent.futures.wait(futures)
    [...]
```

Listing 5.6: fit_round function

```
[...]
aggregated_result: Union[
    Tuple[Optional[Parameters], Dict[str, Scalar]],
    Optional[Weights],
    ] = self.strategy.aggregate_fit(rnd, results, failures)
```

Listing 5.7: aggregate function

```
def aggregate(results: List[Tuple[Weights, int]]) -> Weights:
    """Compute weighted average."""
    # Calculate the total number of examples used during training
    num_examples_total = sum([num_examples for _, num_examples in
    results])

    # Create a list of weights, each multiplied by the related number
     of examples
    weighted_weights = [
        layer * num_examples for layer in weights] for weights,
    num_examples in results ]
    # Compute average weights of each layer
    weights_prime: Weights = [
        reduce(np.add, layer_updates) / num_examples_total
        for layer_updates in zip(*weighted_weights) ]
    return weights_prime
```

A special comment must be made on the number of clients set in the strategy. FedAvg strategy takes as input the minimum number of clients participating in initialization, training and evaluation and the server will remain in a waiting state until the number of registered clients will reach this minimum value. This value takes on a different meaning in the standard version of Flower and in our modified version: in case of no aggregation, the minimum number of clients corresponds to the number of effective clients participating in the FL process, while in case of aggregation, this number corresponds to the number of P4 switches that perform aggregations.

5.4 Communication protocol

The major change in Flower implementation concerns the communication protocol. The standard version of the framework uses gRPC connections for client-server communications and transmits model parameters inside protocol buffer messages. At the time this thesis work began, the implementation of BMv2, the standard behavioural model for P4 systems, did not fully support gRPC, so the communication protocol has been changed and UDP sockets replaced gRPC connections. It is important to note that the development of BMv2 and P4 language is continuously evolving and in the near future different implementations could arise.

5.4.1 Packet format

Figure 5.2 represents a condensed representation of the packet format used in the proposed solution. Packets generated on end devices, both clients and server,

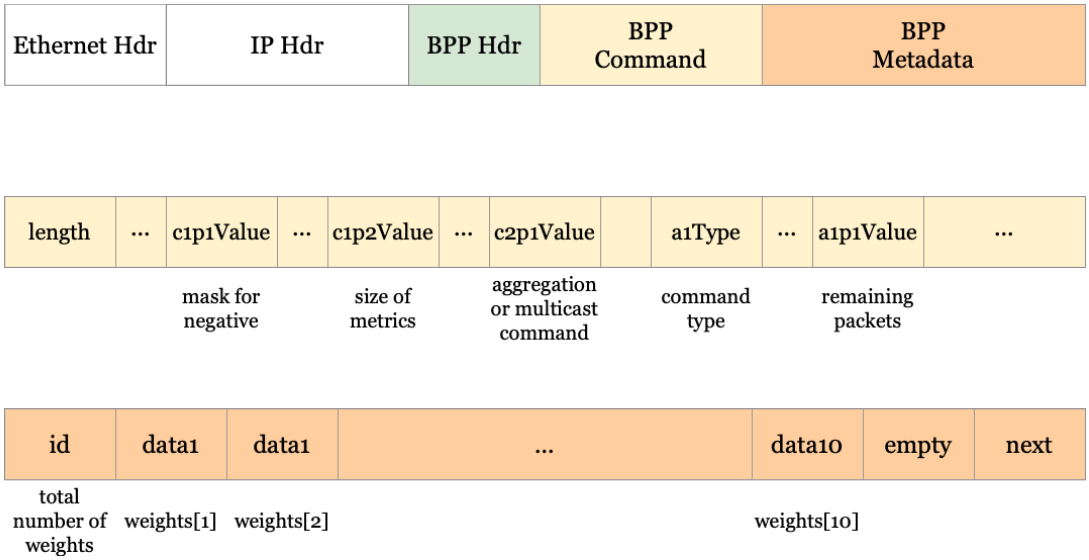


Figure 5.2: Packet format.

consist of an Ethernet header, an IPv4 header, and a BPP block. The BPP block is itself composed of a BPP header, a BPP Command block and a BPP Metadata block. In order to perform switch aggregation, and instruct both intermediate network nodes and end nodes on how to manage packets, fields in BPP command assume specific values. For example, field `c1p1Value` contains a mask on ten bit used to identify negative numbers. Representation in a2 complement is supported in P4 language, but it makes more difficult the aggregation in switches, therefore on end devices is created a mask of ten bits in which each bit indicates if the correspondent parameter in metadata assumes a negative value or not. Another BPP Command field used to transport useful information is `c1p2Value` that contains the size of metrics; this value will be used by the strategy to calculate aggregation of parameters and so the global model.
When an intermediate P4 switch receives a packet, it handles it according to the destination: packets directed to the server must be aggregated, while packets directed to clients need to be forwarded in multicast. To achieve this behaviour field `c1p1Value` assumes different values: 0x1 for aggregation, and 0xbb for multicast. It is important to note that in the proposed implementation, switches directly connected to clients perform similar operations while the switch connected to the server acts in a different way. More in detail, switch S4, presents the behaviour of a traditional switch, since it does not perform any aggregation but has only to forward packets from and to the server; for this reason, the value of `c1p1Value` is changed along the path. In packets directed to the server, the initial value 0x1, indicating that an aggregation has to be performed, is changed to 0x0 after the first

hop, thus switch S4 will not perform aggregations. In packets directed to clients instead, the initial value 0xbb, is changed inside S4 switch into 0xff, indicating that forwarding has to be performed in multicast in the next hop.
BPP Command field `a1Type` is used to transport information about the type of message, for example FIT_INS or FIT_RES. This values is used on end devices to discriminate the stages and operation inside the FL process.
`a1p1Value` contains the number of remaining packets, used in while loop at receiving. In conclusion, `data` values in Metadata Block, are used instead for model parameters, while `id` is used to indicate the number of total number of weights, used in the vector reshaping operations.

5.4.2 UDP socket

A final note concerns UDP sockets and port. Since packets directed to the clients, need to be forwarded in multicast, UDP is the only possibilty, as TCP provides connection only between two endpoints.
In the proposed solution, clients creates a UDP sockets bind on port 10000 both to send and receive packets, while server uses different port for handling registration and FL packets. Specifically, server creates a UDP socket bind on port 6500 to receive registration packet: this socket keep tuned for new registrations that may arrive at any time. When a new registration packet arrives, a dedicated novel socket is created and bind on a port derived from the source IPv4 address of the received packet; this per-user dedicated socket will be used for sending and receiving FL packets.

