# Decentralized Edge Workload Forecasting with Gossip Learning

Alessandro Tundo, Federica Filippini, Francesco Regonesi, Michele Ciavotta, and Marco Savi

***Abstract*—Edge computing has emerged as a crucial paradigm for addressing the growing demands of interconnected devices and large-scale mobile applications by relocating computation and storage services closer to end-users. Edge workloads are inherently volatile and challenging to forecast due to their dependence on factors such as human mobility patterns and geographically-distributed infrastructure, combined with the dynamic nature of edge nodes. Traditional centralized approaches to workload forecasting are inadequate in the context of decentralized and failure-prone edge environments. To address this challenge, this paper investigates workload forecasting using Gossip Learning (GL), an asynchronous peer-to-peer learning protocol. GL allows for the training of forecasting models in a fully-decentralized manner, thereby mitigating single point of failure risks and enhancing overall system robustness. We extended the original protocol across multiple dimensions to improve convergence, reduce communication overhead, and enhance resilience to failures. We evaluated the proposed approach through extensive simulations; the obtained results demonstrate its effectiveness with respect to classical methods, rendering it a promising solution to enhance load balancing and task offloading strategies at the edge, thereby ensuring Quality-of-Service (QoS) and reducing Service Level Agreement (SLA) violations.**

***Index Terms*—Edge Computing, Workload Forecasting, Gossip Learning, Function-as-a-Service, Machine Learning**

## I. Introduction

Edge computing has rapidly evolved to complement and extend cloud computing, addressing the increasing demands of interconnected devices and large-scale mobile applications. This paradigm shifts computation and storage services closer to users by positioning them at the network edge, thereby meeting the requirements for ultra-low latency, location awareness, optimized bandwidth allocation, reduced execution costs, and lower energy consumption [1]. However, edge computing also introduces unique challenges due to its dynamic and geographically-dispersed nature. Edge workloads are particularly volatile and difficult to predict as they depend on several factors like the interaction between human mobility patterns and a geo-distributed computing infrastructure [2] and the particular application scenario (e.g. IoT data ingestion and processing [3]–[5], Machine Learning (ML) model inference and training [6]–[8], or virtual network functions (VNFs) execution [9]–[11]). Moreover, edge nodes are often heterogeneous, with limited capacity, and tend to join and leave the network due to unpredictable link and node failures. Consequently, defining optimal resource allocation and task offloading policies to guarantee Quality-of-Service (QoS) and avoid Service Level Agreement (SLA) violations at runtime is significantly more complex compared to centralized cloud scenarios. Furthermore, security and privacy concerns must be carefully addressed at every decision level, as they are crucial in an ecosystem with a broader attack surface than cloud systems [1].

Research in this area has aimed to tackle these challenges by proposing peer-to-peer (P2P) solutions for managing network dynamics arising at the edge [12]. As for workload prediction, over the last decade, a variety of techniques have been proposed within the context of cloud computing [13]. However, little effort has been put to adopt (variants of) these techniques to P2P-based edge computing. *ML-based workload forecasting* using *P2P* (or *decentralized*) *learning* can be extremely advantageous in a fully-decentralized edge environment [14]: this approach takes advantage of contributions from different nodes to construct a forecasting model tailored to each node. Unlike a model trained only on a node's local data, this method offers better generalization, and preserves privacy while minimizing bandwidth utilization.

Among the variants of P2P learning, *Gossip Learning* (GL) stands out as a decentralized, asynchronous approach [15]. In GL, peers (i.e., participant nodes) communicate directly to update their models [16], eliminating the need for a central system to aggregate locally-trained models, as required in other distributed learning approaches such as Federated Learning (FL) [17]. This feature makes GL particularly suitable for P2P edge environments, as it eliminates the risk of a single point of failure and leverages the existing P2P network to build the workload forecasting model.

Moreover, GL addresses the challenges in developing such a model for a geographically-dispersed environment with unreliable nodes by allowing asynchronous model updates. This method ensures that information eventually disseminates throughout the entire edge network [15]. In addition, by incorporating contributions from various nodes, each node's model can capture a wide range of workload patterns, achieving a balance between generalization and specificity. This balance is essential in a decentralized learning context, allowing nodes to

Manuscript received on May 13, 2025.

This research was partially funded by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP E83C22004640001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and by the Austrian Science Fund (FWF) 10.55776/PAT1668223 and 10.55776/P36870.

Federica Filippini, Francesco Regonesi, Marco Savi, and Michele Ciavotta are with the Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milan, Italy (e-mail: {name.surname}@unimib.it).

Alessandro Tundo is with the Institute of Information Systems Engineering, Technische Universität Wien, Vienna, Austria (e-mail: alessandro.tundo@tuwien.ac.at).

Alessandro Tundo and Federica Filippini equally contributed to the manuscript.

handle out-of-distribution traffic patterns effectively. Finally, a well-generalized model can be adopted by new nodes joining the network, even if they lack historical data to develop their own models.

This paper introduces **Gossip Learning as a solution for decentralized workload forecasting in edge environments**. While our proposed method is generally applicable to a variety of edge computing scenarios, we adopt, as a motivational example, DFaaS [18], a decentralized edge environment that implements the *serverless Function-as-a-Service* (FaaS) model. In this context, we focus on forecasting the workload—a critical step towards enabling improved responsiveness, resource scaling, and distribution at the edge [3], [18]–[22].

We enhance the GL protocol to better align with the characteristics and challenges of decentralized edge computing. In particular, we propose a *performance-based approach* inspired by PENS [23] to selectively merge models from neighbors, increasing the robustness. We also propose a novel merge strategy, *Age-Weighted Overwrite*, which uses the model *age* to probabilistically decide model overwrites. In addition, we explore *FixedUpdates* and *EarlyStopping* criteria to prevent overfitting. Moreover, we implemented and publicly released **a discrete-event simulator** capable of simulating GL with various network topologies and stopping criteria, which is a feature not available in other simulators [24], [25]. An open-source prototype of the simulator is publicly accessible [26].

Finally, we conducted **an extensive empirical evaluation** to assess the applicability of GL for workload forecasting in P2P edge topologies. In particular, we considered five dimensions: *(i)* the performance of GL in this domain, compared to classical approaches (i.e., offline-only, centralized, and Federated Learning); *(ii)* the impact of GL parameters on performance; *(iii)* the scalability and network overhead of GL; *(iv)* the robustness of GL to node failures; *(v)* the demand for computational resources during training. Our experiments leveraged both real workload traces from FaaS applications and synthetic instances. All experimental materials are *publicly available* on Zenodo [27] and on GitHub [26].

