提出面向mobile的FL服务，可提供统一API供第三方软件调用，并能提供跨APP模型训练。无模型性能试验数据，仅测试了该服务的硬件消耗情况。

FLaaS: Federated Learning as a Service

Nicolas Kourtellis
Telefonica Research
Barcelona, Spain
nicolas.kourtellis@telefonica.com

Kleomenis Katevas
Telefonica Research
Barcelona, Spain
kleomenis.katevas@telefonica.com

Diego Perino
Telefonica Research
Barcelona, Spain
diego.perino@telefonica.com

ABSTRACT

Federated Learning (*FL*) is emerging as a promising technology to build machine learning models in a decentralized, privacy-preserving fashion. Indeed, *FL* enables local training on user devices, avoiding user data to be transferred to centralized servers, and can be enhanced with differential privacy mechanisms. Although *FL* has been recently deployed in real systems, the possibility of collaborative modeling across different 3rd-party applications has not yet been explored. In this paper, we tackle this problem and present Federated Learning as a Service (FLaaS), a system enabling different scenarios of 3rd-party application collaborative model building and addressing the consequent challenges of permission and privacy management, usability, and hierarchical model training. FLaaS can be deployed in different operational environments. As a proof of concept, we implement it on a mobile phone setting and discuss practical implications of results on simulated and real devices with respect to on-device training CPU cost, memory footprint and power consumed per *FL* model round. Therefore, we demonstrate FLaaS's feasibility in building unique or joint *FL* models across applications for image object detection in a few hours, across 100 devices.

ACM Reference Format:
Nicolas Kourtellis, Kleomenis Katevas, and Diego Perino. 2020. FLaaS: Federated Learning as a Service. In *1st Workshop on Distributed Machine Learning (DistributedML'20), Dec. 1, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3426745.3431337

1 INTRODUCTION

Machine Learning as a Service (MLaaS) has been on the rise in the last years due to increased collection and processing of big data from different companies, availability of public APIs, novel advanced machine learning (ML) methods, open-sourced libraries, tools for large-scale ML analytics and cloud-based computation. Such MLaaS systems have been predominantly centralized: all data of users/clients/devices need to be uploaded to the cloud service provider (e.g., Amazon Web Services [1], Google Cloud [14] or Microsoft Azure [21]), before ML data modeling can be done.

Federated Learning (*FL*) [17] is a natural evolution of centralized ML methods, as it allows companies employing *FL* to build ML models in a decentralized fashion close to users' data, without the need to collect and process them centrally. In fact, *FL* has been extended and applied in different settings, and recently deployed in real systems (e.g., Google Keyboard [5]). Also, privacy guarantees can be provided by applying methods such as Differential Privacy (DP) [13], or by computing models within a P2P network [4, 16]. In addition, recent start-up efforts in *FL* space (e.g., [9, 11, 22]) aim to provide *FL* support and tools to 3rd-party customers or end-users. However, these solutions have two shortcomings. First, they do not allow independent 3rd-parties to collaborate and build common ML models while protecting users' privacy. This feature would enable different applications to collaborate to build better models thanks to larger and richer datasets, while preserving users' privacy. Second, they do not provide an "as a service model" like existing MLaaS platforms. This feature is critical to enable developers and data scientists to deploy quickly and easily *FL* solutions, also fostering large adoption of the *FL* paradigm.

To enable these two features, there are a few fundamental challenges in *FL* space to be addressed first:

- *How do we enable collaborative modeling across different 3rd-party applications, to solve existing or new ML problems?*
- *How do we perform effective permission and privacy management of data and models shared across collaborating parties?*
- *How do we take advantage of topological properties of communication networks, for better FL modeling convergence, without compromising user and data privacy?*
- *How do we provide collaborative FL models in an "as a Service" fashion?*

In this paper, we propose the *Federated Learning as a Service* (FLaaS), to address such challenges and facilitate a wave of new applications and services based on *FL*. In particular, FLaaS makes the following contributions in *FL* space:

(1) provides high-level and extensible APIs, and an SDK for service usage and privacy/permissions management;
(2) enables the collaborative training of ML models across its customers on the same device using said APIs, in a federated, secured, and privacy-preserving fashion;
(3) enables the hierarchical construction and exchange of ML models across the network;
(4) can be instantiated in different types of devices and operational environments: mobile phones, home devices, edge nodes, etc. (cf. Figure 1);
(5) provides the first, to our knowledge, experimental investigation of on-device training costs of FL modeling on actual mobile devices.

