UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DEVELOPING A DISTRIBUTED FEDERATED LEARNING SYSTEM OVER A DECENTRALIZED FILE SYSTEM

Diploma Thesis

Christodoulos Pappas

Supervisor: Manolis Vavalis

July 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DEVELOPING A DISTRIBUTED FEDERATED LEARNING SYSTEM OVER A DECENTRALIZED FILE SYSTEM

Diploma Thesis

Christodoulos Pappas

Supervisor: Manolis Vavalis

July 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΧΤΙΖΟΝΤΑΣ ΕΝΑ ΚΑΤΑΝΕΜΗΜΕΝΟ ΣΥΣΤΗΜΑ ΟΜΟΣΠΟΝΔΙΑΚΗΣ ΜΑΘΗΣΗΣ ΠΑΝΩ ΑΠΟ ΕΝΑ ΑΠΟΚΕΝΤΡΩΜΕΝΟ ΣΥΣΤΗΜΑ ΑΡΧΕΙΩΝ

Διπλωματική Εργασία

Χριστόδουλος Παππάς

Επιβλέπων/πουσα: Μανόλης Βάβαλης

Ιούλιος 2022

Approved by the Examination Committee:

Supervisor **Manolis Vavalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitris Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Christodoulos Pappas

Diploma Thesis

DEVELOPING A

DISTRIBUTED FEDERATED LEARNING SYSTEM

OVER A DECENTRALIZED FILE SYSTEM

Christodoulos Pappas

Abstract

The rise of resourceful mobile devices that store voluminous and privacy-sensitive data motivates the design of privacy-preserving machine learning protocols. One such protocol is the Federated Learning (FL), a distributed machine learning scheme that enables devices to collaboratively train a model without sharing their data. In this thesis, we focus on the problem of designing an efficient and practical decentralized federated learning protocol. First, we introduce the InterPlanetary Learning System (e.g., IPLS), a new decentralized federated learning system built on top of InterPlanetary File System (e.g., IPFS). Then, we further improve the naive IPLS protocol by relaxing its assumption for direct communication and letting devices communicate indirectly with each other. In addition, we make the protocol robust against malicious aggregators and further improve some critical components of the system.

Keywords:

Federated Learning, Machine Learning, IPFS

Διπλωματική Εργασία

ΧΤΙΖΟΝΤΑΣ ΕΝΑ

ΚΑΤΑΝΕΜΗΜΕΝΟ ΣΥΣΤΗΜΑ ΟΜΟΣΠΟΝΔΙΑΚΗΣ ΜΑΘΗΣΗΣ

ΠΑΝΩ ΑΠΟ ΕΝΑ

ΑΠΟΚΕΝΤΡΩΜΕΝΟ ΣΥΣΤΗΜΑ ΑΡΧΕΙΩΝ

Χριστόδουλος Παππάς

Περίληψη

Η πρόσφατη "έξαρση" υπολογιστικά ισχυρών κινητών συσκευών που αποθηκεύουν μεγάλο όγκο προσωπικών δεδομένων, έδωσε το κίνητρο για την σχεδίαση privacy-preserving προτοκόλων για μηχανική μάθηση. Ένα απο αυτά τα πρωτόκολλα είναι η ομοσπονδιακή μάθηση η οποία επιτρέπει στις συσκευές να εκπαιδεύσουν συνεργατικά ένα μοντέλο μηχανικής μάθησης χωρίς να μοιράσουν τα πρωσοπικά τους δεδομένα. Σε αυτήν την διπλωματική, επικεντρωνόμαστε στην σχεδίαση ενός αποδοτικού και πρακτικού συστήματος αποκεντρωμένης ομοσπονδιακής μάθησης. Πρώτα εισαγάγουμε το Διαπλανητικό σύστημα μάθησης, εάν νέο αποκεντρωμένο σύστημα μάθησης το οποίο είναι "χτισμένο" πάνω από το διαπλανητικό σύστημα αρχείων. Μετέπειτα, βελτιώνουμε το ήδη υπάρχων πρωτόκολλο, με το να "χαλαρώσουμε" την υπόθεση οτι οι συσκευές επικοινωνούν άμεσα μεταξύ, επιτρέποντας να επικοινωνούν έμμεσα. Επιπλέον, κάνουμε το πρωτόκολλο μας ασφαλές ενάντια κακόβουλων aggregators και επίσης βελτιώνουμε κάποια σημαντικά κομμάτια του συστήματος μας.

Λέξεις-κλειδιά:

Federated Learning, Machine Learning, IPFS

Table of contents

Abstract x

Περίληψη xi

Table of contents xiii

List of figures xvii

List of tables xxi

Abbreviations xxiii

1 Introduction 1

1.1 Introduction . . . 1

1.1.1 Centralization vs Decentralization . . . 3

1.1.2 This work . . . 4

1.1.3 Main contributions . . . 4

1.2 Preliminaries . . . 6

1.2.1 IPFS . . . 6

1.2.2 Blockchain-Based FL . . . 6

1.2.3 Pedersen Commitments . . . 7

1.2.4 Vector Commitments . . . 8

2 Introduction to IPLS 13

2.1 IPLS Protocol in a nutshell . . . 13

2.2 Implementation Details . . . 16

2.3 Convergence Rate and Accuracy Evaluation . . . 18

3 IPLS over a Decentralized Storage Network **21**
3.1 Building IPLS on top of Decentralized Storage . . . 21
3.1.1 Notation . . . 21
3.1.2 Security Assumptions . . . 21
3.1.3 Indirect Communication Scheme . . . 22
3.1.4 Directory Service . . . 23
3.1.5 Modified IPLS protocol . . . 24
3.2 Merge And Download Mechanism . . . 30
3.3 Experimental Evaluation . . . 31
3.3.1 Aggregation Performance vs. Variable $|A_i|$ . . . 31
3.3.2 Impact of Merge and Download . . . 33
3.3.3 Performance of Directory Service . . . 34

4 On the Efficiency of Directory Service **37**
4.1 Matrix commitments / Hierarchical vector commitments . . . 37
4.1.1 Matrix Commitment Based on RSA Assumption . . . 38
4.2 Mitigating the Load of the Directory Service . . . 42
4.2.1 Minimizing the amount of data the directory service receives . . . 42
4.2.2 “Offloading” Queries to the IPFS Nodes . . . 45
4.3 Experimental Evaluation . . . 49
4.3.1 RSA Based Vector Commitment Efficiency . . . 49
4.3.2 RSA Vector Commitments Vs Merkle Trees . . . 50
4.3.3 On the efficiency of Matrix Commitments . . . 51

5 Conclusion **55**

Bibliography **57**

APPENDICES **63**

A Cryptographic proofs **65**
A.1 Cryptographic Assumptions . . . 65
A.1.1 RSA Assumption . . . 65
A.1.2 Discrete Logarithm Assumption . . . 65
A.2 Vector Binding Proof . . . 66

A.3 Position Binding Proof 67

B Running IPLS **69**

List of figures

1.1 Figures from [1], displaying the total number of rounds vs time and the rounds per hour vs time, in an actual deployment of federated learning. Observe, that in the best case, each iteration lasts for 7 to 9 minutes. 2
1.2 Figures from [2], comparing two different blockchain-based federated learning protocols with the centralized one. 3

2.1 Aggregation protocol of IPLS, with three participants that are both trainers and aggregators responsible for distinct partitions. 14
2.2 Retrieval of the Up-to-date model in the same setup. 14
2.3 A simple representation of the interaction between the processes used in IPLS. Each IPFS daemon has an IPFS API address, which is used by the IPLS middleware to communicate with it via the IPFS Java HTTP API. On the other hand, the IPLS middleware listens to a port (e.g, 12000) and receives from that port, tasks from IPLS applications. Communication happens with IPC (e.g., sockets) 17
2.4 Convergence of IPLS (decentralized) vs conventional (centralized) FL when using a different number of agents. 19
2.5 Robustness of IPLS in the presence of intermittent connectivity. 20

3.1 Figure depicting one iteration of the IPLS protocol, with only one aggregator responsible for the i^{th} partition and three trainers. The main steps of the protocol are **1.** Upload the gradients to the IPFS network, **2.** Write to the directory service, **3.** aggregator queries the directory service, **4.** Receives the hashes of the gradients, **5.,6.** retrieves them from the IPFS **7.** aggregates them and uploads the update to the IPFS, **8.** the directory service verifies the validity of the update before written to the directory 25

3.2 A snapshot of how trainers download the up-to-date model. They simply query the directory service and upon receiving the hash of each partition they retrieve the updates from the IPFS network. Finally they reconstruct the model by concatenating the updated partitions. 26

3.3 Figure depicting merge and download concept with three IPFS providers. The aggregator sends merge request to his providers (**1.**) and then it receives the aggregated results (**2.**). 31

3.4 Total aggregation delay (top) and total size of data received by an aggregator (bottom) in each iteration, vs. the number of aggregators assigned to each partition. 32

3.5 Aggregation (top) and uploading (bottom) delays 33

3.6 Write delay with 0% and 5% packet loss and variable number of partitions . 34

4.1 A simple resemble of the recursion, when computing $C_{12}, C_{13}, ..., C_{1m}$ from C_1. Note that in the paradigm, m = 9 so we need to find 8 elements for C_1. In addition, we represent $C_i^{e_{k_1}e_{k_2}\cdots}$ with $C_i, \{e_{k_1}, e_{k_2}, ...\}$ 40

4.2 Figure depicting what the trainers do before writing to the directory. They compute the vector commitment of the hash of each gradient partition and also the matrix commitment of the gradient partitions (**1.**,**2.**). Then upload the auxiliary information (e.g., witnesses and the actual hashes) of the commitment schemes to the IPFS network (**3.**), and the Vector and Matrix commitment to the directory (**4.**) 43

4.3 Figure depicting how the directory service offloads reads to the IPFS nodes. In a nutshell, the directory computes the current state of the directory (**1.**) and its secure digest (**2.**). Then sends the directory to the IPFS nodes (**3.**) and its digest to all the IPLS nodes (**4.**). To make a query, the IPLS node simply asks an IPFS node (**5.**), receiving an answer with a witness that the answer is correct (**6.**) 45

4.4 Figure depicting how the representation of the directory using a Merkle tree with four aggregators. The digest of the directory is the H_1. Whenever the a_{11} receives the hashes from an IPFS node, the IPFS node has to additionally send $H_{a_{21}}$ and H_3. a_{11}, verifies the validity of the answer by computing $H_2 = H(H(a_{11}||reply)||H_{a_{21}})$ and checking if $H_1 = H(H_2||H_3)$. 46

4.5 Time needed to compute and open all witnesses vs. number of partitions, in logarithmic scale 49
4.6 Time needed to compute and open all witnesses using a VC scheme and a Merkle Tree vs. number of aggregators, in logarithmic scale 51
4.7 Aggregation (top) and uploading (bottom) delays with variable number of providers. 53

List of tables

4.1 Witness size of RSA based vector commitment scheme and Merkle tree, on different number of aggregators. 50

Abbreviations

e.g.	for example
i.e	in explain
IPLS	InterPlanetary Learning System
IPFS	InterPlanetary File System
DHT	Distributed Hash Table

Chapter 1

Introduction

1.1 Introduction

Traditionally, tremendous volumes of data must be collected, in an individual's desktop or server, to train a neural network or a machine learning model. However, in many applications, those data come from people, commonly containing some personal information about them, and as a result, by collecting all of those data, part of the people's privacy gets compromised. In addition to that, with the establishment of data privacy regulations, e.g., the General Data Protection Regulation (GDPR) [1], came the need to train machine learning models and especially neural networks without "invading" the users' private information. Lately, there has been an enormous effort to provide such services both in research and industry throughout the years. For example, many works enable the training of machine learning models on encrypted data using various cryptographic primitives such as fully homomorphic encryption or secure multi-party computation [3, 4, 5, 6]. However, none of them had such a significant impact as the recently proposed Federated Learning [1, 7].

Federated learning is a relatively new privacy-preserving (or, more correctly, privacy-aware) distributed machine learning paradigm that enables users to collaboratively train a machine learning model without sharing their local data with no one. The federated learning protocol is a simple, iterative protocol, in which, each iteration consists of four distinct steps. The server (or aggregator) that stores the machine learning model selects several clients and sends them the model's parameters. Next, the selected clients train the model locally using

[1] https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_el

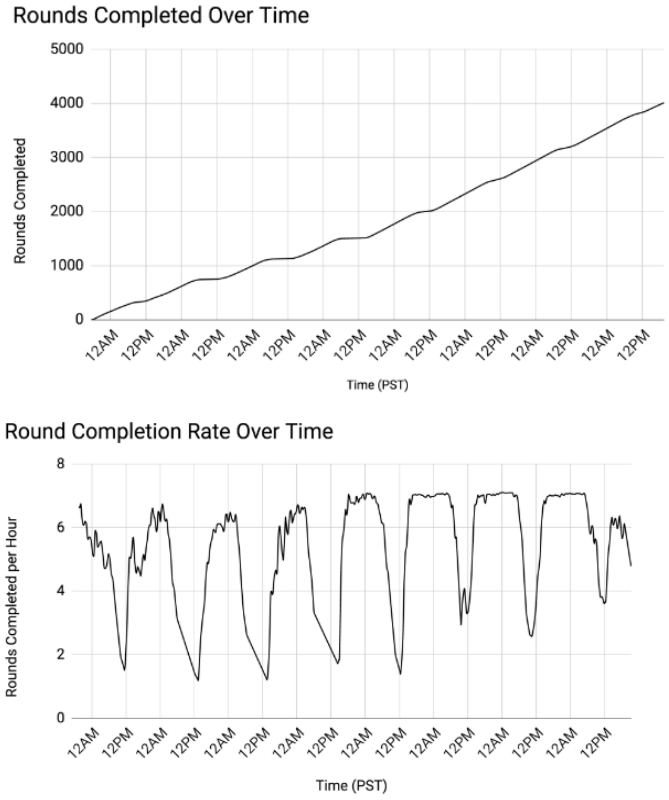

Figure 1.1: Figures from [1], displaying the total number of rounds vs time and the rounds per hour vs time, in an actual deployment of federated learning. Observe, that in the best case, each iteration lasts for 7 to 9 minutes.

their local data and produce some updates. Those updates can be the gradients of the parameters of the model or the parameters of the trained model. Finally, each selected client sends those updates back to the server that aggregates them (e.g., averages them) and forms a new updated global model. This process continues for many iterations until the model converges.

An observant reader might come to the realization that Federated learning is not something new, and in reality, it is a trivial data-parallel distributed machine learning protocol. While that is true, there are several significant differences between Federated learning and data-parallel distributed machine learning. First and foremost, in distributed machine learning, the workers (devices that train the model) are “strong” and reliable devices in terms of computational capability and internet connectivity. That is not commonly true in federated learning, in which workers are commonly IoT devices or smartphones that face intermittent internet connectivity and lack of resources. Moreover, the number of workers in federated learning is magnitudes of times larger than in distributed machine learning. Last but not least, in contrast to federated learning, in distributed machine learning, the server holds the data and distributes them to the workers. Those seemingly minor differences generate new problems

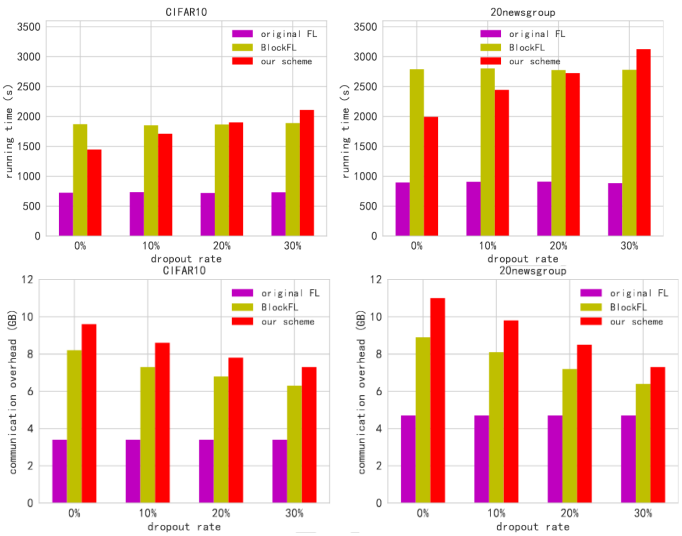

Figure 1.2: Figures from [2], comparing two different blockchain-based federated learning protocols with the centralized one.

that are often hard to solve. All in all, the purely distributed machine learning happens mainly for performance gains in the training process, while federated learning happens mainly to deal with privacy concerns. As it can be seen in Figure 1.1, federated learning doesn't have strict requirements in performance and iteration delay, and in reality a federated learning campaign can last many days.

1.1.1 Centralization vs Decentralization

To date, federated learning is a highly centralized procedure. Although training of the model is distributed among the devices (trainers), a centralized entity, the aggregator, must exist that collects the gradients from the trainers and computes the new model. As in many other systems, centralized federated learning has the inherent problem of a single point of failure. Specifically, a single malicious server can perform various attacks that can easily harm the accuracy or convergence rate of the model (e.g., adding noise to the updated model) or even compromise the privacy of the users by learning some partial information about their data. Although works exist to prevent some attacks [8, 9, 10], due to centralization, they can not inherently deal with forking attacks, Denial of Service, or Sybil attacks.

To overcome the security problems arising from centralized nature of Federated learning, there are has been a noticeable effort to decentralize the process [2, 11, 12, 13, 14, 15]. In the vast majority of those works, decentralization comes through the means of using a blockchain, as it can deal with all the security issues mentioned earlier of centralized FL. In

blockchain-based FL, the blockchain (and consequently the miners) take the role of the aggregator. Every miner has to receive the gradients from all the trainers, aggregate them, and engage in an expensive consensus protocol to add the new updated model to the blockchain. As it is evident (1.2), a blockchain-based solution faces some severe scalability limitations and is significantly less efficient compared to the centralized FL.