The rest of the paper is organized as follows. Section II provides background information that introduces the Gossip Learning protocol. Section III describes our motivational scenario and the approach we propose to tackle the problem of decentralized load forecasting. Section IV provides details on the discrete-event simulator. Sections V and VI present details of the experimental setup and empirical evaluation. Section VII discusses the related work. Finally, Section VIII outlines the conclusions and possible future research directions.

## II. Background

Learning in distributed systems, where data resides on distinct nodes, presents a challenge due to network and computational scalability limitations, and privacy concerns [17].

Federated Learning (FL) [17] was initially proposed by Google as a means of training the model underlying its predictive keyboard (GBoard) while preserving user privacy. In FL, model training is distributed across multiple workers, each holding a private local dataset. A central server coordinates the training process by periodically selecting a subset of workers, elaborating the latest global model and transmitting it to them. The selected workers perform local training on their respective datasets for a predefined number of epochs and return model updates—typically in the form of weight updates or gradients—to the server. The server then aggregates them to refine the global model before the next training round.

FL is currently widely adopted for tasks outside the original scenario, including energy consumption forecasting [28], base station traffic forecasting [29] and many others [30]. Nevertheless, a single aggregation point (i.e., the main server) is still required to reconstruct the global model, which is challenging to achieve in decentralized edge computing scenarios where nodes are generally considered unreliable due to unpredictable failures (e.g. in network links) [1].

Gossip Learning is an optimization framework for the training of ML models in decentralized environments [31]. In GL, all nodes act as equal peers, each independently training a local ML model using its private dataset. Instead of relying on a central server, nodes periodically exchange model updates with randomly-chosen peers in a decentralized, peer-to-peer manner. Through these exchanges, models gradually incorporate knowledge from different nodes as propagated across the network. Unlike Federated Learning, where a single global model is aggregated, Gossip Learning results in a diverse set of models—one per node—each shaped by the specific sequence of updates it has received.

This approach is particularly well-suited for edge environments, as it fully leverages the decentralized nature of the system without requiring a central synchronization point. Additionally, GL has been shown to achieve performance on par to or even surpassing that of FL [15], while offering enhanced robustness. By eliminating a single point of failure, GL mitigates the vulnerabilities associated with centralized architectures, making it more resilient to node failures and network disruptions.

In this work, each node participating in the GL protocol must implement a defined interface that supports two primary operations, formally defined by the asynchronous procedures `MainLoop` and `OnReceiveModel`, as detailed in Algorithm 1. The `MainLoop` routine periodically disseminates model updates $w$ along with contextual information $c$ (e.g. the node's univocal identificator) to a subset of its peer nodes[1]. The `OnReceiveModel` procedure is triggered upon receiving an update from a peer node, integrating the received model with the local one, fine-tuning it using the node's dataset, and updating the context[2].

The following sections (II-A and II-B) provide further details about peer selection and merge strategies.

[1]Model updates may consist of the full set of model parameters, a subset of it, or the model's gradient computed with respect to local data. This work adopts a representation where updates refer to (a subset of) model weights, aligning with the implementation considered in this paper.

[2]For clarity and generality, context updates are explicitly represented by the `UpdateContext` procedure in Algorithm 1. However, they may also be handled within `MERGE` or `TRAIN`, depending on the specific implementation.

**Algorithm 1** Gossip Learning routines

**async procedure** MAINLOOP()
  **while** true **do**
    peers ← SELECTPEERS()
    SEND($(c, w)$, peers)
    wait($\Delta_t$) ▷ $\Delta_t$ is the model transmission time
  **end while**
**end procedure**

**async procedure** ONRECEIVEUPDATE($c_r, w_r$)
  $(c, w) \leftarrow$ MERGE($(c, w)$, $(c_r, w_r)$)
  $(c, w) \leftarrow$ TRAIN($(c, w)$, $D_l$) ▷ $D_l$ is the node dataset
  $c \leftarrow$ UPDATECONTEXT($(c, w)$, $(c_r, w_r)$, $D_l$)
**end procedure**

### A. Peer Selection

Indiscriminately broadcasting model updates to a large number of neighbors is inefficient and impractical under bandwidth constraints. In GL, nodes—particularly those with high connectivity—must carefully select a subset of peers for model transmission to balance communication overhead and learning efficacy. The `selectPeers` routine plays a crucial role in managing the trade-off between exploration and exploitation in decentralized learning. It is responsible for peer discovery [32] and selecting one [15] or, rarely, more peers [33] to send the local model (unicast vs. multicast).

Peer selection strategies can be broadly classified as *static* or *dynamic*. Static strategies maintain a fixed set of target peers throughout training, ensuring stable communication patterns but potentially limiting convergence speed, reliability, and model diversity. In contrast, dynamic strategies adaptively select peers at each iteration, introducing variability that may enhance learning efficiency.

The effectiveness of a peer selection strategy depends on the underlying network topology and application requirements. In fully-decentralized settings, random peer selection provides robustness to node failures [34]. Other methods incorporate topology-aware selection and node prioritization based on factors such as model similarity, learning progress, or network proximity [35]–[37], improving convergence speed and reducing communication overhead. As specified in Section III-C, in this paper we adopt a strategy called *Active-Peer Random Selection*, due to its high robustness to failures.

### B. Merge Strategy

In distributed/decentralized learning, the `Merge` strategy governs how local models are integrated, shaping both convergence dynamics and model diversity. These strategies can be classified based on the nature of exchanged data, the aggregation function employed, and their adaptability.

*Parameter-based merging* aggregates model parameters through methods such as simple or weighted averaging, ensuring stability but often struggling in non-iid settings, where statistical heterogeneity degrades performance [38]. Notable examples include *Overwrite*, and *Simple Average* [15]. In *Overwrite*, the received model completely replaces the current one. In *Simple Average*, the arithmetic mean of the received model weights and the local model weights is computed to mitigate the risk of complete information loss.