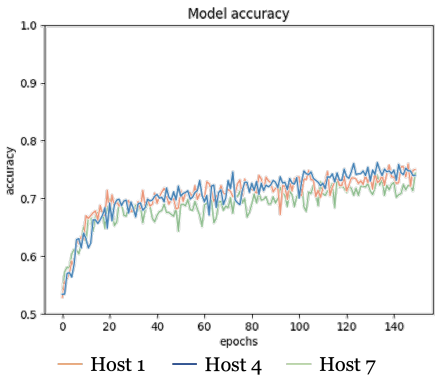
Chapter 6

Experimental results

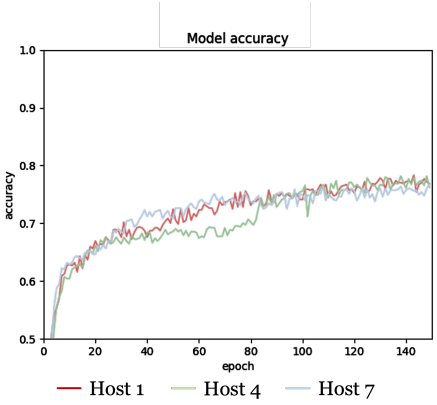
Simulations of the presented solution were performed in an emulated environment; in particular we emulated the behaviour of the proposed network topology using Mininet launching client and server scripts in different terminals. Because of the virtualized nature of these simulations, results are influenced also by the hardware equipment of the machine where they have been conducted. The machine in question presented 20GB of RAM and 4 processors.

Table 6.1 shows a summary of obtained results in terms of training time, network traffic, and final global accuracy and loss.
A special comment must be made about the obtained results. We conducted different simulations in order to compare performances of the system with and without aggregation but to have relevant values two cases have to be explored that differ in the number of clients registered to the server.
The first case takes into consideration an equal number of clients registered to the server. This value, set to three, in the case of aggregation performed in intermediate switches, corresponds to the number of programmable switches. Conducting experiments with this hypothesis we obtained a notable result about accuracy: in a system with aggregation performed in intermediate switches, the accuracy is slightly higher than in the case without aggregation, as well as the loss that results lower in the case with aggregation. This result can be explained by the fact that, even if the number of Clients seen by the Server is equal in both cases, in the first case the real number of participating clients, hidden by the switch, is higher, so it is possible to better train the model. Results about network traffic, are similar in both cases since the size of exchanged packets and their amount is fixed and depends on their structure and the number of rounds.

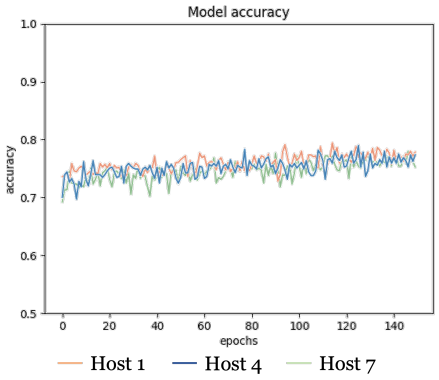
The second case takes in consideration a equal number of client participating in the FL process, which is equal to eight. In this scenario, a significant result is the one about network traffic: having eight Clients registered to the Server, it is



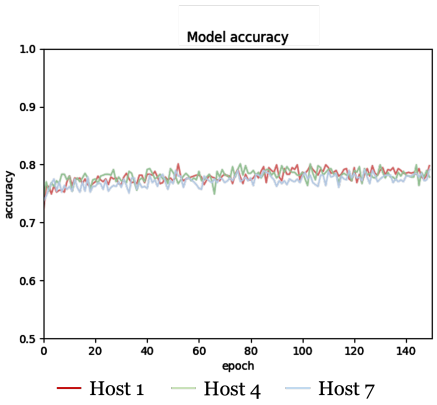
(a) Round 1



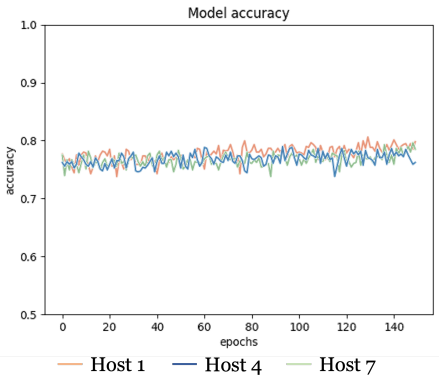
(b) Round 1



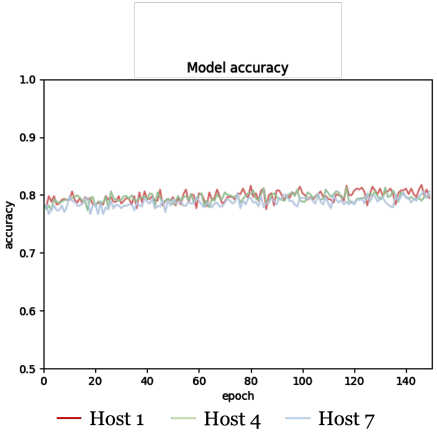
(c) Round 2



(d) Round 2



(e) Round 3



(f) Round 3

Figure 6.1: Accuracy trend on host H1, H4, H7 in system with (on left) and without (on right) aggregation. Rounds set to 3 and epochs set to 150.

possible to see that in the case without aggregation, the traffic through the link connected to the Server is much higher than in the case with aggregation. This is because, in the former each client will communicate directly with the Server, instead in the latter, only one connection per switch is present. This allows reducing the traffic by a factor of S/N where S is the number of switches and N is the number of Clients in the process. Results about accuracy and loss are quite similar in both cases and are respectively 81.35% and 0.41 in the case of aggregation, and 79.2% and 0.45 in the case without aggregation.

The final measurement concerns the training time that is quite similar in both cases and differs from our expectation, in which with the switch aggregation should be smaller. The term training time is intended for the time spent for the entire FL process, so consists of both training and evaluation time during different rounds. In particular, we obtained a training time of 280.76 in case of aggregation and 273.45 in the case without aggregation. This result may be influenced by the virtualized nature of the experimentation system, the specific implementation of BMv2 that we used, and the limited amount of parameters in the model. In order

Metric	**Flower without Aggregation**	**Flower with Aggregation**
Training time [s]	273.453**	280.761*
Network traffic [B]	423 072**	163 984*
Global Loss *	0.4776	0.4105
Global Accuracy * [%]	75.97	81.35

Table 6.1: Comparison between experimental results. Number of rounds=3. *Number of registered clients = 3; **Number of registered clients = 8.

to better represent a real scenario, we performed also measurements changing the parameters of the link. Specifically, changes in the link between S4 and the server have been performed in order to have a bandwidth equal to 50Mbps and a delay of 20ms. Measurements of training and evaluation times during rounds and in different clients are reported in table 6.2.