Independently from the operational environment, in Section 3, we present different ways that FLaaS supports the building of collaborative models across 3rd-party applications in its *FL* environment, along with the challenges FLaaS must address. Then, we detail the FLaaS system design, APIs and software libraries and how FLaaS

DistributedML'20, Dec. 1, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-8182-6/20/12…$15.00
https://doi.org/10.1145/3426745.3431337

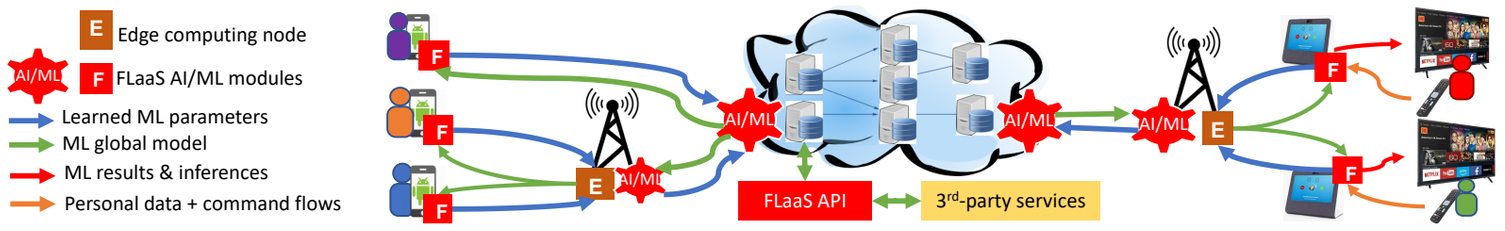


Figure 1: Examples of use cases to be supported by FLaaS. Applications can request FL-modeling support while executed on mobile devices (left-side), or IoT and home devices like personal assistants and smart TVs (right-side). Also, FLaaS can employ edge computing nodes to improve ML convergence, but not compromising users' privacy.

supports collaborative modeling, in Section 4. As a proof of concept, in Section 5, we deploy FLaaS on a mobile phone setting to assess the practical overheads of running such a service on mobiles. We demonstrate FLaaS capabilities in building unique or joint *FL* models for image object detection, for independent or collaborative mobile apps using shared data, respectively. We measure data sharing, ML training and evaluation costs of FLaaS with respect to CPU utilization, memory, execution time and power consumption, for on-device *FL* modeling in the above scenarios. We show that FLaaS can build *FL* models on mobile phones over 10s of rounds in a few hours, across 100 devices, with 10-20% CPU utilization, 10s of MBs average memory footprint and 3-5% battery consumption, per *FL* round and ML model trained.

2 BACKGROUND AND RELATED WORK

Here, we first cover fundamental assumptions on *FL* and the federated optimization problem, and then academic or industrial efforts on the topic of distributed *ML* and *FL*.

2.1 Preliminaries on Federated Learning

Assumptions. The *FL* optimization problem has the following typical assumptions [17, 20]:

- *Massively distributed:* Number of examples (data) available per client (device) is expected to be much smaller than the number of clients participating in an optimization.
- *Limited Communication:* Devices can be typically offline or with intermittent or slow connectivity; their bandwidth can be considered expensive commodity (especially if no WiFi connection is available).
- *Unbalanced:* Data available for training on each device is of different sizes, since users can have different usage profiles (some can be heavy hitters, others on the tail [34]).
- *Highly Non-IID:* Data available on each client is typically non representative of the population distribution, as they only reflect the given client's usage of the device.
- *Unreliable Compute Nodes:* Devices can go offline unexpectedly; there is expectation of faults and adversaries.
- *Dynamic Data Availability:* The subset of data available is non-static, e.g., due to differences in hour, day, country.

Notations. The next analytical formulations use these notations:

- Set of devices $\mathbb{K}$; total number of devices $K = |\mathbb{K}|$;
- A given client is identified as $k \in \mathbb{K}$;
- Total number of rounds R;
- A given round is identified as $r \in [0, R]$;
- Fraction of devices used per round: $C \in [0, 1]$;
- Number of data samples across all clients: n;
- Set of indices of data samples on k: P_k;
- Number of data samples available at k: $n_k = |P_k|$;
- Batch size of data samples used per client: B;
- Number of iterations of device k on local data: $t \in E$;

Federated Aggregation: We consider *FL*-based algorithms with finite-sum objectives of the form:

$$\min_{w \in \mathbb{R}^d} f(w), \quad \text{where} \quad f(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

$f_i(w)$ is defined as loss of prediction on seen examples, i.e., $l(x_i; y_i; w)$, using the trained model with parameters w. All data available in the system (n) are partitioned over K clients, each with a subset of indices P_k. Then, the problem objective can be rewritten as:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w), \quad \text{where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in P_k} f_i(w)$$

Given the assumption of non-IID data, F_k could be an arbitrarily good or bad approximation to f.

