Towards Efficient Decentralized Federated Learning

Christodoulos Pappas*, Dimitrios Papadopoulos†, Dimitris Chatzopoulos‡, Eleni Panagou*
Spyros Lalis*, Manolis Vavalis*
chrpappas@uth.gr, dipapado@cse.ust.hk, dimitris.chatzopoulos@ucd.ie, epanagou@e-ce.uth.gr
lalis@uth.gr, mav@uth.gr
*University of Thessaly, †The Hong Kong University of Science and Technology, ‡University College Dublin

***Abstract*—We focus on the problem of efficiently deploying a federated learning training task in a decentralized setting with multiple aggregators. To that end, we introduce a number of improvements and modifications to the recently proposed IPLS protocol. In particular, we relax its assumption for direct communication across participants, using instead *indirect communication* over a decentralized storage system, effectively turning it into a partially asynchronous protocol. Moreover, we secure it against *malicious aggregators* (that drop or alter data) by relying on homomorphic cryptographic commitments for efficient verification of aggregation. We implement the modified IPLS protocol and report on its performance and potential bottlenecks. Finally, we identify important next steps for this line of research.**

***Index Terms*—Federated Learning, Decentralized Storage, InterPlanetary File System, Verifiable Aggregation, Homomorphic Commitments**

I. Introduction

With the emergence of data privacy laws in Europe and the USA, parties that wish to perform computations and analyses on top of users' sensitive data need to ensure they do not compromise their privacy. Federated Learning (FL) is a relatively new paradigm of distributed machine learning, where data stay "on-the-edge" and never leave the user's device. Instead, users download a global model, train it on their data, and send the locally computed gradients back to an aggregation server. The latter, collects the gradients from the users, aggregates them, and updates the global model. The process is repeated until model convergence is reached.

Although FL has significant success and has been adopted in several applications by various actors (e.g., see [1, 2]), it remains a centralized process where the coordinator of the learning process is the aggregator server. Its responsibilities can range from downloading and aggregating updates to selecting which devices will train the model in the next iteration, etc. However, centralization can lead to serious problems affecting both the performance of the machine learning model and the privacy of the users. To be more specific, a malicious server has the power to alter the global model into a less accurate one in order to perform simpler but faster computations or deny downloading updates from some clients to save bandwidth and power. Prior attempts to address such issues in FL [3, 4], cannot prevent the server from creating "forks" providing different views of the training process to different trainers. For the reasons described above, enterprises or individuals who want to set up their federated learning process may hesitate to trust a single server to be responsible for the learning process.

Decentralized Federated Learning. An alternative approach for federated learning is based on decentralization. *Decentralized federated learning* can be classified into two main categories: (i) purely decentralized schemes where peers communicate directly with others and perform the learning process via gossiping (e.g., [5, 6, 7]), and (ii) blockchain-based approaches where, at a high level, a blockchain takes up the role of the aggregator in centralized federated learning (e.g., [8, 9, 10, 11, 12, 13, 14, 15, 16]). Purely decentralized FL seems tempting, as it does not need any "centralized" authority. However, it may not always achieve the same performance in model accuracy and convergence as centralized FL, and this highly depends on the nature of the dataset and loss function. The blockchain-based approach offers the same convergence guarantees as centralized FL, but does not achieve its scalability, storage requirements, and efficiency. That is because miners have to store all updates into the blockchain, and those who serve as aggregators have to download and aggregate every single update.

Pappas et al., recently proposed IPLS [17] a decentralized FL protocol that is more efficient and lightweight than blockchain-based ones while avoiding the issues of other purely decentralized schemes discussed above. However, IPLS requires the establishment of direct communication links between peers, which, as we elaborate on Section II, may not always be realistic in a cross-device FL setting.

This work. In this paper, we consider the scenario of an individual or a small enterprise that intents to launch an FL task (e.g., using its customers' data for training). Due to scarcity of resources, such a party may not be able to play the role of the aggregator but, due to security and trust issues, does not wish to outsource the aggregation task to a third-party cloud service. A reasonable solution is to launch the task using a decentralized FL service and let the task launcher (owner) devote its limited resources to strictly necessary, lightweight operations. To this end, we propose a solution based on the IPLS protocol, modified in two crucial ways.