Measures have been made also on server; in particular we obtained accuracy and loss values of global model in order to show how aggregation in switches changes results. In table 6.3 and figure 6.2 are reported values during ten rounds of FL.

Comments about model settings

The proposed system uses on clients a custom model composed of 2 hidden layers and that allows to achieve a maximum value of accuracy of around 82%. Analyzing the measured values, it is possible to notice how the value of accuracy increases

Host	Round 1 (TR)	Round 1 (EV)	Round 2 (TR)	Round 2 (EV)	Round 3 (TR)	Round 3 (EV)
H1	50.07	1.13	39.80	2.18	42.52	40.43
H2	49.85	1.36	41.52	0.47	38.41	44.56
H3	50.47	0.74	38.62	3.33	42.73	40.22
H4	44.15	1.46	41.12	0.94	42.19	40.66
H5	53.33	1.78	41.13	0.83	82.50	0.47
H6	53.67	1.09	39.93	1.97	40.29	42.76
H7	44.45	0.74	41.68	0.38	41.35	41.52
H8	55.18	0.65	34.78	7.21	40.40	42.33

Table 6.2: Times on Clients with delay=20ms and bandwidth=50Mbps on link between server and S4. (TR): training time equals to the time between the reception of one server fit request message and the transmission of client response. (EV): evaluation time equals to the time between the reception of one server evaluate request message and the transmission of client response.

Round	Loss	Accuracy [%]
1	0.478444	77.09
2	0.498709	79.83
3	0.422688	79.94
4	0.406568	80.59
5	0.397436	81.56
6	0.389746	81.71
7	0.389897	81.17
8	0.385915	81.93
9	0.378216	83.15
10	0.376298	82.43

Table 6.3: Aggregated Accuracy and Loss values on Server in 10 rounds.

as the number of clients participating in the training increases. Results show also how model settings can influence network traffic and training time. Most of the simulations have been performed using an epoch value equal to 150 and a batch size of 10. Epochs represent the number of times when an entire dataset is passed forward and backward through the neural network only once, while the batch size is the total number of training examples present in a single batch.
We conducted experiments changing the epoch setting and number of rounds to analyze how accuracy and training time changes. Results of these experiments are reported in table 6.4 and represented in figure 6.3.

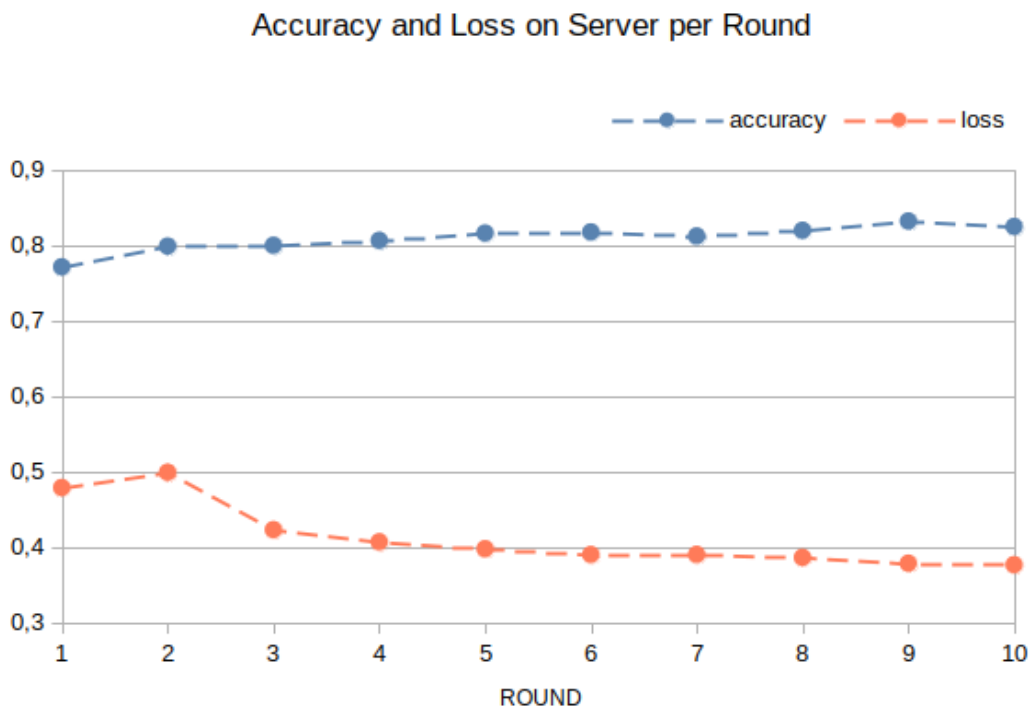


Figure 6.2: Aggregated Accuracy and Loss trend on Server in 10 rounds.

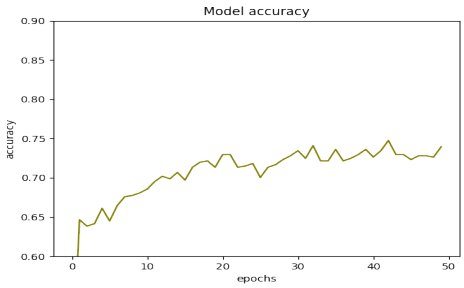
The number of rounds was chosen proportionally to the number of epochs. It is possible to see how increasing the number of epochs the total FL process time decreases, while the average round time increases. Another reflection has to be made about network traffic: increasing the number of epochs and decreasing the number of rounds, the message exchange between clients and server decreases, thus the bottleneck effect. This behaviour can be seen from the results in which with epochs equal to 50, rounds equal to 20, and considering a number of registered clients equal to 3, the amount of traffic is equal to 665.440KB, while setting epochs equal to 500 and rounds equal to 2 we obtain a total amount of traffic equal to 66.54KB. From this point of view, is preferable to have a larger value of epochs and a little value of rounds.
Another consideration has to be made about accuracy and loss of model at the epochs change. Figure 6.3 shows the accuracy trend at epochs change in the first round. With epochs equal to 50, at the end of the first round, the accuracy value reaches the value of 74.3% while at the end of the same round, but with epochs set to 500, an accuracy value of 79.8% is reached. This result leads to the choice of using bigger values of epochs and fewer rounds.

Epochs	**Rounds**	**Total FL time [s]**	**Average round time [s]**
50	20	427.20	21.36
100	10	387.99	38.79
150	7	552.98	78.99
200	5	395.5	79.1
250	4	341.0	85.25
300	3	254.84	84.94
500	2	292.91	146.45

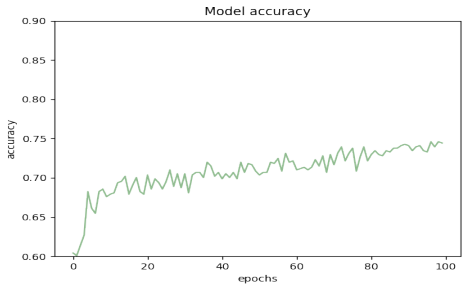
Table 6.4: Total and average times measured in clients at the variance of epochs.

