Spyker

Asynchronous Multi-Server Federated Learning for Geo-Distributed Clients

Zuo, Yuncong; Cox, Bart; Chen, Lydia Y.; Decouchant, Jérémie

DOI
10.1145/3652892.3700778

Publication date
2024

Document Version
Final published version

Published in
Middleware 2024 - Proceedings of the 25th ACM International Middleware Conference

Spyker: Asynchronous Multi-Server Federated Learning for Geo-Distributed Clients

Yuncong Zuo
zuoyuncong@gmail.com
Delft University of Technology

Bart Cox
b.a.cox@tudelft.nl
Delft University of Technology

Lydia Y. Chen
lydiaychen@ieee.org
Delft University of Technology

Jérémie Decouchant
j.decouchant@tudelft.nl
Delft University of Technology

ABSTRACT

Federated learning (FL) systems enable multiple clients to train a machine learning model iteratively through synchronously exchanging the intermediate model weights with a single server. The scalability of such FL systems can be limited by two factors: server idle time due to synchronous communication and the risk of a single server becoming the bottleneck. In this paper, we propose a new FL architecture, Spyker, the first multi-server FL system that is entirely asynchronous, and therefore addresses these two limitations simultaneously. Spyker keeps both servers and clients continuously active. As in previous multi-server methods, clients interact solely with their nearest server, ensuring efficient update integration into the model. Differently, however, servers also periodically update each other asynchronously, and never postpone interactions with clients. We compare Spyker to three representative baselines – FedAvg, FedAsync and HierFAVG – on the MNIST and CIFAR-10 image classification datasets and on the WikiText-2 language modeling dataset. Spyker converges to similar or higher accuracy levels than previous baselines and requires 61% less time to do so in geo-distributed settings.

CCS CONCEPTS

• **Computing methodologies** → **Distributed artificial intelligence**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Byzantine Learning, Asynchronous Learning, Resource Heterogeneity

ACM Reference Format:
Yuncong Zuo, Bart Cox, Lydia Y. Chen, and Jérémie Decouchant. 2024. Spyker: Asynchronous Multi-Server Federated Learning for Geo-Distributed Clients. In *24th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652892.3700778

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700778

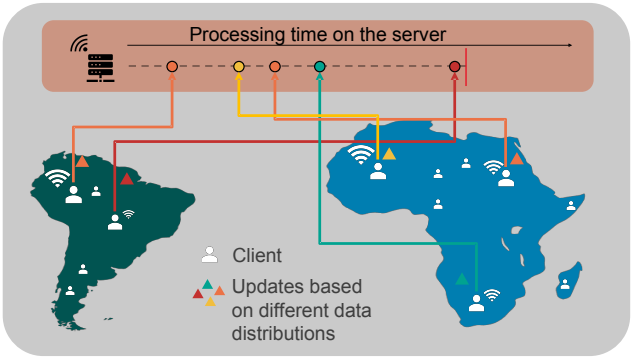


Figure 1: A training round of a synchronous FL system with heterogeneous devices and networks. The times at which the server receives client updates are indicated with colored spheres. The server waits for all client updates to be received to update the model, which results in a mostly idle server.

1 INTRODUCTION

Federated Learning (FL) is an emerging paradigm to train machine learning models iteratively on extensive data that are dispersed across many users and cannot be openly shared due to privacy regulations [24]. FL is widely uses in applications such as mobile keyboard prediction [15], spoken language comprehension [17], and digital healthcare [4, 21, 27, 36].

The most prevalent FL architecture is composed of a single-server and multiple clients, and assumes a synchronous training process over multiple rounds [24]. In each round, the server selects a set of clients that receive the current global model, train it with their local dataset, and send a model update back to the server. Once it has received all model updates, the server aggregates them and computes the new global model that is used in the subsequent round. The training process is repeated until the global model converges.

The duration of a training round in a synchronous FL framework is determined by the longest client update time because a round only completes when all updates of selected clients are received. Clients exhibiting heterogeneous computational capacity and network bandwidth can unfortunately prolong the update time, degrading the efficiency of FL systems.

The computational heterogeneity stems from the difference in their hardware and software stacks; whereas network latency difference is rooted in the geo-distributed settings where clients are spread across several countries or continents [5, 16, 40]. The result

Table 1: Comparison of representative multi-server FL algorithms with Spyker.

	Mutli-Server	Async Client/Server	Async Server/Server
FedAvg [24]	✗	✗	N/A
FedAsync [34]	✗	✓	N/A
HierFAVG [1]	✓	✓	✗
Spyker	✓	✓	✓
Sync-Spyker	✓	✓	✗

of such heterogeneity is often system inefficiency: the server, along with clients near it or those with superior computing power, remain underutilized. Fig. 1 illustrates this problem with a synchronous FL round that involves five clients distributed over two continents. The server has to collect all five client updates before updating the global model. The first four clients become idle after they send an update to the server and the server remains idle until the end of the round. It is worth noting that such heterogeneous settings generally imply a non-Independent and Identicallly Distributed (non-IID) data distribution [9, 26] since data generated by each client can be influenced by their immediate environment.

Asynchronous FL frameworks [6, 31, 34] aim at eliminating performance bottlenecks arising from heterogeneous networks and clients power. Under such frameworks, the server immediately processes a client's model update upon receiving it and returns the updated global model to the corresponding client. Thus, slow clients in an asynchronous FL framework no longer hamper the overall performance. However, this can inadvertently shift the bottleneck to the server. Specifically, if the server is continuously busy processing client updates, it might cause waiting periods for clients, in particular as the number of clients increases.

To prevent potential server-based bottlenecks, one can deploy multiple servers that each interact with a subset of the clients. In particular, hierarchical FL frameworks [1, 19, 22] rely on secondary servers that interact with clients, and on a primary server that merges the models of the secondary servers. This multi-server structure decreases the average communication latency between a client and a server, given that clients upload their model updates to the nearest server. However, in such hierarchical FL frameworks, the primary server still relies on a synchronous procedure to update its global model, which in geo-distributed settings implies that secondary servers stop replying to clients.

We introduce Spyker[1], a novel multi-server and fully asynchronous FL framework that relies on a flat architecture of geo-distributed servers. Moreover, interactions between clients and their nearest server, and between servers themselves are asynchronous to maintain low response times, and reduce both client and server idle times. Spyker accelerates the training of a model in geo-distributed settings, where heterogeneous clients and servers are located in different geographical regions. Table 1 compares Spyker to the relevant related work and highlight its main characteristics

Contributions. This work makes the following contributions to realize the asynchronous multi-server FL vision:

- To support multi-server asynchronous FL, we define the age of a server model and the staleness of a client update. Based on these definitions, a server determines the weight that server models or client updates should be given when aggregating them into its local model.
- To maintain high accuracy despite resource heterogeneity, Spyker triggers an exchange and aggregation of server updates when the ages of two server models differ too much or when the age of a model is sufficiently changed since the last server model aggregation. To minimize the complexity of the server model aggregation process, in particular in asynchronous networks, a single server at a time can trigger the exchange of server models. This decision is helped by a token that servers circulate among themselves and that collects the age of all server models.
- To further maintain accuracy despite data heterogeneity, i.e., clients having non-independent and identically distributed (non-iid) datasets, we detail a local learning rate decay strategy that prevents server models from becoming biased towards the data distributions of fast clients.
- We evaluate Spyker's performance and compare it to three recent and representative FL frameworks, namely FedAvg [24], HierFAVG [1] and FedAsync [34]. We also consider a variant of Spyker, which we call Sync-Spyker, where servers use a synchronous model exchange protocol. We evaluate all frameworks in emulated geo-distributed settings using the MNIST and CIFAR-10 image datasets and the WikiText2 language modeling dataset. We report the accuracy and convergence speed of all frameworks, and evaluate their ability to scale with the numbers of clients and servers.

Our results show that Spyker converges to similar or higher accuracy levels than previous baselines with a 61% shorter running time in geo-distributed settings. Spyker also scales better than other baselines with the number of clients thanks to its flat multi-server architecture. For example, increasing the number of clients from 100 to 200 multiplies the time our three baselines require to reach 90% accuracy by at least 1.64, while Spyker's convergence time is only multiplied by 1.21 (i.e., scalability is improved by 26%).

