HyperAggregate: A sublinear secure aggregation protocol

Miloš Vujasinović
École polytechnique fédérale de Lausanne
milos.vujasinovic@epfl.ch

Supervisors: Lie He, Martin Jaggi

June 2021

Abstract

In this report, we address the issue of scalability of the existing secure aggregation protocols used in decentralized machine learning to a very high number of nodes. As a solution, we propose a novel decentralized aggregation protocol that can be parameterized so that the overall computation overhead scales logarithmically with the number of nodes. The parameterization also affects input privacy of the protocol, ranging from no input privacy to the privacy against a collusion of up to all but 2 nodes. However, stronger privacy guarantees come at the cost of the computation overhead. The protocol in its current version doesn't support users dropping out. We also discuss our implementation of this protocol and measure how well it performs.

1 Introduction

The era of big data has a brought an unique challenge to machine learning. There is an abundance of really good models, however, they are often so big that they require so much data to be trained on in order to extract their potential. An extreme example of this is a state-of-the-art language model called GPT-3 that has staggering 175 billion parameters [1]. Usually, parties wanting to train these models lack enough data to do so, and therefore they collaborate with many other parties that own the data they need. This could be as simple as moving the data of each party to a central node and doing the training there. Unfortunately, this is often not possible due to the importance of the privacy of the data. Sometimes, there are even laws in place to protect this data, as for example in healthcare [2] [3].

The restrictions force data owners to take another approach and it is usually in the form of collaborative learning, such as federated learning. The idea

behind federated learning is simple: models are trained locally by the data owners on data they have and occasionally the local models are communicated and averaged into a single global model. Depending how averaging is done we recognize centralized and decentralized approaches. At simplest, centralized approaches work by moving models in plaintext to a central server and doing averaging there. Unfortunately, 2 problems arise: firstly the central server becomes a bottleneck as all communication goes through it and secondly model updates can be inferred from exposing local models in a plain text which can lead to a leakage of the training data of the given model. The solution to the first problem is using all the available computational power available in the network instead of having nodes being idle while the central server does averaging. Decentralized approaches were born from this idea and they are based on direct communication between owners of local models to average them. The solution to the second problem, however, requires the use of special, so-called secure aggregation protocols. These protocols do aggregation of models without exposing any local model, part of the resulting global model. These protocols come in many flavours and they are available for both centralized [4], decentralized [5] approaches as well as the combination of both [6]. Later in this report, we are going to see how we can use ideas behind AllReduce and centralized federated learning [7] in order to create an efficient decentralized secure aggregation protocol.

2 Cryptographic primitives

In this section we introduce cryptographic primitives used in the protocol.

2.1 Additive secret sharing

Additive secret sharing is a simple n-out-of-n secret sharing scheme. It works by splitting an element x of a field F into n shares $x_1, x_2, \ldots, x_n \in F$ such that the sum of the shares is equal to the element:

$$x = \sum_{i=1}^{n} x_i.$$

The ownership of all n shares is required in order to reconstruct x. Therefore, splitting shares among n owners forces all of them to collaborate and communicate their shares to each other so that they can reveal the initial value.

3 Protocol

In this section, we introduce a novel decentralized secure aggregation protocol. The protocol provides input privacy. In our context input privacy means that local inputs to aggregation remain private to their owners and cannot be inferred

from the exchanged messages during and after aggregation, while the resulting global value is public.

The protocol takes inspiration from trees due to their inherit ability to perform operations in an efficient manner. We take the basic idea behind trees, modify it and enhance with additional properties in order to provide strong privacy guarantees. As a result, we get a structure called *aggregation tree* which is introduced in Definition 3. During the execution of the protocol, nodes are organized in an aggregation tree. For convenience, aggregation actors, aggregation participants, aggregation groups, aggregation levels and aggregation tree may be referred in the rest of the report as actors, participants, groups, levels and tree respectively.

Definition 1 (Aggregation group)**.** Aggregation group $\mathcal{G}$ *on a set of nodes* $\mathcal{S}$ *is a tuple* $(\mathcal{A}, \mathcal{P})$ *where* $\mathcal{A} \subseteq \mathcal{S}$ *is a set of nodes called aggregation actors, and* $\mathcal{P} \subseteq \mathcal{S}$ *is a set of nodes called aggregation participants.*

Definition 2 (Aggregation level)**.** Aggregation level $\mathcal{L}$ *on a set of nodes* $\mathcal{S}$ *is a set of aggregation groups* $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m\}$ *on* $\mathcal{S}$*.*

The participants of level $\mathcal{L}$ *are defined as*

$$\alpha(\mathcal{L}) = \cup_{\mathcal{G}(\mathcal{A},\mathcal{P})\in\mathcal{L}}\mathcal{P}.$$

Similarly, the actors of level $\mathcal{L}$ *are defined as*

$$\beta(\mathcal{L}) = \cup_{\mathcal{G}(\mathcal{A},\mathcal{P})\in\mathcal{L}}\mathcal{A}.$$

Definition 3 (Aggregation tree)**.** Aggregation tree T *on a set of nodes* $\mathcal{S}$ *is a sequence of levels* $(\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k)$ *on* $\mathcal{S}$ *for which the following rules apply:*

1. $|\mathcal{L}_k| = 1$
2. *For every level* $\mathcal{L}$ *and any pair of different aggregation groups* $\mathcal{G}_1(\mathcal{A}_1, \mathcal{P}_1) \in \mathcal{L}$ *and* $\mathcal{G}_2(\mathcal{A}_2, \mathcal{P}_2) \in \mathcal{L}$*:* $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset \wedge \mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$
3. $\forall i \in \{1, 2, \ldots, k-1\}$*:* $\alpha(\mathcal{L}_{i+1}) = \beta(\mathcal{L}_i)$
4. $\forall i \in \{1, 2, \ldots, k\}$*:* $\beta(\mathcal{L}_i) \subseteq \alpha(\mathcal{L}_i)$
5. $\alpha(\mathcal{L}_1) = \mathcal{S}$