Comments about virtualization

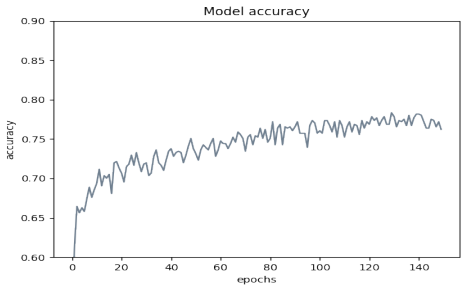
All simulations were performed in an emulated system and this may introduce some latency and resource limitations. Experiments performed on other machines, equipped with fewer resources, led to delays in the training process and more important to packet loss caused by switch resource saturation.
Possible future experimentation might be to try the proposed solution with more complex ML models, and if possible try the solution with a real network and programmable switches.



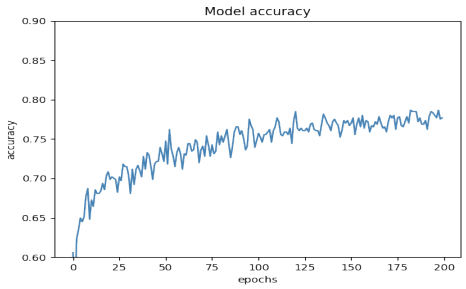
(a) 50 epochs



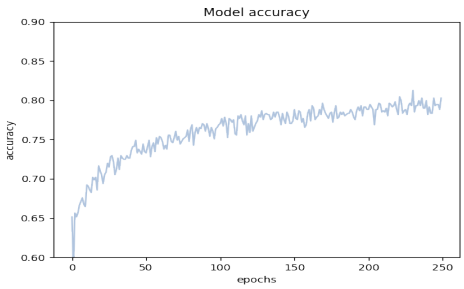
(b) 100 epochs



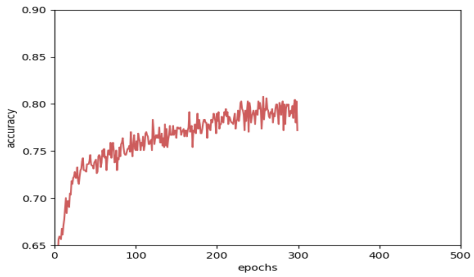
(c) 150 epochs



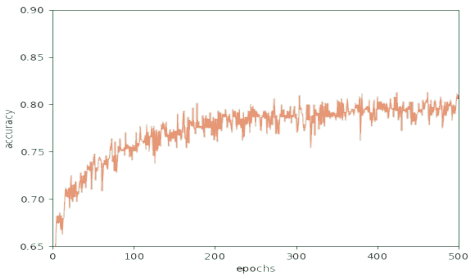
(d) 200 epochs



(e) 250 epochs



(f) 300 epochs



(g) 500 epochs

Figure 6.3: Sensitivity analysis of accuracy on host H1 at the variance of epochs.

Chapter 7

Conclusions

The aim of this thesis work was to exploit the potentialities of programmable switches to accelerate the model training in Federated Learning processes. Specifically, we used P4 switches to compute intermediate aggregations and reduce network traffic, thus alleviating the bottleneck effect on the central FL Server and further accelerating the entire training process. Practically, the standard behaviour of network switches has been modified in order to support gradient aggregation, moreover, extern function support and UDP packet handling have been added to the system.
We conducted experiments using a Mininet emulated network topology, composed of a central server, intermediate P4 switches performing aggregations, and different clients participating in the FL process. With the proposed solution we obtained that the network traffic was reduced by a factor of S/N where S is the number of switches and N is the number of clients in the process. Another notable result is the one about accuracy that is slightly higher than in the traditional FL approach; this can be explained by the fact that the real number of participating clients, hidden by the switches, is higher, so it is possible to better train the model. Lastly, considering the training time for both solutions, we can observe how the aggregation introduces some overhead brought by the P4 additional computation. However, this result is influenced by the virtualized nature of the experimentation system, the specific implementation of BMv2 that we used, the limited amount of parameters in the model, and the absence of challenging network conditions. Despite these factors, the difference between the two solutions is quite negligible, and, along with other metrics, demonstrates the validity of our solution.

7.1 Alternative implementations and enhancements

The proposed solution relies on some hypotheses and implementation details that influenced the final results. In this section, we want to give some ideas for future implementations and improvements.

Dataset size and model

In order to have a testable system, both in terms of times and size of parameters, we choose the "Pima Indians Diabetes Dataset" and a neural network composed only of two hidden layers. Definitely, possible improvements could consist in the enhancement of the training process, for example using EarlyStopping callbacks, in order to reduce the training time, or to increase the number of epochs, since it has been seen that with a large number of epochs, it is possible to achieve a similar number of accuracy working on few FL rounds and to reduce the amount of network traffic.
It would be interesting also to try this solution with different datasets, consisting of many more entries and with more features; possible examples may be CIFAR10[46] and MNIST dataset [47] that have been used in other studies so that comparisons of results can be made.
Because model parameters are sent flattened in BPP Metadata, in order to use other datasets, reshape and flat functions in the custom Flower framework must be adapted to fit the shape of the array parameters.

BPP block structure

An interesting variation concerns the size of BPP blocks that could influence both the FL process time and the amount of network traffic. More in detail, in our implementation we used the standard definition of BPP protocol and block structure, which consists in different headers with fixed size. In particular the Metadata block, contains ten 64B values, thus the number of transmitted packet at each round is equal to the number of parameters divided by ten. Changing the size of metadata values, make it possible to encapsulate more values inside the block and so reduce the amount of packet sent.
Further improvement consist on the queuing of multiple metadata blocks to create larger packets and reducing the amount of Ethernet and IPv4 headers with a consequent reduction in total network traffic.
All these improvements would be possible because the definition of the BPP protocol is all in software: using P4 language in switches and in python inside Flower framework.

Division operations and extern

To perform divisions inside P4 switches, we used extern objects written in C++. We did not perform measures about latencies introduced by this technology but it might be an interesting study. Certainly, the virtualized environment in which we tested the solution introduced some delay but we have reason to believe that the presence of externals also impacts in performance.
Future studies may focus on researching an alternative solution to perform division in P4 systems, as well as on performing these operations using specialized hardware implementations.

