Coordinated Orchestration and Operation of Federated Learning Applications

L. Velasco*, P. González, and M. Ruiz
Optical Communications Group - Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
**e-mail: luis.velasco@upc.edu*

***Abstract*—Edge computing brings processing closer to data sources thus, reducing latency and network capacity utilization. However, data still needs to be moved for training Machine Learning (ML) models, which is against strict data privacy required by some customers. Federated Learning (FL) is a privacy-preserving ML technique where multiple FL clients collaborate to train a ML model. We assume that FL clients run in customer locations and exchange local models to a FL server running in a telecom operator, which can take advantage of powerful hardware accelerators. In this context, the transfer of large ML models upstream, from FL clients to the FL server, and downstream back to the FL clients, might require high-capacity connectivity in the access/metro network segment dedicated to FL applications. In this paper, we propose solutions to reduce connectivity capacity by extending coordination between FL applications and the telecom cloud orchestrator. The proposed algorithms and workflows are experimentally demonstrated and illustrative numerical results showcase the benefits of the proposed coordination.**

Keywords—Federated Learning, Networking for AI

I. Introduction

Federated Learning (FL) is a privacy-preserving Machine Learning (ML) technique where multiple FL clients, coordinated with one or multiple central FL servers, collaborate to train a ML model [1]. Different from conventional centralized ML, where data is collected from different sources and conveyed to a central server for processing, data is used locally to train ML models that are exchanged between the FL clients and the centralized FL servers. Because of their characteristics, many applications of FL have been reported, including healthcare and users' behavior (see, e.g., [2], [3]). The possibility of using hardware accelerators available in the cloud owned by hyperscalers will make possible to train Large Language Models (LLM) for a fraction of the cost of developing such infrastructure internally. However, the cost of transferring the models between locations can be too high, as those LLMs might consist of trillions of parameters and occupy hundreds of GB to TB. This fact makes that telecom operators, owning both the telecom network and a cloud infrastructure, can offer competitive and efficient solutions.

In FL, local models are trained by the FL clients using local data. Once trained, the models are transferred to the FL server, which in turn, trains a global model and sends it back to the FL clients. This loop is repeated for several rounds, where the accuracy of the global model improves at every round. For FL to converge in practical times, not only the training time has to be accelerated, but also time to transfer the models. Therefore, in the case of training large LLMs, high-capacity connectivity between FL clients and the FL server is needed. In view of that, some authors have proposed solutions for making FL communications more efficient (see, e.g., [4]). Furthermore, connectivity capacity can be greatly reduced once the models are transferred and activated back after new models are produced. Such process clearly motivates the collaboration between FL applications and the telecom infrastructure beyond the availability of automatic interfaces for the deployment of FL workloads.

In this paper, we propose the collaboration between FL applications and the telecom operators to reduce, as much as possible, the usage of networking resources. Specifically, we propose coordinated orchestration and operation for on-demand network capacity allocation. In addition, we target at minimizing the number of models that are transferred at every round without major impact on the performance of the models. The rest of the paper is organized as follows. Section II presents the targeted scenario and a simplified control and orchestration architecture is proposed. Section III presents our adaptive FL solution that selects the subset of FL clients that will participate in the FL training at every round. Section IV focuses on defining the needed coordination at the orchestration level to optimize networking resources during the FL operation. Section V reports the experimental results of the implemented workflows and provides numerical evaluation of the proposed coordinated operation. Finally, section VI draws the main conclusions of the work.

II. Federated Learning Orchestration and Operation

Figure 1 presents the scenario for a FL application. We assume that FL clients run in a customer private cloud infrastructure (locations 1 to *n* in Figure 1), e.g., hospitals in a healthcare FL application, whereas the FL server runs in a metro / metro-core location belonging to the telecom operator cloud. Client locations are connected to the telecom cloud infrastructure through an access/metro network. In the control/orchestration plane, a Customer Orchestrator is in charge of the deployment and operation of FL workloads on the targeted hybrid cloud infrastructure. To that end, it coordinates the deployment of workloads on customer cloud sites, e.g., configured as Kubernetes clusters, as well as on telecom cloud sites through the Telecom Orchestrator's north-bound interface. The latter coordinates the deployment of the customer workloads and manages the connectivity between the different locations through the access/metro network by means of a Software-Defined Networking (SDN) Controller.