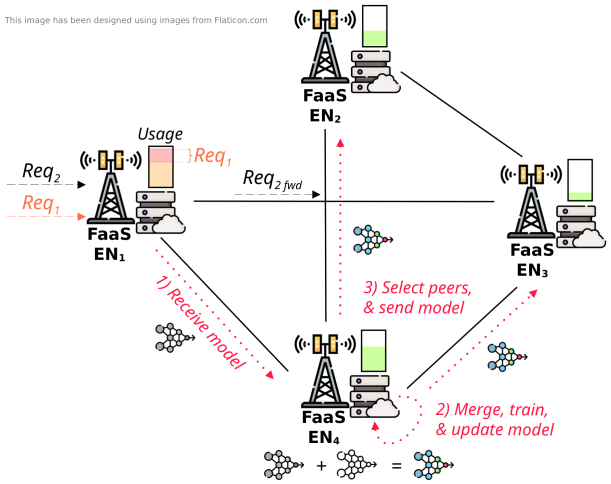


Fig. 1. The motivational scenario representing a set of telecommunication towers equipped with computing and storage capabilities, namely, edge computing nodes (ENs). Each EN hosts an instance of the *DFaaS* platform, initially proposed by Ciavotta *et al.* [18], and participates in the GL protocol.

*Adaptive merging* dynamically adjusts aggregation weights based on inter-model similarity, thereby improving generalization in heterogeneous conditions [15]. Conversely, *gradient-based merging* transmits only gradient updates—exemplified by Decentralized Stochastic Gradient Descent (DSGD)—offering communication efficiency but introducing *gradient staleness*, which may hinder optimization [39]. While research on GL often draws from FL, a direct transfer of insights remains problematic, as FL assumptions do not seamlessly extend to GL settings. A critical gap persists in the literature, underscoring the need for targeted investigations into the behaviors emerging within GL frameworks.

As specified in Section III-C, in this paper we consider, in addition to *Overwrite* and *Simple Average*, two strategies named *Age-Weighted Average*, firstly introduced in [15], and the novel *Age-Weighted Overwrite*.

## III. WORKLOAD FORECASTING WITH GOSSIP LEARNING

This section presents our motivational scenario for workload forecasting in decentralized edge environments. Details about how the GL protocol was adapted to align more closely with the specific characteristics of the problem at hand are then provided and discussed.

### A. Motivational Scenario

Several pertinent use cases for advanced computation at the edge of the network have been identified in the literature. These include mobile augmented reality (MAR) [16], [40], traffic safety applications [41], and IoT data processing [3]–[5] just to mention a few. A reference scenario is defined herein, as illustrated in Fig. 1, which depicts a network of telecommunication towers equipped with computing and storage capabilities, thereby functioning as edge computing nodes

(ENs). Each EN hosts an instance of the DFaaS platform, as initially proposed by Ciavotta *et al.* [18].

DFaaS (Decentralized Function-as-a-Service) is a peer-to-peer edge computing platform that implements the Function-as-a-Service paradigm, enabling each node to execute *serverless functions* in response to client requests. Clients, such as smartphones or other smart devices, initiate function execution requests by selecting the node with the strongest perceived signal—typically the nearest one. When a node's *predicted incoming workload* exceeds its available computing resources, the node dynamically offloads excess requests to the closest underutilized nodes, thereby mitigating congestion and optimizing resource utilization.

A defining characteristic of DFaaS is the absence of a centralized coordinator. All nodes operate in a fully-decentralized manner, functionally identical and equal in their interactions. Consequently, no master node governs network topology or orchestrates task distribution. Instead, a P2P service discovery protocol enables nodes to autonomously identify and communicate with one another, irrespective of the underlying physical network topology or technology. This dynamic discovery mechanism establishes an overlay network that facilitates direct node-to-node communication.

Within this overlay network, DFaaS nodes continuously exchange information regarding their operational state, latency, and network topology changes. Latency data, in particular, enables each node to identify and interact with a subset of peers for which communication latency remains below a threshold, ensuring proximity-based workload distribution.

### B. Gossip Learning for Workload Prediction in DFaaS

Gossip Learning emerges as an ideal framework for training the workload prediction models of DFaaS nodes, aligning naturally with the platform's decentralized architecture. Each DFaaS node relies on an individual prediction model to estimate its incoming workload, a critical factor in maintaining efficient resource allocation. An inaccurate model—whether underestimating or overestimating future loads—can lead to significant inefficiencies: underestimation results in congestion and request rejections, degrading the QoS, while overestimation leads to unnecessary offloading, underutilizing local computational resources.

A purely local training approach, based solely on a node's historical workload data, may fail to capture the spatio-temporal correlations in workload variations across neighboring nodes, particularly in mobility-driven edge applications. Given DFaaS's inherently peer-to-peer nature, conventional Federated Learning techniques that rely on centralized aggregation are ill-suited due to the risk of *vanishing variance*—a phenomenon in which the progressive loss of diversity causes models to converge prematurely to suboptimal solutions. Even hybrid approaches that employ leader election mechanisms [42] would introduce unnecessary computational overhead, disrupting the platform's asynchronous operation.

In contrast, GL seamlessly integrates with DFaaS's overlay network, enabling decentralized and continuous model refinement as nodes asynchronously exchange model updates with their dynamically-discovered neighbors. The protocol can be activated periodically (e.g. daily or weekly) or remain continuously active. Continuous updates enhance adaptability to workload variations by ensuring timely model adaptation, but they also increase communication overhead [43].

**Algorithm 2** Modified `OnReceiveModel` for multi-model merging

```
1: procedure OnReceiveModel(c_r, w_r)
2:     l_r ← getLoss(D_l, w_r)        ▷ Evaluate model on local data
3:     cache.store((c_r, w_r, l_r))
4:     if cache.size() > T then:
5:         models ← cache.get_best_n(N)
6:         w ← merge(models)
7:         w ← train(w, D_l)
8:         c ← UpdateContext(c, w, D_l)
9:         cache.clear()
10:    end if
11: end procedure
```

### C. Extensions to the Gossip Learning Protocol

The GL protocol described in Section II is extended along five key dimensions: 1) We introduce an approach inspired by PENS [23] that enables the simultaneous merging of multiple models, facilitating more effective knowledge integration; 2) We design the *Active-Peer Random Selection* mechanism, which implements a multi-peer selection strategy seeking to enhance convergence speed; 3) We propose the *Age-weighted Overwrite* merge strategy, where the probability of overwriting the current model is proportional to the age ratio between the current and received models; 4) We investigate the role of two stopping criteria—*FixedUpdates* and *EarlyStopping*—in mitigating overfitting and ensuring stable model convergence; 5) Finally, we analyze the impact of a model compression mechanism based on random sampling of network weights, balancing communication efficiency and learning performance.