Definition 3 makes it possible to do the following construction: we start with a set $\mathcal{S}$ and want to partition its elements into a selected number of parts. Each part will correspond to an aggregation group and the elements of the part will be the aggregation participants of the given group. Next, we pick a set $\mathcal{S}'$ subset of $\mathcal{S}$ and partition it in the same number of parts as $\mathcal{S}$ and assign each part to an aggregation group such that each aggregation group receives one part and no two aggregation groups receive the same part. The elements of these parts will be aggregation actors of aggregations groups they are assigned to. The obtained aggregation groups make the level 1 of an aggregation tree. We

then repeat the same process, but instead of starting with $\mathcal{S}$ we start with $\mathcal{S}'$ and partition it in order to get aggregation participants and partition a subset of it to get aggregation actors, this way creating level 2 of the aggregation tree. By repeating the process many times we end up with many levels. The last of these levels has only one aggregation group as defined by the rule 1 of Definition 3, meaning there is no partitioning done when we reach it, instead the whole set is aggregation participants and we pick a subset of it for aggregation actors of the given aggregation group. The goal of doing this is to start with a set $\mathcal{S}$ containing a lot of elements and partition them into smaller sets on which aggregation can be done in a privacy preserving manner efficiently.

The protocol works in the setting with many peers and a single server. There is a communication link between each pair of peers, as well between each peer and the server.

Assumption 1. *Peers are assumed not to drop out during aggregation.*

When a peer wants to do aggregation, it signs up for it by contacting the server and waits. The server keeps track of peers that have signed up and are waiting. When enough peers sign up, the server using a pseudo-random process generates an aggregation tree on a set of peers that have signed up and sends the tree to the peers, notifying them to start the aggregation. The server then empties the list of peers that are waiting for aggregation and waits for new ones to sign up. The number of peers that have to sign up for aggregation can either be all of them or a subset of a parameterized size. It is important to note that the server at no time takes part in aggregation itself, nor does it receive any messages from which it can even try to infer values held by peers. The only purpose of the server is to simplify discovery of peers and generation of aggregation tree.

Aggregation

The rest is the actual aggregation and it can be distinctly divided into 2 phases: the **upward** and the **downward** phase taking place in the given order. We have peers $X_1, X_2, \ldots, X_n$ participating in the aggregation and they want to calculate the sum of the values $D_1, D_2, \ldots, D_n$ they respectively hold, i.e. the final result of the aggregation should be $\sum_{i=1}^{n} D_i$. The values $D_1, D_2, \ldots, D_n$ must be from the same field F.

The upward phase starts with each aggregation participant creating additive secret shares out of the data it holds. The number of shares is equal to the number of the aggregation actors in their aggregation group at level 1. Then, it sends each share to one of these aggregation actors such that each of the aggregation actors receive only 1 of the participant's shares. After this is done every peer that is aggregation actor in one of the groups at level 1 sums up all shares it received from aggregation participants in the aggregation group it is aggregation actor in and sets the sum to be its data at start of the next level. This is repeated for each level of the aggregation tree. At the end we end up

with the aggregation group of the final level and its aggregation actors holding some sums. These aggregation actors then exchange the sums they are holding such that the each actor receives the sums from all the other actors in this group. The aggregation actors then sum all the sums they received together with the one they were holding, as the result getting D. We claim that D is equal to the sum of all values held by the peers participating in the aggregation at the start. The proof of this claim follows trivially from Claim 2. This means that D is the result of the aggregation we are looking for. A formal and more detailed version of this phase for non-distributed setting can be seen in Algorithm 1. Porting the non-distributed version into distributed setting is straightforward, however, requires handling of communicating intermediate values and proper synchronization.

The motivation of each peer for participating in aggregation is to receive the final result of it. Therefore, everyone wants the result, however only the aggregation actors of the group at the final level have the result after the upward phase. The downward phase is there to ensure that all peers participating in the aggregation receive the final result D. It starts by the aggregation actors of the group at the final level sending D to all aggregation participants of the same group. Afterwards, they go to the level below and repeat the same for the groups they were aggregation actors in at that level. This is repeated for all levels until the first one. When a peer receives D it also goes through all groups it is aggregation actor in and sends D to each aggregation participant of any of these groups. We know that all peers participating in the aggregation will receive D from the way aggregation trees are constructed as long as there are no peers dropping out during the aggregation. This claim follows from rules 3, 4 and 5 of Definition 3 and Assumption 1.