This paper is organized as follows. Sec. 3 provides background on FL systems and introduces the most commonly used synchronous, asynchronous, and hierarchical FL algorithms. Sec. 4 describes our system model and gives an overview of Spyker. Sec. 5 discusses Spyker in detail, including the local update and global model exchange strategies. Sec. 6 presents our performance evaluation.

Sec. 2 discusses the related work, and Sec. 7 concludes this paper.

[1]The Spyker was the first four-wheel-drive car. The wheels of such a car are similar to our FL framework's servers: they make progress asynchronously and independently from each other, and occasionally synchronize.

2 RELATED WORK

This section discusses additional related work on asynchronous and multi-server FL that complements Sec. 3.

2.1 Asynchronous FL

ASO-Fed [6] is an asynchronous FL framework that enables wait-free computation and communication. ASO-Fed allows client updates to stream into the federator in different rounds and each local training is based on the newly streamed-in data.

FedAsync [34] is an asynchronous optimization algorithm for FL that guarantees a near-linear convergence and address the staleness problem in the asynchronous setting. Each time an update arrives, the weighted averaging is applied for the current global model and the update.

AsyncFedED is an adaptive asynchronous FL aggregation based on Euclidean distance [31]. If the global model is updated before a client sends back its update, the learning rate of the server should be adjusted based on the staleness of the current update. AsyncFedED evaluates the staleness of an update by its Euclidean distance from the server. Larger staleness leads to a smaller learning rate for its aggregation. The staleness of weight is the relative weight differences.

FedAsync [34] introduces a staleness strategy where the influence of a client's update decreases when its staleness grows as the inverse function of α, the adaptive parameter. A larger α indicates the influence of stale updates should be reduced. FedAsync performs best when the staleness is small, and it can converge faster than FedAvg.

Overall, asynchronous FL approaches incur a high computational load on the server and therefore do not scale well with the number of clients. Intelligent node selection techniques [7, 14, 39] aim at alleviating this burden but are often biased towards specific clients. ASO-Fed and AsyncFedED have been evaluated in non-IID data settings with the Synthetic-1-1, FEMNIST, and Shakespeare datasets, which have inherent data heterogeneity. Both works do not consider scenarios that involve multiple servers.

2.2 Multi-Server FL

Existing multi-server methods [1, 3, 20, 22, 33] add edge servers or assign proxy clients to undertake part of the workload. Hierarchical FL approaches have also been described [1, 19, 22, 33]. Qu et al. demonstrated the convergence of a multi-server FL algorithm [25] where servers indirectly synchronize their models through overlapping client sets. Clustering is frequently used in multi-task and unsupervised FL [2, 13, 30]. Incorporating clustering algorithms in our framework for potential refinement is future work.

Xie et al. proposed a multi-center FL algorithm, FeSEM [35], to address the challenge of data heterogeneity. The main idea is to cluster the updates and assign them to their closest global model (i.e. center). They showed in experiments that the clusters of local updates characterize clients' data distribution thus models generated from similar data distributions are gathered on the same center and aggregated.

HierFAVG [22] is a client-edge-cloud hierarchical framework, which can be considered as a multi-server version of the FedAvg algorithm. Each edge server employs averaging aggregation for its clients. After every certain number of rounds, the cloud server applies averaging strategy to the edge servers. The introduction of the hierarchical structure reduces the communication burden on servers and improves the capability of the single-server FedAvg algorithm while maintaining the stability of synchronous systems.

Despite their ability to support a larger number of clients, current multi-server and hierarchical-server algorithms do not directly address the sensitivity to large system heterogeneity of synchronous systems.

2.3 Geo-distributed Machine Learning

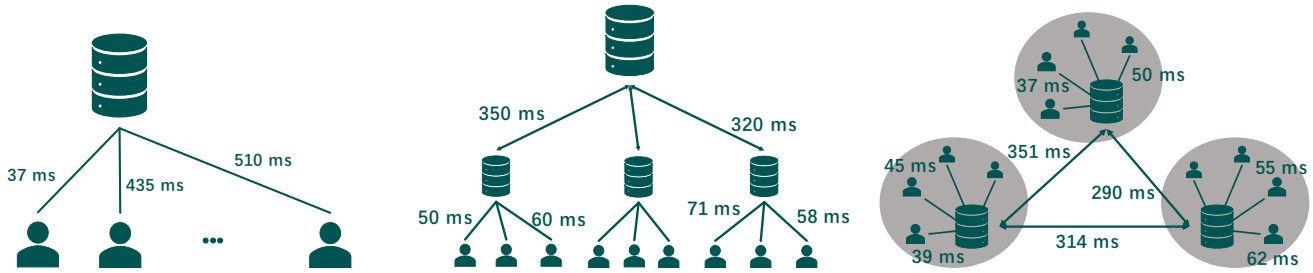
Many distributed ML systems target large-scale ML applications [8, 11, 12], however, they assume that network communication happens within a datacenter. Cano et al. [5] discussed running a machine learning system in geo-distributed settings. They show that one can leverage a communication-efficient algorithm for logistic regression models [23] to improve performance. Several works [16, 18, 38] focused on the design of communication-efficient mechanisms that leaves the ML algorithms unmodified. These works are orthogonal to our approach, since we design asynchronous multi-server FL algorithms for the geo-distributed settings.

3 BACKGROUND

This section provides necessary background on synchronous, asynchronous and multi-server Federated Learning. Fig. 2 illustrates the architectures these paradigms rely on, and Tab. 2 summarizes the notations we use throughout the paper.

Table 2: Notations and parameter values

Symbol		Description
D_k		Local dataset of client k
d_k		Number of data points in D_k
d		Total number of data points $\sum_k d_k$
n		Number of servers
n_C		Number of clients
W_k^t		Weight vector of a model at client k
η_k		Learning rate of client/server k
T_k		Number of local epochs on client k
A_k		Model age of client/server k
Symbol	Value	Description
η_{init}	0.5	Initial client learning rate
η_{min}	10^{-6}	Minimum client learning rate
β	0.05	Decaying rate
h_{inter}	$n_C/5n$	Age drift threshold between server models
h_{intra}	350	Age drift threshold since last global aggregation
ϕ	1.5	Activation rate
η_α	0.6	Aggregation rate (server side)
$s(\tau)$	$(\tau+1)^{-\alpha}$	Staleness parameter



(a) Centralized (b) Hierarchical Multi-Server (c) Flat Multi-Server (used in this work)

Figure 2: Architectures of FL systems

3.1 Synchronous Federated Learning

The first FL framework was proposed by McMahan et al. in 2017 [24]. It relies on a classical centralized architecture where a server interacts with all clients (Fig. 2a). The first FL algorithms, namely FedAvg and FedSgd, are synchronous. In each round, the server selects a group of clients and sends them the latest global model. Upon receiving a global model W^t, a client k trains it on its local dataset D_k to obtain W_k^{t+1} using Eq. 1, where F is the loss function of the classification, and η_k is the learning rate.

$$W_k^{t+1} = W_k^t - \eta_k \frac{\partial F(W^t, D_k)}{\partial W^t}, \tag{1}$$

Afterwards, client k sends its model update W_k^{t+1} to the server. Upon receiving all client updates it expects in a given round, the server aggregates them to compute the next global model W^{t+1} using Eq. 2. In this equation, d_k is the number of data points of client k and d is the total number of data points across all clients.

$$W^{t+1} = \sum_{k=1}^{K} \frac{d_k}{d} W_k^t. \tag{2}$$

The ratio $\frac{d_k}{d}$ weights the participation of a client during aggregation. At the beginning of the following round, the server sends W^{t+1} to the next selected clients. The training process is repeated over multiple rounds until the global model converges. In practical settings, where clients have different computing speeds and are connected to the server with heterogeneous network links, synchronous FL algorithms may lead to the server and some clients staying idle. For example, one client in Fig. 2a would communicate with the server within only 37 ms. This client and the server would stay idle waiting for the clients with 435 ms and 510 ms communication latency with the server to send their update.

3.2 Asynchronous Federated Learning

Asynchronous FL systems, such as FedAsync [34], accelerate convergence in heterogeneous networks and under client heterogeneity. To do so, the server aggregates an update into a global model immediately after it receives it, and returns the new global model to the client. For example, a server using the asynchronous version of FedAvg in classical centralized settings (Fig. 2a) would update the global model using Eq. 3. In this equation, W^t is the t-th global model, W_t^k is the local update client k computed on W^t, $\frac{d_k}{d}$ is the data proportion of the update, and $s(\tau)$ is a staleness parameter that dampens the effect of client k's update if it relates to a model that is older than the one the server currently has.

$$W^{t+1} = W^t - s(\tau) \frac{d_k}{d} \left(W_k^t - W_k^{t+1} \right). \tag{3}$$

An asynchronous FL scheme aims at keeping the clients busy training a model on their datasets, and the server updating its model more frequently, both of which are expected to speed up the model convergence. However, asynchronous schemes also increase the computational and communication loads on the server, which might become a performance bottleneck if the number of clients is too high.