InterPlanetary Learning System (e.g., IPLS) [16] has been proposed as a solution that combines the best of those two worlds. Specifically, it wants to achieve the same or even better efficiency as centralized FL and the same security guarantees as decentralized federated learning. In IPLS, there are multiple non-colluding aggregators as with the blockchain-based FL; however, the aggregation step is distributed among the aggregators, in contrast to the last one. Moreover, we will show that carefully using cryptographic primitives, the aggregation step can be verifiable, meaning that anyone having the updated model and a succinct (constant sized) proof can verify that the updated model is indeed computed honestly, without using any expensive consensus protocol.

1.1.2 This work

In this work, we consider the scenario of an individual or a small enterprise that wants to train a machine learning model via a federated learning protocol. However, those entities lack the resources to set up a server that they fully trust to meet up the scalability requirements of federated learning. Note that even if the model was about 10MB large and the number of trainers on each iteration was 10.000 then the server would have to receive 100GB of data on each iteration which is sometimes prohibited. In addition, the federated learning initiator does not fully trust a third-party cloud service to offload its aggregation step due to the security issues described above. A reasonable solution to this problem would be to let the initiator offload the entirety of the federated learning protocol using the IPLS protocol to the devices, and devote its limited resources to strictly necessary and crucial, lightweight operations.

1.1.3 Main contributions

To begin with, we extend the naive IPLS protocol, so that participants in the federated learning process can communicate reliably and efficiently with each other, regardless their networking limitations. This is done by communicating indirectly, throughout a decentralized storage network. In addition we optimize this type of communication, to further reduce the

aggregation time and in general the time needed for one IPLS round. Furthermore we make our protocol secure against malicious aggregators by using homomorphic hashes. Last but not least, we propose some additional modifications in our protocol, to make some important IPLS components much more efficient.

Overall the main contributions of this work are:

1. Develop IPLS so that peers can communicate indirectly with each other, and deal with some issues regarding mainly the efficiency of their communication.
2. Make IPLS secure against malicious aggregators.
3. Use various cryptographic primitives to make IPLS much more scalable.
4. Provide various measurements for some critical components of IPLS.

1.2 Preliminaries

1.2.1 IPFS

InterPlanetary File System [17] is a distributed file system that enables devices to connect to a peer-to-peer network and share files. In its simplest form, the IPFS network maintains a Distributed Hash table (DHT) [18], which enables peers to find and retrieve files from other peers in the system. Each file in IPFS is addressed by a secure IPFS hash, which is the SHA-256 hash function by default. To retrieve a file from IPFS, a peer has to know beforehand the hash of that file. Then, by using that hash, retrieve from the DHT the peers' addresses that store that file locally. Then the retriever uses a file exchange protocol (e.g., [19]) to download the file. A common misconception about IPFS is that it provides permanent storage and guarantees the availability of the files stored in the system. However, this is not true because IPFS does not inherently has any file replication mechanism. Nevertheless, various protocols do exist (e.g., IPFS cluster, Filecoin [20]) that enable file replication throughout the system and offer file availability guarantees.

1.2.2 Blockchain-Based FL

A decentralized alternative to traditional federated learning, is blockchain-based federated learning. Authors in [21], present an extensive survey on blockchain-based (BCFL) federated learning, while trying to classify the systems in two significant categories, the fully coupled BCFL [13],[12],[11], where trainers are also nodes in the blockchain and participate actively in the block generation and model aggregation, and Flexibly Coupled BCFL [14],[15],[22] where trainers aren't nodes of the blockchain but they just upload their updates in the blockchain while miners are responsible for aggregating the trainers' updates and producing the global model. In the vast majority of those papers, the federated learning process follows the same pattern, which is 1) the trainers when finish training of the model, upload their gradients to the blockchain 2) the blockchain takes the role of the server in the traditional federating learning and is responsible for aggregating those gradients. In more detail, the first step is usually achieved by letting the trainers or their selected blockchain node broadcast their gradients throughout all the other nodes of the blockchain. One noticeable exception can be found in [12], where gradients are stored in the IPFS. The second step gets carried through with selected nodes or even all the nodes in the blockchain network followed

by a consensus protocol, to generate a new block containing the global new model.

Although BCFL can overcome the problems traditional Federated learning faces, issues arise regarding the scalability, storage requirements and efficiency of such architectures. To be more specific, in the majority of the works, trainers' gradients must be stored in the blockchain which depending on the model size and the number of trainers might lead to some serious storage and scalability issues. In addition, to perform the aggregation, each aggregator must download all the gradients stored in the blockchain resulting in unnecessary and redundant operations that can add significant overhead on each training round. As a consequence, though secure and robust, BCFL is an expensive alternative to centralized federated learning.

1.2.3 Pedersen Commitments

Pedersen commitment [23, 24] is a cryptographic commitment scheme that enables a user to commit to a single element or vector of elements, into a constant sized digest, which is commonly (256 bits). Pedersen commitments use a public group, in which finding the discrete logarithm is "hard". Such groups are the prime order groups. Let G be such a group, and g be the generator of that group. Then Pedersen commitments can be described by the two polynomial time algorithms (Gen,Commit) :

$\mathbf{pp} \leftarrow \mathbf{Gen}(\mathbf{1^k}, \mathbf{G}, \mathbf{n})$. This algorithm take as input the security parameter k, and the size of the vector n, and returns some public parameters. Those public parameters are simply random group elements $h_1, h_2, ..., h_n \in G$. This can be computed simply by selecting at random some elements $r_1, r_2, ..., r_n \in |G|$ and then compute $pp = \{h_1, h_2, ..., h_n\} = \{g^{r_1}, g^{r_2}, ..., g^{r_n}\} \in G^n$. It is crucial that $r_1, r_2, ..., r_n$ must remain secret to any other entity in the system.

$\mathbf{C} \leftarrow \mathbf{Commit}(\mathbf{pp}, \tilde{\mathbf{v}})$. This algorithm takes as input the public parameters, the vector $\vec{v}$ to be committed and returns a succinct (and commonly constant sized) digest C. This is computed by $C \leftarrow h_1^{v_1} h_2^{v_2} ... h_n^{v_n} \in G$.

Pedersen commitments, are **vector binding** meaning than no adversary can find two different vectors v, v' that result to the same commitment. Formally,

$$Pr \begin{pmatrix} v, v' \leftarrow A(pp); \\ Commit(pp, v) = Commit(pp, v') \end{pmatrix} \leq negl(k) \tag{1.1}$$

The formal proof is located in the appendix. Moreover Pedersen commitments are homomorphic. That is because $Commit(pp, v_1) \cdot Commit(pp, v_2) = (h_1^{v_{11}} h_2^{v_{12}} ... h_n^{v_{1n}}) \cdot (h_1^{v_{21}} h_2^{v_{22}} ... h_n^{v_{2n}}) =$

$h_1^{v_{11}+v_{21}} h_2^{v_{12}+v_{22}} ... h_n^{v_{1n}+v_{2n}} \in G = Commit(pp, v_1 + v_2)$. In reality Pedersen commitments are also hiding, however, this property is not needed in our work, so we can safely ommit its formalization and the proof.

1.2.4 Vector Commitments

A Vector commitment is a cryptographic primitive that makes it possible to commit to an ordered set of elements (commonly denoted as vector), and then someone can open that commitment to a specific position of the ordered set. Vector commitments can be described by the following tuple of probabilistic polynomial time algorithms $(Gen, Commit, Open, Verify)$. Specifically :

$\mathbf{pp} \leftarrow \mathbf{VC.Gen}(\mathbf{1^k}, \mathbf{n})$. This algorithm take as input the security parameter k, and the size of the vector n, and returns a set containing public parameters.

$\mathbf{C}, \mathbf{aux} \leftarrow \mathbf{VC.Commit}(\mathbf{pp}, \tilde{\mathbf{v}})$. This algorithm takes as input the public parameters, the vector $\tilde{\mathbf{v}}$ to be committed and returns a succinct (and commonly constant sized) digest C and also some auxiliary information.

$\mathbf{W_i}, \mathbf{v_i} \leftarrow \mathbf{VC.Open}(\mathbf{pp}, \mathbf{aux}, \tilde{\mathbf{v}}, \mathbf{i})$. This algorithm takes as input the public parameters pp, the committed vector, and the position $i \leq n$, and returns the element $\mathbf{v_i}$ of the vector and also a witness $\mathbf{W_i}$, that $\mathbf{v_i}$ is indeed the i^{th} element of the committed vector.

$\mathbf{0}, \mathbf{1} \leftarrow \mathbf{VC.Verify}(\mathbf{pp}, \mathbf{C}, \mathbf{v_i}, \mathbf{W_i}, \mathbf{i})$. This algorithm takes as input the public parameters, the commitment C, the element $\mathbf{v_i}$, its witness $\mathbf{W_i}$ and a position i. Returns 1 if the element $\mathbf{v_i}$ is indeed the i^{th} element of the committed vector, otherwise returns 0.

As in every cryptographic primitive, vector commitments must fulfill some security guarantees. Firstly, we require that a vector commitment to be **correct**, meaning that in case v_i is indeed an element of an committed vector $\vec{v}$ with commitment C, then :

$$Pr\begin{pmatrix} W_i, v_i \leftarrow Open(pp, aux, \vec{v}, i); \\ 1 \leftarrow Verify(pp, C, v_i, W_i, i) \end{pmatrix} = 1 \tag{1.2}$$

Next we additionally require vector commitments to be **position binding** meaning that no efficient adversary can correctly open a commitment on two different values at the same position. Formally, given an probabilistic polynomial time adversary A, and public parameters pp, the probability of finding two **different** values v_i, v_i' and two witnesses W_i, W_i', the

probability that $Verify(pp, C, v_i, W_i, i) = Verify(pp, C, v'_i, W'_i, i) = 1$ is negligible with respect to the security parameter k. Specifically:

$$Pr\left(\begin{array}{c} C, W_i, W'_i, v_i, v'_i \leftarrow A(pp); \\ 1 \leftarrow Verify(pp, C, v_i, W_i, i) \wedge \\ 1 \leftarrow Verify(pp, C, v'_i, W'_i, i) \end{array}\right) \leq negl(k) \tag{1.3}$$

Vector Commitment Based on RSA Assumption

Throughout recent years, there has been an "explosion" in the research of vector commitment shemes [25, 26, 27, 28, 29, 30]. In this work we will use, the very simple constructions introduced in [26]. Out of the two constructions presented in that paper, we will specifically focus on the construction based on RSA assumption, because is much simpler and the size of the public parameters is minimal. In more detail the construction is the following:

$\mathbf{pp} \leftarrow \mathbf{Gen}(\mathbf{1^k}, \mathbf{n})$ Select two random primes p_1, p_2 of length n/2 and compute N = $p_1 p_2$. Then select n random distinct primes $\{e_i\}_{i\in[n]}$, pick a random element g and compute $S_i = g^{\prod_{j\in[n]-i} e_j}$. Set pp = $\{N, e_1, e_2, ..., e_n, S_1, S_2, ..., S_n\}$

$\mathbf{C}, \mathbf{aux} \leftarrow \mathbf{Commit}(\mathbf{pp}, \tilde{\mathbf{v}})$ Compute $C = \prod_{i\in[n]} S_i^{v_i} \ mod\ N$. Set aux = {}.

$\mathbf{W_i}, \mathbf{v_i} \leftarrow \mathbf{Open}(\mathbf{pp}, \mathbf{aux}, \tilde{\mathbf{v}}, \mathbf{i})$ Compute $W_i = \prod_{j\in[n]-i} (S_j^{v_j})^{1/e_i} \ mod\ N$. Note that in reality we can't compute the square root of an element of a prime order group. Nonetheless S_j^{1/e_i} can be computed naively by finding $S'_j = g^{\prod_{k\in[n]-\{j,i\}} e_i} \ mod\ N$.

$\mathbf{0}, \mathbf{1} \leftarrow \mathbf{Verify}(\mathbf{pp}, \mathbf{C}, \mathbf{v_i}, \mathbf{W_i}, \mathbf{i})$ Check if the $S_i^{v_i} W_i^{e_i} \ mod\ N = C$.

It is trivial to see that correctness property holds. To prove that the vector binding holds, we make the hypothesis that there exist an adversary A that can win the vector binding game with non-negligible probability. The we create an adversary B that uses A to break the RSA assumption with non-negligible probability thus reaching contradiction. The detailed proof, can be found in the Appendix A.

Another additional property of that construction, is that someone can not only open the commitment to one position, but also to multiple positions. This "kind" of opening, is called subvector opening. Let $I \subset [n]$ be a set of indexes of the vector, then to open to the commitment in I, we must compute the witness, which is $W_I = \prod_{j\in[n]-I} (S_j^{v_j})^{1/e_I} \ mod\ N$, where $e_I = \prod_{i\in I} e_i$. To verify that the subvector belongs to the vector, check the equation $C = (\prod_{i\in I} S_i^{v_i}) W_I^{e_I} \ mod\ N$.

On Efficiency of RSA Vector Commitment

Naively, computing the public parameters, would require $O(n^2)$ time, computing the commitment would require $O(n)$ time, computing a witness W_i would require $O(n^2)$ and verification cost is $O(1)$. Note that the computation of the commitment is done in $O(n^2)$ when we do not have the S_i. In reality those asymptotics are completely unacceptable. However we can reduce those unacceptable asymptotics with various tricks [2].

To begin with, the complexity of opening to a position can be easily minimized. Assuming that we want to open the position i, we have to find a more efficient way to compute the $S'_j = g^{\prod_{k\in[n]-\{j,i\}} e_i}$, $\forall j \in [n] - i$. Let e' be the set of all primes except e_i, thus $S'_j = g^{\prod_{k\in[n]-\{j\}} e'_i}$. Observe that for $j = 0$ to $j = (n-1)/2$, all the values of S'_j contain in their exponent the product $\prod_{k\in((n-1)/2,\ldots,n} e'_k$, while for $j = (n-1)/2$ to $j = n$, all the values of S'_j contain the product $\prod_{k\in1,\ldots,(n-1)/2)} e'_k$. Thus for each value of $S'_{1,\ldots,(n-1)/2}$ we can compute $g^{\prod_{k\in(n-1)/2,\ldots,n} e'_k}$ only once. Continuing this idea recursively we end up with a divide and conquer algorithm with complexity is $O((n-1)log(n-1))$. The same way we can compute subvector openings with complexity $O((n-|I|)log(n-|I|))$. This algorithm, firstly "baptised" in [28], is called root factor, and although there is a more efficient algorithm, this algorithm becomes of great importance later in this work.

We could develop a more efficient opening algorithm by the following way. First we maintain two variables A and W_i, and initialize them with $A = g^{e'_1}$ and $W_i = g^{v_1}$. Next we compute $W_i = W_i^{e'_2} A^{v_2} = g^{e'_2 v_1} g^{e_1 v_2}$ and $A = A^{e'_2}$. We repeat this process until we finish with all primes in e'. Note that the running time is $O(n)$. The same way we can compute subvector openings in $O(n-|I|)$ time. As a result, from quadratic complexities we ended up with linear ones, something that is significantly faster.

In addition, we could pre-compute all the witnesses, so that the opening time be constant. To achieve that we apply a divide and conquer algorithm. First we compute the witnesses $W_{0,\ldots,n/2}, W_{n/2+1,\ldots,n}$ of opening the first half of the vector and the last one. This can be achieved in $O(n)$ time. We continue recursively for the first and the last halves of the vectors. For example in the second step of the recursion we compute $W_{0,\ldots,n/4}$, $W_{n/4,\ldots,n/2}$, $W_{n/2+1,\ldots,3n/4}, W_{3n/4+1,\ldots,n}$. Finally we end up with the recursive equation $T(n) = 2T(n/2)+ O(n)$. As a result precomputing all the proofs requires $O(nlogn)$ time which, while opening,

[2] An outstanding explanation of those tricks can be found in `https://alinush.github.io/2020/11/24/Catalano-Fiore-Vector-Commitments.html`

requires only $O(1)$ time. In our implementation of the vector commitment, we require devices to precompute the openings of their commitments.

Application of Vector commitments

Vector commitments can be used in various ways. One simple application of a vector commitment is that of verifiable storage. Specifically, consider a client who holds a large array of data, and wants to offload its data to a cloud server, and query it, whenever the client needs to retrieve one or more positions of that array. However the client does not fully trust the server that it will reply with the correct data. To "force" the server to behave honestly, the client computes a vector commitment, keeps the digest, and stores the array to the server. Whenever the client needs to get the elements in some position, queries the server which replies with the element and a witness that belongs to the vector. The client accepts the reply if the verification returns 1.

Chapter 2

Introduction to IPLS

2.1 IPLS Protocol in a nutshell

The InterPlanetary Learning System (IPLS) [16] is a decentralized federated learning middleware that enables the training of a machine learning model in Federated manner but without using a centralized server. The key concept of IPLS is to segment the parameters vector of the machine learning model into smaller partitions. This is exactly what happens with large files in IPFS or bit-torrent. The partitions of the model are then separately maintained and aggregated by different participants that are made responsible for these partitions, based on the received gradients. More specifically, IPLS participants have the following roles:

1) Bootstrappers. In IPLS, a bootstrapper is the initiator of a federated learning task. Whenever a participant wants to join the task, it must initially communicate with its bootstrapper. Bootstrappers are assumed to have good network connectivity as they are required to have periodic activity, e.g., to maintain participant registration for the tasks they have launched. Bootstrappers also setup a private IPFS network, on which, they are also bootstrappers.

2) Aggregators are the participants that behave exactly like servers in centralized federated learning but only for a small portion of the model. Aggregators can be responsible for one or more partitions of the model. What an aggregator receives, from the trainers, is only the gradients that it is responsible for. Upon receiving all gradients that it has to, then it aggregates them (using summation) and produces the updated model for those partitions. Lastly it communicates those partitions back to the trainers.