Simple switch and gRPC

The stable version of BMv2, at the time this thesis work began, did not fully support gRPC connections, so we opted for implementing the system using the simple_switch target and using UDP socket connections for communications. The use of UDP, necessary because of the presence of multicast traffic, brings synchronization and reliability problems since it provides best-effort behaviour and does not offer support for packet loss. Possible future enhancement of the proposed solution could concern the implementation of reliability mechanisms in order to reduce the quantity of packet loss that proved tragic in our experiments: if one packet is lost in transmission, the end device cannot rebuild the original message, so the entire process stops.
During these months, new implementations and changes in behavioural model arisen so it would be interesting to change the current implementation of the proposed solution trying to use gRPC connections and the standard Flower framework.

Figure 7.1: Barefoot networks: computational fabrics

Virtualized system vs Real system

In conclusion, since all experiments have been performed in a virtualized Mininet environment, it should be interesting to measure training time and especially

communication time in a real network, using programmable switches. For example, it may be used programmable switches equipped with Intel Tofino, an ASIC switch Ethernet specifically created to be programmed with P4.

7.2 Future prospects in In-network computations

The field of programmable switches and In-Network computations is constantly evolving and proposes different challenges.

Figure 7.2: Challenges and future trends in programmable switches.

Memory capacity

Programmable data planes benefit greatly from stateful processing, as it allows applications to store and retrieve data across packet boundaries. This facilitates a

wide range of new applications (e.g., in-network caching, fine-grained measurements, stateful load balancing, etc.) that were not previously possible in non-programmable networks.
The amount of data stored in the switch is limited by the size of the on-chip memory which ranges from tens to hundreds of megabytes at most. Consequently, the majority of stateful-based applications suffer have trade-offs between performance and memory usage. The size of the on-chip memory of the switch, which ranges from tens to hundreds of megabytes at most, limits the amount of data that can be stored. Because of this, most stateful-based applications suffer from memory and performance trade-offs. Current and future initiatives focus on expanding the available memory on the switch for example using external DRAM.

Resource accessibility

In addition to the on-chip memory size limitation, the data plane developer should be aware of other limitations. First, since the table memory belongs to each stage in the pipeline, the other stages cannot reclaim unused memory from the previous stage. Therefore, memory and match/action processing are fuzzed, which makes placing tables difficult; additionally, sequentially executing operations in the pipeline results in less efficient utilization of resources, particularly when matches and actions are imbalanced.
Interesting researches are exploring centralizing memory into a pool accessed by a crossbar, as well as creating a cluster of processors that can perform arbitrary operations in any order. The disaggregation model should be implemented on hardware targets and analyzed in the future.

Arithmetic Computations

The implementation of arithmetic computations in the data plane poses several challenges. First, there are only a few simple arithmetic computations that can be implemented using programmable switches.vSecondly, only a limited number of operations are supported per packet so line rate execution can be guaranteed; usually, a packet will spend only a few nanoseconds in this pipeline. As a third problem, computations in the data plane consume a number of hardware resources, making it impossible for other applications to run concurrently. This affects a wide variety of applications due to the lack of complex computations in the data plane. For instance, some operations required by AQMs (e.g., square root function in the CoDel algorithm) are complex to be implemented with P4. Additionally, the majority of machine learning frameworks and models work on floating-point values, as well as In-network model updates aggregation requires calculating the average over a set of floating-point vectors, whereas the arithmetic operations on the switch are intended to work with integer values. The current method for overcoming

computation limitations are approximation and pre-computation. In the former, the designer attempts to approximate the desired value by using the few set of supported operations, at the expense of precision; the pre-computation method stores values in match-action tables or registers using computations performed at the control plane (e.g., switch CPU).
Future work can focus on identifying the computations that can be pre-evaluated in the control plane and for creating data plane code and APIs for the control plane based on these identifications. Moreover, P4 developers would be beneficial if they had access to a community-maintained library, which contains P4 code emulating various complex functions.

Control plane intervention

When users delegate tasks to the control plane, the latency and performance of the application are affected. For instance, rerouting-based schemes often use tables to store alternatives routes in congestion control. Because the data plane cannot alter table entries directly, control plane intervention, is required but it would be ideal to minimize this intervention whenever possible, for example, to synchronize the state among switches. Developers have full control over the design of the interaction between the control plane and the data plane and have to try to minimize it.
Research should be carried out on designing algorithms and tools for identifying excessive interactions between the control and data planes and suggesting alternatives (ideally as generated code) to minimize such interactions.

Security

An attacker can intelligently craft a traffic pattern to cause a system to behave unexpectedly, so designers of data plane systems must anticipate the kind of traffic that might damage the system. If, for example, a load balancer is configured to balance traffic through packet header hashing without cryptographic support, it can be tricked by an attacker by creating skewed traffic patterns; another example is attacks against in-network caching. As long as reads dominate data plane requests, caching performs well. If a constant stream of write requests is continuously generated, the load on the storage servers would be imbalanced. A random failure in the switch can cause data to be lost if the system is designed to handle write queries on hot items. Additionally, an attacker can also exploit the memory limitation of the switch and request diverse values, leading to the eviction of pre-cached values.
Recent researches aimed at automatically discovering sensitivity attacks in the data plane trying to derive traffic patterns that would drive the program away from common case behavior as much as possible. Work in this direction should focus on formalized verification of the codes to achieve high assurance. Moreover, the

stability of the data plane should be managed carefully with fast mode changes; future work might integrate self-stabilizing systems for this purpose. Finally, future work should provide security interfaces for collaborating switches that belong to different domains. Lastly, it would be helpful if future work provided security interfaces for collaborating switches that belong to different domains, furthermore, it would be useful to expose different attack patterns for different application types so that data plane developers can avoid the vulnerabilities.

Interoperability

It has been shown that applications that offload their processing logic to the network can achieve significant improvements in performance. Programmable switches, in particular, are especially useful for in-network applications but n spite of such facts, it is very unlikely that mobile operators will replace their current infrastructure in one shot with programmable switches due to significant operational and budgeting costs.
The best approach is that network operators deploy programmable switches in an incremental fashion, which means that P4 switches will be added to the network alongside the existing legacy devices.
In the future, P4 switches could be positioned to support applications such as in-network caching, accelerated consensus, and in-network defense, taking into account the current topology that has legacy devices. Additionally, recent research is considering the use of network taps as a means to replicate production network traffic to programmable switches for analysis; by tapping on legacy devices and processing on P4 switches, operators can benefit from the capabilities of P4 switches without the need to fully replace their current infrastructure. This method can be used in a variety of in-network applications like network-wide telemetry and DDoS detection/mitigation.