978-3-903176-78-2

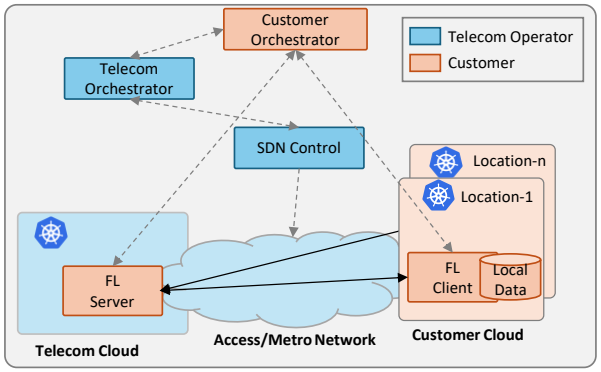

Figure 1 Federated Learning Scenario

Let us assume a FL application that consists of a set C of FL clients and one single FL server. We assume that local models follow the same architecture and configuration, which results into a common set of trainable coefficients (weights and biases) denoted as F. Then, ω_c represents the set of trainable coefficients (local model) of FL client c, being ω_{cf} the f^{th} coefficient. Once local models are trained locally using local data, they are shared with the FL server, which computes the global model Ω, e.g., as the average of the coefficients of the local models, see eq. (1). Once Ω is computed, it is distributed to all clients, that will replace their local model with the global one, and continue to the next round.

$$\Omega = \bigcup_{f \in F} \{\Omega_f\} = \bigcup_{f \in F} \left\{ \frac{1}{|C|} \sum_{c \in C} \omega_{cf} \right\} \quad (1)$$

In the case of FL applications training large models, exchanging local models from all the FL clients at every iteration requires high-capacity connectivity so that local models are transferred in practical times to the central FL server. However, once models are transferred, such capacity remains unused until the next FL round. Therefore, coordination between the FL application and the orchestration system would be beneficial to implement capacity-on-demand approaches ensuring that connectivity capacity is available when it is really needed by the application, while making unneeded capacity resources available for other services.

In addition, some FL clients might observe similar local data during local training, which would result in similar local models being conveyed to the FL server. Therefore, it would be important to find which local models should be transferred to the FL server at every round to optimize the use of connectivity capacity without meaningful impact on the accuracy of the global models.

III. Adaptive Federated Learning

In this section, we propose an adaptive FL mechanism that targets to select the subset of local models that will participate in the federated training at every round. The procedure is executed by the FL server to obtain the subset of FL clients $C^* \subseteq C$ for which the local model needs to be retrieved. To this aim, the FL server maintains: (*i*) a buffer W with the latest version of each local model; and (*ii*) the global models generated in the last two rounds. The selection procedure identifies whether some of the local models are close to the last global models (i.e., they do not bring new knowledge) and can be used for the next round, so their FL clients can be excluded from C^* in the next round. When new local models from clients in C^* are received, they replace the previous ones in the buffer and a new global model is computed following eq. (1).

Algorithm 1 – Client Selection Procedure

INPUT: t, $W=[\omega_c]_{\forall c \in C}$, Ω_t, Ω_{t-1} **OUTPUT**: C^*

1: **if** $t \leq T$ **then return** C
2: $C^* \leftarrow \emptyset$, $\Delta \leftarrow [\emptyset]_{\forall c \in C}$
3: **for** $c \in C$ **do**
4: $d_2 \leftarrow$ L2_norm(Ω_{t-2}, ω_c)
5: $d_1 \leftarrow$ L2_norm (Ω_{t-1}, ω_c)
6: **if** $\alpha \times d_2 < d_1 < d_2$ **then** $\Delta_c \leftarrow d_1 / d_2$
7: **else** $C^* \leftarrow C^* \cup \{c\}$
8: **if** $C^* \leftarrow \emptyset$ **then** $C^* \leftarrow$ arg min(Δ)
9: **return** C^*

Algorithm 1 details the adaptive procedure that runs at the beginning of every round t. The algorithm receives the round number and the buffer W with the previous local models and the last two global models. The algorithm selects all the FL clients during the first T rounds, to be sure that enough knowledge from all the FL clients is captured (line 1 in Algorithm 1). For training rounds after T, the selection of clients is performed. After some initializations (line 2), the distance between a local model and each of the global models is computed (lines 3-5). Distances between two models are computed as the L2 norm in the $|F|$-dimensional space of coefficients F. Two distances (d_1 and d_2) from the local model to the global models are computed at the end of rounds t-1 and t-2.