The model optimization under *FL* assumes a decentralized architecture, which performs a *Federated Aggregation* algorithm with the clients and a central server. Each client k executes a stochastic gradient decent (*SGD*) step (iteration t) on the local data available, $g_k = \nabla F_k(w_t)$. Assuming C=1, at iteration t, the server aggregates all gradients and applies the update on the global model: $w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$, since $\sum_{k=1}^{K} \frac{n_k}{n} g_k = \nabla f(w_t)$, where η is a fixed learning rate. Equivalently, every client can perform the update as: $w_{t+1}^k \leftarrow w_t - \eta g_k$, and the global model is $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$.

In fact, this process can be repeated for $t \in E$ iterations locally per client, before sharing models with the server in R rounds. Therefore, the client can iterate the local update $w^k \leftarrow w^k - \eta \nabla F_k(w^k)$ for E times, before the aggregation and averaging at the central server, per round r: $w_{r+1} \leftarrow w_r - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$.

It becomes apparent that factors such as E iterations per client, C clients participating in each round, and R rounds executed can have high impact on model performance, and communication cost incurred in the infrastructure to reach it. We note that C is usually selected in such a way [5] to account for device unreliability (intermittent connectivity, failed computation, etc.).

2.2 Related Work

Several initiatives from startups and online communities have been recently proposed in the space of decentralized ML. For example, Decentralized Machine Learning (*DML*) [11] is a (now abandoned) blockchain (BC)-based project, enabling its participants to build models in a distributed fashion, while growing its BC network.

Open-source community efforts propose libraries and platforms that will allow users to train ML models in a decentralized, secured and privacy-preserving (PP) fashion. For example, *OpenMined* [22] proposes the libraries *PySyft* and *PyGrid*, by employing multiparty computation (MPC), homomorphic encryption, *DP* and *FL* for secured and PP-ML modeling in a decentralized fashion, in both mobile or desktop environments. In addition, *Datafleets* [9] utilizes *DP*, secure MPC and role-based access control to build models inside or across enterprises, and on edge computing nodes. With similar technologies, *FATE* (Federated AI Technology Enabler) [12] focuses on desktop deployments.

Building on the popular *TensorFlow* (TF) framework, *TensorFlow Federated* (TFF) is an open-source framework for ML and other computations on decentralized data [27]. *coMind* [8] proposes a custom optimizer for TF to easily train neural networks via *FL*. There have also been benchmark frameworks proposed like *LEAF* [7], for learning in *FL* settings, with applications including *FL*, multi-task learning, meta-learning, and on-device learning.

In contrast to all these efforts, FLaaS follows a *FL*-as-a-Service model, with high-level APIs, enabling: a) independent 3rd-party applications (i.e., external to FLaaS operator) to collaborate and combine their common-type data and models for building joint meta-models for better accuracy; b) collaborative 3rd-party applications to combine their partial models into meta-models, in order to solve new joint problems, never before possible due to data siloing and applications' isolation; c) 3rd-party applications to build the aforementioned FLaaS models on edge nodes, desktops, mobile phones or other low-resource (IoT) devices (cf. Figure 1).

3 FLaaS MOTIVATION & CHALLENGES

FLaaS aims at providing to single applications an easy way to use *FL*, without the costly process of developing and tuning the algorithms, as well as to enable multiple applications to collaboratively build models with minimal efforts. Specifically, FLaaS is designed to support the following use cases (some examples in Figure 1):

1. Unique *FL* modeling per individual application for an existing ML problem without the need of developing the algorithms. Traditionally, *ML* modeling is requested uniquely per application aiming to solve a specific, existing ML problem: e.g., a streaming music application (e.g., Spotify) that wants to model its users' music preferences to provide better recommendations.