First, our protocol eliminates the need for direct peer-to-peer communication among FL participants, relying instead on *indirect* communication via a decentralized storage network. We base our solution on IPFS [18], a widely used p2p distributed file system that offers fast and reliable content routing and retrieval. In this way, communication between

mobile and resource-constrained devices is possible while ensuring the liveness and availability of the gradients uploaded by the trainers and the updates computed by the aggregators. We also exploit the decentralized storage network to make the aggregation of gradients much faster and bandwidth-efficient, via a merge-and-download optimization (Section III).

Second, in the scenario explained above, it is plausible that the task launcher would rather not have to "blindly" trust the task participants. To that end, our protocol achieves *verifiable aggregation*, i.e., it ensures that *malicious aggregators* cannot cheat by dropping or altering some of the trainers' updates. We achieve this by using cryptographic vector commitments with homomorphic properties that allow fast verification of the fact that all gradients have been taken into account for the next iteration of the global model (Section IV).

Overall, our contributions can be summarized as follows:

1) We propose a decentralized FL system where communication between participants takes place over a decentralized storage network.
2) We extend our system to be secure against malicious aggregators, ensuring the new model correctly encompasses all the gradients uploaded by trainers in each FL iteration.
3) We introduce a merge-and-download mechanism, exploiting the features of the underlying decentralized storage, to further reduce aggregation time by letting storage nodes pre-aggregate gradients.
4) We implement a prototype of the proposed system, report on its performance, and identify possible bottlenecks. Our code is publicly available.[1]
5) We discuss next steps and future directions towards efficient and verifiable decentralized FL.

Related works in Decentralized FL. Wang et al. [19] present an extensive survey on blockchain-based FL (BCFL), classifying works based on their architecture into two categories: fully coupled BCFL [12, 13, 14], where trainers are also nodes in the blockchain and participate actively in the block generation and model aggregation, and flexibly coupled BCFL [8, 9, 10, 11] where trainers just upload their updates to the blockchain, while miners are responsible for aggregating the trainers' updates and producing the global model. In the majority of these papers, the federated learning process follows the same pattern. First, trainers upload their gradients to the blockchain, and then the blockchain takes the role of the server in the spirit of the centralized FL approach and is responsible for aggregating these gradients. For the first step, usually trainers or their selected blockchain nodes broadcast the gradients to all other blockchain nodes blowing up communication. Some notable exceptions can be found in [11, 13], where gradients are stored in the IPFS storage (similar to what we do as part of our non-blockchain approach). Finally, there is a plethora of decentralized federated learning protocols over D2D networks (e.g., see [20, 21, 22]). Since these operate in different network topologies, they are not directly applicable to our target scenario.

[1]https://github.com/ChristodoulosPappas/IPLS-Java-API

II. Background on IPLS

The InterPlanetary Learning System (IPLS) [17] is a decentralized federated learning (FL) framework. In particular, participants cooperate with each other to train a machine learning model in a federated manner, without using a centralized server. The main idea behind IPLS is to segment the parameters vector of the machine learning model into smaller partitions, which are then separately aggregated by different participants that are made responsible for these partitions, based on the received gradients. More specifically, IPLS participants can assume the following roles:

1) Bootstrappers. In IPLS, a bootstrapper is the initiator of a federated learning task. Whenever a peer 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 peer registration for the tasks they have launched.

2) Aggregators are participants who are in charge of maintaining specific partitions of the model parameters vector. An aggregator receives from the trainers only the gradients for the partitions it is responsible for, and aggregates them (using summation). It then communicates the aggregated gradients to all the trainers. Note that, for efficiency and robustness purposes, it may be useful to assign multiple aggregators to the same partition. In this case, each such aggregator computes a "partial" aggregation for the partition based on gradients they receive from a subset of the trainers. Then, in a synchronization phase, these aggregators communicate with each other to further aggregate the partially aggregated gradients into the the global updated parameters vector for the partition, which is then communicated back to the trainers.

3) Trainers are the main protocol participants who are responsible for iteratively training the model on their respective data. In each iteration, each trainer produces a gradients vector, it splits it into the specified partitions, and sends each gradient partition to one of the designated aggregators. It then receives updates for all model partitions, forms the updated model (by concatenation of the updated partitions), and the process repeats until convergence.