In the case that d_1 is smaller but close to d_2, we assume that the local stored model is close enough to the global one (similarity is controlled by parameter α) and consequently, is considered as reusable (line 6). Ratio Δ_c between current and previous distances is computed. Otherwise, when the condition in line 6 is not met, we assume that the local model needs to be updated and therefore, the FL client needs to be included in the subset of clients C^* (line 7). To ensure that the training of the global model does not stop, update of at least one client model is forced. Then, in the case that subset C^* is empty after processing all local models, the client that requires the minimum increment of α to break the reusability condition is selected for the current training round (line 8). Finally, the algorithm returns the subset of FL clients C^* (line 9).

IV. Coordinated Orchestration and Operation

Let us now define the workflows for the FL application deployment (WF1) and FL application operation (WF2).

WF1 (see Figure 2) starts when a customer issues the deployment of a FL application through the user interface (UI) (step 1 in Figure 2). The request is handled by the Customer Orchestrator, which generates a FL deployment descriptor (step 2). The description includes the configuration and resource allocation for each of the components involved in the FL deployment, i.e., FL clients, FL server(s), and upstream and downstream connectivity. Assuming that the FL server needs to be deployed in the telecom cloud, the Customer Orchestrator

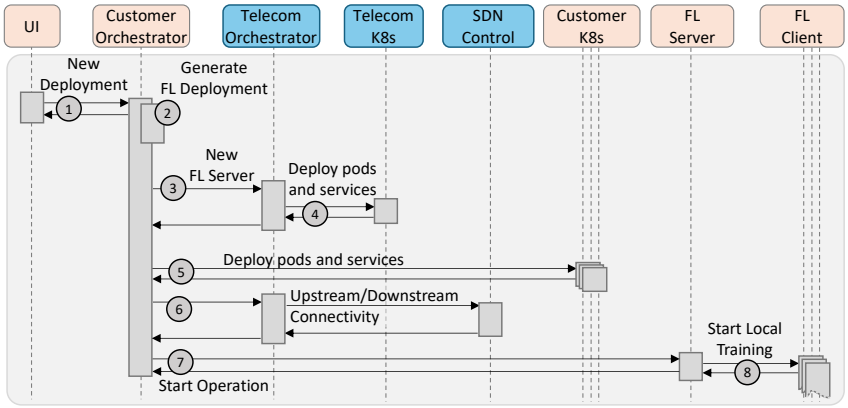

Figure 2 WF1: FL Application Deployment

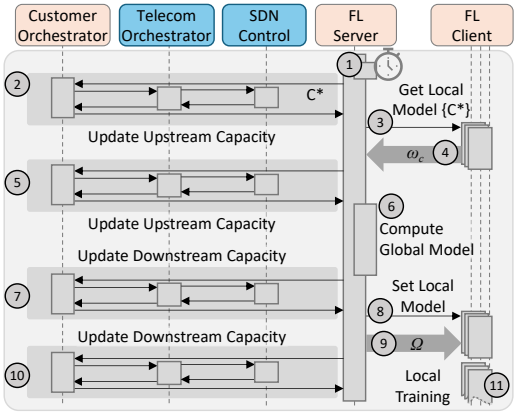

Figure 3 WF2: FL Operation

sends a request to the Telecom Orchestrator for that deployment (step 3). Upon the reception of the request, the Telecom Orchestrator checks it and proceeds with the deployment of the FS server application (step 4). Similarly to the customer cloud, we assume that telecom cloud management is based on Kubernetes. Note that the Telecom Orchestrator makes the decision of the final placement, e.g., based on the computation requirements, available connectivity to the customer locations, etc. Following the deployment of the FL Server, the Customer Orchestrator deploys all the necessary software of the FL clients through each Kubernetes manager in the locations defined on the deployment descriptor (step 5).