Programming simplicity

The P4 programming language is not easy to learn; recent studies have shown many of the existing programs have several bugs that may cause the network to go down completely. Furthermore, since programmable switches have many restrictions on memory and the availability of resources, developers must take into account the low-level hardware limitations when writing the programs. It is known that this process is based on trial and error; developers are rarely sure if their program will "fit" into the ASIC, so they keep compiling and adjusting their codes accordingly. Such problem makes evident when the complexity of the in-network application increases, or when multiple functions are executed concurrently in the same P4 program. Additionally, code modularity is not simple in P4 and the programmers typically rewrite existing functions adjusting them for their own purposes. All

these factors affect the network's cost, stability, and correctness in the long run. Until now, the networking industry operated on switches equipped with fixed-function ASICs, so few programming skills were needed by network operators, but with the advent of programmable switches, operators are now expected to have experience in ASIC programming. In view of the fact that this is a challenging task, future research efforts should consider simplifying the programming workflow for the operators and generating code, for example, graphical tools could be developed to translate workflows into P4 programs.

Modularity and virtualization

It's evident that today's networks require operators to run multiple network functions simultaneously on a single physical switch, despite programmable data planes originally intended to execute a single program at a time.
With the increase in demand for on-switch networking functions, cloud providers are trying to offer them as services. To do this several challenges must be addressed including resource isolation, performance isolation, and security isolation. Programming for P4 programs and functions should be made more modular so that developers can easily integrate multiple hardware services into the hardware pipeline. There are currently attempts to implement data plane virtualization in the literature. Among the challenges that can be explored further is performance degradation from packet recirculation, inflexibility of live reconfiguration, frequent recompilations, and loss of state during data plane reconfiguration.

Practical testing

Verifying the correctness of novel protocols and applications in real production networks is of utmost importance for engineers and researchers. Due to the ossification of production networks (which cannot run untested systems), engineers typically rely on modeling and mimicking the network behavior in a smaller scale to test their proof-of-concepts. One way to model the network is through simulations; while simulations offer flexibility in customizing the scenarios, they cannot achieve the performance of real networks since they typically run on CPUs. Another way to model the network is through emulations. Emulators run the same software of production networks on CPU and offer flexibility in customization; however, they produce inaccurate measurements with high traffic rates and are bound to the CPU of the machine. Finally, emulating testbeds on a smaller scale might produce results different than production networks. Future Initiatives could focus on finding tools that emulate production networks at scale while achieving line-rate performance.

After all these considerations and having analyzed future prospects and possible improvements, the proposed solution proves valid and wants to encourage researchers to extend and develop this work. It has been demonstrated how FL is highly suited for EC applications, as it can take advantage of the processing in intermediate programmable switches and the highly distributed edge devices generating data. Moreover, this thesis work represents an example that can be followed in the implementation of P4 programs, since contains solutions to various challenges proposed by the system.

Appendix A

Division in P4

During the development of this thesis work, different approach has been explored in order to make possible the arithmetic division operation inside the programmable switches. Before implementing these operations using extern functions, we tried to implement division using binary operations. Due to P4 language and resource limitations, this solution works well only if applied on few numbers but by increasing the amount of numbers, the compilation time becomes untenable. The main cause lies in the lack of support of loops in P4 language, so the implementation of multiple division operations relies on loop unrolling technique.
Figure A.1 shows the algorithm used for bit division operations and consists in binary shift and sum, while listing A.1 shows an extract of the P4 code implementation. For our purpose, the implemented solution works on 32 bit values, but the support of larger number could be possible with some modifications in code.

Listing A.1: Binary division implementation

```
bit<32> divide(in bit<32> dividend, in bit<32> divisor){
    [...]
    // construction of intermediate result AQ
    AQ = AQ | (bit<64>)Q;
    Ashift = shiftLeft32ByN(n1, A);
    AQ = AQ | Ashift;

    Ba2 = complementA2(n1, divisor);

    // #STEP 1
        if(Ncycle > 0){
            AQ = AQ<<1; // SHIFT
            AQtmp = AQ;
            // new A
            Ashift = AQtmp & maskA64;
            Ashift = shiftRight64ByN(n1, Ashift);
            Atmp = (bit<(32+1)>) Ashift;
```

```
18              A = (bit<32>) AQtmp;
19              Atmp = (bit<(32+1)>)Atmp + Ba2;      // A-B
20              Atmp = Atmp & (bit<(32+1)>)maskA32;
21
22              check = checkBit1((n1+1), Atmp);
23
24              if(check == 1){
25                  AQ[0:0] = 0;
26                  Atmp = (bit<(32+1)>)A;      // A+B. Restore previous
    value of A
27              }
28              else{
29                  AQ[0:0] = 1;
30                  // put new value of A in AQ
31                  A = (bit<32>) Atmp;
32                  A = A & maskA32;
33                  Ashift = (bit<64>) shiftLeft32ByN(n1, A);
34                  AQ = AQ & maskQ;
35                  AQ = AQ | Ashift;
36              }
37              Ncycle = Ncycle -1;
38          }
39      [...]
40  }
```

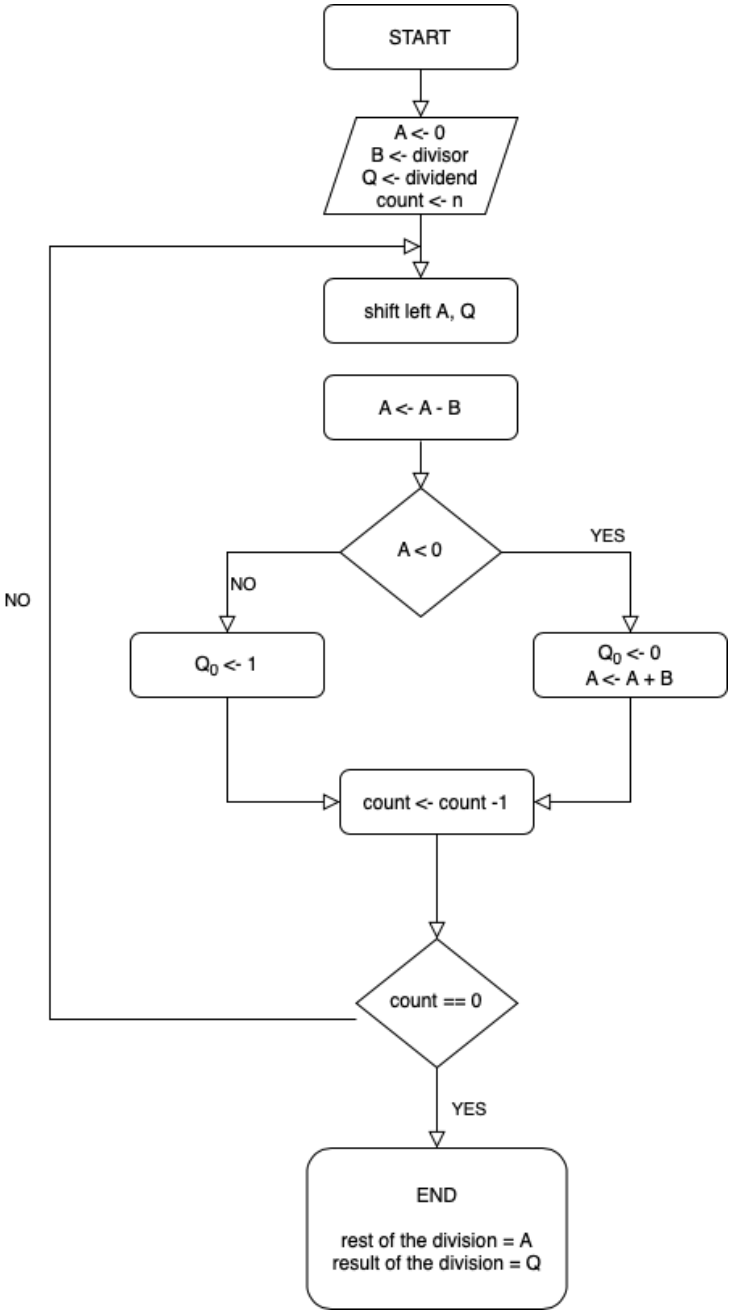


Figure A.1: Division flow chart

Bibliography

[1] *The Future of IoT Miniguide: The Burgeoning IoT Market Continues*. URL: https://www.cisco.com/c/en/us/%20solutions/internet-of-things/future-of-iot.html (cit. on p. 13).