For robustness, security and efficiency purposes, it is important to have multiple aggregators responsible for the same partition. If this is true, then each trainer selects only one

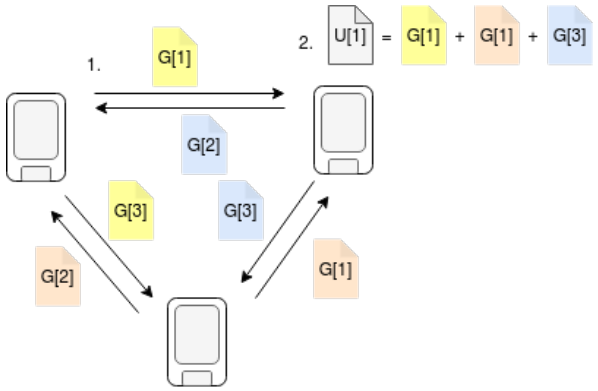

Figure 2.1: Aggregation protocol of IPLS, with three participants that are both trainers and aggregators responsible for distinct partitions.

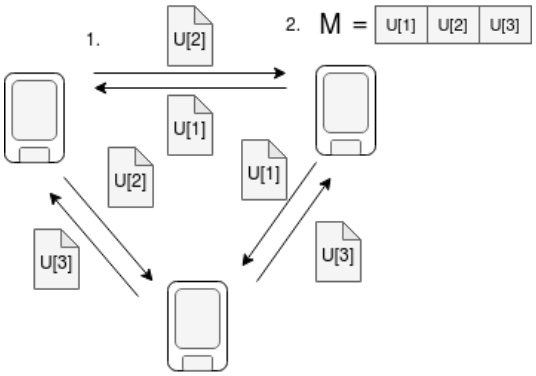

Figure 2.2: Retrieval of the Up-to-date model in the same setup.

aggregator for each partition of the model, to send to it its gradients. Moreover whenever an aggregator collected all its gradients, it aggregates them producing a partial update and then communicates this partial update with all the other aggregators responsible for the same partition.

Upon receiving all the partial updates for the partition it is responsible for, then further aggregates them, computing the global updated partition which is sent back to the trainers.

3) Trainers are the participants who behave exactly like ordinary clients in the centralized federated learning. They firstly download the up to date model. This is done by receiving the update for each partition of the model. Then using the concatenation rule, they form the new model which they are using for training based on their local data. At the end of the local training, they produce some gradients which get segmented, and send each gradient partition to its corresponding aggregator.

Figures 2.1 and 2.2 capture the main steps of the vanilla IPLS protocol. In those figures, we assume that only three participants exist, and they are both trainers and aggregators. In fact, in the first version of IPLS this was a strict requirement; however, in later versions,

this is not mandatory. In 2.1, participants segment their gradients into three partitions upon completing their training and send each gradient partition to the corresponding participant responsible for that partition. The participants, upon collecting the gradients from the partitions they are responsible for, aggregate those partitions. As figure 2.2 suggests, participants, communicate back their updated partitions whenever aggregation is finished. Finally, when the three updated partitions are collected, participants concatenate them and form the new model.

To communicate, participants must establish direct communication links. That, for example, can be achieved by using TCP or UDP, but IPLS uses the ipfs pub/sub protocol [31, 32], because it can deal with intermittent connectivity issues, but also it can be used for different types of communication, for example, multicasting or broadcasting. In addition, vanilla IPLS used two different kinds of synchronization in the aggregation step. The first one was strictly synchronous aggregation, in which an aggregator waits until it receives all the gradient partitions that it has to, and asynchronous aggregator, in which the aggregator, upon receiving a gradient partition, instantly aggregates it and communicates back the newly updated partition. In reality, asynchronous can be much more robust and sometimes faster than strictly synchronous, but it might take many more rounds to converge.

On the communication complexity of the IPLS protocol. Let M be the model and $|M|$ be its size (e.g., in bytes). In the scenario of 2.1,2.2, we see that the model is segmented into three partitions, and assume that the size of each partition is the same, (e.g., $Partition\_Size = |M|/3$). In both 2.1 and 2.2, each participant receives and sends data of size $2 \cdot |M|/3$, thus totally the amounts of data each participant receives and sends is $4 \cdot |M|/3$ and $4 \cdot |M|/3$ respectively. Asymptotically, if each participant was responsible for only one partition and if there were no other participants responsible for the same partition, then, for N participants in the system, each participant would have to download data of size $2(N-1)|M|/N \approx 2|M|$ and upload data of size $2(N-1)|M|/N \approx 2|M|$. As a result, the communication cost for each participant is independent of the number of participants in the system. In reality, however, it would be highly impractical to have only one aggregator responsible for the same partition mainly for robustness reasons.

Dealing with more aggregators responsible for the same partition. Whenever more aggregators responsible for the same partition do exist, they somehow have to communicate with each other in order to compute a "global" update for that partition. To achieve that, each ag-

gregator, upon collecting the gradients from the trainers that selected him, aggregates them and then multicasts that "partial", via IPFS pub/sub, update to the aggregators responsible for the same partition. It then waits until it receives the "partial" updates from all the other aggregators responsible for the same partition.

System initialization. Before the beginning of an IPLS round or even before the beginning of the entire IPLS process, a participant must learn 1) if it will become a trainer, aggregator, or both, 2) if it becomes an aggregator, which partition/s should it be responsible for and 3) if it is trainer, which aggregator should it select, for each partition, to send its gradients. In the vanilla IPLS protocol, a participant is an aggregator and trainer simultaneously. To learn which partitions it will be responsible for, it follows a protocol explained in the next paragraph. Finally, a participant, for each partition of the model it is not responsible for, selects an aggregator at random to send him the gradients for that partition. Note that those policies are very naive, and additional work can be done so that responsibilities assignment and aggregators selection, so that to optimize some aspects of the IPLS system.

Responsibilities assignment protocol. At the very beginning of the IPLS protocol, the only entity present is the bootstrapper of the project. What the bootstrapper does, is to receive requests from participants interested in the project and provide them with useful information about the project. Such information is, for example, the minimum number of partitions p_{min}, an aggregator should be responsible for. When a participant enters the system, it first broadcasts a message asking all other participants to provide it with their responsibilities. Then the participant selects the p_{min} least "popular" partitions (e.g., those that fewer aggregators are responsible for). Then the participant broadcasts its responsibilities. This sub-protocol, can be performed periodically. Clearly, the bootstrapper could also select by himself the responsibilities of the participants; however, by doing that, we would have to rely even more on a central entity.

2.2 Implementation Details

To begin with, IPLS, offers an API that consists of four main methods, the $ipls.init(\cdot)$, $ipls.updatemodel(\cdot)$, $ipls.getmodel()$ and $ipls.terminate()$. The $ipls.init(\cdot)$, takes as input various parameters that are mainly determined by the project and chosen by the bootstraper. A participant calls the $ipls.init(\cdot)$, so that it can to join the project, learn about the respon-

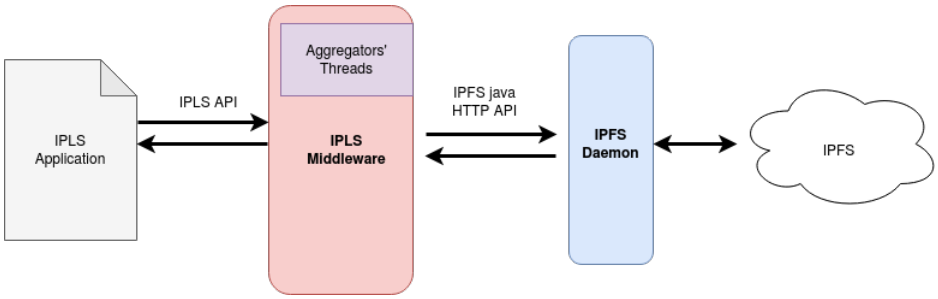

Figure 2.3: A simple representation of the interaction between the processes used in IPLS. Each IPFS daemon has an IPFS API address, which is used by the IPLS middleware to communicate with it via the IPFS Java HTTP API. On the other hand, the IPLS middleware listens to a port (e.g, 12000) and receives from that port, tasks from IPLS applications. Communication happens with IPC (e.g., sockets)

sibilities of other participants and also take some responsibilities. When the participant finished the model training, calls the $ipls.updatemodel(\cdot)$, which takes as input the gradients, segments them into the proper partitions, and sends them to the selected aggregator, while waiting to receive the updated partitions. The updated partitions are then stored in a cache. $ipls.getmodel(\cdot)$, returns the cached updated partitions partitions, and reconstructs the up-to-date model. Finally a participant can use $ipls.terminate()$, in order to leave the project. To see in much more detail the API and its current implementation, visit Appendix B.

Regardless of its version, the dominant processes in IPLS are the IPFS daemon, the IPLS middleware and the application. The **IPFS** daemon, is a process, that performs various networking functionalities in the level of IPFS. For example, the IPFS daemon is responsible for downloading or adding files in the IPFS or executing the IPFS pub/sub protocol. The **IPLS** middleware, is responsible for implementing the IPLS API. Moreover, the IPLS middleware is responsible for executing the tasks of an aggregator, if the participant that runs the middleware is aggregator, something that is "learned" by the $ipls.init()$. Last but not least, the application which uses the IPLS API, is the program written by the project initiator, and is used to collect data, locally train the initiators model and updated the "global" model using the IPLS API. Note that the application communicates with the IPLS middleware via IPLS API, and the IPLS middleware with the IPFS daemon via IPFS java API. The middleware is written in java, but IPLS APIs do exist for java and python.

2.3 Convergence Rate and Accuracy Evaluation

To measure the performance of IPLS, in terms of convergence rate and accuracy, in direct comparison with the centralized federated learning, we used the MNIST dataset [33]. The MNIST dataset contains 60.000 samples, each one representing an 28×28 pixel image of a digit between 0-9, which are uniformly distributed among the peers. We use MNIST to train an Artificial Neural Network so that it can classify the digit an image represents. The ANN we used has four layers ($785 \times 500 \times 100 \times 10$) which contains roughly 443.500 trainable parameters. To simulate the interaction between peers, we used the Mininet [1] network emulator.

In the first experiment 2.4, we measured the convergence of both synchronous and asynchronous aggregation of IPLS and compared it with the convergence rate of the centralized federated learning. Specifically, each peer (agent), is responsible for only one partition of the model, and there are not peers responsible for the same partition. As it can be seen, the performance of the synchronous IPLS is the exact same as the centralized federated learning. On the other hand, as stated earlier, asynchronous aggregation needs more rounds, in order to reach the same accuracy as synchronous and centralized federated learning.

Next, we measured the performance and resilience of IPLS, when peers face intermittent connectivity issues 2.5. Specifically, we used eight peers and dynamically closed and opened their internet connectivity using real traces. We measured the convergence rate of each peer when there is only one peer responsible for a partition and when there are four peers responsible for the same partition. Note that for those experiments, we only used asynchronous aggregation because, with synchronous aggregation, the convergence rate would be the same as if peers had perfect internet connectivity.

From figure 2.5, we can see that if there is only one peer responsible for a partition, then we have the same convergence rate regardless the quality of the connectivity. However, when peers face intermittent connectivity issues, the time needed to converge is significantly more than having perfect connectivity. In contrast, we can see that if more peers are responsible for the same partition, the convergence rate drastically changes. To begin with, the model presents a slower convergence rate when more peers responsible for the same partition exist. However, when peers face intermittent connectivity issues, accuracy does not change too much. In reality, the model needs less time to converge than if we had only one peer respon-

[1] http://mininet.org/

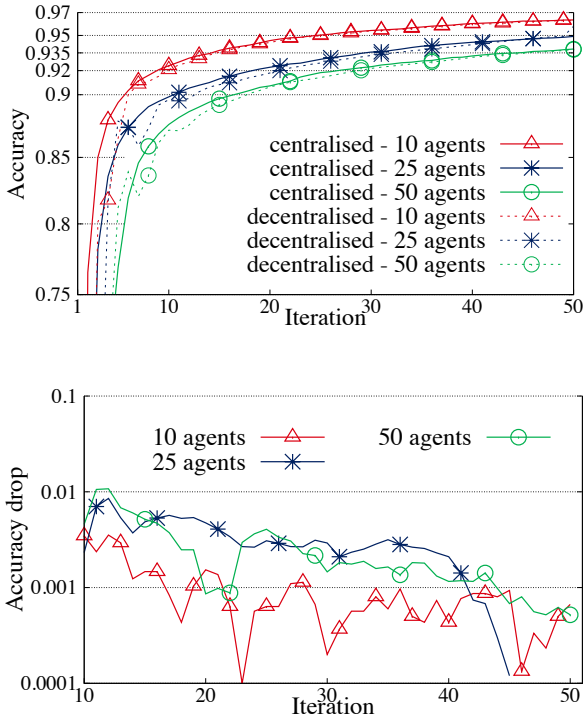

Figure 2.4: Convergence of IPLS (decentralized) vs conventional (centralized) FL when using a different number of agents.

sible for a partition with imperfect connectivity.

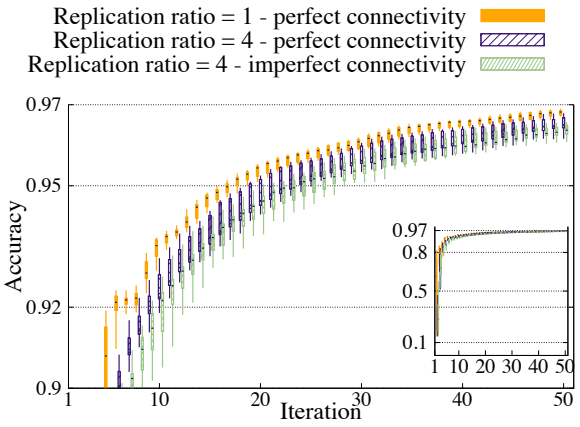

Figure 2.5: Robustness of IPLS in the presence of intermittent connectivity.

Chapter 3

IPLS over a Decentralized Storage Network

3.1 Building IPLS on top of Decentralized Storage

3.1.1 Notation

Let A be the set of all aggregators in the system. We denote as A_i the set of aggregators responsible for the partition i of the model, and a_{ij}, the aggregator j responsible for the partition i. Furthermore, we denote as T the set of all trainers and T_{ij} the set of trainers that send their gradients for the partition i to the aggregator a_{ij}. It is required that $\varnothing = \bigcap_{j \in |A_i|} T_{ij}$ and $T = \bigcup_{j \in |A_i|} T_{ij}$. This means that aggregators responsible for the partition i receive gradients for the partition i from distinct trainers. In practice, these sets can be formed and maintained by a initialization protocol between IPLS participants in the beginning of each training round as shown in 2 or by the bootstrapper itself. Also let p be the number of partitions the model gets segmented.

3.1.2 Security Assumptions

In [16], authors assumed that all parties involved in the protocol were fully trusted. However, this is an unreasonable assumption to make. Although we can assume that the bootstraper is trustworthy because he wants his federated learning campaign to "succeed", meaning that it wants the resulting machine learning model to have as high accuracy as possible, this is not true for all the other entities. Aggregators, in some cases, might have incentives to

behave maliciously. For example, they might perform the aggregation of the gradients with much less accuracy than they should do or even deny receiving gradients from some trainers to save some energy. Such attacks can significantly reduce the accuracy of the model and its convergence rate, resulting in many additional Federated learning rounds and, in general, slower completion time.

In addition, trainers can also perform various attacks. The most simple attack is poisoning their gradients by just adding noise. However, some more sophisticated attacks can be performed, for example, label flipping, resulting in significant misclassifications on the trained model. Last but not least, IPFS nodes that consist of the decentralized storage network can misbehave. IPFS nodes can send incorrect data to the retrievers or simply deny storing some data to save space.

In this work, we will **not** consider the case where IPLS trainers might behave maliciously. For an interested reader, [34, 35, 36, 37, 38] are some interesting works that try to deal with this problem. Instead, we will assume in this work that IPLS aggregators might behave maliciously in the sense that they can alter or drop gradients from some trainers. To deal with such malicious behavior, IPLS must guarantee that an updated partition comes from all the correct sum of the gradients sent from all the trainers. In addition, we assume that the IPFS storage network guarantees the availability of data. That can be done by an incentivized storage network ([20]) or IPFS protocols (i.e., IPFS cluster [1]). However, we assume that IPFS nodes might send incorrect data to the retrievers.

3.1.3 Indirect Communication Scheme

As with many other decentralized federated learning protocols [39, 40, 41] IPLS assumed that participants could communicate directly with each other. However, especially for the system setup of Federated learning, this assumption might not always hold. For various networking issues (e.g., Firewalls, NATs, volatile mobile IP addresses) or even because of the type of network (e.g., 3G, 4G), participants, which commonly are smartphones or IoT devices, can not always establish direct communication links. In addition, even if they could establish direct communication links, their communication might be highly inefficient, especially in terms of energy consumption. Lastly, if an aggregator a_{ij} dropouts, then the trainers T_{ij} will have to send another aggregator their gradients, or else their contribution would be

[1] `https://cluster.ipfs.io/`

lost.

To deal with the plethora of issues arising due to direct communication, we modify the original IPLS protocol so that reliable communication happens indirectly through the means of a decentralized storage network. Nodes that comprise the decentralized storage network are the IPFS nodes, which commonly are relatively powerful devices (e.g., laptops or desktops) with fast and stable internet connectivity. Those IPFS nodes run a full IPFS daemon connected to a public or private IPFS network from a technical perspective. On the other hand, IPLS participants might not run a full IPFS daemon locally, but they might be connected to an IPFS gateway node. In practice, IPLS participants communicate with each other by uploading or downloading their data (gradients, partial updates, or updates) to and from the decentralized storage network.