Next, the Customer Orchestrator requests upstream and downstream connectivity to the Telecom Orchestrator (step 6), which relies on the SDN Controller to configure the actual network nodes. Note that the connectivity between FL nodes needs to support commands and models exchanged between FL clients and the FL server. Therefore, high-capacity is needed when models are exchanged but a minimal capacity needs to be always available. Additionally, FL application upstream's connectivity should be based on unicast individual connections between each FL client and the FL server, whereas downstream connectivity can be based on multicast. Finally, when the deployment is complete, the Customer Orchestrator notifies the FL server to start its operation (7), which in turn notifies the FL clients to start the local training and the deployment phase finishes.

Once the deployment phase has finished, the FL application operation starts and the FL server periodically collects local models from the FL clients, computes a global model, and distributes it back to the FL clients. However, connectivity needs to be managed during these training rounds and the capacity increased and decreased back before and after the models are transferred. Such activity is defined in WF2 (see Figure 3). With a configured periodicity, the FL server starts a training round and runs Algorithm 1 to compute the subset of FL clients C^* from which the local models will be retrieved (step 1 in Figure 3). Then, the FL server sends a request to the Customer Orchestrator to increase the capacity of the upstream connectivity for the FL clients in C^* (step 2). The Customer Orchestrator, forwards the request to the Telecom Orchestrator, which in turn relies on the SDN Controller to actually configure the network nodes. Once the FL server receives a confirmation of the capacity allocation, it notifies the selected FL clients to send their local models (step 3). The last version of the local models are sent using the high-capacity upstream connectivity to the FL server (step 4). The FL server updates buffer W with the received local models ω_c and once the FL server confirms that all model transfers are completed, it sends a request to the Customer Orchestrator to decrease back the upstream capacity (step 5). The FL server computes a global model Ω applying eq. (1) to the local models in W (step 6). Once the new global model is computed, the FL server will update the FL clients. To this end, the FL server requests a capacity increase for the downstream connectivity (step 7). Recall that all FL clients will receive the new global model disregarding the last local model was retrieved in this round. Then, once the capacity update has been confirmed, the FL server notifies all the FL clients (step 8) and distributes the global model (step 9). Once the transfer is completed, the FL server requests to decrease the downstream capacity back to the minimum (step 10) and FL clients use the new global model to continue the local training process (step 11). This process is repeated until the decommissioning of the service.

V. RESULTS

This section first reports the experimental results of the implemented workflows defined in Section IV, and then provides numerical evaluation of the gains of the proposed coordinated operation for medium- and large-size LLMs.

A. Orchestration and Operation Experiments

The setup used to evaluate the proposed WFs runs in a virtualized environment provided by an OpenStack cluster on-premises. Three different locations have been set up, each one running a Minikube Kubernetes cluster [5]. Minikube offers most of the features found in production-ready Kubernetes clusters and is suited for testing environments due to its simplicity of deployment and maintenance. Moreover, Minikube uses the latest API versions of Kubernetes, making the migration to classic Kubernetes cluster seamless. Each emulated cloud resides in a different VM. The Orchestrators and the SDN Controller have been implemented using Python and are deployed in separate Virtual Machines (VM). The Flower FL framework [6] has been used for evaluation of the

Source	Destination	Info
UI	Customer Orch.	POST /deploy_fl
Customer Orch.	UI	HTTP/1.1 202 ACCEPTED
Customer Orch.	Telecom Orch.	POST /register_service
Telecom Orch.	Customer Orch.	HTTP/1.1 201 CREATED
Telecom Orch.	Telecom K8s	Application Data
Telecom K8s	Telecom Orch.	16443 → 54730
Customer Orch.	Customer K8s	Application Data
Customer K8s	Customer Orch.	16443 → 41542
Customer Orch.	Telecom Orch.	POST /set_connectivity
Telecom Orch.	SDN	POST /Setup
SDN	Telecom Orch.	HTTP/1.1 200 OK
Telecom Orch.	Customer Orch.	HTTP/1.1 200 OK
Customer Orch.	FL Server	POST /Start
FL Server	Customer Orch.	HTTP/1.1 200 OK
FL Server	FL Client	POST /Start
FL Client	FL Server	HTTP/1.1 200 OK

Figure 4 WF1 Exchanged Messages

(1)
```
{"service_id": "fl-1",
  "fl_config":{
    "clients": [
      {"id": "client-1", "type": "client", "location": "minikube-2"},
      {"id": "client-2", "type": "client", "location": "minikube-3"}],
    "server": {"id": "server-1", "type": "server", "location": "TelecomCloud"}}
```