Formally, let A_i be the set of the aggregators responsible for the i-th gradient partition, A_{ij} be the j-th aggregator in A_i, and T_{ij} the set of trainers that send their i-th partition to A_{ij}. Note that $\forall i,\ T = \bigcup_{j \in |A_i|} T_{ij}$ where T is the set of all trainers, since every trainer must send its i-th partition to an aggregator. Likewise, $\forall i,\ \varnothing = \bigcap_{j \in |A_i|} T_{ij}$, as each trainer sends its gradient partition i to only one aggregator. In practice, these sets can be dynamically maintained by the bootstrapper or through suitable communication among the participants.

III. Building Federated Learning on top of Decentralized Storage

A. Adversarial Model / Security Assumptions

The IPLS framework operates under the assumption that all three types of involved parties follow the protocol and have no incentive to misbehave. This is a reasonable assumption

for the bootstrapper, who (as the task owner) naturally wants the learning process to be successful, achieving as-good-as-possible model performance. However, this is not necessarily the case for the aggregators, who may have incentives (monetary or otherwise) to hinder the training process or affect the quality of the model [3, 4]. For instance, consider a lazy aggregator who wants to reduce costs by performing less accurate computations, or a competitor of the FL task that purposefully alters its updates.

When designing our protocol we consider *malicious* aggregators that can either *drop or alter* the gradients received by trainers, e.g., to poison the updated model. Specifically, even in the presence of such malicious behavior, we want to guarantee that in each round the updated model is *complete*, i.e., no gradient sent by a trainer has been omitted, and *correct*, i.e., all included gradients contain the values sent by the trainers. Although malicious activity by the trainers is also possible (e.g., poisoning their gradients resulting in degradation of the model's performance, or otherwise diverging from the protocol to slow down the learning process), we do not focus on this issue and it is left for future work. Finally, we assume an underlying distributed storage protocol (e.g., IPFS) guarantees data availability (e.g., via IPFS cluster or incentivized storage [23]), however, we do not assume correctness of retrieved data; this is up to the parties to check.

B. Indirect Communication between Participants

Similar to many other decentralized FL approaches [5, 6, 7], IPLS assumes reliable direct communication among all participants. However, this assumption might be unrealistic in practice, especially for mobile and edge-based participants, such as smartphones and IoT devices. Firstly, various technical limitations, such as firewalls and volatile mobile IP addresses, may make it impossible for participants to establish direct communication links with each other over the Internet. Secondly, even if direct communication is possible, participants may not be online at the same time due to intermittent connectivity or energy limitations, requiring several attempts to connect successfully. Furthermore, if aggregators face dropouts, trainers have to re-send their gradients to alternative ones.

To address this issue, we modify IPLS so that a decentralized storage network is used for reliable *indirect* communication between IPLS participants. In our implementation, we adopt IPFS and we introduce a clean separation between IPLS participants and IPFS nodes.[2] That is, IPLS participants are the trainers and aggregators that actively contribute to the FL process, as described above. On the other hand, IPFS nodes provide the distributed and highly available storage system network, which is used to support the indirect communication between IPLS participants. In practice, IPLS participants communicate indirectly by uploading and downloading their data (gradients, partial updates or updates) to and from the IPFS storage network. Below, we refer to specific parts of its operations, as necessary, and we refer interested readers to [18] for additional details about it.

[2]The implementation of [17] also used the IPFS interface, but only as a means of reliable communication (multicast and broadcast by explicitly using pub/sub) and not for storage.

C. Directory Service

In order to locate stored data (and verify their integrity), IPFS relies on a secure hash function (by default, SHA-256), by computing a *hash address* $Cid = Hash(data)$. Parties that wish to retrieve $data$ must perform a lookup for Cid; without knowing this hash, one cannot find $data$.

To solve this issue, we introduce a *directory service* that operates as follows. Every piece of information uploaded to the decentralized storage network is associated with some "addressing" meta-information. For example the i-th gradient partition from the trainer T_j, can be "addressed" by the tuple $addr = (uploader\_id, partition\_id, iter, type)$, where $iter$ is the number of training round and $type$ is either "gradient", "partial update" or "global update". The directory service will thus maintain a map from this addressing information to the Cid of the corresponding $data$ in IPFS. This map is updated when receiving hashes of gradients or updated partitions from the trainers and the aggregators, respectively.