3.1.4 Directory Service

To route and retrieve data from the IPFS storage network and verify their validity, IPFS and also many other p2p file systems rely on a secure hash function (e.g., SHA-256). As a result, IPLS participants have to know the secure hash of the data they have to download. However, there is no way to learn such a hash function on their own. As a result, an entity must exist, that maps some addressing meta-information about the data to its corresponding IPFS hash and other auxiliary information. It is crucial that this addressing information should be known to all participants and should be easily computed. Such addressing meta-information can be, for example, the id of the IPLS participant who uploaded the data.

To overcome this issue, we introduce the directory service, whose primary responsibility is to map such addressing information to the corresponding hash of the data into a directory maintained strictly by the directory service. In IPLS, such addressing information can be described by the tuple $(uploader\_id, type, partition\_id)$, where uploader id is the IPFS id of the IPLS participant who uploaded the corresponding data, the "type" is a string that indicates if the data are gradients or updates, and the $partition\_id$ is an integer indicating the partition of the model that the data belong. Note that in IPLS, the directory service maps the tuple mentioned above with the IPFS hash of the data and their Pedersen commitment. The reason why the directory service stores the Pedersen commitment will be evident in later sections.

In general, we require the directory service to be immutable, meaning that no one can write a new hash to a tuple already written in the directory. If the directory service were

not immutable, then aggregators or IPFS nodes would easily tamper with the mappings of the directory and replace the hashes of correct data (e.g., gradients) with hashes of poisoned data. It is also easy to observe that both the writes and reads should be relatively fast because the efficiency of communication between IPLS participants depends on how efficient the directory service is. One natural question that arises is who will take the responsibility of becoming the directory service. The bootstraper can become the directory service because, in reality, the directory service receives magnitudes times less data than all the aggregators combined, and the bootstraper will behave honestly because he wants his federated learning campaign to succeed. Though, a blockchain can also take the role of becoming the directory service. Keep in mind that in IPLS, the blockchain takes the role of a directory service and not the role of the aggregator.

3.1.5 Modified IPLS protocol

Each IPLS round in our modified IPLS protocol, can be described in three distinct phases, the **preprocessing phase**, the **training phase** and the **aggregation phase**. Figures 3.1 and 3.2 capture the main steps of the protocol while algorithms 1 and 2 give a much more detailed description of the tasks performed by each entity in the system.

Preprocessing phase

This phase starts at the beginning of a new IPLS round. In this phase, the directory service clears all the entries of its directory that correspond to gradient writes (i.e., they are of the form $(uploader\_id, \text{“}gradient\text{”}, partition\_id)$). Next, from the IPLS participants that are online, it generates the sets A, A_i, T and T_{ij} and informs IPLS participants about their new responsibilities. Note that the way those sets are generated is out of the scope of this work and is left for future work. Also, keep in mind that those sets can be determined solely by the IPLS participants, as it happens with [16].

Furthermore, the directory service selects a time T_{train},T_{aggr}, which are nothing more than long integers presenting time in UTC. T_{train} is used so that IPLS trainers will know when the training phase stops, and T_{aggr} is used the same way but for the aggregation phase. Keep in mind that whenever T_{train} elapses, then the directory service stops receiving writes for gradients from the trainers. In the end, the directory service broadcast to all IPLS participants a message containing the T_{train}, T_{aggr} and the IPLS round indicating the beginning of the

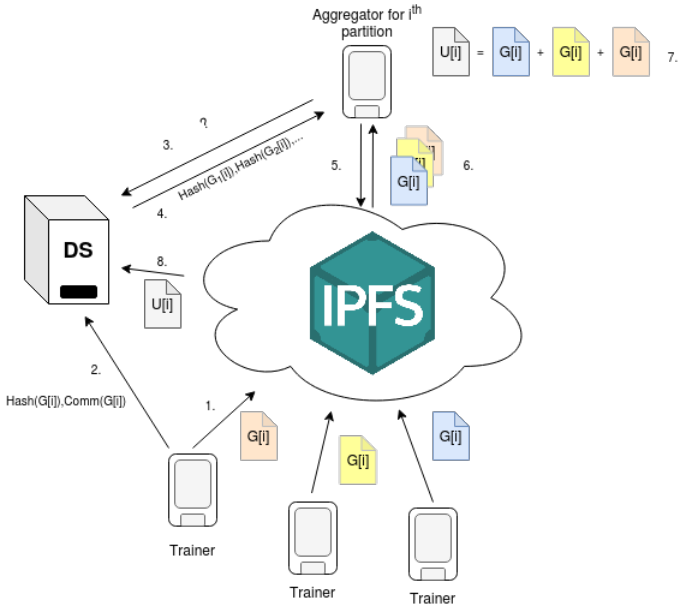

Figure 3.1: Figure depicting one iteration of the IPLS protocol, with only one aggregator responsible for the i^{th} partition and three trainers. The main steps of the protocol are **1.** Upload the gradients to the IPFS network, **2.** Write to the directory service, **3.** aggregator queries the directory service, **4.** Receives the hashes of the gradients, **5.,6.** retrieves them from the IPFS **7.** aggregates them and uploads the update to the IPFS, **8.** the directory service verifies the validity of the update before written to the directory

training phase.

Training Phase

Whenever IPLS trainers receive the message broadcasted by the directory service, they read the directory service to get the hash of the updated partition for each partition of the model. This is done by reading from the directory the tuples $(\text{“}update\text{”}, i), \forall i \in [P]$. Whenever those hashes are retrieved, trainers use the $ipfs.get(\cdot)$, with input, the hash for each partition to retrieve the updated partition from the IPFS network. After collecting all the updated partitions, they construct the machine learning model using the concatenation rule and train it using their local data (i.e., M = U[1]||U[2]||...||U[p]).

When training is completed, then each trainer checks if his current time is less than T_{train}. If this is false, they abort and wait for the next round. Otherwise, they get the gradients of the trained model, which currently are simply the parameters of the trained model. Next, they segment the gradients into the corresponding partitions; for each partition, they append the number 1 and then call the $ipls.add(\cdot)$, for each gradient partition to upload it to the IPFS network. The function $ipls.add(\cdot)$, first computes the IPFS hash and the Pedersen commit-

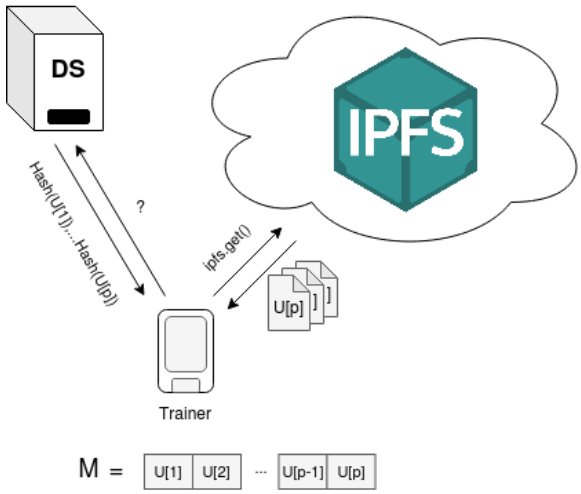

Figure 3.2: A snapshot of how trainers download the up-to-date model. They simply query the directory service and upon receiving the hash of each partition they retrieve the updates from the IPFS network. Finally they reconstruct the model by concatenating the updated partitions.

ment of its input. Then given a predetermined IPFS node, each trainer sends to it the input data and waits to get an acknowledgment from the IPFS node that it stored the data in the IPFS network (by calling $ipfs.add(\cdot)$). Note that this procedure can be done in parallel for each partition of the model.

Whenever the acknowledgment is received, then the trainer sends to the directory service a message containing the addressing information and the hash of the gradient partition (i.e. $(trainer\_id, \text{“}gradient, i)$, $Hash(G[i]), Commit(G[i])$). Upon receiving that message, the directory service checks 1) if the current time is more than T_{train}, 2) if a mapping with the same addressing information is already written to the directory. If none of them is true the directory simply sets $D[(trainer\_id, \text{“}gradient, i)] \rightarrow Hash(G[i]), Commit(G[i])$.

The directory service also maintains for each aggregator a_{ij} a queue, which stores the IPFS hashes for the partition i for each trainer in T_{ij}. At the training phase, each aggregator a_{ij} polls the directory service simply by reading all the IPFS hashes from its queue ($Q_{a_{ij}}$). For example consider that in a given time, $Q_{a_{ij}} = [Hash_1, Hash_2, Hash_3]$. In the first poll a_{ij} will receive $Hash_1, Hash_2, Hash_3$, and the directory service will clear the queue $Q_{A_{ij}} = []$. If nothing is put in the $Q_{A_{ij}}$ until the next poll, then in the next poll, a_{ij} will receive nothing from the directory service.

Whenever aggregators receive new hashes from the directory service, they simply use the function $ipfs.get(\cdot)$, to download the gradient partition that corresponds to the given hash.

Aggregation Phase

Aggregation phase is dominated only by the aggregators and the directory service while the trainers remain idle waiting for the next IPLS round. At the very beginning of the aggregation phase, the directory service, for each partition i computes the digest of the updated partition which is $Commit(U[i]) = \prod_{j \in T} Commit(G_j[i])$ and also the digest of the partial update for each $Commit(PU_j[i]) = A_{ij}$ which is $\prod_{k \in T_{ij}} Commit(G_k[i])$. Note that those computations are very efficient and add only a minor overhead.

Each aggregator a_{ij}, at the start of the aggregation phase, sends its last query to receive the remaining hashes of the gradient partitions from T_{ij}, if any. Then it receives $Commit(PU_j[i])$ $\forall j \in A_i - a_{ij}$ from the directory service. Whenever a_{ij} downloads all the gradients from T_{ij}, aggregates them, and computes its "partial" update $PU_j[i] = \sum_{k \in T_{ij}} G_k[i]$, which is uploaded to the IPFS network by calling $ipls.add(PU_j[i])$. Instead of using the directory service to make the IPFS hash of the computed partial update "publicly known", each a_{ij} multicasts the IPFS hash of $P_j[i]$ to all the other aggregators $\in A_i$. Whenever an aggregator responsible for partition i receives a hash from an another aggregator in A_i, a_{ij}, it downloads the partial update, computes its Pedersen commitment ($Commit(PU_j[i])$), and then checks it is equal with that given by the directory service $Commit_{DS}(PU_j[i])$. If $Commit(PU_j[i]) = Commit_{DS}(PU_j[i])$, then with overwhelming probability, the partial update from $a_j[i]$ is the correct one. In case the equation did not hold, then the aggregator downloads the gradients for the partition i been uploaded from the trainers in T_{ij}, and uploads the correct $PU_j[i]$.

Whenever an aggregator for the partition i computed the U[i], uploads it to the IPFS network and sends its IPFS hash to the directory service. If the directory service has not received any write for the U[i], it downloads it and checks if $Commit(U[i]) = Commit(U_{recv}[i])$. If the equation holds, the directory service stores the tuple $(\text{"}update, i) \rightarrow Hash(U_{recv[i]})$, and stops receiving writes for updates of the partition i. Whenever the updated partitions for all the partitions are gathered, the directory service informs the system that the IPLS round is finished and a new one begins. As it is obvious, the training process can finish before T_{aggr}. In reality, T_{aggr} is used mainly for robustness guaranting that the training process will not stall permanently. This can happen whenever there is only one aggregator responsible for a partition, and this aggregator goes offline.

Algorithm 1 Functions of the directory service and IPFS nodes

function $directory\_service(D = \{\}, A, T)$
  $msg \leftarrow$ recv()
  if $msg.tag = grad\_write \wedge t_{current} < t_{train}$ **then**
    if $msg.addr \notin D$ **then**
      $D[msg.addr] \leftarrow msg.Hash, msg.Commit$
      $t \leftarrow msg.addr.trainer\_id$
      $i \leftarrow msg.addr.partition$
      $a_{ij} \leftarrow get\_Aggr(t)$
      $Q_{a_{ij}}.append(msg.Hash)$
    end if
  end if
  if $msg.tag = upd\_write \wedge t_{current} < t_{aggr}$ **then**
    if $msg.addr \notin D$ **then**
      $i \leftarrow msg.addr.partition$
      $U \leftarrow ipfs.get(msg.Hash)$
      if $Comm(U) = \prod_{t \in T} Comm(G_t[i])$ **then**
        $D[msg.addr] \leftarrow msg.Hash$
      end if
    end if
  end if
  if $msg.tag = read$ **then**
    $i \leftarrow msg.partition$
    $a_{ij} \leftarrow msg.sender$
    $send(a_{ij}, Q_{a_{ij}})$
    $Q_{a_{ij}}.clear()$
  end if
  if $Ended(T_{train})$ **then**
    for each $a_{ij} \in A$ **do**
      $Comm(PU_j[i]) \leftarrow \prod_{t \in T_{ij}} Comm(G_t[i])$
      $Bcast(A_i, PU_j[i])$
    end for
  end if
end function

function $ipfs\_node$
  while $True$ **do**
    $msg \leftarrow recv()$
    $ipfs.add(msg.data)$
  end while
end function

Algorithm 2 Algorithms for one FL training iteration

function upload($addr, data, WriteDs$)
  $cid \leftarrow hash(data)$
  $put(ipfs\_peer, data)$
  if $WriteDs = True$ **then**
    $send(directory, [addr, cid])$
  else
    $pub(A_i, [addr, cid])$
  end if
end function

function trainer(M, A_t)
  $gradU \leftarrow$ train(M) ▷ train model and produce gradient updates
  if $t_{current} > t_{train}$ **then** ▷ Abort if didn't train it in time
    Abort iteration i
  end if
  for each $i \in M.parts$ **do** ▷ Upload gradient updates ∀ partition
    $upload((id, i, iter, gradient), [gradU[i], 1], True)$
  end for
  $T_{train,t_aggr} \leftarrow wait\_new\_round()$
  for each $i \in M.parts$ **do** ▷ get updated partitions
    while cid == NILL **do** ▷ check the DS until you get the Cids
      $cid \leftarrow send(directory, (\text{“}update\text{“}, i))$
    end while
    $modU[i] \leftarrow download(cid)$ ▷ download updated partitions
    $modU[i] \leftarrow modU[i][: size - 1]/modU[i][size - 1]$
  end for
  $M \leftarrow modU$ ▷ build next fully updated model
end function

function aggregator(A^i, T_a, t_{aggr})
  while $T_{ij} \neq \varnothing$ **do** ▷ get gradient updates from my trainers
    $cids \leftarrow send(directory, (a, i))$ ▷ Check if new Cids commited
    for each $(t, cid_t) \in Cids$ **do**
      $gradU^i[t] \leftarrow download(cid_t)$ ▷ Download gradients
      $T_{ij} \leftarrow T_{ij} - t$
    end for
  end while
  $modelU^i[a] \leftarrow \sum gradU^i[t]$ ▷ own updated partition
  $upload((id, i, iter, partial\_update), modelU^i[a], False)$
  $Comm_i \leftarrow recv(directory)$
  while $t_{curr} < t_{aggr} \wedge A_i \neq \varnothing$ **do** ▷ sync with $A_i - A_{ij}$
    $cid, a' \leftarrow recv(A_i)$ ▷ Check if new Cids commited
    $modelU^i[a'] \leftarrow download(cid)$
    if $Comm_i[a'] = Commit(modelU^i[a'])$ **then**
      $A_i \leftarrow A_i - a'$
    end if
  end while
  $modelGlobU^i \leftarrow \sum modelU^i[a']$ ▷ globally updated partition
  $upload((i, iter, update), modelGlobU^i, True)$
end function

3.2 Merge And Download Mechanism

As it might have been obvious from the description of the protocol, the data each aggregator has to receive grow linearly with the number of aggregators responsible for the same partition A_i and the number of trainers T_{ij}. However, it is possible to further reduce the data the aggregator has to download. To achieve that we take observe that there is a possibility that an IPFS node, might hold gradients from multiple trainers that belong to the same T_{ij}. As a consequence, the aggregator instead of downloading from the IPFS node the gradients one-by-one, it could simply request to the IPFS node aggregate those gradients on its behalf, and send him the aggregated result. This mechanism is called Merge and Download 3.3.

To further utilize that concept, we allow aggregators, at the beginning of each iteration or in a couple iterations, to select a set of IPFS nodes. For an aggregator a_{ij}, this set of IPFS nodes is called providers of a_{ij} and it is symbolized as P_{ij}. Trainers in T_{ij} send their gradients for partition i, they choose one IPFS node in P_{ij} and send to it their gradients for the partition i. Whenever gradients from all T_{ij} were selected then the aggregator a_{ij} sends a merge request to its providers who aggregate its stored gradients and send back to the aggregator their results. Note that the aggregator instead of downloading data proportional to $|T_{ij}|$ it now downloads data proportional to $|P_{ij}|$. The same mechanism can be applied for the partial updates. To be more specific a set of IPFS nodes P_i can be selected in order to store the partial updates for the partition i of the model.

A natural question that arises, is how large the P_{ij} should be. If P_{ij} was too small, in the extreme case $|P_{ij}| = 1$, then the single IPFS node would become congested by receiving all the gradient partitions from T_{ij}, while the data the aggregator would have to receive would be the minimum. On the other hand, if $|P_{ij}|$ was extremely large, then the uploading delay from trainers perspective would be low, however the merge and download would probably have no significant impact, because the probability that an IPFS provider hold many gradients from T_{ij} would be small. The number of Providers that can lead to optimal aggregation time is $\sqrt{|T_{ij}|}$, as we will show next. If we assume that all IPFS nodes have roughly the same download speed d then the time it takes for an aggregator a_{ij} to download all its data is $\tau = Partition\_Size \cdot (|T_{ij}|/(d|P_{ij}|) + |P_{ij}|/b)$, where b is the download speed of the aggregator. To minimize τ, we compute $\frac{\partial \tau}{\partial P_{ij}} = 0$, which results to $b \cdot |T_{ij}|/d = |P_{ij}|^2$, which confirms our previous observation.