(6)
```
{"service_id": "fl-1",
  "nodes": [
    { "id": "client-1", "type": "client", "endpoint": "192.168.30.39:30000"},
    { "id": "client-2", "type": "client" "endpoint": "192.168.30.86:30001"},
    { "id": "server-1", "type": "server"}, "endpoint": "192.168.30.101:30002"}],
  "up_links": [
    {"id" : "ul-1", "src": "client-1", "dst": "server-1", "bw_gbps": 0.01},
    {"id" : "ul-2", "src": "client-2", "dst": "server-1", "bw_gbps": 0.01}],
  "down_links": [{"id" : "dl-1", "src": "server-1", "dst": ["client-1", "client-2"],
                  "bw_gbps": 0.01}]}
```

Figure 5 Selected Messages from WF1

Source	Destination	Info
FL Server	Customer Orch.	POST /bandwith_reconfig
Customer Orch.	Telecom Orch.	POST /bandwith_reconfig
Telecom Orch.	SDN Ctl.	POST /resource_reconfig
SDN Ctl.	Telecom Orch.	HTTP/1.1 200 OK
Telecom Orch.	Customer Orch.	HTTP/1.1 200 OK
Customer Orch.	FL Server	HTTP/1.1 200 OK
FL Server	FL Client	GET /get_local_model
FL Client	FL Server	HTTP/1.1 200 OK
FL Client	FL Server	POST /data
FL Server	FL Client	HTTP/1.1 200 OK
FL Server	Customer Orch.	POST /bandwith_reconfig
Customer Orch.	Telecom Orch.	POST /bandwith_reconfig
Telecom Orch.	SDN Ctl.	POST /resource_reconfig
SDN Ctl.	Telecom Orch.	HTTP/1.1 200 OK
Telecom Orch.	Customer Orch.	HTTP/1.1 200 OK
Customer Orch.	FL Server	HTTP/1.1 200 OK
FL Server	Customer Orch.	POST /bandwith_reconfig
Customer Orch.	Telecom Orch.	POST /bandwith_reconfig
Telecom Orch.	SDN Ctl.	POST /resource_reconfig
SDN Ctl.	Telecom Orch.	HTTP/1.1 200 OK
Telecom Orch.	Customer Orch.	HTTP/1.1 200 OK
Customer Orch.	FL Server	HTTP/1.1 200
FL Server	FL Client	GET /set_local_model
FL Client	FL Server	HTTP/1.1 200 OK
FL Server	FL Client	POST /data
FL Client	FL Server	HTTP/1.1 200 OK
FL Server	Customer Orch.	POST /bandwith_reconfig
Customer Orch.	Telecom Orch.	POST /bandwith_reconfig
Telecom Orch.	SDN Ctl.	POST /resource_reconfig
SDN Ctl.	Telecom Orch.	HTTP/1.1 200 OK
Telecom Orch.	Customer Orch.	HTTP/1.1 200 OK
Customer Orch.	FL Server	HTTP/1.1 200 OK

Figure 6 WF2 Exchanged Messages

FL application. Flower implements both clients and servers, the communication, and the orchestration of the FL training session. Since Flower is agnostic to the ML technology, it allows the user to define their custom client and servers. Moreover, Flower offers predefined training strategies defining how the central server should aggregate the data received from the clients. A Web Dashboard UI has been developed using Flask and eases the interaction of the user with the Orchestrator API. The UI also allows the user to track the status of the deployments and the different steps involved.

Let us start with the assessment of WF1. Figure 4 shows a capture of the exchanged messages of the workflow and follows the same numbering as the one used in Figure 2. The deployment phase of the FL application is initiated by the user, who sends a request through the Web Dashboard (message 1 in Figure 4). This request includes the service identifier and a specification of the customer locations where the clients need to be deployed, while leaving the specific server location to be decided by the Telecom Orchestrator (see the details in Figure 5). The request is handled by the Customer Orchestrator, which generates the FL deployment descriptor, including the description and configuration of the pods hosting the FL clients. Several services are defined to ensure communication inside and outside the Kubernetes cluster; specifically, for the external communication between the FL components and the Orchestrators, each pod defines a NodePort service (<IP, Port>). The Customer Orchestrator sends a request to the Telecom Orchestrator including the details needed for the deployment of the FL Server (message 3). The Telecom Orchestrator selects the best cluster for the deployment of the FL server, generates a descriptor for the deployment and asks the Telecom Kubernetes manager to allocate the necessary resources on the Telecom Cloud, including the pod and the NodePort service (message 4). Next, the Customer Orchestrator communicates with each Kubernetes manager in the customer locations involved in the deployment of the FL clients to allocate and deploy the selected pods and services (message 5). The response to the Customer Orchestrator includes the assigned NodePort service for the FL server.