*1) Configurable Number of Merged Models:* We extend the vanilla GL protocol to support the simultaneous merging of multiple models, drawing inspiration from the PENS (Performance-based Neighbor Selection) algorithm [23]. PENS improves convergence in GL by prioritizing model exchanges between peers with similar data distributions, thereby mitigating challenges posed by non-iid data.

Following PENS, the `OnReceiveModel` procedure is modified (see Algorithm 2). Instead of immediately merging incoming models, each received model is first assessed on local data and stored in a cache, potentially replacing a previous version from the same peer. When the cache accumulates $T$[3] models, the $N$ top-performing (lowest loss) models are selected, merged, and the resulting model is trained locally. Finally, the cache is cleared to prevent stale updates and counteract vanishing variance.

*2) Active-Peer Random Selection:* While similarity-based selection (like the one implemented in PENS [23]) performs well in stable environments, it struggles in dynamic settings where network topology and workloads shift over time. To

[3]The merging threshold $T$ is a controllable parameter, constrained by the node's minimum degree.

address this limitation, we propose a peer selection mechanism that remains resilient under such changes.

Rather than relying on feature similarity, we leverage the DFaaS P2P network's peer sampling service to enable decentralized and scalable neighbor discovery. It prioritizes peers based on latency, selecting topologically-closer nodes to balance similarity with reduced communication overhead. This locality-aware selection provides a crucial advantage: it inherently adapts to evolving network conditions, seamlessly integrating new nodes without requiring explicit similarity estimation. Furthermore, unlike traditional GL, which predominantly relies on unicast propagation, we employ multicast to improve communication efficiency. To prevent excessive message flooding, we introduce a stochastic selection mechanism that restricts interaction to a controllable fraction of active peers. This ensures a diverse model exchange while keeping message overhead manageable.

*3) Age-Weighted Overwrite:* In this paper, we propose a novel model merging strategy, termed the *Age-Weighted Overwrite* strategy, which synthesizes the *Overwrite* approach (discussed in Section II-B) with the *Age-Weighted Average* method, as introduced in [15].

The *Age-Weighted Average* [15] strategy modulates the merging process by incorporating a weighting factor that accounts for the *age* of each model. Formally, the age of model $i$, denoted as $t_i$, is defined as the cumulative number of data records on which the model has been trained, where each record is counted anew upon every training pass. This formulation ensures that models trained on larger or more frequently used datasets are assigned higher ages, reflecting their accumulated learning history. The primary objective of this approach is to mitigate the adverse effects of integrating *outdated* models—i.e., models trained on substantially fewer data points than the current one. The model age information can be communicated through the context $c$, facilitating efficient coordination among distributed agents.

The proposed Age-Weighted Overwrite strategy introduces a probabilistic *overwrite* mechanism, wherein the received model may entirely replace the current model with probability $p$, which is dynamically computed based on the relative ages (defined as in [15]) of the models. Specifically:

$$p = \begin{cases} 1 - \left(\frac{t}{t_r}\right)^2, & \text{if } t_r > t, \\ \left(\frac{t_r}{t}\right)^2, & \text{if } t_r \leq t. \end{cases}$$

with $t$ and $t_r$ representing the ages of the local and received models, respectively.

*4) Stopping Criteria:* In its classical form, GL assumes that the protocol is either continuously running or running for a fixed amount of time. While the literature offers numerous sophisticated techniques for terminating training to prevent overfitting (e.g. early stopping, as discussed in [44]), these methods are often challenging to adapt to the GL paradigm. This is because each node operates with a distinct model, making it difficult to define a general stopping criterion that can effectively control the entire procedure.

In this work, we propose and examine the potential of employing two stopping criteria for GL in our scenario:

1) *FixedUpdates*: each node is constrained to perform a fixed number of updates. A new update is performed each time a node receives a message from a neighbor. In the event that a message is received subsequent to the threshold being reached, it is discarded. This is the simplest criterion, but when applied it does not take into account the quality of the resulting model.
2) *EarlyStopping*: each node implements an early stopping mechanism. Upon each update, the merged model is trained for a fixed number of epochs. Subsequently, the validation loss is evaluated and compared to that of the best trained model so far. If the validation loss does not show an improvement of at least a quantity equal to the parameter designated as `min_delta`, a counter is increased; otherwise, it is reset to zero. Upon reaching a value designated as `patience`, the stopping criterion is met. The primary distinction between this approach and a classical early-stopping procedure [44] is the way how the validation loss is evaluated. In traditional approaches, the comparison is conducted between a global reference model and a local one. In this approach, however, the validation loss is evaluated after each update on a model—result of a merge operation—that may originate from a distinct neighbor with respect to previous merged models.

In both criteria, GL must address the issue of nodes terminating their training at a different point in time. This presents a challenge, as other nodes may continue selecting a node that is no longer active in training activities, resulting in the inefficient utilization of network resources and a slower convergence speed. Nevertheless, this issue can be addressed by establishing two fundamental rules:

1) Whenever a node receives a message including a model after its stopping criterion has been satisfied, it discards it and notifies the sender that it is no longer active;
2) A node must keep the `MainLoop` running for as long as it has some active neighbors, regardless of whether it has stopped or not, to ensure that its model—which is final if it has met its stopping criterion—can be sent to them. This ensures that those neighbors can continue updating their models until they also meet the stopping criterion.

More details on node states and transitions in our simulations will be provided in Section IV-A.

*5) Model Compression via Subsampling:* The exchange of model updates among peers in decentralized learning systems introduces significant network overhead, particularly as models grow in size and complexity. This communication burden not only affects scalability but also exacerbates vanishing variance.

To address these challenges, *model compression* has been proposed as a means to reduce network overhead, mitigate vanishing variance, and enhance generalization [45]–[47]. A seminal work in this area explores strategies for improving communication efficiency in FL through compression techniques such as quantization, sparsification, low-rank approximation, and subsampling [45]. However, in the specific domain of GL, the use of model compression techniques remains underexplored. A handful of studies have proposed subsampling, where only a fixed percentage of randomly-

selected model weights is shared with selected peers [15], [43]. Building upon this body of work, our paper investigates *subsampling* as a model compression technique due to its proven computational efficiency.