3.3 Multi-server Federated Learning

Multi-server FL systems rely on multiple servers that interact with disjoint subsets of the clients to limit their computational and communication loads. Hierarchical FL systems (Fig. 2b), such as HierFAVG [1], are multi-server and rely on a principal server to maintain the global model [1, 19, 22, 33]. In those systems, a model is updated in two times in each round. The first aggregation is managed by the edge servers that aggregate model updates from a group of clients and send their own model update to the principal server. Upon receiving all model updates from the servers, the principal server executes the second-level aggregation and generates the global model for the new round. The global model is then distributed to all servers, and eventually to all clients. Previous hierarchical FL frameworks are synchronous. In geo-distributed settings, where the servers would be located far from each other, a round in a hierarchical FL framework involves several successive long distance communications and model aggregations, which would limit its speed.

4 OVERVIEW OF SPYKER

In this section, we start the presentation of Spyker by providing the key ideas behind its design.

Flat multi-server architecture. To get the best of the asynchronous and hierarchical FL paradigms, Spyker relies on multiple servers that directly interact with each other and with clients in a fully asynchronous manner. The servers are organized in a flat multi-server infrastructure (Fig. 2c). Each server is the center of a star sub-network, and interacts with a group of clients that are assigned to it based on geographical proximity to minimize communication latency. In this way, communication latency between

clients and servers are reduced, and servers are always able to process client or server updates. On the contrary, the related studies mainly consider that clients interact with a single server.

Asynchronous training and communications. Clients interact with their assigned server in the same way they would in a classical asynchronous FL system: they receive a global model from the server, train it over their local dataset and send their model update back to the server. Spyker's servers rely on asynchronous communications to exchange their models and benefit from the dataset of their respective clients. To minimize the complexity of this procedure, servers rely on a token-based strategy to trigger the asynchronous exchange of their models. Model aggregation also happens asynchronously and in a peer-to-peer fashion among servers. The model exchange can only be initiated by the server that holds the token, which prevents potential concurrent and redundant model broadcasts. At a given point in time, servers might maintain slightly different models.

Handling model staleness and fast clients. Since all interactions in Spyker are asynchronous, servers take into account model staleness at all levels to maintain accurate models. To do so, servers maintain the age of their model. A server increases the age of its model whenever it processes a client model update, or the model of another server. Spyker uses the age of models to dampen the contribution of an old model update when processing it. Similarly, Spyker uses a learning rate decay to limit the influence of clients that frequently produce updates.

Preventing server models from drifting apart. The frequency at which servers synchronize their models significantly influences the performance of the system in presence of data and resource heterogeneity. It is therefore important to ensure that a server model does not become biased toward the datasets of its clients, and that it leverages all client datasets. The model exchange and aggregation procedures allow servers to asynchronously homogenize their models. To prevent server models from drifting too much from each other during the training process, servers exchange and aggregate their models whenever they detect that their ages differ too much, and whenever a server has updated its model too many times since the last model exchange.

We assume that all servers and clients are honest, i.e., they do not launch data or model poisoning attacks [10, 32], or privacy attacks [29, 37], and never deviate from their protocol specification.

5 SPYKER DETAILS

In this section, we present the details of Spyker's building blocks. We first detail how servers and clients interact. We then discuss Spyker's aggregation algorithm that merges two server models and the token-based algorithm that triggers the asynchronous exchange of server models. Tab. 2 details our notations and the values of all parameters.

5.1 Local Training and Decaying Learning Rates

Alg. 1 describes the asynchronous interactions between a client and a server in Spyker, which are divided into two main procedures.

The first procedure, LocalTraining, is executed by a client C_k to train a model over its local dataset (Alg. 1, ll. 1-7), and is triggered upon receiving a model W_i^t, along with its age A_i and a learning rate η_k from a server. The client then trains the model over its dataset using the specified learning rate (Alg. 1, ll. 4-5), and sends the trained model back to the server along with the age A_i (Alg. 1, l. 7). Note that instead of having the client sends the model's age with its update, the server could remember the age of the model it has sent to every client. We adopted this option to simplify our pseudocode.

Algorithm 1 Interactions between clients and servers

η_i: server S_i's learning rate
η_k: client C_k's learning rate (given by S_i)

1: **[Client C_k] Procedure** LocalTraining(W_i^t, A_i, η_k)
2: ▷ *receive model W_i^t with age A_i from server S_i*
3: $W_k^t = W_i^t$
4: **for each** epoch $\in T_k$ **do**
5: update W_k with learning rate η_k
6: $W_k^{t+1} = W_k^t$
7: send $\left(W_k^{t+1}, A_i\right)$ to server S_i

8: **[Server S_i] Procedure** Aggregation($W_k^{t'}, A_k$)
9: ▷ *receive model $W_k^{t'}$ with age A_k=t' from client C_k*
10: $w_k^t = A_i - A_k$
11: $W_i^{t+1} = W_i^t + \eta_i \cdot w_k^t \cdot \left(W_k^{t'} - W_i^t\right)$
12: $A_i = A_i + 1$
13: $u[k] = u[k] + 1$ ▷ *num. updates received per client*
14: $\eta[k] = \text{Decay}(\eta[u[k]], u[k], \overline{u})$
15: send $\left(W_i^{t+1}, A_i, \eta[k]\right)$ to client C_k
16: checkSynchronization()

The second procedure, Aggregation, is executed by a server upon receiving a client update (Alg. 1, ll. 8-15). The server first determines the age difference between its current model and the model it sent to the client (Alg. 1, l. 10), and uses it to possibly decrease the impact of the received update on its model when updating it (Alg. 1, l. 11). The server then computes the learning rate η_k that the client should use for its next local training (Alg. 1, l. 14), before returning its new model W_i^{t+1} along with A_i and η_k to client C_k (Alg. 1, l. 15). Finally, the server verifies whether it should trigger a synchronization of server models (Alg. 1, l. 16) if its model age has sufficiently increased since the latest synchronization by calling function checkSynchronization() (defined at l. 17 in Alg. 2).

Let us provide more information on the way a server in Spyker updates the learning rate of a client (Alg. 1, l. 14) to maintain high model accuracy despite clients and network heterogeneity, i.e., despite the fact that some fast clients might consistently send updates at short intervals, while slow clients might contribute infrequently. Spyker adopts an adaptive strategy that tailors the impact of each client update. More precisely, a server uses function Decay (Alg. 1, l. 14) to decrease the impact of the fast clients that frequently generate updates on its model. The definition of this decay function relies on the observation that fast clients update more frequently and bias the model, and that therefore reducing the learning rate

of fast clients with the decay function reduces their impact on the aggregated model. To do so, the server maintains the number of updates it has received from each client in an array u, and therefore increments $u[k]$ (Alg. 1, l. 13) upon receiving a model from C_k. The server adjusts client C_k's learning rate using the following Decay function:

$$\text{Decay}(\eta[u[k]], u[k], \overline{u}) = \begin{cases} \eta_k & \text{if } u[k] < \overline{u} \\ \max\left(\eta_{min}, \eta\left[u[k]\right] - \beta(u[k] - \overline{u})\right) & \text{if } u[k] \geq \overline{u} \end{cases}$$

The Decay function takes three inputs: $u[k]$ is the number of updates that the server received from client C_k; $\eta[u[k]]$ is the learning rate a client would use without decay, which classically decreases with $u[k]$; and $\overline{u}$ is the average number of updates that clients have sent to server S_i. Decay also uses two parameters: η_{min} is a lower bound for learning rates; and β is the decaying rate. We empirically determined that $\beta = 0.05$ and $\eta_{min} = 10^{-6}$ provide the best results (cf. Tab. 2). The value of β controls the impact of fast clients: a small β benefits fast clients while a small β benefits slow clients.