Using the NodePort details of the FL clients and FL server, the Customer Orchestrator sends a request to the Telecom Orchestrator to create the upstream and downstream connectivity of the FL application (message 6 in Figure 4). The details of message 6 are presented in Figure 5. The specification of the connectivity includes the service id, the NodePort, i.e., the IPs and ports of each of the endpoints exposed by the different FL components, and the specification of the initial capacity for the upstream and downstream connectivity. The Telecom Orchestrator processes the connectivity setup request and forwards it to the SDN Controller which in turn configures the involved network nodes. Finally, the FL training operation can start, so the Customer Orchestrator notifies the FL server to

start (message 7), and the FL server notifies the FL clients that the operation can start with the local training (message 8).

Once WF1 finishes, FL operation starts following WF2. Figure 6 shows the exchanged messages implementing the workflow following the numbering used in Figure 3. Periodically the FL server runs a FL training round and executes Algorithm 1 to find the subset of FL clients (C^*) for the current training round. Before the FL server can retrieve the local models from the FL clients, it must reconfigure the capacity of the uplink connectivity. To this end, the FL server sends a notification to the Customer Orchestrator with the service id and the list of FL clients (C^*) for which the increment of network capacity is required (message 2 in Figure 6). The Customer Orchestrator creates a request for the Telecom Orchestrator, which translates the request into a resource reconfiguration that the SDN Controller can handle. Once the updated network capacity is available for the selected links, the FL server sends a notification to each of the selected FL clients to transfer their local models (message 3). The selected FL clients receive the notification and transfer the last version of their trained local model to the FL server (message 4). Upon completion, the FL Server has all the required local models from the selected FL clients, so it issues a notification to the Customer Orchestrator to scale down the capacity of the uplinks, which coordinates with the Telecom Orchestrator and the SDN Controller (message 5). Then, the FL server computes a new global model applying eq. (1) to the new local models in the buffer W. Once this new global model is generated, the FL server updates the local models of the selected clients. To this end, it issues a notification to the Customer Orchestrator to reconfigure the downstream connectivity (message 7). When the connectivity is ready, the FL server notifies the FL clients to receive the new global model (message 8) and it transfers the global model to all FL clients (message 9), so the FL clients start training a local model based on the just received global one. Finally, the FL server notifies the Customer Orchestrator to reduce the capacity of the downstream connectivity (message 10). This process repeats until the decommissioning of the service.

B. Numerical Evaluation

For numerical evaluation purposes, we deployed a FL application with 5 FL clients and one single FL server. The system was implemented using the Flower framework (v1.25.0) [6] and every participant (FL clients and FL server) was hosted on a separate VM provisioned with 16 CPU cores and 32 GB of RAM running Ubuntu 24.04.2 LTS3. To replicate real-world conditions, artificial network latency (up to 2 s) was configured at the VM network interface level using the Linux traffic control (tc) subsystem.

As for the model being trained, we used the FLAN-T5-Base LLM [7], which consists of around 247 million parameters, leading to model instances of around 1.1 GB. A subset of around 25.000 samples of the Yahoo Answers Questions-and-Answers corpus [8] was used and partitioned across five clients using a native function in Flower that applies the Dirichlet-based strategy to simulate moderately non-identically and independently distributed data [6]. Figure 7 shows the data partition for each FL client (identified by their Id), which reflects the difference between them.