One question that arises is how this directory will be instantiated (both for efficiency and security). Since the directory service receives orders of magnitude fewer data per iteration than the aggregators combined do, we believe it is reasonable to assume this will be run by the (trusted) bootstrapper of the FL task. If this is not feasible, an alternative approach is to use a distributed, blockchain-based directory service [24].

D. Implementation Details

Algorithm 1 shows the details of our proposed FL scheme. We consider several aggregators responsible for a partition, and assume that a set of trainers is allocated to each aggregator. We explain the operations for the case of honest participants for simplicity. The necessary changes to achieve security against malicious aggregators are described in Section IV.

Participants (both trainers and aggregators) store data to the decentralized storage via the UPLOAD function (lines 1-5). This sends $data$ to an IPFS node via a put method and receives an acknowledgment. It then sends the hash Cid of the data and the corresponding addressing information to the directory service. To retrieve gradient partitions or partial updates, aggregators poll the directory service to receive the hashes of the data they have to download (lines 28-34 and 37-42). Then, they download the actual data via the IPFS get method. Likewise, trainers learn the hashes of the updated partitions (16-22) and retrieve them. Note that the trainers, before sending their gradient partitions, append the value 1 to each partition to be used for averaging the received updated partition (14, 20-21).

In each iteration (training round), participants receive a schedule that contains the iteration (number) of the learning process and two UTC timestamps, the t_{train} and t_{synch}. The first timestamp is a time threshold indicating when the trainers have to upload their gradients. The latter one sets a maximum threshold on when the iteration must finish. Knowledge about

the end of synchronization time is needed to prevent the stall of the learning process if all the aggregators responsible for the same partition are unavailable. However, with a great number of A_i, this is highly unlikely. Finally, whenever an aggregator from A_i does not respond, another aggregator downloads his gradients on his behalf (not shown in Algorithm 1 for brevity).

Algorithm 1 Algorithms for one FL training iteration

function UPLOAD($addr, data$)
  $cid \leftarrow hash(data)$
  $put(ipfs\_peer, data)$
  $send(directory, [addr, cid])$
end function

function TRAINER($M, A_t, t_{train}, t_{sync}$)
  $gradU \leftarrow$ 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 $\forall$ partition
    $upload((id, i, iter, "gradient"), [gradU[i], 1])$
  end for
  for each $i \in M.parts$ **do** ▷ get updated partitions
    while cid == NILL **do** ▷ check the DS until you get the Cids
      $cid \leftarrow check\_directory(A_t[i], i)$
    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($A^i, T_a, t_{aggr}, t_{sync}$)
  while $T_{ij} \neq \varnothing$ **do** ▷ get gradient updates from my trainers
    $cids \leftarrow check\_directory(a, i)$ ▷ 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
  $upload((id, i, iter, "partial\_update"), modelU^i[a])$
  while $t_{curr} < t_{sync} \wedge A_i \neq \varnothing$ **do** ▷ sync with $A_i - A_{ij}$
    $cids \leftarrow check\_directory(a, i)$ ▷ Check if new Cids commited
    for each $(a', cid_t) \in cids$ **do**
      $modelU^i[a'] \leftarrow download(cid_t)$
    end for
  end while
  $modelGlobU^i \leftarrow \sum modelU^i[a']$ ▷ globally updated partition
  $upload((i, iter, "update"), modelGlobU^i)$
end function

E. Merge and Download

As described above, each aggregator A_{ij} has to download data of size $D = (|T_{ij}| + |A_i| - 1) \cdot Partition\_Size$, hence communication scales linearly with the number of trainers from whom A_{ij} has to download gradients partitions and the number of the aggregators responsible for the same partition. To reduce communication costs on the aggregators' side, we take advantage of the fact that some gradient partitions that correspond to the same aggregator might be stored to the same IPFS node. To exploit this, instead of explicitly downloading each gradient from that IPFS node, the aggregator sends a set of hashes and requests to "pre-aggregate" the gradient partitions for those hashes and send only the aggregated result. We call this mechanism *merge-and-download*. To make the best out of this mechanism, we can assign to each aggregator a set of IPFS nodes P_{ij}, also called *providers* of this aggregator (an IPFS node can be provider for multiple aggregators). Then we require that for the i-th partition, a trainer $T \in T_{ij}$ is required to to upload its gradients to a node from P_{ij}, while the remaining of the protocol proceeds as before but with aggregators issuing merge-and-download requests.