Correctness and properties

Claim 1. *For each aggregation group the sum of values aggregation participants in it have before doing upward phase on this group is equal to the sum of all shares that aggregation actors in it receive from these aggregation participants.*

Proof. Pick an arbitrary group $\mathcal{G}(\mathcal{A}, \mathcal{P})$ that for aggregation participants have $P'_1, P'_2, \ldots, P'_m \in \mathcal{P}$ which have values $D'_1, D'_2, \ldots, D'_m$ respectively when the upward phase for this group starts. The group also has aggregation actors $A'_1, A'_2, \ldots, A'_n \in \mathcal{A}$. By construction of aggregation tree we know that each aggregation group must have at least one aggregation actor and one aggregation participant. Each aggregation participant P'_i splits their value D'_i into shares $D_i'^{(1)}, D_i'^{(2)}, \ldots, D_i'^{(n)}$ such that $\sum_{l=1}^{n} D_i'^{(l)} = D'_i$ and sends them to aggregation actors in that order: $D_i'^{(l)}$ is sent to A'_l. At the end each aggregation actor A'_l has received $D_1'^{(l)}, D_2'^{(l)}, \ldots, D_m'^{(l)}$, and summing them we get $\sum_{i=1}^{m} D_i'^{(l)}$. By Assumption1 we know that none of the shares is lost. By summing up all shares the aggregation actors received we get $\sum_{l=1}^{n} \sum_{i=1}^{m} D_i'^{(l)}$ which can be rewritten as $\sum_{i=1}^{m} \sum_{l=1}^{n} D_i'^{(l)} = \sum_{i=1}^{m} D'_i$ proving the claim. □

Algorithm 1 The upward phase of the aggregation protocol

Input:
- F - field
- T - the aggregation tree of the aggregation
- *nodes* - List of nodes in the aggregation
- *data* - Values from F held by *nodes*

Output: Sum of values held by *nodes*

for $node \in nodes$ **do**
  $value_{0,node} \leftarrow data_{node}$
end for
for $level\_index, level \in$ ascending_by_index(indexed(levels(T))) **do**
  $S \leftarrow empty()$
  for $group \in$ groups($level$) **do**
    $group\_participants \leftarrow$ participants($group$)
    $group\_actors \leftarrow$ actors($group$)
    for $participant \in group\_participants$ **do**
      for $actor\_index, actor \in$ ascending_by_index(indexed($group\_actors$)) **do**
        $last\_share \leftarrow value_{(level\_index-1),participant}$
        if $actor\_index \neq |group\_actors|$ **then**
          $S_{participant,actor} \leftarrow$ random_from_field(F)
          $last\_share \leftarrow last\_share - S_{participant,actor}$
        else
          $S_{participant,actor} \leftarrow last\_share$
        end if
      end for
    end for
    for $actor \in group\_actors$ **do**
      $value_{level\_index,participant} \leftarrow \sum_{p \in group\_participants} S_{p,actor}$
    end for
  end for
end for
$final\_level \leftarrow$ last(levels(T))
$final\_group\_actors \leftarrow$ actors(first(groups($final\_level$)))
return $\sum_{actor \in final\_group\_actors} value_{|\text{levels}(T)|,actor}$

Claim 2. *After doing the upward phase on a level the sum of all shares aggregation actors in this level received is equal to the sum of all values aggregation participants of this level started the upward phase for the level with.*

Proof. From definition 3 we know that each peer cannot be aggregation participant in more than one aggregation group at a level. By using this fact and applying Claim 1 to each group on a level and summing respective sides for all the groups the claim follows. □

Observation 1 (Privacy guarantee)**.** *Let o be the smallest number of aggregation actors in any aggregation group of an aggregation tree. In that case, the protocol is safe against a collusion of up to $o-1$ peers in the aggregation as long as the total number of peers in aggregation is greater than o.*

Observation 1 tells us about the privacy guarantees of the protocol. It is relatively easy to see why this holds: each model is split into shares among all aggregation actors, meaning that all actors of an aggregation group would have to collude in order to be able to reconstruct a model. However, an exception to this rule is an aggregation tree with one aggregation group, in which all aggregation actors are at the same time aggregation participants of the group. If all but one aggregation actors collude in this case, the model of the remaining aggregation actor can be obtained by subtracting the sum of the models of the colluding actors from the result of the aggregation. This observation also reveals two important properties. First, performing aggregation on an aggregation tree with even a single group with only one aggregation actor is not privacy preserving. Therefore, when speaking of secure aggregation we assume that every aggregation group has at least 2 aggregation actors. Secondly, having only one aggregation participant in a group is safe in case of passive adversaries as long as the number of aggregation actors is at least 2.

For a set of peers there are many aggregation trees that satisfy Definition 3, however in this report, for simplicity, we consider only aggregation trees with an equal or relatively equal number of aggregation participants, as well as the number of aggregation actors in each group. As the purpose of using aggregation trees is reducing number of nodes at each level, we consider that in a group the number of aggregation actors is strictly smaller than the number of aggregation participants, yet greater than 1 in order to ensure privacy.

Theorem 1 (Aggregation overhead)**.** *The computation overhead of performing aggregation with N nodes on an aggregation tree with each group having p aggregation participants and a aggregation actors using our protocol is*

$$O\left((a+g)\log_{\frac{g}{a}}\frac{N}{g}\right).$$

Proof. There are 3 distinct stages in our protocol: 1) propagating aggregates from the aggregation participants of the first level until the aggregation actors of the aggregation group of the final level; 2) exchanging aggregates between

the aggregation actors of the aggregation group of the final level and summing them and 3) propagating the result of the aggregation downward to all peers.

Before calculating the cost of each phase let's calculate the number of levels in the aggregation tree: a node can be an aggregation actor only once and an aggregation participant also once on the same level. Because of this and the fact that only nodes remaining at the next level are aggregation actors at the current one we have that the number of nodes remaining at the next level is equal to number of aggregation actors in a group times the number of aggregation groups at the current level. If we have x nodes left at the current level, they form $\frac{x}{g}$ groups and each group has a aggregations actors, meaning there is $\frac{x}{g}a$ nodes left at the next level. We also know that the final level has the number of nodes left at most equal to the size of an aggregation group. From this we can write equation $N(\frac{a}{g})^l = g$ where l is the number of levels in the aggregation tree. By solving it we get $l = \log_{\frac{g}{a}} \frac{N}{g}$.