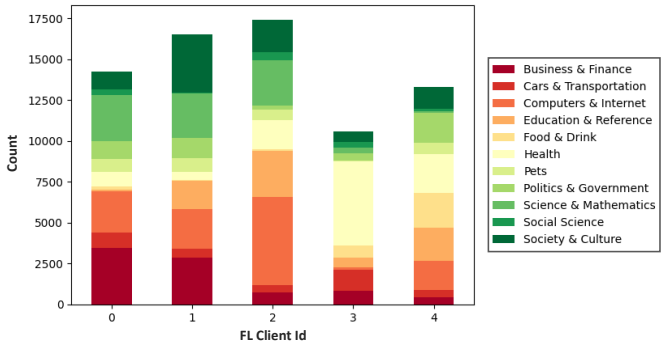

Figure 7 Client data partitions

Table 1 Performance Reference Metrics

Metric		FLAN-T5 (Experimental)	GPT-3 (Estimated)
Model Size	*# Params*	2.5e8	1.8e11
	GB	0.98	942
Execution Time (s)	*Client (epoch)*	1560	5875
	Server	5700	2850
Comm. Time (s)	*10 Gb/s*	4.28	1628
	100 Gb/s	2.23	172

Table 1 presents some reference performance metrics including: (*i*) the aforementioned model size in terms of number of parameters and size; (*ii*) the execution time of one single training epoch in a FL client, as well as the time to compute the global model in the FL server; and (*iii*) the communication time required to send models from one FL client to the FL server and vice versa, assuming network connections of 10 Gb/s and 100 Gb/s. The numbers show the experimental measurements using the FLAN-T5 model, as well as estimations for a larger model based on the GPT-3 one, based on scaling up the experimental measurements. From now on, we will consider the GPT-3 case, since it is more realistic in terms of size and more demanding in terms of network capacity requirements, which motivates the use of hybrid cloud infrastructures. The values in Table 1 have been used to generate scenarios for the comparison of three different approaches, namely: (*i*) *static*, which assumes that connectivity between clients and server is not dynamically managed; (*ii*) *fixed*, where connectivity is dynamically managed and the set of clients is fixed (all clients are selected at every round); and (*iii*) *adaptive*, that adds client selection to dynamic connectivity management.

Figure 8 shows the network capacity reduction of the fixed approach with respect to the static one, for rounds of incremental duration. This figure illustrates the gains of timely coordinating the connectivity of the FL application, i.e., providing large connectivity only when clients and server need to communicate to exchange models. The results are presented for both connection speeds; in both cases, connectivity is always ensured with a minimum of 1 Mb/s capacity. The results show from 2 to 3 orders of magnitude of average capacity reduction when rounds last for a few hours. Note that, in case

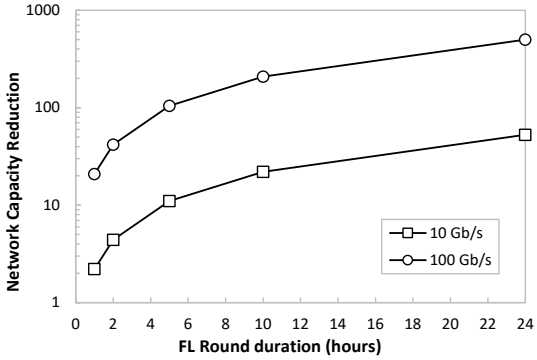

Figure 8 Fixed vs Static comparative analysis

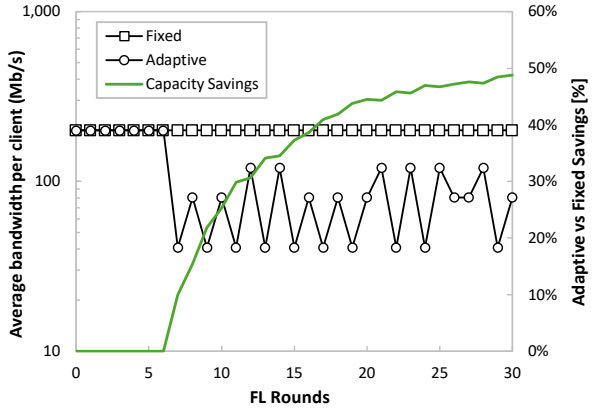

Figure 9 Adaptive vs Fixed comparative analysis

of a periodical daily training (i.e., a new round starts every 24 hours), the average capacity of the fixed approach is 50 times and 500 times less than that of the static approach, for 10 Gb/s and 100 Gb/s connections, respectively.