## IV. Discrete-Event Simulator

We designed a *discrete-event simulator* capable of running both the GL classical protocol and the proposed extensions. The simulator is scenario-agnostic, meaning it is not restricted to the specific context described in Section III-A.

Events are processed sequentially based on their timestamps, that is, when a new event occurs, it is added to a time-ordered *priority queue*, whose elements are processed by the simulator one at a time. The simulation ends when the queue is empty. Optionally, a simulation can be configured so that each node will periodically test its current model against a set of benchmark models, which shall be the same for all nodes (e.g. a model trained in a centralized fashion). By computing the loss function for the predictions made at each step, it is possible to analyze the training progress, both in terms of loss reduction and convergence speed among the models in different nodes.

The simulator allows configuring the capacity of each link by specifying the time required to transfer model weights—expressed in simulated seconds. It is possible to define both indirect and direct links to create a wide variety of network topologies, with configurable download and upload bandwidths. Finally, the simulator supports both link and node failures in a customizable way. We have integrated both the *carbontracker* [48] and the *scalene* [49] profilers to gather information on the power, energy, CPU and memory consumption in several phases of the GL protocol.

In the following, we detail the most relevant aspects of the simulator: the support of different node states and transitions between them, and the simulation workflow including the types of generated events.

### A. Node states and transitions

Nodes can be in four different states, namely, `ACTIVE`, `TRAINING`, `STOPPED`, and `FAILED`. A node is `ACTIVE` if it continues accepting incoming messages from neighbors and updating its own model. A node is `TRAINING` when an *update* is triggered. This occurs when an `ACTIVE` node receives a message from a neighbor and later begins training its merged model. Upon completion of the training, depending on whether the stopping criterion is met, the node can return to the `ACTIVE` state or transition to the `STOPPED` one. All messages received while the node is in the `TRAINING` state are discarded. A node is `STOPPED` when it meets the stopping criterion for training. Periodically, every node undergoes a binomial trial with probability $p$ to determine if it should transition to the `FAILED` state. In this state, the node ceases processing and discards all incoming messages. After a configurable delay, it reverts to the `ACTIVE` state.

Fig. 2 shows an example network with 7 nodes. Green nodes are `ACTIVE` (i.e., nodes 3 and 5), node 2 in blue is `TRAINING`, three nodes are `STOPPED` (i.e., gray nodes 1, 4, and 6), and 1 node is `FAILED`. Among `STOPPED` nodes, only node 6 (dark gray) is not sending its model weights, because both its neighbors are also stopped (i.e., nodes 1 and 4).

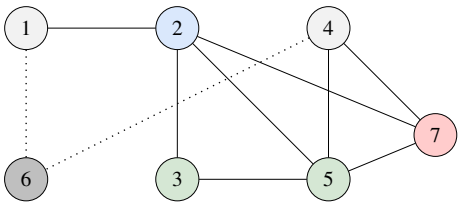


Fig. 2. Network with 7 nodes with different states. Green nodes (3 and 5) are `ACTIVE`, blue node (2) is `TRAINING`, gray nodes (1, 4, and 6) are `STOPPED`, and the red node is `FAILED`. Node 6 is not sending its model weights because its neighbors (nodes 1 and 4) are also stopped.

### B. Workflow and event types

The simulator supports five main event types, i.e., `ReceiveModel`, `SendModel`, `SaveModel`, `FailedNodeEvent`, and `RecoveryNodeEvent` that will be introduced in the following by describing the simulation workflow.

According to the protocol specification, each node must periodically send a copy of its model weights to the target neighbors. To do so, it should first check if it has any `ACTIVE` neighbor. If not, the event processing terminates and the node stops sending model weights. Otherwise, the node selects some targets to which it will send a copy of its serialized (and possibly compressed) model weights. Both target selection and model compression strategies depend on the current simulator configuration. After that, the simulator triggers the `ReceiveModel` events for the selected target neighbors and the next `SendModel` event for the current node, which occurs right after the transmission of all models to neighbors is completed.

Please note that during the initialization phase of a simulation, an instance of the `SendModel` event is generated for each of the nodes and enqueued in the first 60 seconds of simulated time in a random fashion. This ensures that the nodes start building their model asynchronously, eliminating the bias of a sequential start.

As said, a node can receive messages from its neighbors only if it is in `ACTIVE` state; otherwise, any received message is discarded. Upon receiving a message, the model weights are stored locally until $T$ models are received from distinct nodes. Once this threshold is reached, the $N$ best performing ones, in addition to the local one, are merged into a single model using the chosen merge strategy, and this undergoes training for a fixed number of epochs. After the training process is completed, the `SaveModel` event is triggered. Separating these two events allows the simulator to evaluate the computational time required to conduct the training.

Upon completion of a round of model updates, a node must evaluate the new model performance by computing the validation loss. If this is lower than the one of the previous model, indicating improved performance, the node should store the updated model parameters. Additionally, the node must assess its stopping criterion to determine its next state.

Finally, at regular intervals, the simulator probabilistically determines whether a node should transition to the failing state. If so, it generates both a `FailedNodeEvent`, marking the node's transition to a state where it halts processing and discards incoming messages, and a `RecoveryNodeEvent` with a future timestamp, scheduling the node's return to the `ACTIVE` state. An open-source prototype of the simulator is publicly accessible [26].

## V. Experimental Setup

This section provides details on the datasets preparation (Section V-A), the ML model used for all the experiments (Section V-B), and finally, the training methodology and evaluation metrics we considered (Section V-C).

### A. Datasets Preparation

We considered two different types of workload traces as datasets for our experiments: synthetic traces, which we considered to perform an initial training campaign and an ablation study validating the different components of our GL protocol, and real traces extracted from the workload of Microsoft's Azure Functions [50]. In both cases, since no information is provided in the Azure dataset about which nodes serve the incoming requests, the mapping between workload traces over time and the nodes in the P2P network has been generated based on real data related to the positions of telecommunication towers and the movement of people within the city of Porto (Portugal). The underlying assumption behind exploiting mobility traces in the data generation process is that people's movement within a city is a good proxy for network traffic generated by users at the far edge.

In the following, we focus on the adopted procedures for network and traffic generation. In particular, the first and last steps are common to both the synthetic and real-traces dataset, while the traffic generation process has been carried out differently in the two cases.