2. Unique *FL* model trained in a joint fashion between two or more collaborative applications for an existing ML problem. That is, a group *G* of applications interested in collaborating to build a shared ML model that solves an existing problem, identical and useful for each application, but on more, shared and homogeneous data. For example, Instagram, Messenger and Facebook (owned by the same company) may want to build a joint ML model for better image recognition, on images of similar quality and scope, but coming from each application's local repository.

3. Unique *FL* model trained in a joint fashion between two or more collaborative applications, as in case (2), but for a novel, never explored ML problem. For example, an application for planning your transportation (e.g., Uber, GMaps, or Citymapper) may want to model your music preference while on a specific transportation type (e.g., bicycle, bus, car, etc.).

Several challenges arise while supporting these use cases under a Federated Machine Learning setting, which we elaborate next.

Permission management across applications and services: Mobile and IoT systems provide mechanisms to grant application and services access to data such as mobile sensors, location, contacts or calendar. Such access is typically given at a very coarse granularity (e.g., all-or-nothing), and can be unrestricted or, more recently, granted per application. On top of these traditional permissions, FLaaS has to provide mechanisms to specify permissions across applications and services to share data and models among them. Further, it has to provide security mechanisms to guarantee these permissions are respected.

Privacy-preserving schemes: In FLaaS deployment scenarios and use cases, multiple applications and services can be involved in the *FL* execution. In order to guarantee the privacy of customers' data, it is critical to leverage privacy-preserving mechanisms in the construction of *FL* models. In FLaaS, we plan to leverage Differential Privacy (*DP*) to provide further privacy guarantees to participating clients. *DP* noise can be introduced at different stages of the *FL* system: in the data source at the client side, also known as local-*DP* [25, 29, 30, 33], at the central server side [13] while building the global model, or at an intermediate stage such as edge computing nodes [19] or base stations [23], or with hybrid methods and hierarchical methods [6, 28, 31]. However, introducing *DP* noise in the ML pipeline reduces model utility [32] as it affects convergence rate of the *FL*-trained model. Note that, while FLaaS plans to build on existing DP solutions, finding an optimal way to add *DP* noise in *FL* is an open research problem, and beyond the scope of this work.

Exchange model across a (hierarchical) network with FL: As depicted in Figure 1, FLaaS can build models in a hierarchical fashion across different network layers: end-user device, ISP edge nodes, or the central server. Recent works considered the hierarchical *FL* case, where multiple network stages are involved in the training process [6, 19, 23]. Such efforts showed convergence and accuracy can be improved with proper design under such settings. FLaaS will build on these works to realize its hierarchical use cases. However, the design of optimal hierarchical *FL* methods is an open research problem beyond the scope of this work.

Training convergence and performance: As mentioned earlier, the usage of *DP* in multi-stage training and the hierarchical *FL* approach impact the convergence and performance of *FL* models. However, in FLaaS, the possibility of building cross-application models introduces another dimension, potentially impacting model convergence and performance. This is a relevant research problem that FLaaS will need to address in the near future.

Platform usability: Every service platform should enable a customer to use its services with limited overhead and knowledge of the underlying technology. On the one hand, existing commercial MLaaS platforms (e.g., AWS [1], Google Cloud [14] or Azure [21]) provide users with APIs and graphical user interfaces (GUI) to configure and use ML services in cloud environments. However, these APIs are not designed to deal with cross-application model building, nor tailored for *FL* services. On the other hand, existing *FL* libraries (e.g., TFF [27] or OpenMined [22]) are still in prototype phase and cannot support a service model, and do not provide GUIs, or high-level service APIs. They also do not support cross-application ML modeling as FLaaS does. FLaaS builds on these existing works

1 可移植性的重要性

and provides high-level APIs to support model building across applications on the same device and across the network, and software libraries or Software Development Kits (SDKs) for application developers to include the service in their apps and devices (cf. Sec. 4.2).

4 FLaaS SYSTEM DESIGN

4.1 Service Main Components

The FLaaS design comprises three main system components:
Front-End: Main interface for customers (e.g., app or service developers), to bootstrap, configure, and terminate the service. It runs a GUI, processes front-end API calls from customers (or the GUI), and calls functions on the Controller to execute customer requests.
Controller: Takes as input commands received from Front-End, executes the required steps to configure the service, e.g., initialize the model, set appropriate permissions, etc. Once the service starts, the Controller is in charge of monitoring service health, budget, and terminating execution of ML modeling when requested.
Central Server and **Clients** (e.g., mobile or home devices, edge nodes): are the elements actually in charge of executing the *FL* algorithms and protocol (cf. Sec. 2.1). The Central Server, hosting the Controller and Front-End, is under the administrative domain of FLaaS, while the Clients are typically in another domain, e.g., at user side. The Server also runs a FLaaS Global module responsible for the federated aggregation of received models. Each Client runs a FLaaS Local module directly provided by the FLaaS provider. In addition, every application on the FLaaS Clients needs to embed a software library, providing the required functions that can be accessed via client APIs.