Now, we can calculate the cost of each stage. Firstly, each aggregation participant creates a additive secret shares from its data and sends a share to each actor of its group at the current level, both costing $O(a)$. After an aggregation actor receives the shares, it sums them, and since the number of aggregation participants in a group is g there are g additive shares received by each actor and they can be aggregated in $O(g)$. These operations are done in parallel by nodes on a level and therefore, the cost of a single level is $O(a + g)$. Since there is $\log_{\frac{g}{a}} \frac{N}{g}$ levels, the total cost of the first stage is $O((a + g) \log_{\frac{g}{a}} \frac{N}{g})$. In the second stage, there is a aggregation actors that send each other partial aggregates they received and sum the received afterwards. Both of these operations are done in $O(a)$ and in parallel by the actors, meaning the total cost of the second stage is $O(a)$. Finally, in the third stage, we have each node sending the resulting model to all aggregation participants in the groups the node plays a role of aggregation actor. A node can be aggregation actor in at most l groups (1 on each level), and each group has roughly g participants. Since this can be done in parallel by all nodes, the cost of the third stage is $O(l\ g) = O(g \log_{\frac{g}{a}} \frac{N}{g})$.

By adding the costs of all stages we get that the total computation overhead of the protocol is

$$O\left((a + g) \log_{\frac{g}{a}} \frac{N}{g}\right).$$

□

Theorem 1 reveals the dependency of aggregation overhead on the way aggregation tree is generated and can help us optimize the protocol in order to reduce it. However, even for a simple selection of parameters, e.g. $g = 4$ and $a = 2$ we get that the overhead of the protocol is $O(\log N)$, which is significantly better than the previous work. In general, g and a are in practice supposed to be so small compared to N that the dominant factor in the overhead would always be $\log_{\frac{g}{a}} \frac{N}{g}$ for large N.

Combining Theorem 1 and Observation 1 reveals a unique trade-off. Observation 1 tells that having higher number of aggregation actors in a group is beneficial for privacy. However, increasing the number of aggregation actors while keeping the number of aggregation participants constant, decreases the base of the logarithm in the overhead, as well as the factor $a + g$, which results in the higher overhead. In turn, decreasing the base of the logarithm requires increasing the number of aggregation participants, which additionally increases the factor $a + g$. Therefore, the optimization of the protocol is not straightforward and in practice would require a lot of testing to get right. Keeping these things in mind while using the protocol is critical and setting them properly is highly dependant on an use case.

4 Implementation

In this section we discuss a prototype implementation of the introduced protocol. The code is available at `https://github.com/mvujas/HyperAggregate`.

The prototype is implemented in `Python`. The code is organized in several subpackages many of which depend on each other, two main being `client` and `server` subpackage which implement client and server logic respectively. Other packages are `netutils` that implements classes that simplify network communication; `aggregation_profiles` that offers to a programmer an easy-to-use interface for representing aggregation logic, data preparation before sending over network and splitting data into shares; and finally there is subpackage `shared` that collects model classes and functionality shared by client and server side that doesn't fit into neither `netutils` nor `aggregation_profiles`.

4.1 Network communication

Network communication is done using library `ZeroMQ`[1]. We provide a class that offers functionality of receiving and sending messages to anybody. Despite having a server we use this class on peers, as well as the server, the reason being relatively simple: servers usually use request-response pattern, i.e. they wait for a request and respond to it immediately when it comes, however, in our case the server also waits for enough peers to sign up for aggregation, then generates an aggregation tree and contacts the signed up peers to notify them about the structure of the tree. This requirements go out of the way of the standard request-response pattern and are more similar to the way peers communicate between each other. The socket abstraction class, `ZMQDirectSocket` follows synchronous programming model, meaning it exploits multithreading and multiprocessing in order to wait for incoming messages in parallel to the execution of the rest of the code. Every instance of this class creates and starts a new process and a thread: the process runs in an infinite loop waiting for messages to come, when a message comes it is pushed to a queue. The given queue is visible to the main process and the created thread can access it. The thread runs an

[1] `https://zeromq.org/`

infinite loop and whenever a new message is available in the queue the thread gets it and calls a callback function on it. This callback function is passed to the instance of the class upon its creation. We note that the usage of a thread instead of a process for message processing is important as threads created by the main process share memory with it, while processes use a separate memory, therefore making it easier for threads to work with objects that are already in the memory of the main process. The question that may arise is why is a process used for receiving and a thread for processing messages instead of having a single thread that would do the both. When a message is received it is put to a buffer of a small limited size until it's read. If we used only a single thread for both after receiving a message the thread would run the callback function. The callback function may run for a long time and in the meantime many more messages could come causing an overflow of the buffer and as the result we would have a loss of data which we try to avoid by using the described architecture.

Messages are `Python` objects that are serialized before sending. Each message has a type. For every type there is a specific callback that handles it. A group of classes called message routers are responsible for identifying the correct callback to call based on the type of a message. The main server and client class, `SchedulingServer` and `PrivacyPreservingAggregator` respectively, are both subclasses of the base message router. This can be seen in Figure 1 which displays the most important parts of the system using a class diagram.