Clearly, there is a trade-off between the number of an aggregator's designated providers, and the aggregation completion time and its communication complexity. In the extreme case where $|P_{ij}| = 1$, aggregator A_{ij} needs only one aggregated partition (which is actually its partial update) but the IPFS node might get congested, slowing down aggregation. On the other extreme, if $|P_{ij}| = |T_{ij}|$, although the IPFS nodes will not get congested, communication becomes expensive and this again affects the aggregation time. The number of IPFS providers P_{ij} that appears to achieve the best aggregation time is approximately $\sqrt{|T_{ij}|}$, as we show next. Assuming 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.

IV. Security Against Malicious Aggregators

In this section we describe the necessary modifications in the above described algorithms in order to achieve verifiable aggregation against malicious aggregators, as described in our adversarial model in Section III-A. This is achieved by using *homomorphic commitments* to succinctly represent gradients in a secure way. We first describe the cryptographic scheme we use, and then explain how it is used in our protocol. We note that this approach has previously been used in FL for verifiable aggregation [3], albeit in the centralized setting.

A. Pedersen Vector Commitments

Cryptographic vector commitments allow a party that holds a vector $\mathbf{v}$ of values to produce a commitment C that has constant size, independent of $\mathbf{v}$. Subsequently, the vector owner can "open" C to show that the pre-image was $\mathbf{v}$. Crucially, it should be impossible to produce two different vectors $\mathbf{v}$, $\mathbf{v}'$ as valid openings for the same C, a property known as *vector binding*. We focus on commitments with *homomorphic properties*: Given only $C1, C2$ (for vector pre-images $\mathbf{v}_1$, $\mathbf{v}_2$), one can efficiently compute commitment C with pre-image $\mathbf{v}_1 + \mathbf{v}_2$. We consider the vector version of the classic Pedersen commitment [25] (e.g., see [26]) that is vector-binding under the discrete logarithm assumption.

Without going into all the technical details, a Pedersen vector commitment C for $\mathbf{v}$, is computed as $C = \prod_{i=0}^{n-1} h_i^{v_i}$ where $\{h_i\}_{i=0}^{n-1} \in G^n$ is a public parameters vector from a cyclic prime-order group G. The commitment C itself is a single group element. Given the vector and the commitment,

one can verify it is a valid pre-image by re-running this computation. It is also easy to see that the above commitments are homomorphic: if C_1, C_2 are commitments for vectors $\mathbf{v_1}$, $\mathbf{v_2}$, then $C = C_1 \cdot C_2$ is a valid commitment for $\mathbf{v_1} + \mathbf{v_2}$, where $\cdot$ is group multiplication in G.

B. Modifications in our Protocol

Utilizing the above commitment, we first require trainers to include the commitment C for each gradient vector $data$ to the addressing information sent to the directory service. Then, the directory service maps each partition of the model to the *total accumulated* commitment for it, For example, consider the i-th partition, for which the N trainers have sent commitments $C_{i1}, \ldots, C_{iN}$. The directory stores the total accumulated commitment for this partition $C_i = \prod_{k \in [N]} C_{ik} \in G$. On the aggregator's side, after collecting and aggregating all committed gradients by the trainers, as part of uploading to IPFS the updated partition, it sends to the directory service its addressing information. The remaining step is to verify that this updated partition is a pre-image of the independently computed partition commitment C_i. This guarantees that no trainer's data has been left out or altered by the aggregators. This can be performed by any participant (trainer or bootstrapper) but for simplicity we assume it will be performed by the directory service. Even in this case, the directory service only needs to access the updated model (much less than the total size of gradients data sent by all the trainers).

In case multiple aggregators are assigned to each partition, the directory would have to check each partial update, increasing the performance overhead. This can be avoided as follows. First, the directory also stores for each aggregator A_{ij} that has been assigned the i-th partition with corresponding set of trainers T_{ij}, the accumulated commitment $\prod C_{ik}$, where k takes the values of the trainer indexes in $[N]$ that correspond to trainers in T_{ij}. Then, when the trainer upload time window elapses, aggregators retrieve these accumulated commitments from the directory for each aggregator responsible for common partitions. Aggregators use the IPFS pub/sub functionality to publish their IPFS hashes for their partial updates. When an aggregator downloads a partial update it verifies that the partial update is indeed the pre-image of its corresponding accumulated commitment. Only the first aggregator who achieves the *true* globally updated partition writes back to the directory.