5.2 Token-Triggered Server Model Aggregations

In addition to interacting with their clients, Spyker's servers synchronize their models using Alg. 2. Since the age of a server model impacts the way its client updates are aggregated, Spyker triggers an asynchronous server model synchronization whenever the ages of server models are drifting too much or when a server has processed a large enough number of client updates to make sure that all client datasets are fairly represented in server models.

To avoid the complexity of having to handle multiple concurrent synchronizations, servers rely on a token so that only one server can trigger the model broadcasts and aggregations. This token contains a synchronization ID *bid* that allows a server to broadcast their model only once per synchronization. Only the server holding the token can trigger a synchronization, but other servers might indirectly learn about an ongoing synchronization and broadcast their models to each other. A server broadcasts its model either because it holds the token and because some conditions are met (more details below), or by receiving another server's model with an unknown synchronization ID. To maintain fairness, the token is circulated among all servers. To accelerate synchronizations, whenever necessary, servers can broadcast their age so that the token holder triggers a synchronization. Note that we assume that links are FIFO, which can easily be enforced if it is not the case, so that a server receives the models of any other server according to increasing synchronization IDs and can process them all.

Upon executing the ServerInit initialization procedure (Alg. 2, l. 1), a server S_i initializes several variables to coordinate the exchange of models between servers. The token initially resides at a randomly chosen server (here server S_1), and contains a synchronization ID *bid* set to 1 and a vector of server model ages that are all initially equal to 0. The hashmap *cnt* is used to count the number of models that have been received for a given synchronization ID. The known age of server models is kept in $ages_i$. Variable *didBroadcast* contains the set of synchronization IDs for which S_i has broadcast its model. The Boolean variable *ongoingSynchro* is used by the server holding the token to initiate a synchronization only once before forwarding the token. The model age at which the latest synchronization happened, and the current model age are respectively stored in $A_{i,prev}$ and A_i. Finally, a server randomly initializes its model.

The token keeps visiting every server following a random ring-based topology. Upon arrival of the token (Alg. 2, ll. 11– 16), a server S_i updates the ages of the models that it maintains using the token's information. It then calls the CheckSynchronization procedure (Alg. 2, l. 17) to determine whether a model synchronization should be triggered. The frequency at which server model exchanges happen is controlled by the h_{inter} and h_{intra} parameters (cf. Tab. 2): h_{inter} is the threshold for the maximum age difference between different server models; and h_{intra} is the threshold for the maximum age difference between a server model's current age and the age it had during the last server model synchronization. If the inter-server age difference exceeds threshold h_{inter}, or if the intra-server age difference exceeds threshold h_{intra}, then server S_i might trigger a server model exchange. First, if S_i holds the token and has not already triggered a server model exchange, then it broadcasts its current model W_i^t of age A_i along with the current broadcast ID $t.bid$ to all the other servers. Second, if S_i does not hold the token, it can broadcast its model age to all other servers, so that the server that holds the token can execute procedure RcvAge (Alg. 2, l. 8), which updates the age of a server model and possibly triggers a model synchronization.

Procedure RcvModel (Alg. 2, ll. 27-38) is executed when a server receives a server model, and is also in charge of relaying the token. Upon receiving a server model from a server S_i, server S_j first updates its local age for server model i. It then verifies whether it has already broadcast is model for the synchronization ID indicated by server S_i using its local *didBroadcast* set, and if necessary broadcasts its model and updates the relevant variables. A server that receives a server model then executes the aggregation procedure ServerAgg (Alg. 2, l. 41), which is described in details in the next section. Finally, the server that holds the token makes sure that it receives the model of all other servers (using the *cnt* hashmap) before forwarding the token to its successor on the ring of servers.

5.3 Aggregation of Server Models

Following their exchange of models, servers aggregate them asynchronously immediately after they are received. Alg. 2 also details the procedures that a server S_i follows to aggregate a model that it receives from another server S_j. This algorithm first calculates the weight that should be used to update the local server model with the received one. The aggregation weight $w_{i,j}$ is computed based on the age of the two server models A_i and A_j using a sigmoid function that is parameterized using parameter ϕ (Alg. 2, ll. 43-44):

$$w_{i,j} = \frac{1}{1 + e^{-a}}, \quad \text{where} \quad a = \frac{\phi(A_j - A_i)}{A_i}$$

The age difference $A_j - A_i$ indicates whether W_j^t is more mature than W_i^t in terms of the number of updates both models have been trained on. The influence of W_j^t increases with this age difference. Denominator A_i makes the absolute weight relative to the current age of server S_i. As a model age increases, it becomes more stable

Algorithm 2 Token-triggered aggregation of server models.

1: **[Server S_i]** ServerInit()
2: $token = (i{\neq}1)?\ NULL : \{bid{=}1, ages{=}[0, \cdots, 0]\}$
3: $cnt = ages_i = \{\}$ ▷ *hashmaps*
4: $didBroadcast = \emptyset$
5: $ongoingSynchro$ = False
6: $A_{i,prev} = A_i = 0$
7: set W_i^0 to random model

8: **[Server S_i] Procedure** RcvAge(A_j)
9: $ages_i[j] = \max(ages_i[j], A_j)$
10: checkSynchronization()

11: **[Server S_i] Procedure** RcvToken(t)
12: **for** $j \in [1, N]$ **do**
13: $ages_i[j] = \max(ages_i[j], t.ages[j])$
14: $token = t$
15: $t.bid = t.bid + 1$
16: checkSynchronization()

17: **[Server S_i] Procedure** checkSynchronization()
18: **if** $\max(ages) - \min(ages) \geq h_{\text{inter}}$ **or** $A_i - A_{i,pre} \geq h_{\text{intra}}$ **then**
19: **if** *hasToken* **and** (**not** *ongoingSynchro*) **then**
20: $A_{i,pre} = A_i$
21: $ongoingSynchro$ = True
22: send $(W_i^t, A_i, t.bid)$ to all servers
23: $didBroadcast = didBroadcast \cup \{t.bid\}$
24: $cnt_i[token.bid] = 1$
25: **else**
26: send A_i to all servers
27: **[Server S_j] Procedure** RcvModel(W_i^t, A_i, bid_i)
28: $ages_j[i] = \max(ages_j[i], A_i)$
29: **if not** $bid_i \in didBroadcast$ **then**
30: $didBroadcast = didBroadcast \cup \{bid_i\}$
31: $A_{j,pre} = A_j$
32: send $\left(W_j^t, A_j, bid_i\right)$ to all servers
33: ServerAgg(W_i^t, A_i)
34: **if** $token \neq$ NULL **and** $t.bid = bid_i$ **then**
35: $cnt[bid_i] = cnt[bid_i] + 1$
36: **if** $cnt[bid_i] = n$ **then**
37: $t.ages = ages_j$
38: send token t to the next server on ring
39: $token$ = NULL
40: $ongoingSynchro$ = False

41: **[Server S_i] Procedure** ServerAgg(W_j^t, A_j)
42: process model W_j^t of age $A_j = t$ from server S_j
43: $a = \frac{\phi(A_j - A_i)}{A_i}$
44: $w_{i,j} = (1 + e^{-a})^{-1}$
45: $W_i^{t+1} = W_i^t + \eta_a \cdot w_{i,j} \left(W_j^t - W_i^t\right)$
46: $A_i = \left(1 - \eta_a \cdot w_{i,j}\right) A_i + \eta_a \cdot w_{i,j} \cdot A_j$

and the impact of other server models should be decreased. The sigmoid function ensures that $w_{i,j}$ remains between 0 and 1. The derivative of this function becomes 0 and results in a weight of 1 when the relative model age difference is too large. Parameter ϕ indicates in which range the sigmoid function is active. A larger ϕ leads to a smaller active range, leaving a larger area for the weight of 0 and 1.

The aggregation rate η_a also scales the influence of other servers during model aggregation. When the aggregation with server S_i happens locally at server S_j, a large η_a indicates that server S_i could greatly influence server S_j's model, and $\eta_a = 1$ means it could replace server S_j's model when their ages are equal. Parameter η_a needs to be carefully tuned. If η_a is too large, the influence of the server itself is eliminated; if η_a is too small, the server learns too little from its peers thus its model could be biased to its clients' data distribution. We empirically determined $\phi = 1.5$ and $\eta_\alpha = 0.6$ to provide the best results (cf. Tab. 2).

Server S_i finally updates its local model age after aggregating a server model using the weight $w_{i,j}$ that has been applied to the model aggregation (Alg. 2, l. 46). As the aggregation of a server model embeds more than one client update, we apply a weighting strategy to increase a model's age after aggregating another server's model.