[2] *5 Applications of Federated Learning*. URL: https://www.hitechnectar.com/blogs/applications-of-federated-learning/ (cit. on p. 15).

[3] George Pu, Yanlin Zhou, Dapeng Wu, and Xiaolin Li. «Server Averaging for Federated Learning». eng. In: (2021) (cit. on p. 21).

[4] Chamath Palihawadana, Nirmalie Wiratunga, Anjana Wijekoon, and Harsha Kalutarage. «FedSim: Similarity guided model aggregation for Federated Learning». eng. In: *Neurocomputing (Amsterdam)* (2021). ISSN: 0925-2312 (cit. on p. 21).

[5] Jer Shyuan Ng, Wei Yang Bryan Lim, Zehui Xiong, Xianbin Cao, Jiangming Jin, Dusit Niyato, Cyril S Leung, and Chunyan Miao. «Reputation-aware Hedonic Coalition Formation for Efficient Serverless Hierarchical Federated Learning». eng. In: *IEEE transactions on parallel and distributed systems* (2021), pp. 1–1. ISSN: 1045-9219 (cit. on p. 22).

[6] Hongzhi Guo, Weifeng Huang, Jiajia Liu, and Yutao Wang. «Inter-Server Collaborative Federated Learning for Ultra-Dense Edge Computing». eng. In: *IEEE transactions on wireless communications* (2021), pp. 1–1. ISSN: 1536-1276 (cit. on p. 22).

[7] Dong-Jun Han, Minseok Choi, Jungwuk Park, and Jaekyun Moon. «FedMes: Speeding Up Federated Learning With Multiple Edge Servers». eng. In: *IEEE journal on selected areas in communications* 39.12 (2021), pp. 3870–3885. ISSN: 0733-8716 (cit. on p. 22).

[8] Frank Po-Chen Lin, Seyyedali Hosseinalipour, Sheikh Shams Azam, Christopher G Brinton, and Nicolo Michelusi. «Semi-Decentralized Federated Learning With Cooperative D2D Local Model Aggregations». eng. In: *IEEE journal on selected areas in communications* 39.12 (2021), pp. 3851–3869. ISSN: 0733-8716 (cit. on p. 22).

[9] Lumin Liu, Jun Zhang, S.H Song, and Khaled B Letaief. «Client-Edge-Cloud Hierarchical Federated Learning». eng. In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. Vol. 2020-. IEEE, 2020, pp. 1–6. ISBN: 1728150892 (cit. on p. 22).

[10] Ramin Firouzi, Rahim Rahmani, and Theo Kanter. «Federated Learning for Distributed Reasoning on Edge Computing». eng. In: *Procedia computer science*. Procedia Computer Science 184 (2021), pp. 419–427. ISSN: 1877-0509 (cit. on p. 22).

[11] Wei Yang Bryan Lim, Jer Shyuan Ng, Zehui Xiong, Jiangming Jin, Yang Zhang, Dusit Niyato, Cyril Leung, and Chunyan Miao. «Decentralized Edge Intelligence: A Dynamic Resource Allocation Framework for Hierarchical Federated Learning». eng. In: *IEEE transactions on parallel and distributed systems* 33.3 (2022), pp. 536–550. ISSN: 1045-9219 (cit. on p. 22).

[12] Luo Shouxi, Fan Pingzhi, Xing Huanlai, and Yu Hongfang. «Domain-Specific Transport Protocols for Edge-Accelerated Cross-Device Federated Learning». eng. In: *IEEE Communications Magazine* (2021) (cit. on p. 23).

[13] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander Wolf. «NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres». eng. In: *Proceedings of the 10th ACM International on conference on emerging networking experiments and technologies*. CoNEXT '14. ACM, 2014, pp. 249–262. ISBN: 9781450332798 (cit. on p. 23).

[14] Fahao Chen, Peng Li, and Toshiaki Miyazaki. «In-Network Aggregation for Privacy-Preserving Federated Learning». eng. In: *2021 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*. IEEE, 2021, pp. 49–56. ISBN: 9781665432856 (cit. on pp. 24, 25).

[15] Amedeo Sapio et al. «Scaling Distributed Machine Learning with In-Network Aggregation». eng. In: (2019) (cit. on pp. 24, 26, 27).

[16] Amedeo Sapio, Ibrahim Abdelaziz, Marco Canini, and Panos Kalnis. «DAIET: A System for Data Aggregation Inside the Network». eng. In: ACM, 2017 (cit. on p. 27).

[17] Fan Yang, Zhan Wang, Xiaoxiao Ma, Guojun Yuan, and Xuejun An. «SwitchAgg: A Further Step Towards In-Network Computation». eng. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on field-programmable gate arrays*. FPGA '19. ACM, 2019, pp. 185–185. ISBN: 1450361374 (cit. on p. 28).

[18] Yi-Bing Lin, Shie-Yuan Wang, Ching-Chun Huang, and Chia-Ming Wu. «The SDN approach for the aggregation/disaggregation of sensor data». eng. In: *Sensors (Basel, Switzerland)* 18.7 (2018), p. 2025. ISSN: 1424-8220 (cit. on p. 29).