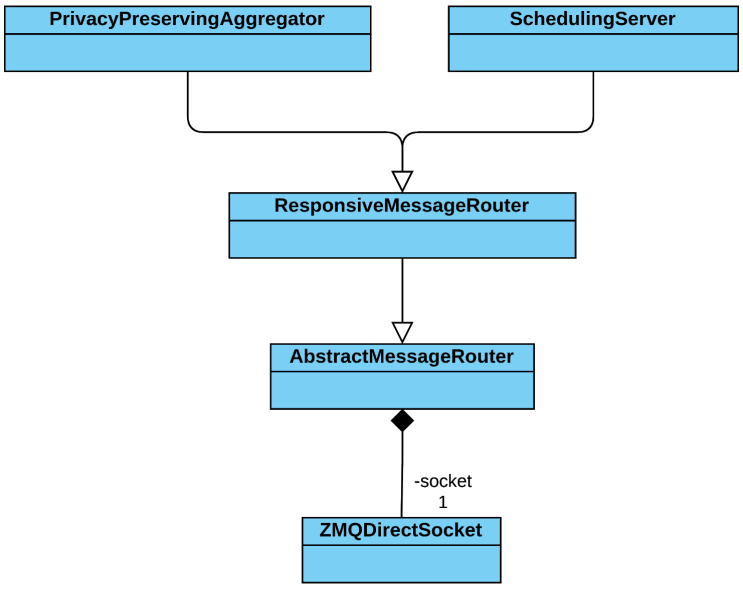

Figure 1: Simplified class diagram of the system

4.2 Server

The server is there with a single purpose described in the protocol: It waits for enough peers, in this implementation called clients, to sign up for aggregation. The number of peers that have to sign up for aggregation before the server starts generating an aggregation tree is called target size and it is specified upon creation of the server object. The server also accepts parameters used for tree generation, namely group size and the number of actors in each group. Aggregation tree is generated by randomly partitioning the set of peers participating in the aggregation in groups of size specified by the parameter group size. For each aggregation group the specified number of actors is picked from the aggregation participants on this level. Aggregation participants of the next levels are aggregation actors of this one. This is repeated until no more than a single group can be generated from the available participants. The generation of the tree terminates only if at each level there is less participants than at the previous or if there is only one level with a single group. Because of this group size must either be equal to the target size or higher than the number of actors in each group.

When the generation of a tree is done the server sends to each peer that has signed up for the aggregation groups that the peer is in either as an aggregation actor or an aggregation participant, as well as an aggregation profile telling the peer how to do some parts of the aggregation. The peers are also sent an identifier for aggregation which they use to make distinction between messages that are late from previous aggregations they have participated in and the aggregation they are doing at the moment.

4.3 Aggregation profiles

Aggregation can be done in many different ways on many different types and there is no generic way to make a single logic that will work for all. For example, additive secret sharing is only one type of secret sharing, however, we may at some point want to use a different secret sharing scheme that creates shares out of data in a different way and as well uses a different operation other than addition to aggregate these shares. Aggregation profiles offer flexibility to a programmer to implement another secret sharing schemes, as well as work with different types or prepare aggregates differently for the transport over network. An aggregation profile requires implementation of several methods:

- A method that transforms data into the format in which we intend to send it over network. This format doesn't have to be a low level format since the data object will be serialized by the code itself.
- A method that transforms data from the format in which we intend to send it over network into the format which can be used by application.
- A method that creates a list of shares from data in the format in which we intend to send it over network such that the resulting shares are in the same format.

- A method that aggregates shares in format we intend to send them over network and produces result in the same format.

As it can be seen most of these methods work on data in the format in which we intend to send it over network. This way we avoid an overhead of unnecessarily converting data back and forth between the two formats during aggregation. Instead at the start of aggregation all peers doing aggregation transform their data chunk into the format specified for sending over network, and all operations are done on it while in this format. Only at the end, when the resulting aggregate is received it is transformed back.

As an example, we provide an aggregation profile that defines averaging of `PyTorch`[2] models, precisely their state dictionaries, using additive secret sharing. State dictionaries are prepared by converting tensors of type `float` corresponding to layers of the model into `numpy`[3] arrays of type `int64`. The conversion is done by multiplying each entry with 10 brought to a parameterized degree, so that only certain number of decimals is preserved. The degree is equal for all tensors and for all peers participating in the same aggregation. We assume parameters of the model aren't too small nor too big to have a significant loss of information in this process. Even if this is the case it is assumed that programmer will take care of this before aggregation, as this is usually an indicator of a bad model prone to overfitting. The preparation is done by pairing the resulting state dictionary of `numpy` arrays with an integer equal to 1 making a 2-element tuple. We create additive shares of a number by picking uniformly pseudo-random values in `int64` range as shares until there is one share left to create, it is calculated by subtracting the sum of the selected values from the input number. Additive shares of an array are calculated similarly: we generate arrays of the same size as the input array with values picked in uniformly pseudo-random way from `int64` range, the final share is calculated by subtracting the sum of selected shares from the input array. The result of creating shares on our prepared format is a list of 2-element tuples where the first element is a state dictionary of `numpy` arrays such that if we sum corresponding arrays for each share under their keys of the dictionary we get the starting state dictionary. The other element is a just a number split among shares in the aforementioned way. The aggregation of a set of shares is done by summing arrays of all shares under corresponding entries of their state dictionaries, while the second element of the tuple is obtained by summing the second element of all shares. After aggregation is done the final result is transformed back by dividing each value of each array in the state dictionary by the second element of the tuple and converting them back into `PyTorch` tensors of type `float` in the reversed process of the one already explained. Note that the second element of the tuple is at the end equal to the number of peers that participated in the aggregation as all of the peers set it to be equal to 1 at the start. Therefore, the resulting state dictionary is equal to the average of state dictionaries of all peers that participated in the aggregation.

[2] https://pytorch.org/
[3] https://numpy.org/

4.4 Client

Client side closely follows the protocol introduced in Section 3, however, it adds few optimizations on top of it.