6 PERFORMANCE EVALUATION

In this section, we describe our experimental settings, and compare Spyker to state-of-the-art representative FL algorithms. We show the superior accuracy of Spyker over previous baselines, demonstrate its improved scalability with the number of servers, and report the positive impact of our learning rate decay function on accuracy. We also report the network consumption of all algorithms.

6.1 Settings

Baselines. We compare Spyker with three recent Federated Learning algorithms: FedAvg [24], FedAsync [34] and HierFedAvg [22]. FedAvg is the original synchronous FL framework, while FedAsync and HierFedAvg are respectively state-of-the-art asynchronous and hierarchical FL frameworks. We also implemented a synchronous version of Spyker, which we call Sync-Spyker, that uses a synchronous server-server model aggregation procedure. Sync-Spyker periodically triggers synchronous model exchanges between servers (i.e., ignoring the influence of h_{inter} to trigger model exchanges). Periodically, the servers broadcast their model, along with their age, and wait for all other server models. Upon receiving all models, a server aggregates them following a deterministic order (i.e., using the server IDs). Following this exchange and aggregation of server models, all servers own the same model. In Sync-Spyker, when the server synchronization algorithm is started, servers stop processing

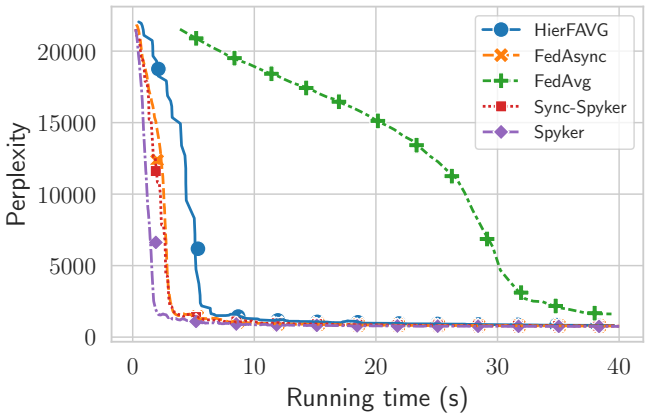


Figure 3: WikiText2: Perplexity wrt. time. (lower is better)

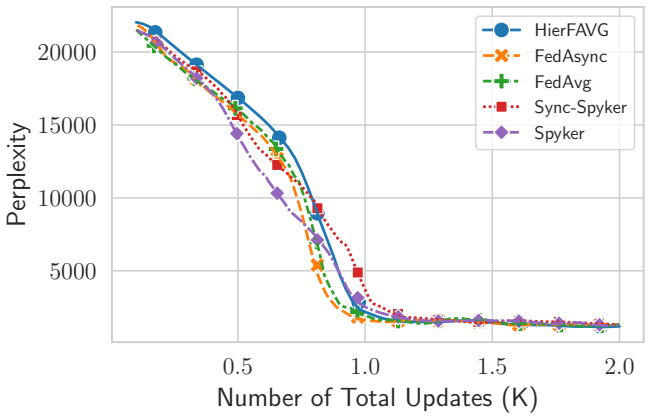


Figure 4: WikiText2: Perplexity wrt. # updates (lower is better)

local updates from clients, and instead store them. Client updates are then processed when all the server models have arrived and have been aggregated.

Datasets. We conduct experiments on the MNIST and CIFAR-10 image collections, and on the WikiText2 language modeling dataset. For a given experiment, a dataset is split into subsets of equal sizes that are assigned to different clients. To introduce client data heterogeneity, we assign l labels to each client. A smaller l indicates a higher degree of non-independent and identically distributed (IID) data distribution and thus larger heterogeneity among clients [9]. We set l to 2 in non-IID data experiments.

Computation and Network Delays. Our experiments take into account the computations that clients execute. Time is maintained at each client, and is advanced depending on the procedure it executes during the experiments. Tab. 3 details the average computing times of the server aggregation procedure of all FL frameworks. These values were obtained by benchmarking all algorithms using the Python *time* package. To simulate client heterogeneity, we sample each client's training delay from a Gaussian distribution $N(\mu, \sigma^2)$, where $\mu = 150$ and $\sigma = 7.5$. We use the same training delays per client across all experiments. Since servers might be located in different regions of the world (e.g., Amsterdam and Sydney), we set the communication delays among servers according to the Amazon Web Service network latency [28] as shown in Tab. 4. We assume that a client and its nearest server are in the same geographical location and therefore set their communication delay using the diagonal of Tab. 4. We assume that all network links have a 100 Mbps bandwidth. For each experiment, we detail the number of servers and clients that we use along with the parameter values.

Model. For MNIST, we use a CNN model with two convolutional and two fully connected layers. For CIFAR-10, we use a CNN model with three convolutional and two fully connected layers. For WikiText2, we use the next character Long Short-Term Memory (LSTM) neural network model, which has been designed for character-level text generation tasks. The language model consists of an embedding layer that is used to capture the semantic and syntactic properties of characters, an LSTM layer that captures dependencies between characters and generates coherent text, and a fully connected layer

Table 3: Computation time required per procedure (ms).

Local Training	200
Model Aggregation in Sync-Spyker	2
Model Aggregation in Spyker	2
Model Aggregation in FedAVg	15
Model Aggregation in HierFAVG	15
Model Aggregation in FedAsync	2

Table 4: Communication delays between geographical locations (ms).

	Hongkong	Paris	Sydney	California
Hongkong	1.41	194.9	132.28	155.13
Paris	197.91	0.9	278.83	142.25
Sydney	132.06	280.11	2.56	138.47
California	154.96	142.79	138.57	2.14

that transforms the LSTM's hidden states into a probability distribution over all possible characters in the vocabulary. The initial local learning rate of clients η_k is 0.05. In asynchronous settings, we use $\alpha = 0.5$ for the staleness weighting in FedAsync and a global learning rate of $\eta = 0.6$ for the client-server update.

6.2 Accuracy

We now use 100 clients that are equally distributed across 4 servers located in an AWS region. Fig. 3, Fig. 5 and Fig. 7 show that Spyker converges faster than the baselines in terms of elapsed time for all three datasets. Spyker and Sync-Spyker converge as quickly as the baselines in terms of number of processed client updates.

We also evaluated the ability of the 5 FL frameworks to scale with the number of clients. We evaluated the time each algorithm needs to reach 90% accuracy with MNIST and 4 servers using 100, 200 and 300 clients. Tab. 5 reports this time for all algorithms for 200 and 300 clients divided by the time measured with 100 clients. When the number of clients in the system increases, the convergence time and required number of updates of the various methods increase

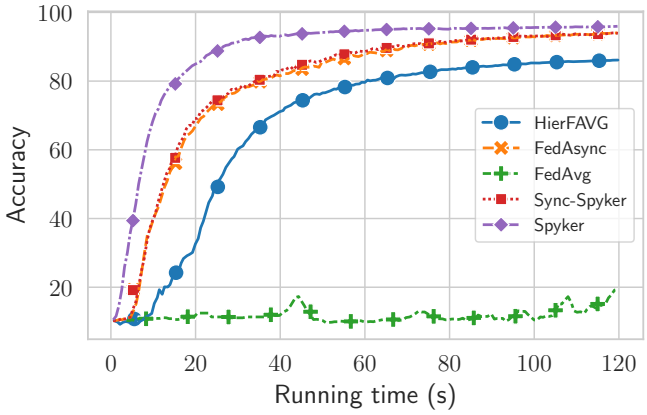


Figure 5: MNIST: Accuracy wrt. time (higher is better)

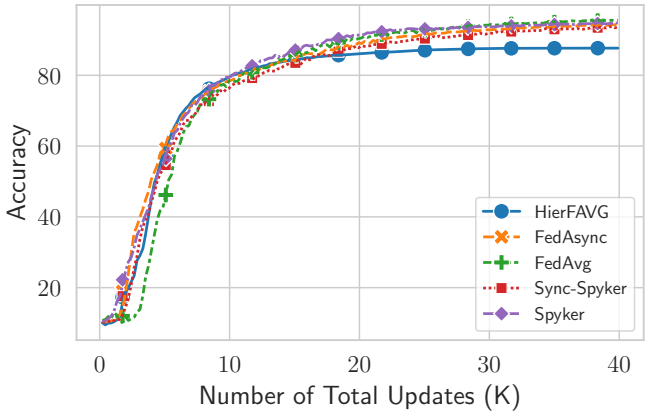


Figure 6: MNIST: Accuracy wrt. # updates (higher is better)