*1) Network Generation:* The first step involves generating a network of interconnected nodes distributed over a geographical area. Specifically, several networks have been generated by considering the central area of Porto city (latitude in $[41.1369, 41.1690]$, longitude in $[-8.6338, -8.5862]$) and sampling real antenna positions from the `OpenCellID`[4] public dataset. An example is shown in Fig. 3.

After selecting the desired number $n$ of nodes, we generated networks with the desired degree of edge connectivity $k$[5]; note that, considering telecommunication networks, it is safe to assume a connectivity greater than 1 to ensure a minimum level of redundancy and resilience against link failures.

Our algorithm weights the links according to *Harvesine distance* between nodes, which is a measure of the distance between two points on the surface of a sphere. This metric is often used to compute geographical distances, especially in navigation, as it provides more accurate measures than the classical Euclidean distance.

[4]https://opencellid.org

[5]A graph is $k$-edge-connected if it remains connected whenever less than $k$ edges are removed; this occurs if each node has at least $k$ neighbors.

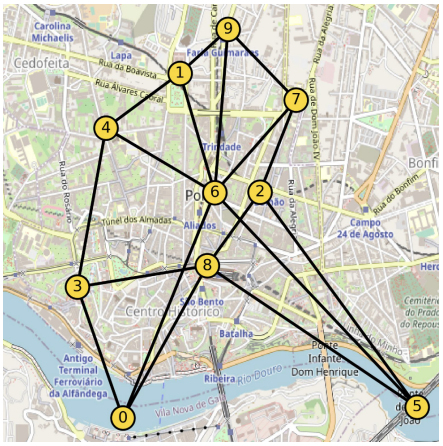


Fig. 3. Example of a generated 3-edge connected network with 10 nodes.

*2) Traffic Generation:* This step has been detailed differently according to whether synthetic or real function traces are considered. First of all, in both cases we have divided the considered region of the Porto city into a $20 \times 20$ grid, which simplifies the dataset generation process and enables a rapid creation of alternative datasets. Considering a specific network generated as in the previous stage, each grid is assigned to a different tower based on its distance from the tower locations. To guarantee a realistic traffic profile towards the towers, we have considered the `Porto taxi`[6] dataset, which encompasses GPS trajectories of taxis operating in the city of Porto, sampled every 15 seconds, over the course of a year spanning from 2013 to 2014.

To generate the synthetic workload traces, for each cell of the grid we constructed a time series representing the number of taxis present at any given moment, and we assumed that a function request is generated whenever the GPS coordinates of a taxi are sampled and directed towards the tower associated to the grid the taxi is into. Considering the Azure function traces, instead, we proceeded by associating each application owner with one of the taxis in the Porto dataset. All requests coming from a specific owner (i.e., taxi) are then directed towards the tower associated with the grid where the taxi is at a given moment in time. It is relevant noting that the Azure traces include the number of per-minute function invocations over 14 consecutive days; in our work, we have considered aggregated invocations over 15-minute time intervals to limit the noise in the workload signal.

*3) Pre-processing:* For each possible configuration $(n, k)$, where $n$ represents the number of nodes and $k$ is the network edge connectivity, 10 network variants and their corresponding node datasets are generated. For simplicity, the unique pair given by a configuration and one of its instances will be referred to as *experimental setting*. The datasets produced for each experimental setting undergo a series of pre-processing steps to prepare them for training. In particular, a total of $\nu - \tau$ time sequences is generated by each node to be used for training, where $\nu$ is the number of generated observations and $\tau = 4$ is the number of past timesteps to be used for

[6]https://www.kaggle.com/datasets/crailtap/taxi-trajectory

predictions. Obtained sequences are reshaped into a training record $(X_i, Y_i)$, where $X_i$ is the feature array for each time step and $Y_i$ is a scalar value representing the true number of requests to be predicted for each timestep. The 20% of prepared sequences form the test set, while the 10% of those in the training set are kept for validation.

The resulting datasets (one per node) contain a univariate time series representing the number of requests, towards a specific serverless function, that are generated by clients connected to the base station at each specific time step. Not considering the presence of multiple serverless functions is not a limitation; indeed, as shown in [51], the best approach for FaaS workload forecasting is training a single global univariate model on samples from different time series (i.e., related to different functions) and use it to predict the incoming traffic for all functions, meaning that our approach can be readily extended to consider multiple functions. Global univariate models should be preferred over alternatives because: *(i)* serverless functions often exhibit similar patterns, so a model performing well on one time series is likely to do so also on others; and *(ii)* the multiplicity of functions is too high to train either a separate univariate model for each function or a multivariate model with such a large number of outputs.

### B. Machine Learning model

The ML model trained by each node is an LSTM (Long Short-Term Memory) neural network, widely recognized as the *de facto* standard for handling time series data due to its robustness against the vanishing gradients problem [52]. Although transformers often yield better results, their superior performance for time-series forecasting is still in question [53]; the large number of trainable parameters makes it impractical to continuously serialize and transmit them over the network.

The architecture and hyperparameters tuning were conducted manually. The final model comprises 2 stacked LSTM layers with 50 units each (using the *tanh* activation function), followed by a fully connected (FC) layer with 32 units (using the *ReLU* activation function), and a single-unit output layer. A dropout rate of 20% is applied before each FC layer as a regularization technique. The Adam optimizer [54] was chosen, with an initial learning rate of 0.001.

We used the Mean Squared Error (MSE) as a loss function, as is commonly done for regression problems. MSE is particularly suitable for this application as it penalizes larger errors more heavily, making it effective in cases where the magnitude of the error is critical, not just its proportion to the true value.

### C. Training methodology and evaluation metrics

For each tested *experimental setting*, including a specific set of GL protocol parameters, we run 10 different simulations so that the same configuration and network features get tested on different network variants. Each simulation produces $n$ models $\tilde{M}_i$ (one per node), whose performance is compared with those of two benchmark approaches: *single-node* models $M_i$, trained exclusively on local data, and a *centralized* model $M_C$, trained on combined data obtained by aggregating and shuffling the training records from all nodes. For a subset of our experiments (i.e., Sections VI-A, VI-B1 and, VI-B3) we also compared our approach with Federated Learning (FL) using FedAvg [17] as aggregation strategy.