If we think back about downward phase of the protocol we would realise that there is an excessive number of messages sent that are not necessary: each peer sends the aggregated model to all aggregation participants of the groups that it plays a role of aggregation actor in, meaning that each aggregation participant receives the same model from every aggregation actor in a group, repeated for each group it is in as aggregation participant. Therefore, whenever a participant receives the aggregated model it sends a message that it doesn't need it anymore to all aggregation actors of groups it is an aggregation participant in starting from the top of the tree. Before sending the model in downward phase to an aggregation participant, an aggregation actor first checks whether it has received a message from the participant telling the actor that the participant doesn't need the model, and if that is the case the actor aborts sending the model to the participant.

Client side works in a synchronous manner from outside the class, meaning that when a thread from outside calls the method that does aggregation, the execution of the code after this call continues only when the result of the aggregation is returned. There are 4 states a client could be in:

- The client isn't doing aggregation, nor is signed up for any.
- The client sent a request to sign up for aggregation, and waits a response from the server to confirm whether the sign-up was successful.
- The client has signed up for aggregation successfully, but waits for the server to send it an aggregation tree for the aggregation.
- The client has obtained an aggregation tree and is doing the aggregation with other peers.

4.5 Middleware

This protocol was built as a part of a bigger project called `DeAI`[4], a system for decentralized machine learning, that unlike the protocol is written in `JavaScript`.

In order to enable the use of the protocol from `JavaScript` we provide a middleware that takes the task of translating calls from `JavaScript` to `Python`. The translation is accomplished by having a `Python` server visible only locally and running client side aggregation logic. When making a call from `JavaScript` it sends a request with a model to the `Python` server, which further contacts the aggregation server signing up for aggregation and later doing the aggregation upon receiving an aggregation tree. After receiving the final result of the aggregation the `Python` server responds to the request by returning the aggregated model back to the `JavaScript` client. These two sides of middleware

[4]https://github.com/epfml/DeAI

may use different libraries to work with models meaning that the same models could be represented in a different way. For this reason, all models exchanged between the `JavaScript` client and the `Python` server must be in a predefined intermediate format.

5 Results and discussion

In this section we benchmark and discuss the performance of the protocol, as well as the overhead the middleware adds on top of it.

The benchmarks are done on MNIST [8] dataset which contains 28×28 grayscale images of handwritten digits. It is split into 60000 training and 10000 test images. The training data is split uniformly among all peers in equal shares. The benchmarks are done on a single 16 GB memory machine, meaning there is no speed of network to take into account, while the performance of CPU is significantly hindered due to multiple peers being run in parallel and sharing CPU time. Besides the CPU, the programs also share the same memory, which in some of the tests with a high number of peers causes the memory to become full and forcing the machine to use virtual memory, in turn slowing aggregation. The tests that caused this occurrence will be noted. The benchmarks are done on a six layer neural network with input of size 28×28 with layers in the following order:

- Convolutional layer with 32 output channels, kernel size 3×3 and stride 1×1 followed by ReLU activation function
- Convolutional layer with 64 output channels, kernel size 3×3 and stride 1×1 followed by ReLU activation function
- Max pooling layer with kernel size 2×2
- Dropout layer with probability of zeroing an element equal to 0.25
- Dense layer with 128 output neurons followed by ReLU activation function
- Dropout layer with probability of zeroing an element equal to 0.5
- Dense layer with 10 output neurons followed by ReLU activation function

In the first set of tests we measure the amount of time it takes between receiving an aggregation tree from the server and obtaining the result of aggregation. We also compare these times with the amount of time it takes for one of the simplest decentralized secure aggregation protocols with additive secret sharing to do aggregation with the same number of peers. The aforementioned protocol works by having each of the peers participating in aggregation split their model into the number of additive shares equal to the number of peers and sending a share to each of them. In the next step, each peer sums up the shares it received and sends the result to all other peers. Finally, peers sum up all results they received in the second step and this sum is equal to the sum

of all models. For convenience, we will call this protocol `A2A protocol`, and it just so happens that our protocol is a generalization of this one. By generating an aggregation tree with a single aggregation group where all peers are both aggregation actors and aggregation participants we have our protocol working just like `A2A protocol`. The mean time of aggregation is obtained over 100 times it took to do aggregations recorded by peers. The times are measured with 3, 6, 12, 16, 24, 30, 36, 42, 51 and 60 peers.

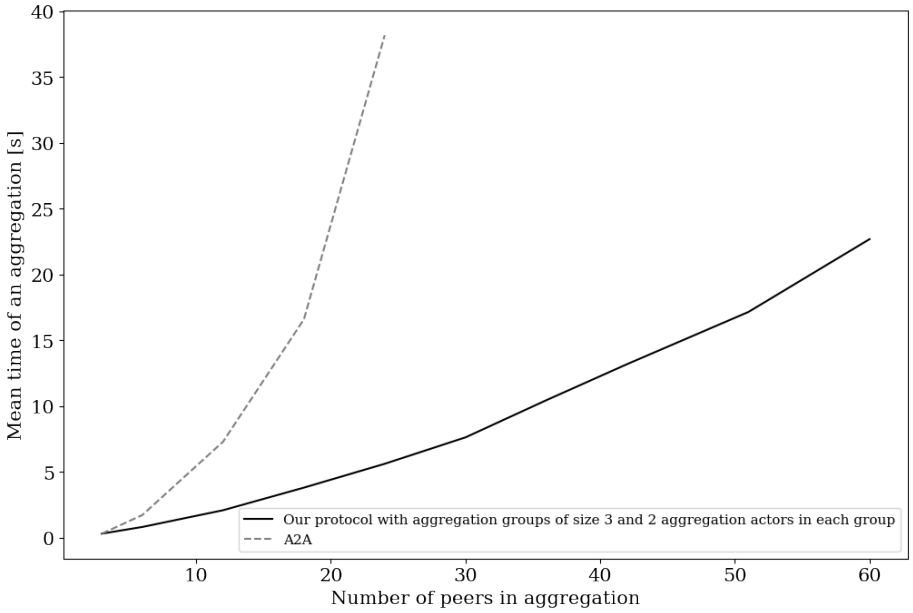

Figure 2: Comparison of the effect the number of peers have on the time of aggregation between our protocol and `A2A protocol`