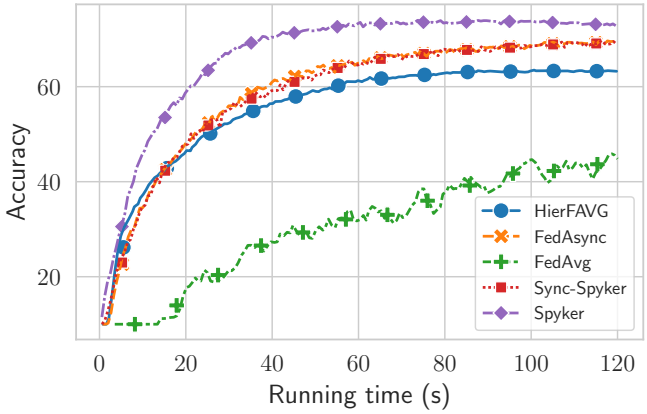


Figure 7: CIFAR-10: Accuracy wrt. time (higher is better)

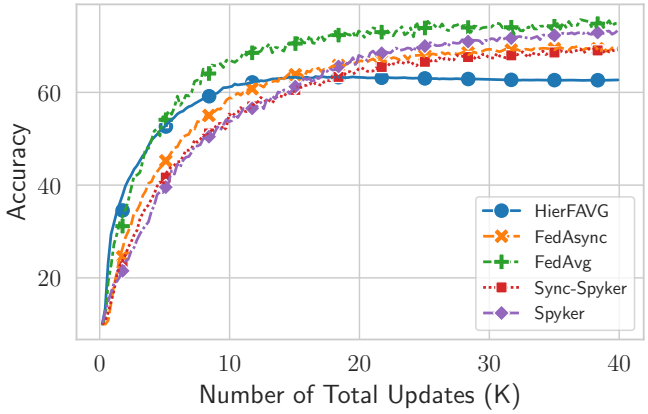


Figure 8: CIFAR-10: Accuracy wrt. # updates (higher is better)

Table 5: Multiplicative factor for the amount of time and number of updates necessary for algorithms to reach a 90% accuracy with 200 and 300 clients compared with their own execution with 100 clients. A low value indicates that an algorithm scales well with the number of clients.

	200 clients		300 clients	
	Time	Updates	Time	Updates
FedAsync	2.42	3.63	5.99	9.01
FedAvg	1.98	4.05	2.44	7.26
HierFAVG	1.64	3.18	1.75	8.19
Spyker	**1.21**	**2.43**	**1.43**	**4.34**
Sync-Spyker	1.61	2.68	1.90	4.87

following different trends. Spyker is the method that scales the best. For example, it requires 21% more time to converge with 200 clients than with 100 clients while other baselines (excluding Sync-Spyker) require at least 64% more time. It also requires 2.43 times as many updates to converge while other baselines require at least 3.18 times as many. On the other side of the spectrum, FedAsync's required time and number of updates to converge increase the most. With 200 clients, its convergence time is multiplied by 2.42, while its required number of updates is multiplied by 3.63 Spyker not only exhibits the fastest convergence, but it also scales the best with the number of clients.

Fig. 4, Fig. 6 and Fig. 8 show the perplexity of all the baselines, Spyker and Sync-Spyker on WikiText-2, and their accuracy on MNIST and CIFAR-10 depending on the number of client updates processed. One can see that Spyker and Sync-Spyker do not always require the least number of updates to reach a given perplexity or accuracy. This is not a problem in practice as it makes more sense to evaluate the evolution of perplexity or accuracy based on elapsed time during a deployment. However, these figures also show that Spyker reaches similar perplexity or accuracy levels as FedAvg, which generally converges the fastest based on the number of updates (but not based on elapsed time, where it is the worst performing baseline).

6.3 Impact of Multiple Servers in Asynchronous FL

In asynchronous FL systems, the server aggregates an update immediately after receiving it. However, a busy server can cause a queueing of updates that await being processed. To further illustrate

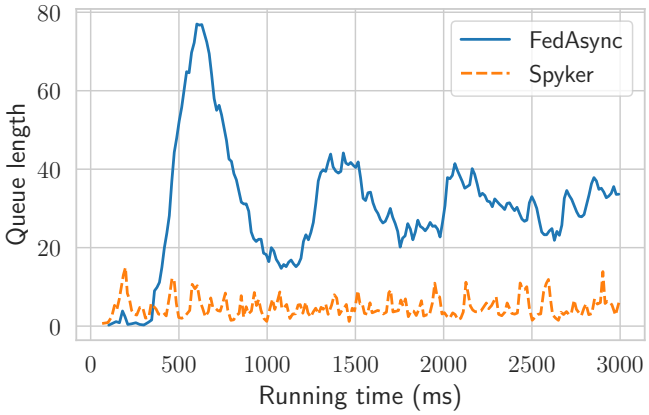


Figure 9: Number of updates queued at a server with Spyker and FedAsync

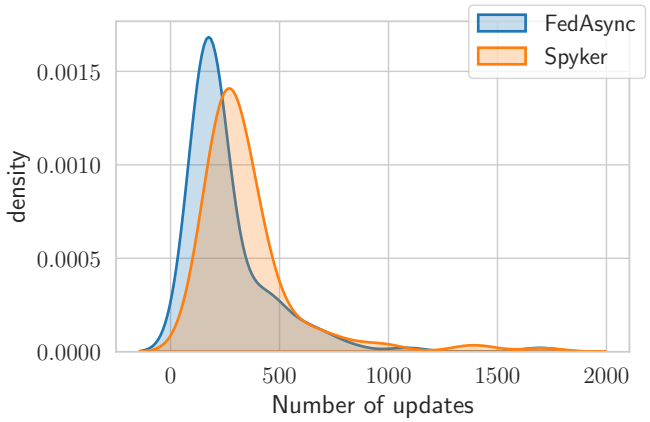


Figure 10: Density distribution of the number of updates per client with Spyker and FedAsync

the benefits of introducing a multi-server system to an asynchronous FL framework, we evaluate the update queuing phenomenon with Spyker (4 servers) and FedAsync (1 server) with 200 clients. The local training delays of clients are generated using a Gaussian distribution with a mean of 150 ms and a standard deviation of 60 ms. Processing delays are set using Tab. 4.

Fig. 9 shows that FedAsync's queue length increases to nearly 80 updates from 300 ms to 600 ms and always stays above 20 during the rest of the experiment. Client updates are processed more efficiently by the 4 servers that Spyker uses, as server queues never contain more than 20 updates.

We further evaluated the time needed for FedAsync and Spyker to reach 90% and 95% accuracy on the MNIST dataset. For these experiments, the time difference observed is only caused by resource heterogeneity as we set all network latency's to the same value. The results are shown in the bottom three rows of Tab. 6. Spyker reaches a 90% and a 95% accuracy respectively 38% and 25% faster than FedAsync.

Table 6: Time to reach 90% or 95% accuracy with FedAsync and Spyker. *Lat* uses AWS latency [28], and *No lat* assumes uniform latency between servers (same mean in both cases).

Network	Method	Time 90%	Time 95%
Lat.	FedAsync	59*s*	125*s*
	Spyker	22*s*	51*s*
	Improvement	**-61%**	**-58%**
No lat.	FedAsync	40*s*	75*s*
	Spyker	25*s*	56*s*
	Improvement	**-38%**	**-25%**

To demonstrate that Spyker mitigates the impact of high communication delays, we conducted a group of experiments with the latencies of Tab. 4. The results are shown in the top three rows of Tab. 6. Compared with the bottom three rows, the top three rows show a larger difference between Spyker and FedAsync. Spyker appears 61% faster to reach 90% accuracy, and 58% faster to reach 95% accuracy than FedAsync. This difference can be explained by the fact that Spyker decreases the distance between servers and clients compared to FedAsync, which uses a single server.

6.4 Number of Clients per Server

We evaluate the effect of imbalanced number of clients allocated to each server. We consider four different scenarios. The first one consists of 4 servers with each 25 clients. In the three other scenarios, one of the servers has more clients (i.e., it has 52, 63, or 70 clients) while the remaining clients are divided evenly over the remaining servers. Table 7 show the accuracy of the global model and the time necessary for it to converge. These results show that slight imbalance in the client distribution degrades accuracy the most (-14.5% with the second most balanced scenario) and that increasing imbalance always increases the convergence time. In comparison, we observed that HierFAVG's performance is less impacted by this kind of imbalance, but its performance nonetheless is always worse than Spyker's.