4.2 APIs and Software Libraries

Front-End APIs: FLaaS can be configured via front-end APIs, or a GUI that uses the front-end APIs under the hood. These APIs can be classified in three types, as follows:

- *DATA APIs* allow customers to describe data types the model takes as input for training or produces as output after inference. This is specified via JSON format and includes name and type of each input feature or output data column.
- *MODEL APIs* enable customers to create an ML model, define model type and parameters, or choose the option of parameter self-tuning. Also, these APIs allow customers to specify properties in the ML modeling, in case the model is built across (partially) different data from multiple customers, or as an average/ensemble across models.
- *PERMISSION APIs* enable customers to specify if and which other customers (e.g., apps) can access said data, or how other customers can access the model for inference, or to build models to solve new ML problems.

Client APIs: A set of functions need to be embedded in the application code to realize FLaaS functionality. To this goal, we design a software library (currently implemented as an Android SDK, cf. Sec 5) providing the implementation of such functions that are then exposed via a set of APIs. The library includes main functions such as: (i) Authenticate API to the Central Server. (ii) Support on-device training for ML applications, including: load a model (either pre-trained or with initial parameters), add training samples, conduct

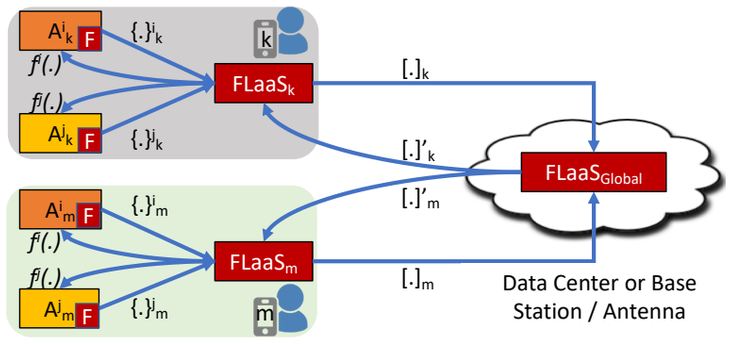


Figure 2: Overview of FLaaS ML modeling architecture.

model training, predict from test samples, and save or load model parameters to device memory. (iii) Exchange/share data between FLaaS-enabled apps on-device, before local *FL* training takes place.

While training on-device a new model is possible, it requires a significant amount of processing power and many communication rounds, making it impractical in user devices with limited resources. Transfer Learning (*TL*) [26] is an ML technique that takes a model built for a basic task T (e.g., image recognition) and reuses it as the starting point for a new task, but related to T, to speed up training and improve model performance. FLaaS employs *TL* for various scenarios (e.g., image object recognition, text classification, item recommendations, etc.) as it is suitable for currently resource-constrained mobile devices and networks.

4.3 FLaaS Algorithmic Design

We now provide algorithmic details of how the main use cases outlined in Sec. 3 are supported by FLaaS design.
FL modeling per application for existing ML problems: We assume a set of apps, $i \in \mathbb{A}$, installed on device $k \in \mathbb{K}$, are interested in building *FL* models with FLaaS. In this case, each app wants its own model built on its local data.

Figure 2 outlines the general interactions of two apps i and j with the FLaaS Local module running on user devices k and m. The apps communicate to the FLaaS Local their *FL* models built. Thus, for app i and device k, in Fig. 2 we define $\{.\}^i_k = F^i_k(w)$.

FLaaS Local collects all such models from individual apps, and transmits them in a compressed format to FLaaS Global:

$$[.]_k = [F^i_k(w)], \forall i \in \mathbb{A}, \forall k \in \mathbb{K}$$

Subsequently, FLaaS Global performs Federated Aggregation across all reported local models and builds one global weighted average model per app, which it then communicates back to the participating devices per app, i.e., in Fig. 2:

$$[.]'_k = [f^i(w)], \forall i \in \mathbb{A}, \forall k \in \mathbb{K}$$

Finally, the FLaaS Local module distributes the global model to each app, i.e., in Fig. 2:

$$f^i(.) = f^i(w), \forall i \in \mathbb{A}$$

Jointly-trained FL modeling between group of apps for existing ML problem: In the following scenario, we assume a group of two or more apps, $i \in \mathbb{A}$, installed on device $k \in \mathbb{K}$, are interested in collaborating and building a common *FL* model with FLaaS. This model will be shared among all apps but will be built jointly on each application's local data.