Merge-and-download can be extended in a similar way. When receiving an aggregation from an IPFS node, aggregators check if the commitment of the aggregation equals to the product of commitments that supposedly belong to it.

V. Experimental Evaluation

To evaluate the performance of our protocol, we conducted a series of experiments to measure the aggregation time and synchronization times and the communication sizes, as well as the impact of adopting the homomorphic commitments for verifiability. To accurately estimate these, we used the *mininet*[3] network emulator to simulate network conditions assuming aggregators and trainers have the same network bandwidth capabilities. Our testbed was a AWS c5ad.12xlarge instance running Ubuntu 18.04 LTS with 48 virtual CPUs and 96GB RAM. For our implementation we modified the publicly available IPLS code of [17] adding approximately 3000 Java lines. For Pedersen commitments we used the Bouncy Castle[4] implementation over elliptic curves secp256r1 and secp256k1.

Impact of merge-and-download. First, we examine the impact of the merge-and-download optimization by measuring the aggregation and gradient uploading delays for one iteration. We measure aggregation delays as the time interval between the write of the first gradient hash in the directory until all uploaded gradients are aggregated. The upload delay from the trainers' perspective is measured as the time between uploading the gradients to an IPFS node until the receipt of the store acknowledgment from the IPFS nodes. The experiment was run with 16 trainers, partition size 1.3MB, one aggregator per partition, and variable number of IPFS node providers P_{ij}. Aggregator and trainers had 10Mbps of bandwidth.

The results are shown in Figure 1. As expected, more providers implies smaller upload delays but larger aggregation delays. The number of providers that offers the best trade-off between aggregation and upload delay is roughly $\sqrt{16} = 4$, as expected from our analysis in Section III-E. Likewise, the upload delay with 4 IPFS providers is roughly the same as with 8 providers and also the aggregation delay is roughly the same with that of 1 or 2 providers. Comparing our approach with IPLS from [17], in Figure 1(top) we also include the overhead of our indirect communication imposes for 8 providers (labelled 8 (naive)) vs. the direct communication that [17] requires (labelled 8 (direct)). This shows that if we want to relax the strong direct communication assumption, merge-and-download is an essential mechanism to maintain efficiency.

Performance vs. variable $|A_i|$. To explore how aggregation and synchronization delay change when multiple aggregators are assigned to a partition, we conducted experiments with a setup of 16 trainers, 8 IPFS nodes, and a variable number of aggregators $|A_i|$ per partition. We segmented the model into 4 partitions of 1.1MB each, making each aggregator responsible for only 1 partition and setting the communication bandwidth to 20Mbps. We first deploy 4 aggregators, so $|A_i| = 1$, then 8 aggregators for $|A_i| = 2$ for each partition, and so on. To isolate the impact of $|A_i|$ for this experiment we do not use the merge-and-download mechanism.

Figure 2 shows that as the number of aggregators responsible for a same partition increases, the gradients aggregation delay decreases by almost half for each additional aggregator responsible for the same partition. That is because each aggregator has to download fewer gradients. On the other hand, the synchronization overhead increases, as expected, because increasing the number of aggregators for the same partition entails increased data communication for synchronization. That said, the total aggregation delay steadily decreased as $|A_i|$

[3]http://mininet.org/

[4]https://www.bouncycastle.org/

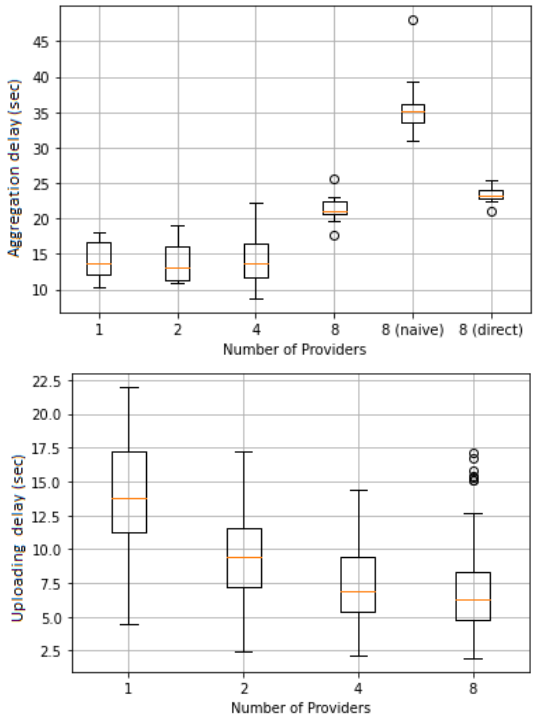

Fig. 1: Aggregation (top) and uploading (bottom) delays with variable number of providers.