Table 7: Effect of imbalanced number of clients per server on accuracy.

	Spyker			
	25 clients	52 clients	63 clients	70 clients
Accuracy	96.4%	−14.5%	−11.9	−7.63%
Duration (s)	120	+17.76	+33.04	+53.08

6.5 Impact of Learning Rate Decay

We ran two experiments to demonstrate the positive impact of learning rate decay. We first used Spyker and FedAvg each with 100 clients to perform the classification task on the MNIST dataset. We gathered statistics on the number of updates of every client and plotted a Kernel Density Estimate (KDE) plot to show its distribution. The most desirable KDE plot for a FL system should exhibit a single concentrated peak, which would mean that clients contribute equally. Fig. 10 shows the KDE plots of Spyker and FedAsync.

FedAsync has a more concentrated distribution of updates with a steep peak at around 200 updates compared to a gentler peak for

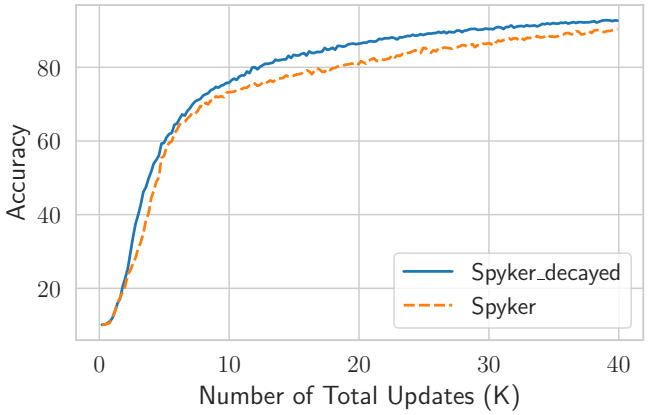


Figure 11: Accuracy with and without a learning rate decay on MNIST with 4 servers and 100 clients (25 per server)

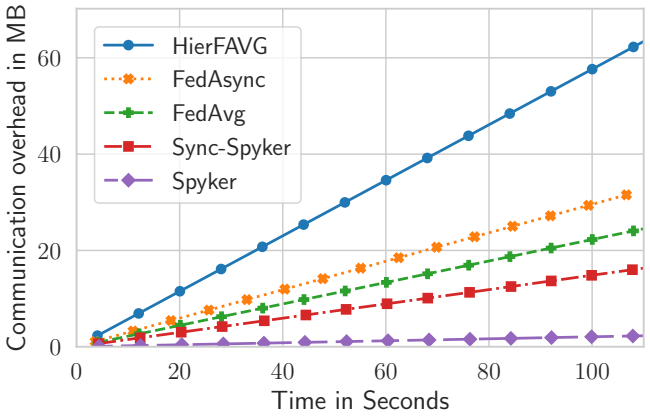


Figure 12: Overall network consumption of Spyker compared to the baselines

Spyker at around 300 updates. This difference is explained by the fact that FedAsync uses a single server, whose latency distribution is wider than the ones Spyker's servers have with their local clients. On the other hand, by introducing multiple servers, the range of possible communication delays with Spyker becomes smaller and the differences between client training times becomes the major source of heterogeneity. We also observe a peak at around 1,400 updates for Spyker and 1,700 for FedAsync, which comes from the presence of fast clients near the server, which could cause a biased global model due to their higher influence on the server's model if left unaddressed. This analysis justifies the need for a client learning rate decay mechanism that customizes the learning rate of clients according to the number of updates they contribute. The impact of the updates that the most active clients generate is therefore dampened to balance the overall contribution of clients. Interestingly however, the decay function would use a smaller decay ratio if the system as a whole would catch up with the fast clients.

In the second experiment, we evaluated Spyker's convergence speed with and without our learning rate decay mechanism. Fig. 11 shows that introducing the learning rate decay mechanism increases Spyker's convergence speed, and therefore that it reduces the impact of client data heterogeneity.

6.6 Bandwidth consumption

We evaluated the bandwidth consumption of all FL algorithms over time. We measured the number of bytes transferred over 110 s during server-server and server-client model communications. Fig. 12 details the network consumption of all algorithms with the MNIST dataset, 4 servers and 100 clients equally distributed over all servers. FedAvg has the lowest bandwidth consumption with 2.28 MB consumed, which can be explained by its synchronous communications and its use of a single server. Spyker has the highest bandwidth requirements with 63.4 MB transferred because of its asynchronous server-server and server-client communications. This network consumption is however very reasonable for modern networks. For each client update, the aggregated global model is relayed back to the client, which results in higher bandwidth requirements. Similarly, Sync-Spyker is the second most network intensive algorithm and transmits 32.5 MB over the network since it maintains asynchronous client-server interactions. Unlike Spyker's asynchronous multi-server aggregation, Sync-Spyker's synchronous multi-server aggregation halts the client updates, which results in lower communication overhead. FedAsync's asynchronous client updates incur 24.5 MB of data transmissions over the network, which is higher than with FedAVG. HierFAVG's synchronous client-server communication results in lower communication overhead of 16.35 MB but the hierarchical server aggregation requires relatively higher bandwidth.

7 CONCLUSION

We described Spyker, the first fully asynchronous multi-server FL algorithm. Spyker focuses on practical scenarios where clients are distributed over the world, and it addresses the performance bottlenecks that naturally appear in presence of weak clients or heterogeneous networks. Spyker scales better with the number of clients than previous works. Spyker relies on an aging mechanisms that servers use to support asynchrony of all exchanges and on a token-based algorithm that servers rely on to synchronize their model exchanges. Our experimental results show that Spyker requires less time to converge than three representative algorithms – FedAsync, HierFAVG, and FedAvg, and is also faster than its partially synchronous variant Sync-Spyker where servers synchronize themselves using a synchronous algorithm. Spyker also leverages a learning rate decay method to maintain high accuracy despite client heterogeneity. Future work includes exploring the possibility of integrating clustering algorithms in Spyker to enable servers to group clients based on possible similarities in their data distributions.

ACKNOWLEDGMENTS

This work was partially supported by the EU Horizon Europe Research and Innovation Programme under Grant No. 101073920 (TENSOR).

REFERENCES

[1] Mehdi Salehi Heydar Abad, Emre Ozfatura, Deniz Gündüz, and Özgür Erçetin. 2020. Hierarchical Federated Learning ACROSS Heterogeneous Cellular Networks. In *ICASSP*. 8866–8870. https://doi.org/10.1109/ICASSP40776.2020.9054634

[2] Andreas Adolfsson, Margareta Ackerman, and Naomi C. Brownstein. 2019. To cluster, or not to cluster: An analysis of clusterability methods. *Pattern Recognit.* 88 (2019), 13–26. https://doi.org/10.1016/J.PATCOG.2018.10.026

[3] Seyed Mohammad Azimi-Abarghouyi and Viktoria Fodor. 2022. Multi-Server Over-the-Air Federated Learning. *CoRR* abs/2211.16162 (2022). https://doi.org/10.48550/ARXIV.2211.16162 arXiv:2211.16162

[4] Theodora S. Brisimi, Ruidi Chen, Theofanie Mela, Alex Olshevsky, Ioannis Ch. Paschalidis, and Wei Shi. 2018. Federated learning of predictive models from federated Electronic Health Records. *Int. J. Medical Informatics* 112 (2018), 59–67. https://doi.org/10.1016/J.IJMEDINF.2018.01.007

[5] Ignacio Cano, Markus Weimer, Dhruv Mahajan, Carlo Curino, Giovanni Matteo Fumarola, and Arvind Krishnamurthy. 2017. Towards Geo-Distributed Machine Learning. *Data Engineering Bulletin* 40, 4 (2017), 41–59. http://sites.computer.org/debull/A17dec/p41.pdf

[6] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. 2020. Asynchronous Online Federated Learning for Edge Devices with Non-IID Data. In *BigData*. 15–24. https://doi.org/10.1109/BIGDATA50022.2020.9378161

[7] Zheyi Chen, Weixian Liao, Kun Hua, Chao Lu, and Wei Yu. 2021. Towards asynchronous federated learning for heterogeneous edge-powered internet of things. *Digit. Commun. Networks* 7, 3 (2021), 317–326. https://doi.org/10.1016/J.DCAN.2021.04.001