FLaaS Federated Learning as a Service

如何aggregate不同app的model?

Thus, in Figure 2, we can redefine the general interactions of a group G of apps i and j (i.e., $G = \{i, j\}$) with FLaaS Local running on devices k and m, in order to build such a joint model among them. In fact, we point out at least three different ways that such joint model can be built, by sharing different elements with FLaaS Local (Fig. 2):

1. Sharing local data: $\{.\}_k^i = \{x^i; y^i\}_k$

Apps in G share with FLaaS Local data they are willing to provide in the collaboration. FLaaS Local collects all shared data, which should have the same format, and performs *SGD* on them. For this way to be possible, participating applications must be willing, and permitted to share user data across applications.

2. Sharing personalized gradients: $\{.\}_k^i = \{\nabla F_k^i(w_t), \epsilon\}$

Apps share with FLaaS Local their personalized gradient for iteration t along with the error ϵ, which was acquired after training their local model for the t^{th} iteration. In this case, FLaaS Local uses the received gradients g_k^i per iteration t to incrementally correct the locally built joint model. Then, it releases back to apps the improved joint model, before receiving new updates in the next iteration.

3. Sharing personalized model: $\{.\}_k^i = F_k^i(w)$

Apps share with FLaaS Local their complete personalized models built on their data, after they perform E iterations. In this case, FLaaS Local performs Federated Aggregation on the received models, thus, building a joint model that covers all apps in G, with a generalized, albeit, local model. In the second and third ways, the apps do not need to worry about permissions for sharing data, as they only share gradients or models. Note that the user data never leave the device, in any of the aforementioned cases.

Then, for any of these ways, FLaaS Local reports to FLaaS Global the model jointly built on data or model updates, i.e., :

$$[.]_k = [F_k^G(w)], \forall i \in \mathbb{A}, \forall k \in \mathbb{K}$$

Subsequently, FLaaS Global performs Federated Aggregation across all collected local models and builds a global weighted averaged model, for each collaborating group $G \in \mathbb{G}$ of applications. Then, it communicates each such global model back to the participating devices, i.e., in Fig. 2:

$$[.]'_k = [f^G(w)], \forall G \in \mathbb{G}, \forall k \in \mathbb{K}$$

Finally, FLaaS Local distributes the global model to each application of the collaborating group G, i.e., in Fig. 2:

$$f^i(.) = f^G(w), \forall i \in G$$

Jointly-trained FL modeling between group of apps for a new ML problem: In this scenario, we assume a primary app i is interested in solving a new ML problem but does not have all data required to solve it. Therefore, it comes to an agreement with other, secondary apps (j) to receive such needed data ($x^{j'}$) to build the new model, using FLaaS. Notice that these additional data are a subset of the full data that secondary apps produce, i.e., $x^{j'} \subseteq x^j$. In a similar fashion as before, the collaborating apps must share data or models, in order to enable joint model building (Figure 2). In fact, we point out at least two ways that such joint model can be built, by sharing different elements with FLaaS Local:

1. Sharing local data:

1a. Primary application i: $\{.\}_k^i = \{x^i; y^i\}_k$

1b. Secondary applications j: $\{.\}_k^j = \{x^{j'}\}_k$

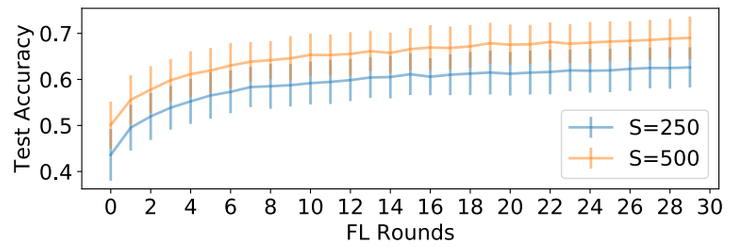


Figure 3: FLaaS test accuracy per FL round, for two sample sizes S available per device. Error bars: accuracy variability for 100 simulated clients.