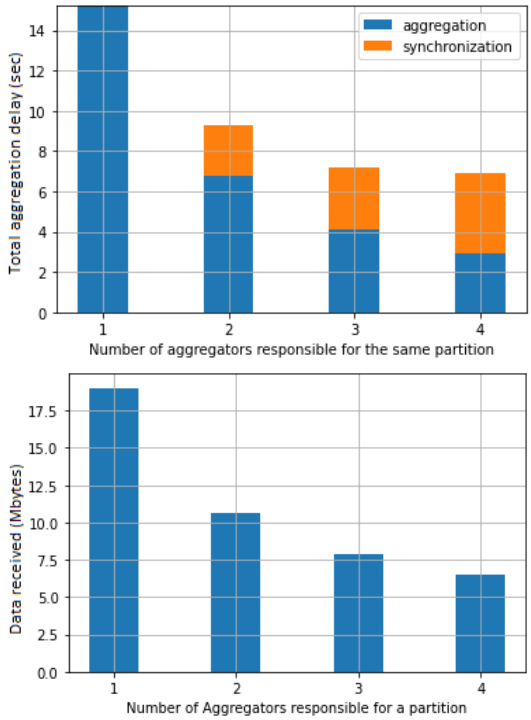

Fig. 2: Total aggregation delay (top) and total size of data received by an aggregator (bottom) in each iteration, vs. the number of aggregators assigned to each partition.

increases in our experiment, albeit at a progressively smaller rate due to the increased overhead of synchronization as $|A_i|$ grows. Overall, increasing the number of aggregators not only makes the system more robust, but also more efficient.

Impact of verifiability on performance. Next, we focus on how performance deteriorates when we adopt the modifications from Section IV for verifiable aggregation. Figure 3 shows the time needed for a trainer to compute the commitment of its gradients when varying the size of the model's parameters. We first note that this overhead can quickly become the bottleneck for our protocol. Even for medium-sized models (5M-10M parameters) like MobileNetV1, GoogleNet and SuffleNet, the computation cost of the commitments becomes severely expensive ($\approx$ 4-9 min). On the other hand, aggregators are in a much better position when they verify the validity of partial updates from other aggregators for the same partition, as a partition contains only a fraction of the model's parameters. Moreover, our Pedersen implementation is rather straight-forward and there is plenty of room for further optimizations (e.g., see [27, 28]), which we leave as future work.

Convergence and Accuracy. Because we segment and distribute the aggregation task to multiple aggregators, it easily follows that both the model's convergence rate and final accuracy will be exactly the same as that of traditional FL. Therefore we omit relevant measurements.

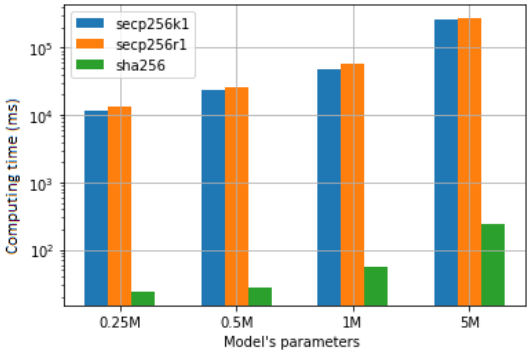

Fig. 3: Time needed to compute the SHA-256 hash and Pedersen commitment using secp256k1 and secp256r1 curves vs. size of model parameters (in logarithmic scale).

VI. Future Work and Open Questions

Our proposed protocol leaves plenty of room for further improvement both in terms of efficiency and security, some of which we briefly overview below.

Minimize the query load of the directory service. As shown in Algorithm 1, the directory service receives hashes and addressing information for each partition of the model and for each trainer, while also replying to multiple queries. It is possible to reduce its load by delegating the storage of its maps to the IPFS network, making the IPFS nodes responsible for replying to map queries. Moreover, instead of writing the hash of each partition to the directory service, trainers only need to send an accumulation over the hashes of gradient partitions.