Let us now compare the additional gain from selecting specific FL clients at every round. Figure 9 compares the fixed and the adaptive approaches, assuming FL rounds of 24 hours, and plots the results as a function of the number of rounds. The adaptive method has been configured with parameters: T=6 and α=0.5. The average network capacity per client in the upstream direction is plotted, showing constant value in the fixed case since all FL clients send models at every round. In contrast, after the initial period of full FL client selection, the adaptive approach considered a variable number of FL clients at every round, leading to overall network capacity savings. We observe that after 30 rounds the relative savings reaches around 50%, which is a remarkable additional gain to be added to the overall benefits of the proposed coordination approach. Figure 10 completes the study, by showing that the behaviour of loss and accuracy of both fixed and adaptive approaches is similar, which eventually validates the latter approach.

VI. Concluding Remarks

Coordinated orchestration and operation of FL applications deployed over hybrid customer–telecom cloud infrastructures have been proposed and experimentally assessed. The main objective was to reduce the connectivity utilization required to exchange models during FL training. The solution enables capacity-on-demand operation and adaptive participation of FL clients at each training round.

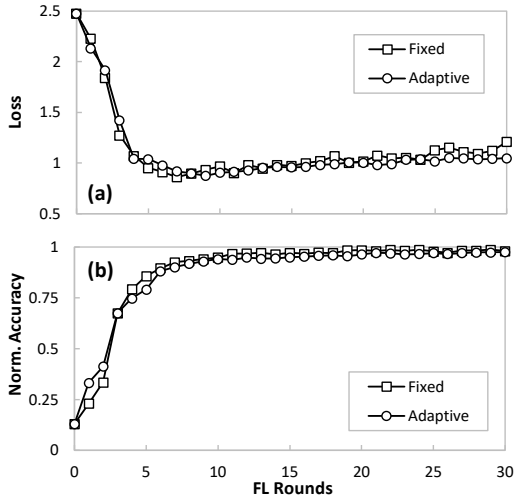

Figure 10 Loss (a) and accuracy (b) of fixed and adaptive approaches

Experimental validation demonstrated the feasibility of dynamically deploying FL workloads and reconfiguring network connectivity using cloud-native technologies such as Kubernetes for workload management and SDN for transport network control. Numerical evaluations for training LLMs demonstrated the effectiveness of the proposed coordination. Two LLMs were considered: FLAN-T5 was used for the experimental assessment of the proposed solution, while the much larger GPT-3 was selected for illustrative numerical estimations. For the latter, static provisioning would require model transfer times of approximately 172 s over 100 Gb/s links, leading to highly inefficient average bandwidth utilization. Results showed that dynamic connectivity capacity management can reduce average network capacity requirements by two to three orders of magnitude as compared to the static provisioning. Additional savings of approximately 50% were showed through adaptive client selection without degradation in loss or accuracy. The results demonstrated that coordinated orchestration of FL applications and telecom infrastructures is essential for efficient FL operation.

Acknowledgement

The research leading to these results has received funding from the European Union's Horizon Europe research and innovation programme under G.A. No. 101092766 (ALLEGRO), the project PID2024-157824OB-I00 funded by MICIU/ AEI/10.13039/501100011033 /FEDER, UE and from ICREA. The authors thank the students Xi Chen and Dongwei Li.

[1] B. McMahan *et al*., "Communication-Efficient Learning of Deep Networks from Decentralized Data," in Proc. AISTATS, 2017.
[2] S. Matta *et al*., "Federated Learning for Privacy-Preserving Healthcare Data Sharing: Enabling Global AI Collaboration," in American J. of Scholarly Research and Innovation, 2025.
[3] M. Isaksson and K. Norrman, "Secure Federated Learning in 5G Mobile Networks," in Proc. GLOBECOM 2020.
[4] C. Wu *et al*., "Communication-efficient federated learning via knowledge distillation," Nature Communications, vol. 13, 2022.
[5] Minikube [On-line] https://minikube.sigs.k8s.io/docs/
[6] Flower FL Framework [On-line] https://flower.ai/
[7] H. Chung *et al*., "Scaling Instruction-Finetuned Language Models," arXiv, 2022. https://doi.org/10.48550/arXiv.2210.11416
[8] "Yahoo! Answers Topic Classification", dataset, https://www.kaggle.com/datasets/bhavikardeshna/yahoo-email-classification.