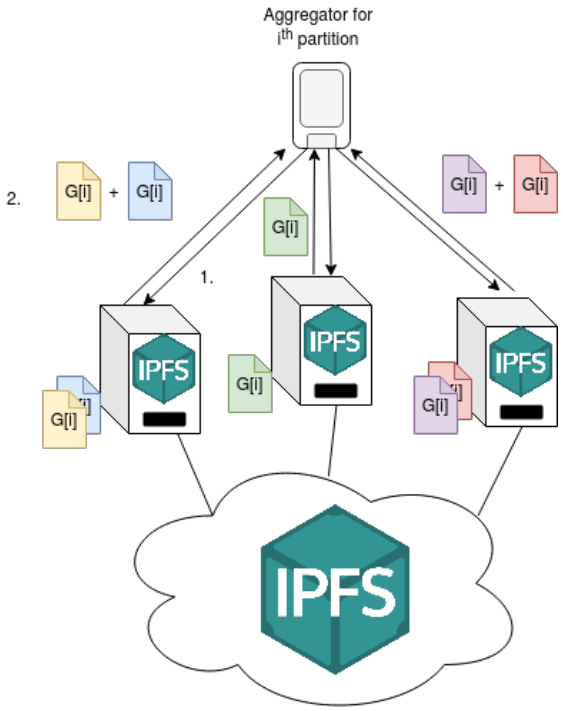

Figure 3.3: Figure depicting merge and download concept with three IPFS providers. The aggregator sends merge request to his providers (**1.**) and then it receives the aggregated results (**2.**).

3.3 Experimental Evaluation

To conduct the experiments and measure the performance of the modified IPLS protocol, we used a testbed which consists of AWS c5ad.12xlarge instance running Ubuntu 18.04 LTS with 48 virtual CPUs and 96GB RAM. Regarding the functionalities of the protocol, we measured the impact of the number of aggregators responsible of the same partition, in terms of aggregation speed, the impact of merge and download scheme with variable number of IPFS providers and also the efficiency of writes in the directory service with variable number of partitions. In order to make the measurements as realistic as possible, we used the mininet emulator and assumed that both trainers and aggregators have the same bandwidth capabilities.

3.3.1 Aggregation Performance vs. Variable $|A_i|$

In this experiment, we measured the performance of the aggregation with variable number of aggregators responsible for the same partition. For those experiments we deployed 16 trainers, 8 IPFS nodes and variable number of aggregators. We segmented the model into 4 partitions of size 1.1 MB each, and each aggregator was responsible for only one partition of

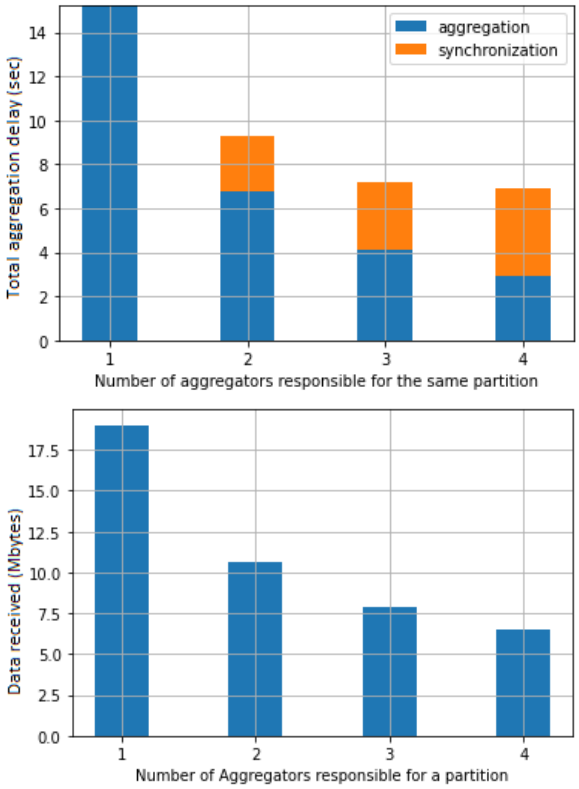

Figure 3.4: Total aggregation delay (top) and total size of data received by an aggregator (bottom) in each iteration, vs. the number of aggregators assigned to each partition.

the model. Firstly, we used 4 aggregators and as a result $|A_i| = 1$ for each i. Then we used 8 aggregators so that $|A_i| = 2$, and so on until reaching $|A_i| = 4$. Last but not least IPLS participants had communication bandwidth of 20MBps.

Note that in figure 3.4 the total aggregation delay is the time from the beginning of the iteration until the computation of the updated partition. In the Figure 2, the y axis represents the amounts of data been received by a single aggregator in an IPLS iteration. Before going into the details of our results, it would be wise to find the theoretical asymptotics of the aggregators. If we roughly assume that an aggregator is responsible for only one partition, and each aggregator for the same partition receives approximately equal number of gradients, then the amounts of data each aggregator has to download is $D_{a_{ij}} = (\frac{|T|}{|A_i|} + |A_i| - 1) \cdot Partition\_Size$ and the delay would be approximately $T_{aggr} = D_{a_{ij}}/BW$, where BW is the bandwidth of the aggregator.

As can be easily verified, both figures in 3.4 agree with our theoretical assumption. To be more specific, we can see that as the number of aggregators responsible for the same partition increases, both the aggregation delay and the downloaded data significantly decrease. Moreover, as the number of aggregators for the same partition increases, so does the overhead for

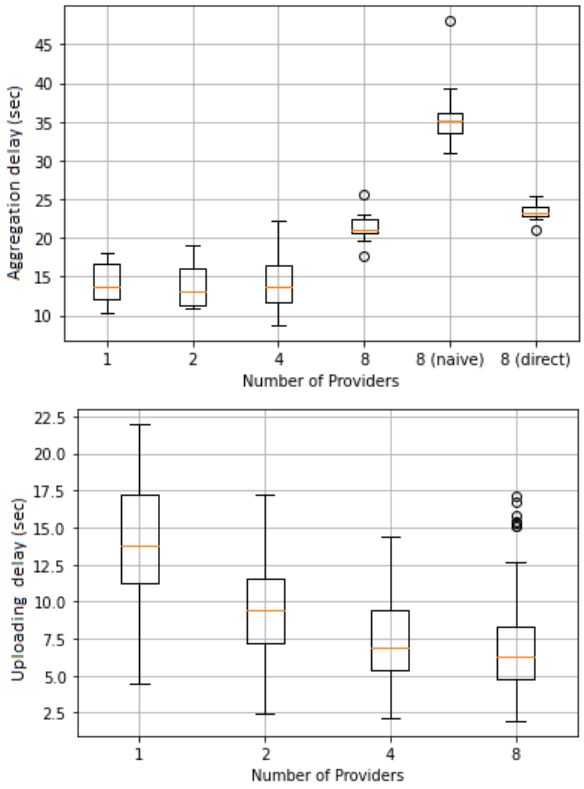

Figure 3.5: Aggregation (top) and uploading (bottom) delays with variable number of providers.

synchronization, which is simply the download and aggregation of the partial updates. In fact, if we continued our experiment, we would observe that the both the total aggregation delay and the data received would start increasing again.

3.3.2 Impact of Merge and Download

To measure the impact of merge and download mechanism, we contacted an experiment, that consists of 16 trainers, an aggregator responsible for a partition of size 1.3MB and a variable number of IPFS Providers. Both the trainers and the aggregator have bandwidth of 10MBps while IPFS providers had bandwidth of 100MBps. In those experiments, we measured the aggregation delay, which is the time interval in which the aggregator receives the first write from a trainer in the directory service, until it computes the updated partition. In addition, we measured the uploading delay, which is the time needed for a trainer to upload its partition to the IPFS network.

From our theoretical result, it is expected that with 4 IPFS providers, we will have the minimum total aggregation time, because both uploading delay and aggregation delay will be small. That can easily be verified by the Figure 3.5, because the aggregation delay with

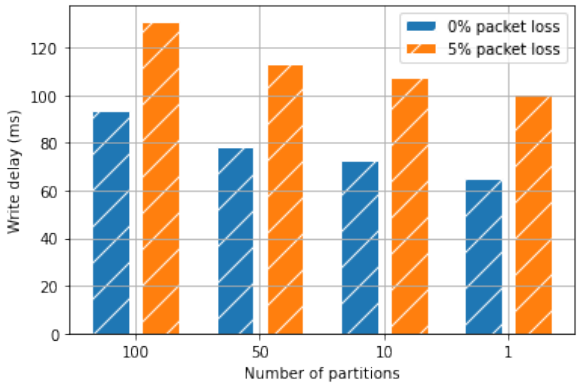

Figure 3.6: Write delay with 0% and 5% packet loss and variable number of partitions

4 IPFS providers is relatively the same as the aggregation delay with 1 or 2 IPFS providers. Moreover, the uploading delay with 4 IPFS providers is similar to the uploading delay of 8 trainers, while with 1 IPFS provider, we can see that it becomes congested receiving the gradients from the trainers.

In addition, we used this experiment, to compare merge and download, with simple indirect communication (8 (naive)) and also direct communication (8 (direct)). As it can be seen, purely indirect communication, gives the worst aggregation delay. However, keep in mind, that this happens mainly because the aggregator downloads sequentially the gradients, by using $ipfs.get()$ method, which adds significant overheads, probably due to querying the IPFS DHT. We could make the downloading much more efficient, by concurrently downloading many files by using multiple threads. Overall, the results of this experiment show that if we want to deploy indirect communication between IPLS participants, then Merge and Download is a crucial mechanism for the efficiency of the IPLS protocol.

3.3.3 Performance of Directory Service

In this experiment, we measured the performance of the writes in the directory service, by inspecting the time it takes for a trainer to send the write message to the directory service, until it receives the acknowledgment that its write is done correctly. We conducted this experiment with 1000 trainers, and a directory service that it had bandwidth of 100MBps. To make the writes as fast as possible, we made a modification in the original protocol and instead of sending write messages via pub/sub we are sending them over UDP. That is because IPFS pub/sub is built over TCP which may add noticeable overhead [32]. Roughly, as shown in 3.6, write delay depends linearly to the number of model partitions, and as it is obvious the less

we partition the model, the faster the writes. Moreover, due to retransmissions and timeouts, the write delay with 5% packet loss in the network, is much larger than the write delay with 0% packet loss. All in all, we can see that even with a moderately large number of trainers, the writes remain quite fast even with 5% packet loss.

Chapter 4

On the Efficiency of Directory Service

4.1 Matrix commitments / Hierarchical vector commitments

We introduce a new cryptographic scheme called matrix commitment. As its name suggests, a matrix commitment scheme enables committing to an order set of vectors instead of elements. As in vector commitments, matrix commitments can be described by the tuple of four probabilistic polynomial time algorithms $(Gen, Commit, Open, Verify)$.
$\mathbf{pp} \leftarrow \mathbf{MC.Gen}(\mathbf{1^k}, \mathbf{n}, \mathbf{m})$. Takes as input the security parameter k, and also the dimension of the matrix. Return some public parameters.

$\mathbf{C}, \mathbf{aux} \leftarrow \mathbf{MC.Commit}(\mathbf{pp}, \mathbf{M})$. Takes as input the public parameters and the matrix $M \in D^{m \times n}$ and return a constant sized digest and also some auxiliary information.

$\mathbf{W_i}, \mathbf{M_i} \leftarrow \mathbf{MC.Open}(\mathbf{pp}, \mathbf{aux}, \mathbf{M}, \mathbf{i})$. Takes as input the matrix M, the auxiliary information, the public parameters pp, and also the index of the row we want to open. Returns the i^{th} row of the matrix and also a witness W_i that M_i is indeed the i^{th} row of the matrix.

$\mathbf{0}, \mathbf{1} \leftarrow \mathbf{MC.Verify}(\mathbf{pp}, \mathbf{C}, \mathbf{M_i}, \mathbf{W_i}, \mathbf{i})$. Returns 1, if M_i is the i^{th} row of the matrix, else 0.

Let **m** be the number of rows and **n** be the number of columns. In IPLS setting, we set **m** to be the number of partitions, and **n** be the number of elements of each partition. Usually, it holds that $\mathbf{m} \leq \mathbf{n}$.

Matrix commitments must fulfill the **correctness** property as defined in the equation 1.2

for elements but also for rows and columns. Matrix commitments must be **row binding** meaning that given an PPT adversary A and public parameters pp, the probability of finding two **different** rows of the matrix $\vec{M}_i, \vec{M}'_i$ and two witnesses W_i, W'_i, is negligible with respect to the security parameter k. Formally,

$$Pr\left(\begin{array}{c} C, W_i, W'_i, M_i, M'_i \leftarrow A(pp); \\ 1 \leftarrow MC.Verify(pp, C, M_i, W_i, i) \wedge \\ 1 \leftarrow MC.Verify(pp, C, M'_i, W'_i, i) \end{array}\right) \leq negl(k) \tag{4.1}$$

Last but not least, we require Matrix commitments to be additivelly **homomorphic**, meaning that $Commit(pp, M_1) \cdot Commit(pp, M_2) = Commit(pp, M_1 + M_2)$.

4.1.1 Matrix Commitment Based on RSA Assumption

Before discussing our proposed construction, it is important to clarify that we can construct matrix commitments in a trivial way, by simply using vector commitments that enable subvector openings. For example, commit could be used as it is and in the open we could simply compute the witness of opening a subvector that represents the row of the corresponding matrix. Someone could also precompute the witness of each row of the matrix, in the commit algorithm, so that open would have $O(1)$ complexity. In the vector commitment based on RSA Assumption for example, we could precompute the witnesses of each row, the same way as we compute the witnesses for each position, but halting prematurely in the m^{th} level of the recursion. That, would require $O(m \cdot n \cdot log(m))$ time. In reality, we would like something that its complexity depends solely on m and not n. The construction we present bellow, offers faster witness precomputation, but with the requirement that $m < n$. Concisely, our scheme combines integer commitment schemes [42, 43], with the RSA vector commitment scheme presented in 1.1.3. In fact, we use integer commitments to commit each individual row of the matrix, and then, use the RSA based vector commitment to "bind" them together.

$\mathbf{pp} \leftarrow \mathbf{MC.Gen}(\mathbf{1^k}, \mathbf{n}, \mathbf{m})$. Take input the security parameter k, and also the dimension of the matrix. Select two random safe primes p_1, p_2 of length n/2 and compute N = $p_1 p_2$. Then select n random distinct integers $\{r_i\}_{i \in [n]} \in Z_N$, and n random distinct primes $\{e'_i\}_{i \in [m]}$. Then pick a random element g $\in Z_N^*$ and compute $S_i = g^{2r_i}$. The reason why we multiply r_i

by 2, is because we want $S_i \in QR[N]$ (i.e the group of quadratic residues) as in [43] [1]. Set pp = $\{N, e'_1, e'_2, ..., e'_m, S_1, S_2, ..., S_n\}$.

$\mathbf{C}, \mathbf{aux} \leftarrow \mathbf{MC.Commit}(\mathbf{pp}, \mathbf{M})$. For each row of the matrix compute $C_i = \prod_{i\in[n]} S_i^{v_i} mod N$. Finally compute $C = \prod_{i\in[m]} C_i^{\prod_{j\in[m]-i} e'_j} \; mod \; N$. Set aux = $\{C_1, C_2, ..., C_m\}$.

$\mathbf{W_i}, \mathbf{M_i} \leftarrow \mathbf{MC.Open}(\mathbf{pp}, \mathbf{aux}, \mathbf{M}, \mathbf{i})$. Compute $W_i = \prod_{j\in[m]-i} C_j^{\prod_{k\in[n]-\{i,j\}} e'_k} \; mod \; N$. Return W_i and the i^{th} row of the committed matrix M.

$\mathbf{0}, \mathbf{1} \leftarrow \mathbf{MC.Verify}(\mathbf{pp}, \mathbf{C}, \mathbf{M_i}, \mathbf{W_i}, \mathbf{i})$. Compute $C_i = \prod_{k\in[n]} S_k^{M_{ik}} \; mod \; N$ and finally check if $C_i^{\prod_{k\in[n]-i} e'_k} W_i^{e'_i} = C$.