Guarantee availability of gradients in IPFS network. Data unavailability can lead to significant slowdown and performance deterioration for the learning process. Hence we would like to take measures to ensure it (an alternative direction would be to extend our scheme to work with partially unavailable gradients or updates). This can be achieved by utilizing storage-incentive mechanisms for IPFS such as the Filecoin protocol [23]. However, since in our protocol both gradients and updates only needed for a short period of time, it may be sufficient to simply replicate them through a predetermined number of IPFS nodes. In that direction, it would be preferable to ensure a uniform allocation of gradients to nodes, to reduce the possibilities of collusion between malicious participants and potentially malicious IPFS nodes, e.g., based on the hash of the gradients and the nodes id's.

Delegating verification to the aggregators. In our current protocol, the directory service needs to access the updated partition and check if it matches the accumulated commitment. To reduce the directory service overhead, we can delegate the responsibility for proving this matching to the untrusted aggregators themselves, e.g., using cryptographic arguments [29, 30]. However, proving statements for a SHA-256 pre-image is notoriously computation-intensive, hence it would first make sense to replace it in IPFS with a proof-friendly hash [31]. This would allow aggregators to efficiently prove that the hash and Pedersen commitment come from the same gradients and partial updates, respectively.

Acknowledgements

The work was supported by Protocol Labs under grant PL-RGP1-2021-053. Dimitrios Papadopoulos was supported by the Hong Kong RGC under grant GRF-16200721.

References

[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] Yiqiang Chen, Xin Qin, Jindong Wang, Chaohui Yu, and Wen Gao. Fedhealth: A federated transfer learning framework for wearable healthcare. *IEEE Intelligent Systems*, 35(4):83–93, 2020.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] Yang Zhao, Jun Zhao, Linshan Jiang, Rui Tan, Dusit Niyato, Zengxiang Li, Lingjuan Lyu, and Yingbo Liu. Privacy-preserving blockchain-based federated learning for iot devices. *IEEE Internet of Things Journal*, 8(3):1817–1829, 2020.

[12] 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.

[13] 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.

[14] 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.

[15] Jiawen Kang, Zehui Xiong, Chunxiao Jiang, Yi Liu, Song Guo, Yang Zhang, Dusit Niyato, Cyril Leung, and Chunyan Miao. Scalable and communication-efficient decentralized federated edge learning with multi-blockchain framework. In *BlockSys 2020*, pages 152–165, 2020.

[16] Austine Zong Han Yapp, Hong Soo Nicholas Koh, Yan Ting Lai, Jiawen Kang, Xuandi Li, Jer Shyuan Ng, Hongchao Jiang, Wei Yang Bryan Lim, Zehui Xiong, and Dusit Niyato. Communication-efficient and scalable decentralized federated edge learning. In *IJCAI 2021*, pages 5032–5035.

[17] 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.

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

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

[20] Anousheh Gholami, Nariman Torkzaban, and John S. Baras. Trusted decentralized federated learning. In *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2022.

[21] Jianyu Wang, Anit Kumar Sahu, Zhouyi Yang, Gauri Joshi, and Soummya Kar. Matcha: Speeding up decentralized SGD via matching decomposition sampling. In *2019 Sixth Indian Control Conference (ICC)*, pages 299–300. IEEE, 2019.

[22] Hong Xing, Osvaldo Simeone, and Suzhi Bi. Decentralized federated learning via sgd over wireless d2d networks. In *IEEE SPAWC 2020*, pages 1–5, 2020.

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

[24] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In *SIGMOD 2020*, pages 637–652, 2020.

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

[26] 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.

[27] Bodo Möller. Algorithms for multi-exponentiation. In *International Workshop on Selected Areas in Cryptography*, pages 165–180, 2001.

[28] Fábio Borges, Pedro Lara, and Renato Portugal. Parallel algorithms for modular multi-exponentiation. *Applied Mathematics and Computation*, 292:406–416, 2017.

[29] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Eurocrypt 2013*, pages 626–645. Springer, 2013.

[30] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM CCS*, pages 1304–1316, 2016.

[31] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *USENIX Security 2021*, pages 519–535.