TABLE I
IMPACT OF DIFFERENT STRATEGIES THAT REDUCE NETWORK USAGE ON GL GENERALIZATION CAPABILITIES AND TRAINING TIME

| Technique | Impact on | |
|---|---|---|
| | Generalization | Training time |
| Reduce the number of updates | Negative | Positive |
| Introduce early stopping | Negative | Positive |
| Weights sampling | Negative | No impact |
| Slow down the `MainLoop` | No impact | Negative |
| Low target sampling probability | No impact | Negative |

We considered the MSE our main evaluation metric to compare model performance, which is also the loss function used for training as mentioned in Section V-B. Furthermore, we computed its mean (MMSE) and standard deviation (STD) across all trained models and simulations, along with the *Relative Standard Deviation* (RSD), defined as $\frac{STD}{MMSE}$. Since it is scale-invariant, RSD provides a meaningful measure of how similar the errors obtained by different models are.

These metrics were computed in the *classical* way by comparing the predictions made by a node model with the true values contained in its local test set. However, this was not sufficient to evaluate the generalization capabilities of the trained models, whose optimization is the main goal of a decentralized learning approach (see Section I). Therefore, we defined *generalization metrics* by computing MSE, MMSE, STD and RSD, for each model, using as test set the union of all the node neighbors test sets. In the latter case, it is thus possible to evaluate the capability of the model to recognize patterns experienced by other nodes.

## VI. EMPIRICAL EVALUATION

This section presents and analyzes the results of all experiments performed to validate the GL protocol in our context. In particular, a preliminary comparison among the centralized approach, the single-node (i.e., local-only) training, and GL featuring different merge strategies is conducted in Section VI-A. Section VI-B presents the results of an ablation study aimed at exploring the impact of the different GL protocol configuration parameters, and a scalability analysis is discussed in Section VI-C. A summary of our observations on how different techniques that reduce the network usage affect the GL generalization capabilities and model training times is reported in Table I. Section VI-D analyses the resilience of GL with respect to the failure of different percentages of nodes during training. A validation of our GL protocol considering real function invocation traces coming from the Microsoft Azure dataset [50] is reported and discussed in Section VI-E. Finally, GL is assessed in terms of energy and resource utilization in Section VI-F.

### A. Comparison with benchmark approaches

Our initial experiments evaluate whether GL can serve as a viable alternative to *single-node*, *centralized*, and *Federated*

TABLE II
BASELINE NETWORK AND GL PROTOCOL CONFIGURATION

| Parameter | Value |
|---|---|
| Nodes $n$ | 10 |
| Edge-Connectivity $k$ | 3 |
| Updates | 1 epoch $\times$ 100 updates |
| Target sampling probability $\pi$ | 50% |
| Weights sampling | No |

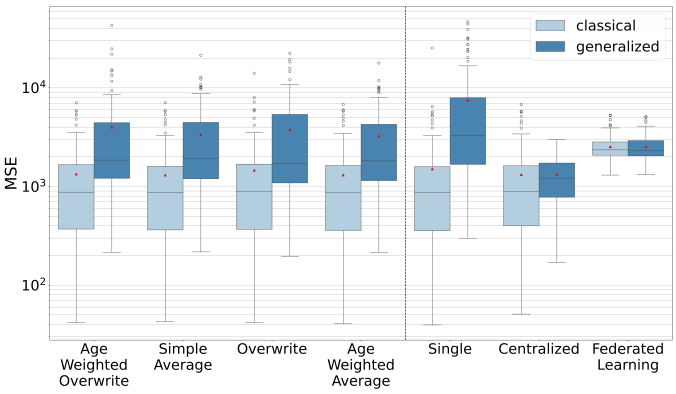


Fig. 4. Comparison among *GL* with *different merge strategies*, *single-node* and *centralized* training, and *Federated Learning* (red triangles denote the average MMSE across all experiments).

*Learning* approaches. To do so, we conducted experiments using configuration parameters consistent with those reported in the literature [15] (see Table II), comparing the impact of different merge strategies. In these experiments, we did not consider the protocol extensions proposed in Section III-C apart from using the *Age-Weighted Overwrite* merge strategy.

We performed ten simulations across all the generated variants of a 10 nodes 3-edge connected network. The MSE and average MMSE values, considering both the *classical* and *generalized* version (see Section V-C), are reported in Fig. 4.

We observe that the values of metrics computed in the *classical* way are roughly equivalent for all approaches except for FL. In contrast, the generalization performance of GL models is better than those of *single-node* models, even though worse than the ones of the centralized model, and FL, which, however, shows a reduced variance. Notably, FL achieved generalization performance on par with GL while exhibiting reduced variance. This outcome is anticipated, as FL trains a unified model for all nodes by averaging their parameters, thereby slightly penalizing individual node performance in favor of enhanced overall generalization. Furthermore, the Age-Weighted Overwrite strategy, although marginally inferior to the Age-Weighted Average strategy in terms of MMSE, still slightly improves the base Overwrite strategy in terms of classical metrics. Finally, we can observe from the extension of box-plot bars that, with the only exception of the generalization performance of the *centralized* model, all the other strategies achieve comparable values of STD and RSD. Collectively, these results underscore the robustness of the GL protocol with respect to the adopted merge strategy across both classical and generalization metrics.

TABLE III
THE MEAN AND STANDARD DEVIATION OF MSE FOR DIFFERENT NUMBERS OF UPDATES AND TRAINING EPOCHS

| Training | Classical | | | Generalization | | |
|---|---|---|---|---|---|---|
| | MMSE | STD | RSD | MMSE | STD | RSD |
| GL 1 ep. $\times$ 100 updates | 1309 | 1454 | 1.11 | 3256 | 3189 | 0.98 |
| GL 1 ep. $\times$ 20 updates | 2584 | 6049 | 2.34 | 6232 | 7778 | 1.25 |
| GL 1 ep. $\times$ 50 updates | 1404 | 1707 | 1.22 | 3884 | 3984 | 1.03 |
| GL 2 ep. $\times$ 20 updates | 1386 | 1621 | 1.17 | 5594 | 7034 | 1.26 |
| GL 2 ep. $\times$ 50 updates | 1303 | 1440 | 1.11 | 3822 | 3939 | 1.03 |
| GL 5 ep. $\times$ 20 updates | 1292 | 1425 | 1.10 | 5307 | 6798 | 1.28 |
| GL 5 ep. $\times$ 50 updates | 1270 | 1389 | 1.09 | 4814 | 5212 | 1.08 |
| FL 1 ep. $\times$ 100 updates | 2530 | 806 | 0.32 | 2535 | 802 | 0.32 |
| FL 1 ep. $\times$ 20 updates | 3189 | 1130 | 0.35 | 3192 | 1134 | 0.36 |
| FL 1 ep. $\times$ 50 updates | 2452 | 689 | 0.28 | 2453 | 692 | 0.28 |
| FL 2 ep. $\times$ 20 updates | 2575 | 830 | 0.32 | 2571 | 830 | 0.32 |
| FL 2 ep. $\times$ 50 updates | 2553 | 705 | 0.28 | 2567 | 724 | 0.28 |
| FL 5 ep. $\times$ 20 updates | 2718 | 932 | 0.34 | 2721 | 918 | 0.34 |
| FL 5 ep. $\times$ 50 updates | 2730 | 967 | 0.35 | 2728 | 958 | 0.35 |
| Single (100 ep.) | 1510 | 2768 | 1.83 | 7544 | 9852 | 1.31 |
| Centralized (100 ep.) | 1322 | 1403 | 1.06 | 1324 | 701 | 0.53 |