As it can be seen in Figure 2, our algorithm performs significantly better than `A2A protocol`, however, we note that in both cases we have virtual memory coming in play, at 24 in case of `A2A` and at 51 and 60 in case of our protocol. We note that from the way the tests are performed the instances running in parallel share resources: memory, CPU time. Meaning, that the protocols run the number of peers times slower than they would otherwise if all peers had their own resources. Therefore, we should also look at Figure 3 which shows the time of aggregation divided by the number of peers. The figure shows a curve that closely resembles a line and the logarithmic curve, however it requires benchmarking with a higher number of peers in the environment not affected by limited memory in order to see which of these curves it actually is.

Our protocol is designed for secure aggregation with a high number of peers. However, instead of doing a single aggregation with all peers, it also makes sense

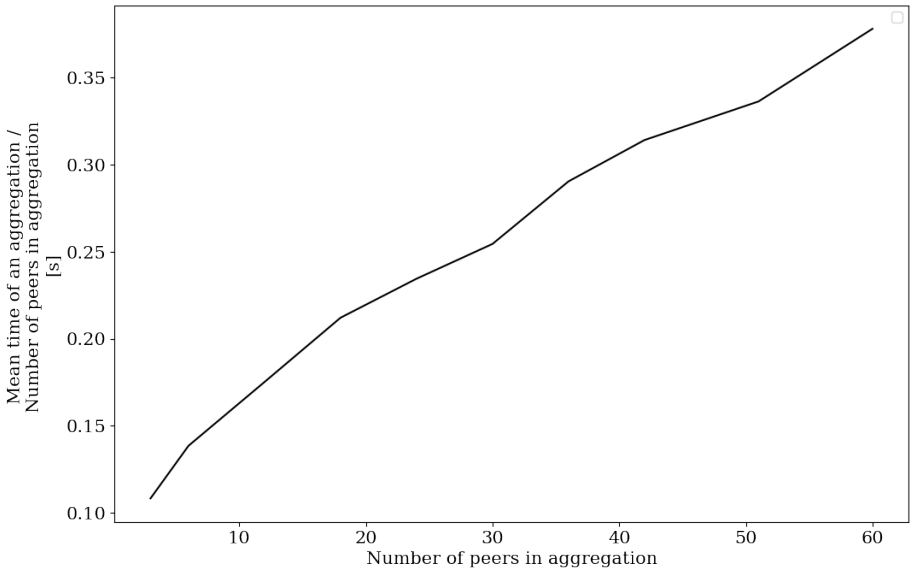

Figure 3: The effect the number of peers have on the time of aggregation divided by the number of peers for our protocol

to do many aggregations with very few peers. The benefit of doing many small aggregations is that each aggregation takes less time, however by aggregating all models, the resulting one will benefit from all data, while in small aggregations the resulting models will depend mostly on the data the models of peers in aggregation have been trained on. The idea is that if a peer participates in many small aggregations with different peers every time eventually the model would benefit from the data of all peers. We test this by comparing the mean number of iterations it takes a peer to reach 98% accuracy on MNIST dataset in both cases. The mean is calculated over all peers that started the given test, e.g. if 15 peers started a test the mean is calculated by averaging values each of them got. The results are shown in Figure 4. As we can see the difference at the start is relatively small and starts growing with the total number of peers, however the proportion between the two doesn't seem to change significantly. Keeping in mind that the time it takes for an aggregation with 3 peers stays constant, while the time for a single aggregation with all peers grows with the number of them, this implies that it is beneficial to do many small aggregations rather than big ones in order to reach target accuracy faster. It is important to note that we do not test the best possible accuracy each approach can achieve, but rather how fast they can reach an arguably high accuracy.

Finally, we measure the overhead the middleware adds on top of aggregation time. This is tested with 3 peers by aggregating simple two layer neural networks having only dense layers of size 2×100 and 100×1 respectively. We obtain

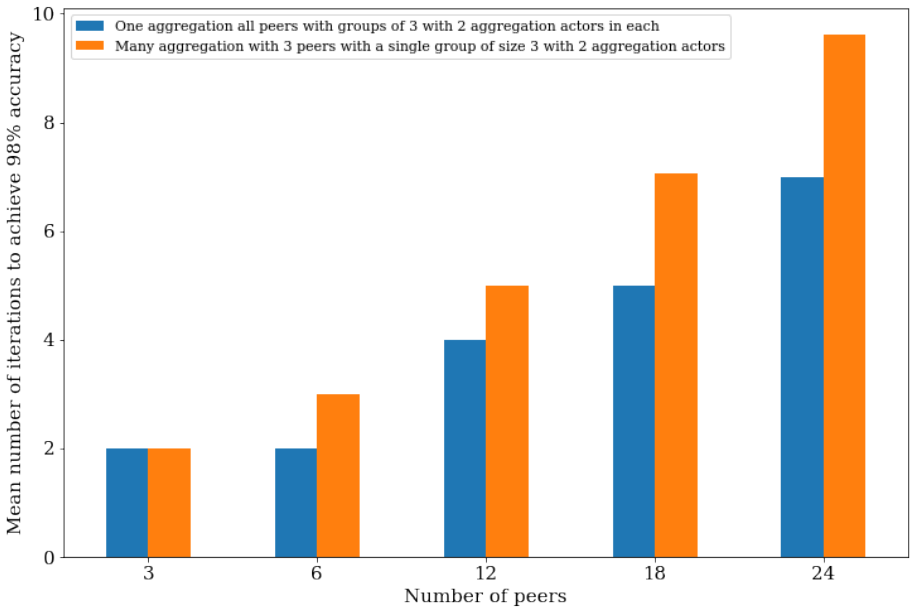

Figure 4: Comparison of the number of iterations it takes to reach 98% accuracy on MNIST dataset between a single aggregation with all peers and many small aggregation with 3 peers each