[19] Shie-Yuan Wang, Jun-Yi Li, and Yi-Bing Lin. «Aggregating and disaggregating packets with various sizes of payload in P4 switches at 100 Gbps line rate». eng. In: *Journal of network and computer applications* 165 (2020), p. 102676. ISSN: 1084-8045 (cit. on p. 29).

[20] Minkoo Kang, Gyeongsik Yang, Yeonho Yoo, and Chuck Yoo. «TensorExpress: In-Network Communication Scheduling for Distributed Deep Learning». In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 25–27. DOI: 10.1109/CLOUD49709.2020.00014 (cit. on p. 29).

[21] *Processori Intel Tofino Intelligent Fabric*. URL: https://www.intel.it/content/www/it/it/products/network-io/programmable-ethernet-switch/tofino-3-product-brochure.html (cit. on pp. 31, 32).

[22] Pat Bosshart et al. «P4: Programming Protocol-Independent Packet Processors». In: *SIGCOMM Comput. Commun. Rev.* 44.3 (2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890 (cit. on pp. 31, 35).

[23] Barefoot Networks. *"Use cases"*. Tech. rep. URL: https://www.barefootnetworks.com/use-cases/ (cit. on p. 31).

[24] A. Weissberger. *Comcast: ONF Trellis software is in production together with L2/L3 white box switches.* Tech. rep. URL: https://tinyurl.com/y69jc7sv (cit. on p. 31).

[25] M. Nishiki N. Akiyama. *P4 and Stratum use case for new edge cloud.* Tech. rep. URL: https://tinyurl.com/yxuoo9qv (cit. on p. 31).

[26] Stordis GmbH. *New STORDIS advanced programmable switches(APS) first to unlock the full potential of P4 and next generation software defined networking (NG-SDN).* Tech. rep. URL: https://tinyurl.com/y3kjnypl (cit. on p. 31).

[27] Open Networking Foundation (ONF). *Stratum – ONF launches major new open source SDN switching platform with support from Google.* Tech. rep. URL: https://tinyurl.com/yy3ykw7g (cit. on p. 31).

[28] P4.org Community. *P4 gains broad adoption, joins Open Network- ing Foundation (ONF) and Linux Foundation (LF) to accelerate next phase of growth and innovation.* Tech. rep. URL: https://p4.org/p4/p4-joins-onf-and-lf.html (cit. on p. 31).

[29] Facebook engineering. *Disaggregate: networking recap.* Tech. rep. URL: https://tinyurl.com/yxoaj7kw (cit. on p. 31).

[30] Open Compute Project (OCP). *Alibaba DC network evolution with open SONiC and programmable HW.* Tech. rep. URL: //www.opencompute.org/files/OCP2018.alibaba.pdf (cit. on p. 31).

[31] S. Heule. *Using P4 and P4Runtime for optimal L3 routing.* Tech. rep. URL: https://tinyurl.com/sy2jkqe (cit. on p. 32).

[32] Edgecore. *Wedge 100BF-32X, 100GbE data center switch.* Tech. rep. URL: https://tinyurl.com/y365gnqy (cit. on p. 32).

[33] STORDIS. *The new advanced programmable switches are avail- able.* Tech. rep. URL: https://www.stordis.com/products/ (cit. on p. 32).

[34] Cisco. *Cisco Nexus 34180YC and 3464C programmable switches data sheet.* Tech. rep. URL: https://tinyurl.com/y92cbdxe (cit. on p. 32).

[35] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. «An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends». eng. In: *IEEE access* 9 (2021), pp. 87094–87155. ISSN: 2169-3536 (cit. on pp. 32–34).

[36] Lijun Dong and Richard Li. «Information exchange oriented clustering for collaborative vehicular system». In: *2018 27th Wireless and Optical Communication Conference (WOCC)*. 2018, pp. 1–5. DOI: 10.1109/WOCC.2018.8372700 (cit. on p. 37).

[37] Lijun Dong and Renwei Li. «Latency Guarantee for Multimedia Streaming Service to Moving Subscriber with 5G Slicing». In: *2018 International Symposium on Networks, Computers and Communications (ISNCC)*. 2018, pp. 1–7. DOI: 10.1109/ISNCC.2018.8531002 (cit. on p. 37).

[38] Lijun Dong and Richard Li. «Enhance Information Derivation by In-Network Semantic Mashup for IoT Applications». In: *2018 European Conference on Networks and Communications (EuCNC)*. 2018, pp. 298–303. DOI: 10.1109/EuCNC.2018.8442543 (cit. on p. 37).

[39] Lijun Dong and Richard Li. «Distributed Mechanism for Computation Offloading Task Routing in Mobile Edge Cloud Network». In: *2019 International Conference on Computing, Networking and Communications (ICNC)*. 2019, pp. 630–636. DOI: 10.1109/ICCNC.2019.8685537 (cit. on p. 37).

[40] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. «Flower: A Friendly Federated Learning Research Framework». In: *CoRR* abs/2007.14390 (2020). arXiv: 2007.14390. URL: https://arxiv.org/abs/2007.14390 (cit. on p. 40).

[41] PeterH Bennett, ThomasA Burch, and Max Miller. «Diabetes Mellitys in American (Pima) Indians.» In: *The Lancet* 298.7716 (1971). Originally published as Volume 2, Issue 7716, pp. 125–128. ISSN: 0140-6736. DOI: https://doi.org/10.1016/S0140-6736(71)92303-8. URL: https://www.sciencedirect.com/science/article/pii/S0140673671923038 (cit. on pp. 43, 62).

[42] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. «Adaptive Federated Optimization». In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=LkFG3lB13U5 (cit. on pp. 49, 50).

[43] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. «Communication-Efficient Learning of Deep Networks from Decentralized Data». In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 20–22 Apr 2017, pp. 1273–1282. URL: https://proceedings.mlr.press/v54/mcmahan17a.html (cit. on p. 49).

[44] Mohammed Aledhari, Rehma Razzak, Reza M. Parizi, and Fahad Saeed. «Federated Learning: A Survey on Enabling Technologies, Protocols, and Applications». In: *IEEE Access* 8 (2020), pp. 140699–140725. DOI: 10.1109/ACCESS.2020.3013541 (cit. on p. 50).

[45] Jeferson Santiago da Silva. *p4environment*. URL: https://github.com/engjefersonsantiago (cit. on pp. 55, 58).

[46] *Cifar10 dataset*. URL: https://www.tensorflow.org/datasets/catalog/cifar10 (cit. on p. 76).

[47] *MNIST dataset*. URL: https://www.tensorflow.org/datasets/catalog/mnist (cit. on p. 76).