It is trivial to see that the scheme is correct and also homomorphic. It is homomorphic, because $MC.Commit(M_1) \cdot MC.Commit(M_2) = \prod_{i\in[m]} C_{1i}^{\prod_{j\in[m]-i} e'_j} \cdot \prod_{i\in[m]} C_{2i}^{\prod_{j\in[m]-i} e'_j} = \prod_{i\in[m]} (C_{1i}C_{2i})^{\prod_{j\in[m]-i} e'_j} = \prod_{i\in[m]} (\prod_{j\in[n]} S_i^{M_{1ij}+M_{2ij}})^{\prod_{j\in[m]-i} e'_j} = MC.Commit(M_1 + M_2)$

Proof of Row Binding Property

To prove that the scheme is row binding, we introduce a probabilistic polynomial time adversary **A** that can win the row binding game with non-negligible probability in the security parameter. Let $P[W]$ be that probability. We create an adversary **B** that wants to win an RSA challenge and uses as a subroutine the adversary **A**. Specifically, **B** receives from a challenger a triplet of values (N, e, g) and wants to find y s.t $y^e = g \, mod \, N$. **B** selects n,m, computes S_i the exact way as $MC.Gen()$ would work, but selecting g as a base. Then selects at random $i' \in [n]$, and sets $e'_{i'} \leftarrow e$. As a result $pp \leftarrow \{N, e'_1, e'_2, ..., e'_m, S_1, ..., S_n\}$. Then calls A(pp), and receives rows M_i, M'_i, W_i, W'_i, C. Then he checks if the rows are different. If this is true, then $C_i^{\prod_{k\in[n]-i} e'_k} W_i^{e'_i} = C_i'^{\prod_{k\in[n]-i} e'_k} W_i'^{e'_i}$, or $(C_i/C'_i)^{\prod_{k\in[n]-i} e'_k} = (W'_i/W_i)^{e'_i}$

In case $W'_i = W_i$, then there must be $C_i = C'_i$, with $M_i \neq M'_i$. However by [43] we know that finding M_i, M'_i s.t $M_i \neq M'_i \wedge C_i = C'_i$, is equivalent to factoring the modulus N, meaning that we can obtain a non-trivial factor [2]. We call that event DLOG and P_{DLOG} be the proba-

[1] We could also pick simply any random number r_i, as it happens with [42]. However, [42] does not prove the binding property of the commitment scheme when we commit to a vector

[2] Alternatively, [28, 44] we can see that this can happen only with negligible probability

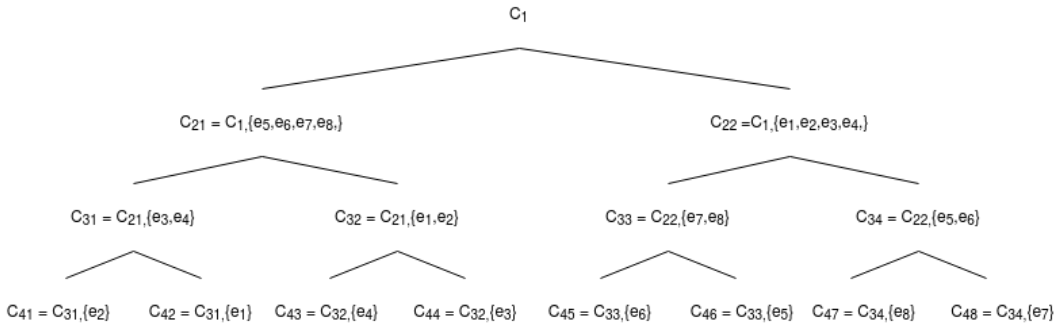

Figure 4.1: A simple resemble of the recursion, when computing $C_{12}, C_{13}, ..., C_{1m}$ from C_1. Note that in the paradigm, m = 9 so we need to find 8 elements for C_1. In addition, we represent $C_i^{e_{k_1} e_{k_2} \cdots}$ with $C_i, \{e_{k_1}, e_{k_2}, ...\}$.

bility that happens. Otherwise, if $W_i' \neq W_i$, then **B** can compute $\delta = \sum_{j \in [n]} 2r_j(M_{ij} - M'_{ij})$, and result in $g^{\delta \prod_{k \in [n]-i} e'_k} = (W'_i/W_i)^{e'_i}$. Since $gcd(\delta \prod_{k \in [n]-i} e'_k, e'_i) = 1$, we can efficiently compute α, β such that $\alpha\delta \prod_{k \in [n]-i} e'_k + \beta e'_i = 1$. By applying shamir's trick [45] we obtain g = $(g^{(\alpha\delta \prod_{k \in [n]-i} e'_k)}(W'_i/W_i)^{\beta})^{e'_i}$. As a result **B** returns $y = g^{(\alpha\delta \prod_{k \in [n]-i} e'_k)}(W'_i/W_i)^{\beta}$ to the challenger. Let P_{RSA}, be the probability that **B** wins the RSA challenge. Because the challenge is uniformly distributed so does e. As a result, the public parameters pp **B** generates, will the same distribution as if they where in honestly generated. Thus $P_{RSA} = P[W \wedge i = i' \wedge \neg DLOG] = P[W] \cdot P[i = i']P[\neg DLOG] = P[W](1 - P_{DLOG})/m$

Complexity Evaluation

Considering the above construction, $MC.Commit$ can be computed naively in $O(n \cdot m) + O(m^2)$ time, the opening of one row is computed in $O(m^2)$ time, and the verification is done in $O(n + m)$ time. However as in vector commitment example, there can be a little space for some improvement.

Note that we could pre-compute the proofs for each row of the matrix in the following way. Having $\{C_1, C_2, ..., C_m\}$, we can compute in $O(m^2 log(m))$ time m^2 elements $\{C_{12}, C_{13}, ..., C_{1m}, ..., C_{m(m-1)}\}$ = $\{C_1^{\prod_{k \in [n]-\{1,2\}} e'_k}, C_1^{\prod_{k \in [n]-\{1,3\}} e'_k}, ..., C_m^{\prod_{k \in [n]-\{m,m-1\}} e'_k}\}$. This can be done, by computing using the root factor for each C_i, with prime set $e''_i \leftarrow e' - e'_i$ as it can be seen in 4.1. Computing the root factor for an commitment C_i takes $O(mlogm)$, thus for all m commitments we will need $O(m^2 logm)$ time. Then to find the witness of a row of the matrix we can compute $W_i = \prod_{j \in [m]-i} C_{ji} mod N$, which can be done in $O(m)$ field multiplications. Thus we can pre-compute the witness W_i for each row of the matrix in $O(m^2(log(m) + 1))$ time. This procedure can be embedded in the Commit algorithm, and the auxiliary parame-

ters could simply contain the witness for each row of the matrix. In that case, the Open will simply be executed in $O(1)$ time.

Witnesses Aggregation

This construction also enables witness aggregation. Let M_1,M_2 be two matrices, C_1,C_2 their corresponding matrix commitments, and W_{1i}, W_{2i} be the witnesses of opening the row i of the matrix. Then, it holds that $W_{1i} \cdot W_{2i} \, mod \, N$, is the opening witness for the row i of the matrix $M_1 + M_2$. To see why this is the case, let C_3 be the commitment of the matrix $M_1 + M_2$, then $W_{1i} \cdot W_{2i} mod N = \prod_{j \in [m]-i} (C_{1j} C_{2j})^{\prod_{k \in [n]-\{i,j\}} e'_k} mod N = \prod_{j \in [m]-i} C_{3j}^{\prod_{k \in [n]-\{i,j\}} e'_k} mod N$.

4.2 Mitigating the Load of the Directory Service

Based on the current deployment of the directory service, whenever each trainer writes to the directory, it has to send the IPFS hash and the Pedersen commitment of each gradient partition and its addressing information. For example, approximately for each partition, a trainer has to send to the directory service his IPFS id (256 bits), the partition id (32 bits), the hash of that partition (256 bits), and its Pedersen commitment (256 bits). Thus, the trainer has to write 800bits (100bytes) for each partition. Totally for each trainer, the directory service has to receive D = $(64 + 4) \cdot p$ + 32 bytes = 68p + 32 bytes. For example, let p = 500, then D = 34032 bytes $\approx$ 34KB. In contrast, if p = 1000, then $D \approx 64KB$ which is twice as much. Back with p = 500, although 34KB might seem like a small amount of data, imagine that for 10^3 trainers, the directory service has to receive 34MB in each iteration, and for $p = 10^6$, roughly 34GB. In other words, not only the load of the directory service scales linearly to the number of trainers, which comes naturally, but also to the number of partitions.

In addition, what makes the task of the directory service even more intense is that aggregators have to frequently read from it the hashes of the gradients that they have to download. That is, because aggregators periodically poll the directory service, which in each query has to make a lookup to its data structures and reply accordingly. In this chapter, we present some modifications to the original protocol, in which we try to make the role of the directory service as "lightweight" as possible. Firstly, we minimize the data the directory service receives by making the length of the write messages for each trainer constant and independent of the number of partitions. Next, we minimize the load of the directory service in terms of reads by distributing the entirety of the directory to the IPFS nodes, who are responsible for replying to queries in a verifiable manner. In this chapter, we assume that the messages come from the directory service are signed meaning that everyone can verify that a message indeed originates from the directory service. That can be easily achieved by utilizing a digital signature algorithm.

4.2.1 Minimizing the amount of data the directory service receives

As seen from the previous simple example, the amount of data the directory service receives depends on the number of partitions and the number of IPLS trainers. Although there is no way to make the load of the directory service independent of the number of trainers, it is

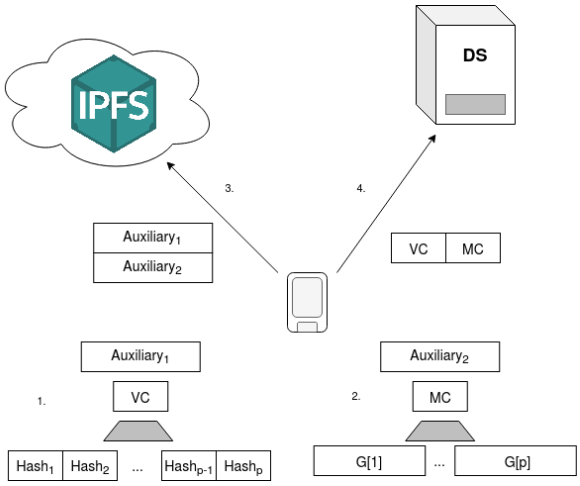

Figure 4.2: Figure depicting what the trainers do before writing to the directory. They compute the vector commitment of the hash of each gradient partition and also the matrix commitment of the gradient partitions (**1.**,**2.**). Then upload the auxiliary information (e.g., witnesses and the actual hashes) of the commitment schemes to the IPFS network (**3.**), and the Vector and Matrix commitment to the directory (**4.**)

possible to make it independent of the number of partitions. In other words, the IPLS protocol can be modified so that instead of receiving data of size $O(p|T|)$, it will receive data of size only $O(|T|)$. Intuitively, this can be achieved by forcing trainers to make write as there was only one partition, which represents all the partitions of the model.

Considering IPFS hashes, the trainers, instead of sending all the p hashes of the model to the directory service they can just send only a vector commitment of those hashes. Let $Hash_i$, be the IPFS hash of the i^{th} partition of the model, then trainers can compute $C \leftarrow VC.Commit(pp, [Hash_1, ..., Hash_p])$, and send only C to the directory service while sending $[Hash_1, ..., Hash_p]$ to the IPFS nodes as it can be seen by 4.2. That can easily be done by broadcasting them using the IPFS pub/sub protocol. The directory service for each tuple of the form $(trainer\_id, \text{“}gradients, i)$ maps it to the C it received and the corresponding Pedersen commitment for the partition i. To retrieve the $Hash_i$, the aggregator responsible for the partition i, will retrieve from the directory service the commitment C. Then it will ask one of the IPFS nodes to send it the $Hash_i$ and a witness that the $Hash_i$ indeed belongs to the position i of the commitment C. To achieve that, IPFS nodes use the $VC.Open()$ algorithm described earlier. Note that this comes in two flavors. Either the trainer sends only the IPFS hashes to the IPFS nodes, and lets them compute the witnesses, or each trainer precomputes all the witnesses and also sends them so that IPFS nodes would not have to compute anything. Last but not least, the aggregator, before downloading the gradient partition, verifies that the $Hash_i$ is a valid element calling $VC.Verify()$ using as C the commitment it read from the

directory service. The protocol's security lies in the fact that the Vector Commitment used is secure, meaning that no IPFS node can open the vector commitment to a different hash and thus will be forced to send the correct one.

What remains is to apply the same concept to the Pedersen commitments. At first glance, it might be tempting to do the exact same thing as we did for the IPFS hashes. If we did that, the directory service would have a commitment $C \leftarrow VC.Commit(pp, [Commit(G[1]), ... , Commit(G[p])]$ for each trainer. However, by having such commitments, the directory would be incapable of computing the product of the Pedersen commitments, which are needed for verification purposes, but only their sum, something that is useless. Simply put, the directory service will not be able to compute at the beginning of the aggregation phase $Comm(U[i])$ $\forall i \in [p]$ and $Comm(PU_j[i]) \forall a_{ij}$. Keep in mind that the security of our protocol relies on computing and distributing those commitments honestly.

To overcome that issue, we use the proposed matrix commitment scheme, where the model is represented as a matrix M whose rows are the partitions of the model. In more detail, each trainer t computes the matrix commitment $C_t, aux \leftarrow M.Commit(pp, P)$. Then it sends the commitment C_t to the directory service and the auxiliary information to the IPFS nodes. Whenever an aggregator a_{ij} queries the directory service, it will also get C_t. Whenever it asks an IPFS node for a hash of a gradient partition i, it will also get a witness that the gradient partition is the i^{th} row of the C_t. To compute the commitments of the updated partitions or the partial updates, the directory service simply computes $Comm(U) \leftarrow \prod_{t \in T} C_t$ and $Comm(P_j[i]) \leftarrow \prod_{t \in T_{ij}} C_t, \forall a_{ij}$. To verify an updated partition U[i], the aggregator sends to the directory service a witness $W_{U[i]}$ that the U[i] is the i^{th} row of the $Comm(U)$. Keep in mind that $W_{U[i]}$ can easily be computed by aggregating the witnesses (look 4.1.1).

For example, imagine that there is only one aggregator responsible for partition i, and three trainers. The trainers will send to the directory service C_1,C_2 and C_3 and the director service will compute $C_t = MC.Comm(U) = \prod_{i \in [3]} C_i \, mod \, N$. The aggregator will receive those commitments from the directory service and will also receive W_{1i}, W_{2i} and W_{3i} from the IPFS nodes, and upon downloading and aggregating the gradient partitions, it also aggregates the witnesses and computes $W_{U[i]} = \prod_{j \in [3]} W_{ji} mod N$. Then the aggregator writes the update to the directory and also sends $W_{U[i]}$. The directory service, before writing to the directory the update for the i^{th} partition, it checks if $MC.Verify(pp, C_t, U[i], W_{U[i]}, i) = 1$. The security of the protocol, derives from the row binding property of the Matrix Commitment scheme.

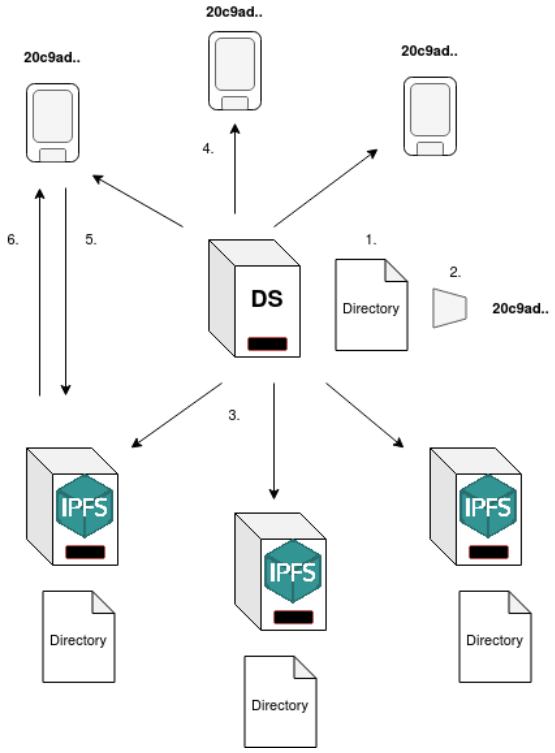

Figure 4.3: Figure depicting how the directory service offloads reads to the IPFS nodes. In a nutshell, the directory computes the current state of the directory (**1.**) and its secure digest (**2.**). Then sends the directory to the IPFS nodes (**3.**) and its digest to all the IPLS nodes (**4.**). To make a query, the IPLS node simply asks an IPFS node (**5.**), receiving an answer with a witness that the answer is correct (**6.**)

Figure 4.2 captures the main steps trainers have to follow to write to the directory service.

4.2.2 "Offloading" Queries to the IPFS Nodes

Except from writes that mainly come from the trainers, the directory service also has to receive and reply to tremendous amounts of queries that mainly originate from aggregators. "Offloading" the responsibility of replying to queries, from the directory service to the IPFS Nodes, can significantly benefit the system's efficiency in total. That is because the workload of answering queries gets distributed among the IPFS nodes, thus enabling faster response times. However, "offloading" such a functionality is not a trivial task. For example, if the directory was simply replicated among the IPFS nodes, then IPFS nodes, based on our security assumptions, could reply to participant's queries with invalid answers. Thus, in some way, IPFS nodes must be "forced" to reply correctly to the queries, or, in other words, give the IPLS participants the capability of verifying the validity of the answers.

The key concept is to let the directory service compute a constant digest of the entirety of

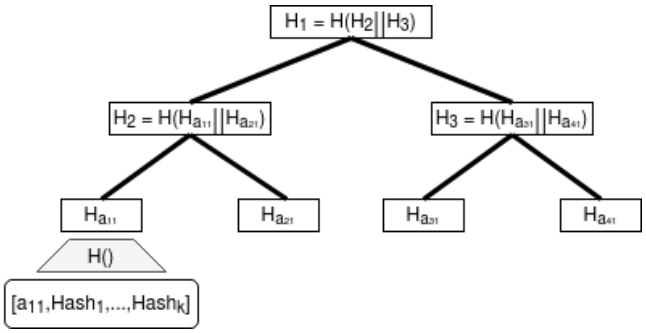

Figure 4.4: Figure depicting how the representation of the directory using a Merkle tree with four aggregators. The digest of the directory is the H_1. Whenever the a_{11} receives the hashes from an IPFS node, the IPFS node has to additionally send $H_{a_{21}}$ and H_3. a_{11}, verifies the validity of the answer by computing $H_2 = H(H(a_{11}||reply)||H_{a_{21}})$ and checking if $H_1 = H(H_2||H_3)$.

the directory and communicate that digest to all IPLS participants. Then participants can use this digest to verify the correctness of the IPFS node's answer. To simplify the explanation, assume that the directory service computes such a digest at the beginning of the aggregation phase. Later, we will show how this can be extended by erasing that assumption. Moreover, because the vast majority of queries come from aggregators, we will focus on these kinds of queries. However, queries from trainers can be extended similarly. Last but not least, we will ignore the modifications been made in 4.2.1, and we will assume that the answers to the queries are the same as in 3.1.5, although the proposed protocol is "compatible" with the modifications of 4.2.1, and those two protocols can be combined easily.