Apps share with FLaaS Local the data they are willing to provide. FLaaS Local collects all shared data and performs *SGD* in an iterative fashion, to build the final local model.

2. Sharing personalized model:

2a. Primary application i: $\{.\}_k^i = F_k^i(w)$

2b. Secondary applications j: $\{.\}_k^j = F_k^{j'}(w)$

Apps provide trained local models that solve portion of the overall new problem, after E iterations.

Then, FLaaS Local builds a meta-model (e.g., based on hierarchical or ensemble modeling), to solve the new problem at hand. In either case, again, no data leave the device. Then, for either of the ways described, FLaaS Local reports to FLaaS Global the model jointly built, i.e., in Fig. 2:

$$[.]_k = [F_k^{i'}(w)], \forall i' \in \mathbb{A}', \forall k \in \mathbb{K}$$

Note: $\mathbb{A}'$ is the set of primary apps building the novel models and does not include secondary apps helping. Subsequently, FLaaS Global performs Federated Aggregation across collected models and builds a global weighted averaged model for each primary app model requested, and communicates each such global model back to participating devices, i.e., in Fig. 2:

$$[.]'_k = [f^{i'}(w)], \forall i' \in \mathbb{A}', \forall k \in \mathbb{K}$$

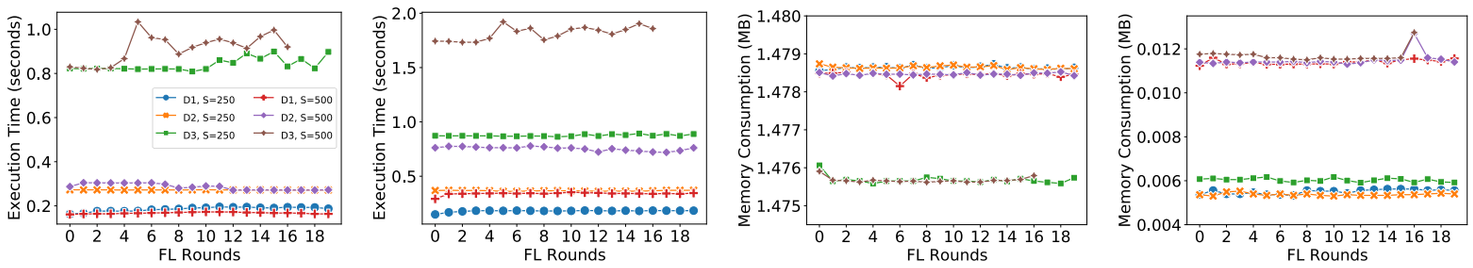
Finally, FLaaS Local distributes the global model to each primary application, i.e., in Fig. 2:

$$f^i(.) = f^{i'}(w), \forall i' \in \mathbb{A}'$$

5 FLaaS PROOF OF CONCEPT

We now present a proof-of-concept (PoC) FLaaS implementation in the mobile setting. We discuss early experimental results to showcase viability of two FLaaS use cases (Sec. 4.3): single app ML modeling and data sharing for ML modeling by two collaborative apps.

PoC FLaaS Implementation. On the client side, we focus on Android OS 10 (API 29). Specifically, we implement the FLaaS module as a standalone user level app and the library to be embedded in FLaaS-enabled apps as an SDK. Both FLaaS module and SDK leverage TensorFlow Lite 2.2.0 and Transfer API library from Google [15]. Exchange of data between FLaaS-enabled apps and the FLaaS Local module is performed using the OS's BroadcastReceiver [2]. The Central Server is implemented using Django 3.07 with Python 3.7. Note that currently, Controller and Front-End are not implemented. Finally, the FLaaS-enabled apps used for the PoC are toy apps only performing the FLaaS functionality and storing a set of data. The current FLaaS version is implemented in 4.6k lines



(a) (b) (c) (d)

Figure 4: Average execution time (a) and memory consumed (c) per image loaded for each FL round, and execution time (b) and memory consumed (d) for model training per epoch for each FL round. 3 device types and 2 sample sizes were tested.

of Java code (for the *FL*-related modules), and 2k lines of Python for server modules.