[8] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*. 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[9] Bart Cox, Lydia Y. Chen, and Jérémie Decouchant. 2022. Aergia: leveraging heterogeneity in federated learning systems. In *Middleware*. 107–120. https://doi.org/10.1145/3528535.3565238

[10] Bart Cox, Abele Mălan, Lydia Y. Chen, and Jérémie Decouchant. 2024. Asynchronous Byzantine Federated Learning. arXiv:2406.01438 [cs.LG] https://arxiv.org/abs/2406.01438

[11] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *EuroSys*. 4:1–4:16. https://doi.org/10.1145/2901318.2901323

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. (2012), 1232–1240. https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html

[13] Don Kurian Dennis, Tian Li, and Virginia Smith. 2021. Heterogeneity for the Win: One-Shot Federated Clustering. In *ICML (Proceedings of Machine Learning Research, Vol. 139)*. 2611–2620. http://proceedings.mlr.press/v139/dennis21a.html

[14] Jiangshan Hao, Yanchao Zhao, and Jiale Zhang. 2020. Time Efficient Federated Learning with Semi-asynchronous Communication. In *ICPADS*. 156–163. https://doi.org/10.1109/ICPADS51040.2020.00030

[15] Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated Learning for Mobile Keyboard Prediction. *CoRR* abs/1811.03604 (2018). arXiv:1811.03604 http://arxiv.org/abs/1811.03604

[16] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *NSDI*. 629–647. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

[17] Zhiqi Huang, Fenglin Liu, and Yuexian Zou. 2020. Federated Learning for Spoken Language Understanding. In *COLING*. 3467–3478. https://doi.org/10.18653/V1/2020.COLING-MAIN.310

[18] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I. Jordan. 2014. Communication-Efficient Distributed Dual Coordinate Ascent. (2014), 3068–3076. https://proceedings.neurips.cc/paper/2014/hash/894b77f805bd94d292574c38c5d628d5-Abstract.html

[19] Li Li, Haoyi Xiong, Zhishan Guo, Jun Wang, and Cheng-Zhong Xu. 2019. SmartPC: Hierarchical Pace Control in Real-Time Federated Learning System. In *RTSS*. 406–418. https://doi.org/10.1109/RTSS46320.2019.00043

[20] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated Optimization in Heterogeneous Networks. In *MLSys*. https://proceedings.mlsys.org/paper_files/paper/2020/hash/1f5fe83998a09396ebe6477d9475ba0c-Abstract.html

[21] Xiaoxiao Li, Yufeng Gu, Nicha C. Dvornek, Lawrence H. Staib, Pamela Ventola, and James S. Duncan. 2020. Multi-site fMRI analysis using privacy-preserving federated learning and domain adaptation: ABIDE results. *Medical Image Anal.* 65 (2020), 101765. https://doi.org/10.1016/J.MEDIA.2020.101765

[22] Lumin Liu, Jun Zhang, Shenghui Song, and Khaled B. Letaief. 2020. Client-Edge-Cloud Hierarchical Federated Learning. In *ICC*. 1–6. https://doi.org/10.1109/ICC40277.2020.9148862

[23] Dhruv Mahajan, S. Sathiya Keerthi, S. Sundararajan, and Léon Bottou. 2013. A Functional Approximation Based Distributed Learning Algorithm. *CoRR* abs/1310.8418 (2013). arXiv:1310.8418 http://arxiv.org/abs/1310.8418

[24] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS (Proceedings of Machine Learning Research, Vol. 54)*. 1273–1282. http://proceedings.mlr.press/v54/mcmahan17a.html

[25] Zhe Qu, Xingyu Li, Jie Xu, Bo Tang, Zhuo Lu, and Yao Liu. 2023. On the Convergence of Multi-Server Federated Learning With Overlapping Area. *IEEE Transactions on Mobile Computing* 22, 11 (2023), 6647–6662. https://doi.org/10.1109/TMC.2022.3200016

[26] Mohamed Rashad, Zilong Zhao, Jeremie Decouchant, and Lydia Y. Chen. 2024. TabVFL: Improving Latent Representation in Vertical Federated Learning. *CoRR* (2024). arXiv:2404.17990 [cs.LG] https://arxiv.org/abs/2404.17990

[27] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletarì, Holger R. Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N. Galtier, Bennett A. Landman, Klaus H. Maier-Hein, Sébastien Ourselin, Micah J. Sheller, Ronald M. Summers, Andrew Trask, Daguang Xu, Maximilian Baust, and M. Jorge Cardoso. 2020. The future of digital health with federated learning. *npj Digit. Medicine* 3 (2020). https://doi.org/10.1038/S41746-020-00323-1

[28] Amazon Web Services. 2024. AWS Latency Monitoring. https://www.cloudping.co/grid

[29] Aditya Shankar, Lydia Y. Chen, Jérémie Decouchant, Dimitra Gkorou, and Rihan Hai. 2024. Share Your Secrets for Privacy! Confidential Forecasting with Vertical Federated Learning. arXiv:2405.20761 [cs.LG] https://arxiv.org/abs/2405.20761

[30] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet Talwalkar. 2017. Federated Multi-Task Learning. In *NeurIPS*. 4424–4434. https://proceedings.neurips.cc/paper/2017/hash/6211080fa89981f66b1a0c9d55c61d0f-Abstract.html

[31] Qiyuan Wang, Qianqian Yang, Shibo He, Zhiguo Shi, and Jiming Chen. 2022. AsyncFedED: Asynchronous Federated Learning with Euclidean Distance based Adaptive Weight Aggregation. *CoRR* abs/2205.13797 (2022). https://doi.org/10.48550/ARXIV.2205.13797 arXiv:2205.13797

[32] Rui Wang, Xingkai Wang, Huanhuan Chen, Jérémie Decouchant, Stjepan Picek, Nikolaos Laoutaris, and Kaitai Liang. 2023. MUDGUARD: Taming Malicious Majorities in Federated Learning using Privacy-Preserving Byzantine-Robust Clustering. arXiv:2208.10161 [cs.CR] https://arxiv.org/abs/2208.10161

[33] Xing Wang and Yijun Wang. 2022. Asynchronous Hierarchical Federated Learning. *CoRR* abs/2206.00054 (2022). https://doi.org/10.48550/ARXIV.2206.00054 arXiv:2206.00054

[34] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2019. Asynchronous Federated Optimization. *CoRR* abs/1903.03934 (2019). arXiv:1903.03934 http://arxiv.org/abs/1903.03934

[35] Ming Xie, Guodong Long, Tao Shen, Tianyi Zhou, Xianzhi Wang, and Jing Jiang. 2020. Multi-Center Federated Learning. *CoRR* abs/2005.01026 (2020). arXiv:2005.01026 https://arxiv.org/abs/2005.01026

[36] Jie Xu, Benjamin S. Glicksberg, Chang Su, Peter B. Walker, Jiang Bian, and Fei Wang. 2021. Federated Learning for Healthcare Informatics. *J. Heal. Informatics Res.* 5, 1 (2021), 1–19. https://doi.org/10.1007/S41666-020-00082-4

[37] Jin Xu, Chi Hong, Jiyue Huang, Lydia Y. Chen, and Jérémie Decouchant. 2022. AGIC: Approximate Gradient Inversion Attack on Federated Learning. In *41st International Symposium on Reliable Distributed Systems, SRDS 2022, Vienna, Austria, September 19-22, 2022*. IEEE, 12–22. https://doi.org/10.1109/SRDS55811.2022.00012

[38] Yuchen Zhang and Xiao Lin. 2015. DiSCO: Distributed Optimization for Self-Concordant Empirical Loss. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 37)*. 362–370. http://proceedings.mlr.press/v37/zhangb15.html

[39] Chendi Zhou, Hao Tian, Hong Zhang, Jin Zhang, Mianxiong Dong, and Juncheng Jia. 2021. TEA-fed: time-efficient asynchronous federated learning for edge computing. In *CF*. 30–37. https://doi.org/10.1145/3457388.3458655

[40] Jie Zhu, Shenggui Li, and Yang You. 2022. Sky Computing: Accelerating Geo-distributed Computing in Federated Learning. *CoRR* abs/2202.11836 (2022). arXiv:2202.11836 https://arxiv.org/abs/2202.11836