To begin with, for each aggregator a_{ij}, the directory service maintains a queue $Q_{a_{ij}}$ that contains the IPFS hashes added from the last query of the aggregator. At the beginning of the aggregation phase, the directory service computes the hash of the concatenation of the elements of that queue and also the id of the aggregator (e.g., $H(a_{ij}||Hash_1||Hash_2||...||Hash_k)$) using a secure hash function. Then, the directory service creates a constant-sized digest of the directory, defined as C_D. That can be done 1) by using vector commitments, by assigning every position, to only one aggregator, or 2) by simply using a Merkle tree [**?**] as seen from 4.4. Note that vector commitments can be slower than Merkle trees; though, Merkle trees have logarithmic-sized witnesses in contrast to the constant-sized witnesses of the vector commitments. Next, the directory broadcasts to the participants the digest C_D, while distributing the directory to the IPFS nodes. Then the IPFS nodes, upon receiving a query from an aggregator a_{ij}, compute/find the witness $W_{a_{ij}}$, and send the hashes and the witness. To verify the validity of the reply, the aggregator uses the hashes and the witness it just received

and also the C_D, it computes the $v \leftarrow H(a_{ij}||Hash_1||Hash_2||...||Hash_k)$ and checks if $VC.Verify(pp, v, C_D, W_{a_{ij}}) = 1$.

We can erase the constrain of restricting aggregators to query the directory only after the training phase by letting the directory service compute the digest of the current state of the directory periodically throughout the training phase and publish it to the participants. One slight modification is that the directory service does not remove any element from the aggregator's queues. Instead, this responsibility is "transferred" to the IPFS nodes, who maintain their local queue for an aggregator that queries them. If an IPLS participant makes a query while holding an out-of-date digest, then the IPFS node replies with the up-to-date condition of the directory and additionally sends the aggregator the up-to-date digest. As made clear earlier, the digest is signed by the directory service so that participants would be assured that the digest is authentic. Algorithms 3 and 4 contain the core logic of the protocol. As we can see, the directory service does nothing more than simply receiving writes from trainers, periodically updating the C_d, and sending the updates to the IPFS nodes and also the new digest to the IPLS nodes. Note that the directory service does not have to send the updates to every single node but to only a tiny number of them. The efficiency of the distribution of the directory updates among IPFS nodes is guaranteed by the pub/sub protocol. As it can be seen in Algorithm 3, each IPFS node maintains two different sets of queues, where each queue element of the set belongs to an aggregator. The Q set of queues is append only and is mainly used to compute the new digest of the directory, while the TQ is used the exact same way as the directory service used its queue in section 3.1.5. As it seems from the algorithmic description, IPFS nodes have to recompute the digest of the directory, and also compute the witnesses. However this might not be always the case. In fact the directory service can pre-compute all the witnesses so that IPFS nodes could open the digest with $O(1)$ time complexity. In that case however, the directory service has to send much more data. Also computing the witnesses in some cases might not be very computational expensive, as it happens with Merkle trees.

Algorithm 3 Algorithm for the directory service

function directory_service($D = \{\}, A, T$)
  while $T_{curr} < T_{aggr}$ **do**
    $sleep(T_{period})$ ▷ Wait for T_{period} while receiving writes and updating D
    $D_{new} \leftarrow D$
    $Q \leftarrow update\_queues(D_{new} - D_{old})$ ▷ Update the queues by appending the new writes
    $state\_vector \leftarrow [hash(Q_{a_{11}}), ..., hash(Q_{a_{kp}})]$
    $C_d, aux \leftarrow VC.Commit(pp, state\_vector)$ ▷ Compute the digest of the new state
    $pub(IPFS\_Nodes, D_{new} - D_{old})$ ▷ Publish the updates to the IPFS nodes
    $pub(IPLS\_Nodes, C_d)$ ▷ Publish the new digest to the IPLS nodes
    $D_{old} \leftarrow D$
  end while
end function

function ipfs_node
  while $True$ **do**
    $msg \leftarrow recv()$
    if $msg.tag == query$ **then** ▷ In case the message is query
      $a_{ij} \leftarrow msg.sender$ ▷ Get the id of the aggregator
      $W_i \leftarrow VC.open(pp, aux, state\_vector, a_{ij})$ ▷ Compute the witness
      $send(a_{ij}, [C_d, seq, W_i, TQ_{a_{ij}}])$ ▷ Send the new hashes if any with the witness
      $TQ_{a_{ij}} = []$
    end if
    if $msg.tag == ds\_update$ **then** ▷ In case the message comes from the DS
      $seq \leftarrow msg.seq$
      $D = D \cup msg.D'$ ▷ Update the Directory
      $Q \leftarrow update\_queues(D)$ ▷ Update the queues
      $TQ \leftarrow append\_updates(D')$
      $state\_vector \leftarrow [hash(Q_{a_{11}}), ..., hash(Q_{a_{kp}})]$
      $C_d, aux = VC.Commit(pp, state\_vector)$ ▷ Compute the digest
    end if
  end while
end function

Algorithm 4 Algorithms for one FL training iteration

function read($IPFS\_Node, data, seq, a_{ij}, recv\_hashes$)
  $send(IPFS\_Node, [a_{ij})$
  $msg \leftarrow recv(IPFS\_Node)$
  if $msg.seq < seq$ **then** ▷ In case the sequence number returned by the IPFS node is smaller then abort and find a new IPFS node
    $abort$
  else
    $recv\_hashes.append(msg.hashes)$
    $v \leftarrow Hash(recv\_hashes)$
    if $VC.Verify(pp, v, msg.C_t, msg.W_{a_{ij}}) = 1$ **then**
      $returnmsg.hashes$ ▷ If the verify is successful return the hashes
    else
      $abort$ ▷ Else abort and find another IPFS node
    end if
  end if
end function

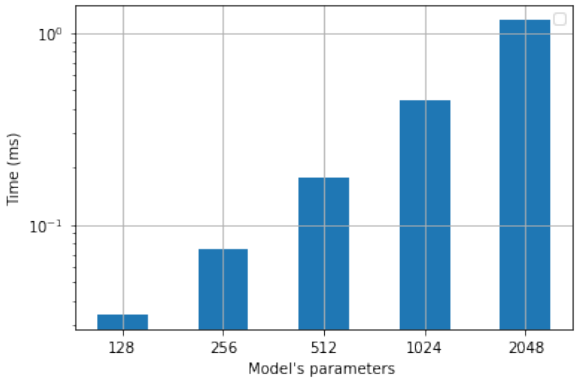

Figure 4.5: Time needed to compute and open all witnesses vs. number of partitions, in logarithmic scale

4.3 Experimental Evaluation

To conduct the experiments and measure the performance of the cryptographic algorithms been used, we used a Dell-Inspiron laptop, with Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8 and 16GB RAM. Specifically, we conducted experiments measuring the performance of the vector commitment scheme based on RSA assumption with different number of partitions but also with variable partition sizes and also the time needed for the directory service or IPFS nodes to compute the constant digest of the directory and precompute all the witnesses. In addition, we measured the performance of the proposed matrix commitment scheme in both single threaded and multi-threaded environment.

4.3.1 RSA Based Vector Commitment Efficiency

In this experiment, we measure the performance of the RSA Based Vector commitment, whenever it is used by the trainers to commit to the IPFS hashes of each gradient partition. Specifically, we measure the time it takes for a trainer to compute the commitment of the vector of IPFS hashes, and also precompute all the witnesses, with variable number of model partitions. As it is obvious, the smaller the number of partitions the faster the compute times. However, even for a relatively large number of partitions (e.g 2048), we can see that the overhead of computing the vector commitment and the witness of each element is very low, roughly one second only as we can confirm by looking at 4.5. Note that this overhead could be easily minimized by computing only the commitment and not the witnesses. In that case however, IPFS nodes would have to compute the witness for every single query, which in some cases could be an overkill.

Number of aggregators	RSA Vector Commitment Witness size	Merkle tree Witness size
2048	128 bytes	352 bytes
4096	128 bytes	384 bytes
8192	128 bytes	416 bytes
16384	128 bytes	448 bytes

Table 4.1: Witness size of RSA based vector commitment scheme and Merkle tree, on different number of aggregators.

4.3.2 RSA Vector Commitments Vs Merkle Trees

In section 4.2.2, we mentioned that the directory service can express the entirety of the directory, by using a vector commitment scheme or simply a Merkle Tree. In this experiment, we measured the time needed to compute the digest of the directory, the exact same way as it has been described in 4.2.2, with variable number of aggregators. Also whenever as in the previous example, the committing the directory using a vector commitment scheme, means that the directory not only computes the commitment but also the the witnesses. From the Figure 4.6, it becomes apparent that Merkle trees, completely outperform the RSA based vector commitment scheme.

Even for large number of aggregators, computing the digest of the directory using merkle trees is far less than one second, in fact 380 ms, comparing to a Vector commitment tree, which is always more than 1 second even with small number of aggregators. It is crucial for the performance of the system, the computation time of the directory to be small, because rate which the directory service publishes the new "version" of the directory, depends on that time. For example, if we had 16384 aggregators, and the directory used a VC scheme, then the directory service can publish a new version of the directory at least every 20 seconds. On the other hand however, the witnesses of the VC scheme are constant regardless the number of aggregator, where in contrast the size of the witness in the Merkle tree is logarithmic but in any case fairly small as table 4.3.2 suggests. Overall, we can conclude that Merkle trees for directory representation should be preferred over RSA based vector commitments mainly for efficiency purposes.

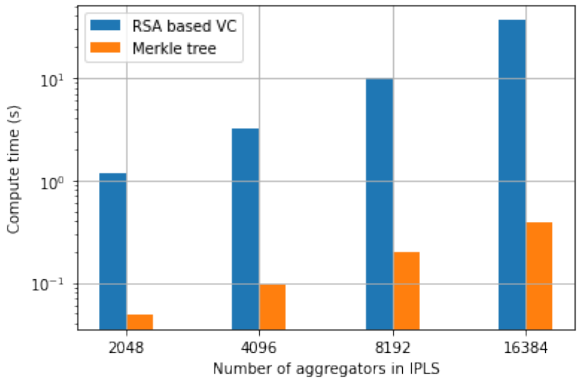

Figure 4.6: Time needed to compute and open all witnesses using a VC scheme and a Merkle Tree vs. number of aggregators, in logarithmic scale

4.3.3 On the efficiency of Matrix Commitments

In this subsection, we measured the performance of our proposed Matrix Commitment construction. In more detail, we explicitly focus on measuring 1) the computation time needed to calculate the matrix commitment digest, 2) the time needed for computing the witness for each row of the matrix. We used a model of approximately one million parameters [3], and we segmented it in different number of partitions (i.e different number of rows). Specifically we segmented the model into (**m**=) 128,256 and 512 partitions (rows), each one of with (**n**=) 8192,4048 and 2024 parameters respectively. In addition, we used multiple threads in order parallelize both the commitment computation but also the witnesses extraction, consequently reducing runtime costs as it will become obvious later.

First things first, it is clear that the time needed to compute the digest of the Matrix with $|M|$ elements, is the same regardless how the matrix is organized (e.g number of rows and columns). What changes is the time needed to compute all the witnesses on different number of rows. As it can be seen in Figures 4.7, with the increasing number of rows, the overhead in computing all the witnesses, becomes a significant bottleneck. This is mainly due to the $O(m^2 logm)$ time complexity, making the scheme seemingly impractical for a large number of rows. Further research should be conducted so that the time complexity will be minimized from $O(m^2 logm)$ to $O(mlogm)$. Quantitatively, the time it takes to compute the matrix commitment for an 8MB model, is roughly 30 seconds. Note that this time is linear to the size of the model, so, for a 16MB model, it would take 60 seconds. To compute all the witnesses, for 128 rows, the time it takes is only 3 seconds, while for 256 and 512 rows is 14 seconds and 67 seconds respectively.

[3] 1,048,576 parameters to be specific, which is a model of size roughly 8MB

To mitigate the severe computational overhead, we parallelize the algorithms introduced in 5, by simply utilizing the pthreads C library [4]. To parallelize the computation of the digest, we simply segmented the model into equal length chunks, the same number as threads in the system. Each thread computes in parallel the commitment of the chunk that it has. When all partial commitments computed from the thread then the main thread simply multiplies them to compute the final digest. To parallelize the witness extraction algorithm, each thread performed the "root-factor" algorithm for a selected row commitment C_i. When a thread completes the computation of the "root-factor" for the C_i, it selects another C_i' until other commitment of a row is left. In our experiments, we computed the digest and the witnesses using 1,2 and 4 threads. As it becomes obvious from the figures 4.7, when we use 2 threads, the computations time is reduced by half, and when we use 4 threads the computations time is almost 1/4 the computations time when using only one thread.

[4]https://man7.org/linux/man-pages/man7/pthreads.7.html

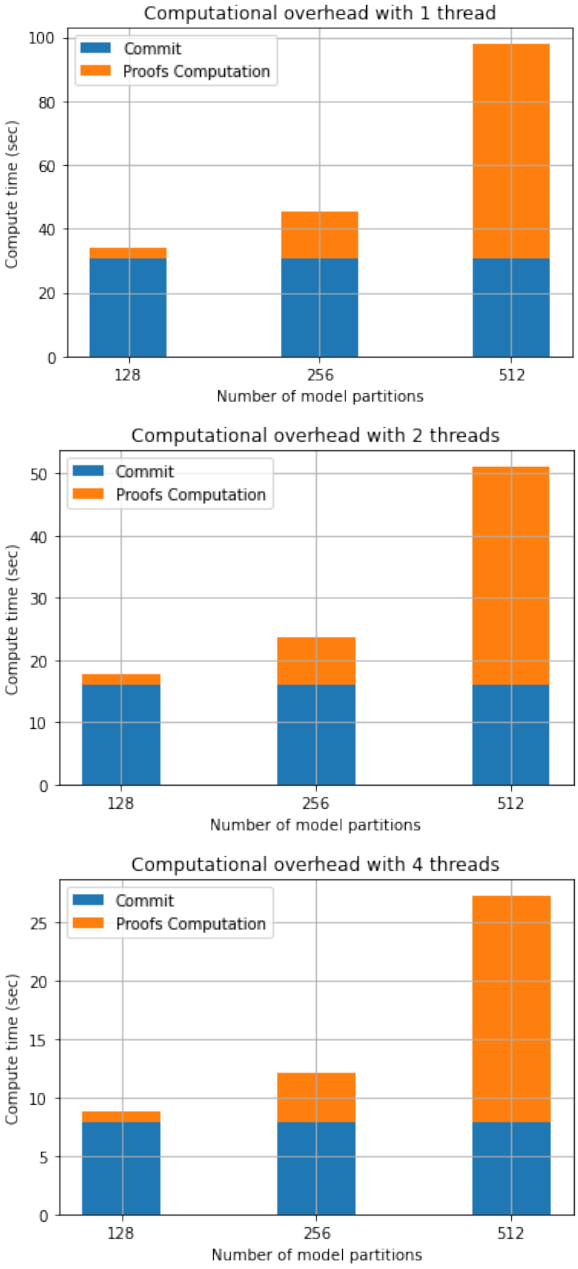

Figure 4.7: Aggregation (top) and uploading (bottom) delays with variable number of providers.

Chapter 5

Conclusion

In this work, we managed to introduce a practical and scalable decentralized and p2p federated learning system. We achieved that by relaxing the communication requirements and letting peers communicate indirectly over a decentralized storage network. In addition, we made IPLS byzantine aggregator robust. In contrast to the blockchain-based solutions, where redundancy in the aggregation is a strict requirement so that the aggregation will be robust, we overcome this problem by using homomorphic commitments. Using such cryptographic tools makes the aggregation step not only faster and scalable but also much more efficient in terms of energy and resource consumption. Although IPLS is much more practical than its counterparts, there is plenty of space for improvement, and many things can be done.

To begin with, it is critical to use much more efficient cryptographic schemes and search for an asymptotically faster matrix commitment scheme. In addition, although the directory service can be deployed as a single trusted entity for some applications, this is not true for other essential applications. As a result, a distributed directory service must be developed, which will have roughly the same efficiency as a centralized directory service. Moreover, data availability is not guaranteed by IPFS on its own, as mentioned earlier, and experimenting with different protocols that make availability of data achievable, is crucial. Last but not least, it is important to explore ways to deal with malicious trainers.

Bibliography

[1] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving Google keyboard query suggestions. *arXiv preprint arXiv:1812.02903*, 2018.

[2] Chen Fang, Yuanbo Guo, Jiali Ma, Haodong Xie, and Yifeng Wang. A privacy-preserving and verifiable federated learning method based on blockchain. *Computer Communications*, 186:1–11, 2022.

[3] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2017.

[4] Harry Chandra Tanuwidjaja, Rakyong Choi, Seunggeun Baek, and Kwangjo Kim. Privacy-preserving deep learning on machine learning as a service—a comprehensive survey. *IEEE Access*, 8:167425–167447, 2020.

[5] Ping Li, Jin Li, Zhengan Huang, Tong Li, Chong-Zhi Gao, Siu-Ming Yiu, and Kai Chen. Multi-key privacy-preserving deep learning in cloud computing. *Future Generation Computer Systems*, 74:76–85, 2017.

[6] Owusu-Agyemang Kwabena, Zhen Qin, Tianming Zhuang, and Zhiguang Qin. Mscryptonet: Multi-scheme privacy-preserving deep learning in cloud computing. *IEEE Access*, 7:29344–29354, 2019.

[7] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[8] Xiaojie Guo, Zheli Liu, Jin Li, Jiqiang Gao, Boyu Hou, Changyu Dong, and Thar Baker. Verifl: Communication-efficient and fast verifiable aggregation for federated learning. *IEEE TIFS*, 16:1736–1751, 2020.