that time between receiving an aggregation tree and calculating the result of the aggregation on average takes 0.1015 seconds; peers on average spend 0.0455 seconds waiting for an aggregation tree after signing up for aggregation; while the middleware adds overhead of only 0.0067 seconds on average. In the real world scenario the majority of time would be spent waiting for enough peers to sign up for aggregation, as we use a script to start all peers at the same time, and the time of aggregation itself would take much longer due to network bandwidth which is a bottleneck in real world. Unlike the other two values, the middleware overhead stays the same as in the test as both sides of the middleware run locally and therefore aren't subject to network bandwidth. This results show that the overhead of the middleware is insignificant compared to the time it takes for the rest of aggregation, to the degree that the use of the middleware likely wouldn't be noticed at all by a human observer.

6 Future work

We've seen the protocol and the proof of concept confirming that the protocol works. However, there are still aspects of the protocol we haven't discussed mainly as they still require improvements and polishing. The aims of the future work should be in the areas of handling peers dropping out during aggregation and eliminating the need for a central server in aggregation tree generation.

Working in network environment is tricky and having users drop out or not being available is inevitable. Therefore, in order to use the protocol in practice it must be robust enough to deal with these scenarios without losing much efficiency. The naive approach of dealing with the issue would be to restart aggregation with the remaining nodes whenever a node drops out. The applicability of this approach highly depends on the ratio between the probability that a peer disconnects during an aggregation and the number of peers in the aggregation. For example, if the probability is small as well as the number of peers in the aggregation, the aggregation likely won't even restart at all, while if the probability is high as well as the number of peers the aggregation may repeat many times. However, the purpose of creating this protocol was doing aggregation with an extremely high number of peers and in such environment even the small probability would likely cause multiple repetitions. A possible solution would probably require dealing with a peer dropping out differently in the groups it plays a role of aggregation actor compared to those in which it plays a role of aggregation participant. For example, the aggregation actors of an aggregation group could do an intersection of all aggregation participants in the group they received a share from and only keep shares from those participants that all of them have received a share from. This approach works for level 1, however, in order to apply it on higher level it would require special tree organization such that aggregation actors of a group at one level must be aggregation participants in the same group at the next level. This way the aggregation actors of the latter could check whether all of them received a share from all aggregation actors of the group at the previous level and keep their shares only

if this is the case. Handling aggregation actors dropping could be done either by using a secret sharing scheme that offers possibility of existence of redundant shares, such as Shamir secret sharing [9], or having backup aggregation actors that would replace aggregation actors that drop out.

We have to assume certain degree of trust in the central server as it plays the key role in generation of aggregation trees and the current version of the protocol doesn't have a way to make sure that the resulting tree is truly pseudo random and not manipulated in any way. This raises a question of possibility of the server colluding with peers in an aggregation. Observation 1 always holds, however even if there is a number of colluding peers higher than the threshold there is still a probability that the colluding peers will not end up as the aggregation actors in the same group and therefore won't be able to learn anything. However, by colluding with the server the tree could be manipulated in such way that would make this attack unavoidable. For this reason, there should either be a way the server can prove that an aggregation tree was generated in a truly pseudo-random process or remove the server altogether and come up with a way for peers to self organize into an aggregation tree instead.

7 Conclusion

This report introduces a novel parameterizable secure aggregation protocol aimed for use in decentralized machine learning with an extremely high number of nodes and it can be tuned for resistance to collusion, as well as aggregation overhead. The protocol doesn't support peers dropping out during aggregation.

Benchmarks performed on a prototype implementation of the protocol show significant improvement compared to a really simple secure aggregation scheme, as well as that doing many aggregations with a small number of peers might be less time consuming way of achieving high accuracy than doing big aggregations with all peers.

Acknowledgements

I would like to thank the members of Machine Learning and Optimization Laboratory at École polytechnique fédérale de Lausanne, especially Lie He and Martin Jaggi, without whose feedback and suggestions this work wouldn't be possible.

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray,

B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[2] "The eu general data protection regulation," https://gdpr.eu/, accessed: 13-06-2021.

[3] C. for Disease Control and Prevention, "Health insurance portability and accountability act of 1996 (hipaa)," https://www.cdc.gov/phlp/publications/topic/hipaa.html, 2018, accessed: 13-06-2021.

[4] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[5] B. Jeon, S. M. Ferdous, M. R. Rahman, and A. Walid, "Privacy-preserving decentralized aggregation for federated learning," 2020.

[6] J. So, B. Guler, and A. S. Avestimehr, "Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning," *CoRR*, vol. abs/2002.04156, 2020. [Online]. Available: https://arxiv.org/abs/2002.04156

[7] L. He, S. P. Karimireddy, and M. Jaggi, "Secure byzantine-robust machine learning," *arXiv preprint arXiv:2006.04747*, 2020.

[8] Y. LeCun, "The mnist database of handwritten digits," *http://yann.lecun.com/exdb/mnist/*, 1998.

[9] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.