Experimental Setup. We evaluated the performance of our implementation with respect to ML Performance, Execution Time, Memory Consumption, CPU Utilization and Power Consumption using three Android devices. *D1*: Google Pixel 4 (2.42 GHz octa-core processor with 6GB LPDDR4x RAM). *D2*: Google Pixel 3a (2.0 GHz octa-core processor with 4GB LPDDR4 RAM). *D3*: Google Nexus 5X (1.8 GHz hexa-core with 2GB LPDDR3 RAM). We updated all devices to the latest supported OS (D1 & D2: Android 10; D3: Android 8.1), and disabled automated software updates, battery saver and adaptive brightness features when applicable. We further set the device under Flight Mode, enabled WiFi access, connected to a stable WiFi 5GHz network, and set the brightness level to minimum.

As a base network, and to initialize the Transfer Learning process, we use MobileNetv2 [24], pre-trained with ImageNet [10] dataset with image size 224x224. As a head network (used for the model personalization), we use a single dense layer, followed by softmax activation (SGD optimizer with 0.003 learning rate). As a dataset for model training and testing, we use CIFAR-10 [18], equally split and distributed across all experimental devices before the experiment.

In our experiments, we applied parameter values used in other *FL* works with CIFAR-10: 20 samples per batch, 50 epochs, 20 *FL* rounds, and 250 or 500 samples per user (S), corresponding to the two scenarios of individual app or joint *FL* modeling of two apps via data sharing. For measuring ML performance, we simulated *FL* on 100 users (devices) for the two scenarios in 30 *FL* rounds. For measuring the on-device cost, we assume 10 users' worth of data and execute *FL* modeling for the two scenarios on the real devices.

FLaaS ML Accuracy. Figure 3 plots the average test accuracy for the two scenarios, per *FL* round. These results, acquired using 100 simulated devices, show that when two apps share data (S=250+250), 10% better accuracy can be achieved with the model trained jointly at FLaaS Local, than individual models with half data (S=250).

FLaaS On-device Memory & Execution Costs. Figures 4(a), 4(b), 4(c) and 4(d) show cost in execution time and memory, averaged across 10 users' data, as computed on the three real devices, and with two use case scenarios per *FL* round. First, we note that all costs reported are, on average, similar through the various *FL* rounds. This means that a device joining any of the *FL* rounds is expected to have similar cost regardless if it is the beginning of the *FL* process or later. Second, we find that execution in the newer devices, D1 and D2, is significantly faster than the older device D3, and this is true for both image loading, and especially model training. Third, and expected, doubling the sample size as required in the second use case scenario (joint modeling), has the same execution cost per image loaded, but practically doubles the execution cost for model training. Forth, we observe that the average memory cost for loading images is same across devices, regardless of scenario. Fifth, and expected, when the sample size doubles (S=500), the model training consumes about double the memory than for S=250. Finally, we measure the cost of data sharing between apps for materializing the second use case scenario. We find that sending 250(500) samples between apps takes an extra time of 54.7(65.9)ms, and consumes 1.1(1.6)MBs of memory, on average, demonstrating the viability and low cost of this option for joint *FL* modeling between apps on the same device.

FLaaS On-device Power & CPU Costs. We measured power consumption and CPU utilization as estimated by the Battery Historian [3] tool (plots omitted due to space). The mean CPU utilization per round for S=250(500) is 16.6(27.0)%, 18.1(34.0)% and 31.1(33.3)%, for the 3 devices, respectively. Interestingly, for S=250(500), D1 and D2 have similar average power consumption of 75.6(123.2) and 81.0(138.0)mAh, respectively, per *FL* round. Surprisingly, D3 has 10-12x higher power consumption than D1 or D2, of 924.8(1563.4)mAh. This result is a consequence of the longer execution time and higher CPU utilization of D3 in comparison to D1 and D2, indicating that older devices may not be suitable for FL (or generally ML) model training.

6 CONCLUSIONS AND DISCUSSION

In this paper, we presented FLaaS, the first to our knowledge Federated Learning as a Service system enabling 3rd-party applications to build collaborative, decentralized, privacy-preserving ML models. We discussed challenges arising under these settings, and highlighted approaches that can be used to solve them. We also presented a FLaaS proof of concept under mobile phone settings, showing the feasibility of our design and potential benefits of collaborative model building. Our long-term goal is to finalize FLaaS design and deployment for large scale evaluation. However, we argue that many of the challenges raised in the FLaaS design represent fundamental open research problems in the Federated Learning space.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from EU H2020 Programme, No 830927 (CONCORDIA), No 871793 (ACCORDION) and No 871370 (project PIMCITY). The paper reflects only the authors' views and the Commission is not responsible for any use that may be made of the information it contains.