[9] Guowen Xu, Hongwei Li, Sen Liu, Kan Yang, and Xiaodong Lin. Verifynet: Secure and verifiable federated learning. *IEEE Transactions on Information Forensics and Security*, 15:911–926, 2019.

[10] Anmin Fu, Xianglong Zhang, Naixue Xiong, Yansong Gao, Huaqun Wang, and Jing Zhang. Vfl: a verifiable federated learning with privacy-preserving for big data in industrial iot. *IEEE Transactions on Industrial Informatics*, 2020.

[11] Kentaroh Toyoda and Allan N Zhang. Mechanism design for an incentive-aware blockchain-enabled federated learning platform. In *2019 IEEE Big Data*, pages 395–403. IEEE, 2019.

[12] Yuzheng Li, Chuan Chen, Nan Liu, Huawei Huang, Zibin Zheng, and Qiang Yan. A blockchain-based decentralized federated learning framework with committee consensus. *IEEE Network*, 35(1):234–241, 2020.

[13] Davy Preuveneers, Vera Rimmer, Ilias Tsingenopoulos, Jan Spooren, Wouter Joosen, and Elisabeth Ilie-Zudor. Chained anomaly detection models for federated learning: An intrusion detection case study. *Applied Sciences*, 8(12):2663, 2018.

[14] Youyang Qu, Longxiang Gao, Tom H Luan, Yong Xiang, Shui Yu, Bai Li, and Gavin Zheng. Decentralized privacy using blockchain-enabled federated learning in fog computing. *IEEE Internet of Things Journal*, 7(6):5171–5183, 2020.

[15] Yunlong Lu, Xiaohong Huang, Ke Zhang, Sabita Maharjan, and Yan Zhang. Low-latency federated learning and blockchain for edge association in digital twin empowered 6G networks. *IEEE Transactions on Industrial Informatics*, 17(7):5098–5107, 2020.

[16] Christodoulos Pappas, Dimitris Chatzopoulos, Spyros Lalis, and Manolis Vavalis. Ipls: A framework for decentralized federated learning. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–6. IEEE, 2021.

[17] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[18] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[19] Alfonso De la Rocha, David Dias, and Yiannis Psaras. Accelerating content routing with bitswap: A multi-path file transfer protocol in ipfs and filecoin. 2021.

[20] Juan Benet and Nicola Greco. Filecoin: A decentralized storage network. *Protoc. Labs*, pages 1–36, 2018.

[21] Zhilin Wang and Qin Hu. Blockchain-based federated learning: A comprehensive survey. *arXiv preprint arXiv:2110.02182*, 2021.

[22] Chuan Ma, Jun Li, Ming Ding, Long Shi, Taotao Wang, Zhu Han, and H Vincent Poor. When federated learning meets blockchain: A new distributed learning paradigm. *arXiv preprint arXiv:2009.09338*, 2020.

[23] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

[24] Stefan Brands. An efficient off-line electronic cash system based on the representation problem; centrum voor wiskunde en informatica. *Computer Science/Departement of Algorithmics and Architecture, Report CS-R9323*, 1993.

[25] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *Theory of Cryptography Conference*, pages 499–517. Springer, 2010.

[26] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[27] Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 2019.

[28] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[29] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.

[30] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–35. Springer, 2020.

[31] Joao Antunes, David Dias, and Luís Veiga. Pulsarcast: Scalable, reliable pub-sub over p2p nets. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–6. IEEE, 2021.

[32] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks. *arXiv preprint arXiv:2007.02754*, 2020.

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

[34] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. *arXiv preprint arXiv:2012.13995*, 2020.

[35] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. *Advances in Neural Information Processing Systems*, 30, 2017.

[36] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):1–25, 2017.

[37] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in byzantium. In *International Conference on Machine Learning*, pages 3521–3530. PMLR, 2018.

[38] Haibo Yang, Xin Zhang, Minghong Fang, and Jia Liu. Byzantine-resilient stochastic gradient descent for distributed learning: A lipschitz-inspired coordinate-wise median approach. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 5832–5837. IEEE, 2019.

[39] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. *Advances in Neural Information Processing Systems*, 30, 2017.

[40] István Hegedűs, Gábor Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 74–90. Springer, 2019.

[41] Jingyan Jiang, Liang Hu, Chenghao Hu, Jiate Liu, and Zhi Wang. Bacombo—bandwidth-aware decentralized federated learning. *Electronics*, 9(3):440, 2020.

[42] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 125–142. Springer, 2002.

[43] Geoffroy Couteau, Thomas Peters, and David Pointcheval. Removing the strong rsa assumption from arguments over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 321–350. Springer, 2017.

[44] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 256–266. Springer, 1997.

[45] Adi Shamir. On the generation of cryptographically strong pseudorandom sequences. *ACM Transactions on Computer Systems (TOCS)*, 1(1):38–44, 1983.

APPENDICES

Appendix A

Cryptographic proofs

In this appendix chapter, we give the formal proofs of vector binding and position binding of the Pedersen commitments and vector commitments respectively.

A.1 Cryptographic Assumptions

A.1.1 RSA Assumption

Let $N \leftarrow Gen(1^k)$, be a probabilistic polynomial time algorithm that returns a number N, which is a product of at least two primes of size at least k each. The RSA assumption, states that given a number e relatively prime to $\phi(N)$ and a random $y \in Z_N^*$, it is "hard" to find x s.t $x^e = y \, mod N$. Formally, it is believed that for any probabilistic polynomial time adversary **A**, that :

$$Pr\begin{pmatrix} N \leftarrow Gen(1^k); \\ e, y \leftarrow Z_N^*; \\ x \leftarrow A(N, e, y) \end{pmatrix} \leq negl(k) \tag{A.1}$$

A.1.2 Discrete Logarithm Assumption

Let $(G, g, p) \leftarrow Gen(1^k)$, be a probabilistic polynomial time algorithm that returns description of the group G, a generator g of the group G, and its order p which is a prime number. The discrete logarithm assumption, states that for any probabilistic polynomial time adversary **A**, given (G, g, p) and a group element y, it find a number $a \in Z_p$ s.t $y = g^a$ only with negligible probability in k. Formally:

$$Pr\left(\begin{array}{c}(G,g,p) \leftarrow Gen(1^k);\\ g^a \leftarrow G;\\ a \leftarrow A(G,g,g^a,p)\end{array}\right) \leq negl(k) \tag{A.2}$$

A.2 Vector Binding Proof

As equation 1 states, it should be "infeasible" for **all** probabilistic polynomial time (e.g., PPT) adversaries A, to find two different vectors, that their commit results to the same Pedersen commitments. To prove this statement, we can create a new adversary B. That adversary receives from an challenger the parameters of a discrete log problem. Those parameters are the prime order group G, a generator of the group g, and also an element of the group h. Note than in reality it is considered hard to find x, s.t $g^x = h$. In other words it holds that

$$Pr[DLog] = Pr[x \leftarrow B(G,g,h); g^x = h] \leq negl(k).$$

For contradiction, imagine that an PPT adversary A exists, who wins the vector binding challenge with non-negligible probability. This means that

$$Pr[W] = Pr[v, v' \leftarrow A(pp); Commit(pp,v) = Commit(pp,v') \wedge v \neq v'] > 1/p(k),$$

where $p(\cdot)$ is a polynomial function. We will show how to construct an PPT adversary B, who "uses" the adversary A, so that B can win the discrete log challenge with non-negligible probability. This is done by the following simple steps:

1. B receives from the challenger the (G,g,h).

2. B creates the public parameters of the Pedersen Commitment scheme. It selects the number of elements n, choose randomly $i \in [n]$, and then choose randomly n-1 elements $r_1, ..., r_{i-1}, r_{i+1}, ..., r_n$. Then set $pp = \{g^{r_1}, ..., g^{r_{i-1}}, h, g^{r_{i+1}}, ..., g^{r_n}\}$. It is important to mention that pp, must have the same distribution as if it would have been computed by simply calling $Gen(\cdot)$. Observe that this is true because h is randomly uniform.

3. B uses A by giving him the public parameters that it computed. Upon calling $A(pp)$, it takes two vectors v and v'. First it checks if Commit(pp,v) = Commit(pp,v'). If the

equation does not hold then abort. Otherwise check if $v_i \neq v'_i$.

4. The last equation holds with probability at least $1/n$. Then B computes $h^{v_i - v'_i} = g^{\sum_{k\in[n]-i} r_k(v'_k - v_k)}$. By doing that it can find x = $(v_i - v'_i)^{-1} \sum_{k\in[n]-i} r_k(v'_k - v_k)$, which is returned to the challenger.

Note, that the pp, can be computed in polynomial time, A(pp) runs in polynomial time and also $(v_i - v'_i)^{-1}$ can be efficiently computed in polynomial time. As a result B runs in polynomial time. It is straightforward that the probability that B succeeds is $Pr[DLog] \geq Pr[W]/n$. However we now that $Pr[DLog]$ is negligible and as we assumed earlier $Pr[W]/n$ is non-negligible, thus reaching a contradiction. As a result for any PPT adversary, $Pr[W] \leq negl(k)$.

A.3 Position Binding Proof

As equation 2 states, it should be "infeasible" for all PPT adversaries A, to compute two different values that open to the same position of the same vector commitment. For this proof we consider an PPT adversary B, that wants to solve the RSA challenge with non-negligible probability. Specifically, B receives from an RSA challenger the challenge which is the tuple (N, z, e). Note that B has to find y, s.t $y^e = z$, with non-negligible probability.

For contradiction, imagine that a PPT adversary A exists that wins the position binding challenge with non-negligible probability. This means that $Pr[W] = Pr[v_i, v'_i, W_i, W'_i, C \leftarrow A(pp); VC.Verify(pp, v_i, W_i, C, i) = VC.Verify(pp, v'_i, W'_i, C, i) = 1 \wedge v_i \neq v'_i] > 1/p(k)$, where $p(\cdot)$ is a polynomial. We will show how B can use A so that to win the RSA challenge thus reaching in contradiction. Specifically, B executes the following algorithm:

1. B receives from the challenger the (N, z, e).

2. B selects n and computes $pp \leftarrow G(1^k, n)$. Then it selects $k \in [n]$ and sets $e_k = e$. Moreover it sets $g = z$. As a result $S_i = z^{\prod_{k\in[n]-i} e_k}$ $\forall i \in [n]$.

3. B runs the adversary A, by giving him the public parameters. It then takes $v_i, v'_i, W_i, W'_i, i, C$ and checks if $k = i, v_i \neq v'_i$ and $VC.Verify(pp, v_i, W_i, C, i) = VC.Verify(pp, v'_i, W'_i, C, i) = 1$. If this is not true then B aborts.

4. Otherwise, it knows that $S_i^{v_i} W_i^{e_i} = S_i^{v'_i} W_i^{e_i}$, which can be written as $(W'_i/W_i)^{e_i} = S_i^{v_i - v'_i} = z^{(v_i - v'_i) \prod_{k \in [n]-i} e_k}$. Note that gcd($(v_i - v'_i) \prod_{k \in [n]-i} e_k$,$e_i$) = 1. Thus applying schamir's trick B gets $z = z^{\alpha e_i} (W'_i/W_i)^{\beta e_i} = (z^{\alpha} (W'_i/W_i)^{\beta})^{e_i}$. Note that if $W_i = W'_i$, then we can factor with non-negligible probability.

5. B returns to the RSA challenger y = $z^{\alpha} (W'_i/W_i)^{\beta}$.

It is not hard to see that B runs on polynomial time. The probability that B succeeds is $Pr[W]/n$ which is non-negligible, which contradicts the RSA assumption. As a result for every PPT adversary A, $Pr[W] \leq negl(k)$.

Appendix B

Running IPLS

To run multiple IPLS nodes and experiment with the IPLS middleware on your computer, you must install the IPFS. Instructions on downloading IPFS are given in `https://docs.ipfs.io/install/ipfs-desktop/#ubuntu`. Next, you must set up as many IPFS daemons as many IPLS participants and IPFS storage nodes you want to have in our system. For example, if you want to run an experiment with 10 IPLS participants (including the IPLS bootstrapper/Directory service) and 5 IPFS storage nodes, then you need to set up 15 IPFS daemons. You can set up several IPFS nodes locally by following the guidelines of `https://stackoverflow.com/questions/40180171/how-to-run-several-ipfs-nodes-on-a-single-machine`. Then, you must set up a private IPFS network and add all nodes just created in that private network. Follow the guidelines of`https://medium.com/@s_van_laar/deploy-a-private-ipfs-network-on-ubuntu-in-5-steps-5aad95f7261b`. When IPFS nodes setup is finished, then start running them by typing "ipfs daemon –enable-pubsub-experiment" in your terminal. To download IPLS, you must visit the `https://github.com/ChristodoulosPappas/IPLS-Java-API` and import the project into your IDE. Note that the easiest way to run the IPLS middleware is by using your IDE, but you can also convert the middleware into a .jar program. Moreover, you should download the IPLS python API from `https://github.com/ChristodoulosPappas/IPLS-python-API`.

To conduct various experiments with IPLS, first, you have to start the IPLS middleware. That is done by running the Middleware.java, which takes seven parameters. Those parameters are:

- The port number (-p), the IPLS middleware listens and receives tasks from the appli-

cation.

- The number of partitions (-pa) the model will get segmented. Note that all IPLS participants must segment the model in the same number of partitions.
- The minimum number of partitions (-mp) a peer should be responsible for. If a peer is not responsible for any partition (e.g, trainers) then -mp = 0.
- The number of IPLS participants (-n) must enter the project before it begins. For example, if -n=3, the Directory service (or IPLS bootstrapper) waits until 3 IPLS participants enter the project.
- Indicator of indirect communication (-i). If -i=0, then all IPLS participants must communicate directly. Else if -i=1, all IPLS participants must communicate indirectly using the decentralized storage network.
- The training time (-training). This is the time IPLS trainers have in order to finish their training. For example, if -training=10, then IPLS trainers have 10 seconds to train the model and upload their gradients.
- Indicator if merge and download is used (-aggr). If -aggr=0, then aggregators do not use merge and download. Until now, merge and download is an experimental feature; thus, it is highly recommended not to be used.

For example, "-p 12000 -pa 3 -mp 0 -n 3 -i 1 -training 10 -aggr 0" is valid example of parameters assignment. The example indicates that an IPLS middleware listens to the port 12000, segments the model in 3 partitions, is responsible for no partition, communicates indirectly as any other IPLS participant, has to train its model in 10 seconds, and merge and download is not used. To start an IPFS node, someone simply needs to start the Decentralized_Storage.java class, with program parameter the IPFS address API of the IPFS daemon that is going to become the IPFS storage node (e.g, "/ip4/127.0.0.1/tcp/5006" or "/ip4/127.0.0.1/tcp/5001").

Using the IPLS python API, someone can create their own IPLS application. The API consists simply of the following two methods:

- **init(api_ipfs_address,model_file,bootstrappers,model_size,model,is_bootstrapper).** This function is used to initialize the IPLS middleware and ask it to join the IPLS project. This method is given as input the IPFS address API of the IPFS daemon, which

the IPLS middleware will communicate, the model_file where the initial model parameters are stored, and the list of the IPLS bootstrappers (commonly, there is only one bootstrapper). In addition, it also takes as input the size of the model, the model (commonly the compiled Keras or TensorFlow model), and a flag whether the IPLS middleware will act as a bootstrapper or not.

- **fit(model,X,Y,batch_size,iter).** This method takes as input the Keras model, the local data of the node X, the corresponding labels Y, the batch size, and the number of iterations the IPLS participant will run. This method should be seen as the fit() method Keras or TensorFlow have. The difference is that inside IPLS fit, the actual IPLS API is used to train the model in a Federated learning fashion.

Note that the API is a class, so a constructor is needed. The object's constructor takes only one input: the port the IPLS middleware is listening to. An example of how to write an IPLS application is given in ipls_example.py.

To run IPLS locally, first of all, someone has to start all the IPFS daemons, then start the IPLS middlewares and the IPFS storage nodes. Each middleware and IPFS storage node must be assigned to only one IPFS daemon. This assignment is done by using the IPFS API addresses. Then first run the IPLS application for the bootstrapper and afterward run all the other IPLS applications. For example, if someone wants to run the ipls_example.py, with 3 IPLS participants, one bootstrapper, and 2 IPFS storage nodes, he should start 6 IPFS daemons. Then start 4 IPLS middlewares with the following parameters:

- -p 12000 -pa 3 -mp 0 -n 3 -i 1 -training 10 -aggr 0
- -p 12001 -pa 3 -mp 1 -n 3 -i 1 -training 10 -aggr 0
- -p 12002 -pa 3 -mp 1 -n 3 -i 1 -training 10 -aggr 0
- -p 12003 -pa 3 -mp 1 -n 3 -i 1 -training 10 -aggr 0

Then start 2 IPFS storage nodes (e.g., start 2 different Decentralized_Storage processes with parameters ip4/127.0.0.1/tcp/5005 and ip4/127.0.0.1/tcp/5006). Finally run the IPLS application by executing:

1. python3 ipls_example.py 0 4 /ip4/127.0.0.1/tcp/5001 1 My_IPFS_ID
2. python3 ipls_example.py 1 4 /ip4/127.0.0.1/tcp/5002 0 Bootstrapper_ID

3. python3 ipls_example.py 2 4 /ip4/127.0.0.1/tcp/5003 0 Bootstrapper_ID

4. python3 ipls_example.py 3 4 /ip4/127.0.0.1/tcp/5004 0 Bootstrapper_ID

Where Bootstrapper_ID is the IPFS id of the bootstrapper (e.g, 12D3KooWCyJZJphf9 z1Dbd2sJ KYc11PVV2RBVA9HQjNz26oMANgR).