## B. Impact and robustness of GL protocol parameters

We experimented with various configurations of the GL protocol to evaluate how they balance model generalization capabilities, network transmission costs (i.e., network usage), and training time. In all subsequent sections, except for Section VI-B3, we present results using the Age-Weighted Average merge strategy. This strategy is slightly more effective in terms of generalization metrics.

*1) Number of epochs per update and number of updates:* We evaluated the impact on performance and convergence speed of varying the number of updates per node and training epochs per update, while keeping all other parameters unchanged. The results presented in Table III indicate that increasing the number of epochs per update can yield performance comparable to or even superior to that of a higher number of updates with fewer epochs, based on classical metrics. Regarding generalization power, as expected, reducing the number of updates negatively impacts performance. However, the GL models still outperform those trained on a single node, and different configurations of FL for classical metrics, suggesting that a balance can be achieved between desired accuracy and communication frequency among nodes.

This behavior happens because, when increasing the number of epochs per update, models have more time to fit a node dataset after they are transferred. This increases the model divergence (see Fig. 5), but at the same time implies a faster and more robust training progress with respect to the predictions performed on the local node datasets, albeit at the expense of generalization.

*2) Weights compression through subsampling:* This approach applies only to average-based merge strategies, intending to reduce the message size and, consequently, network usage (which decreases proportionally to the sampling rate if the number of updates per node remains constant).

We considered sampling rates of 0.2, 0.4, 0.6, 0.8, and 1 (i.e., no sampling), and set the other parameters as in Table II. Experiments results (see Fig. 6) highlight how higher sampling rates improve the performance in terms of classical metrics, since they limit the impact that neighbors have on the model

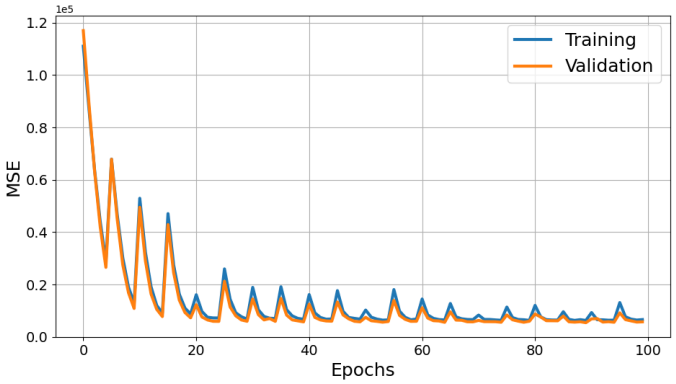


Fig. 5. MSE during training for one node during a sample experiment with 20 updates $\times$ 5 training epochs each. A spike occurs whenever a new model arrives (the most it differs from the local model, the highest).

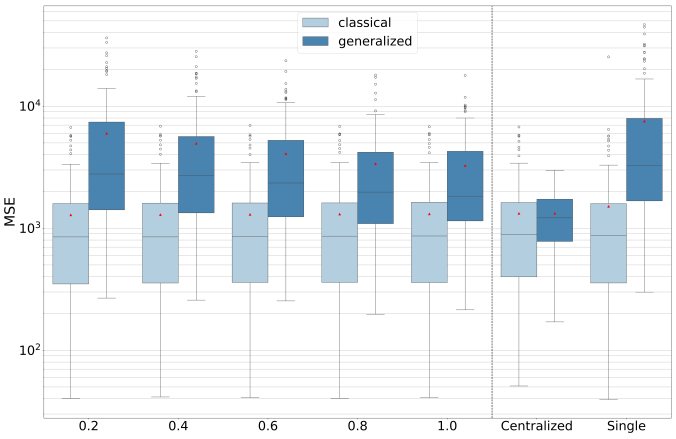


Fig. 6. Comparison between GL with different compression rates and benchmark methods (red triangles: average MMSE across all experiments).

development in each node. As before, while generalization capabilities are negatively affected by model sampling, the MMSE of GL models remains significantly lower than the one achieved by the *single-node* model, which promotes the effectiveness of the paradigm.

*3) Peer Sampling:* One simple method to lower the number of sent messages, thereby reducing network costs, is to decrease the *target sampling probability* $\pi$, i.e., the likelihood that a node will choose to communicate with its target neighbors. Table IV, experiments with $\pi$ values ranging from 0.25 to 1.0 reveal that variations in $\pi$ exert only a marginal influence on the performance of GL, while FL displays higher sensitivity, affecting both *classical* and *generalization* metrics.

*4) Number of merged models:* To evaluate the impact of the PENS-inspired variant described in Section III-C1, we run two experiments considering $N = 3$ merged models, one with no compression and one with 20% weights sampling. In both cases, we observed an improvement in terms of generalization capabilities, while the classical metrics remain roughly unchanged (see Table V).

However, it is relevant to note that using such a technique increases the pressure on the network: since each node has to wait until it receives a message from $N > 1$ (here, 3) different neighbors before triggering an update, the overall

TABLE IV
MSE MEAN AND STANDARD DEVIATION WITH DIFFERENT TARGET SAMPLING PROBABILITIES $\pi$

| Experiment Strategy | $\pi$ | Classical MMSE | Classical STD | Classical RSD | Generalization MMSE | Generalization STD | Generalization RSD |
|---|---|---|---|---|---|---|---|
| GL Age-Weighted Avg. | 25% | 1303 | 1447 | 1.11 | 3304 | 3285 | 0.99 |
| GL Age-Weighted Avg. | 50% | 1309 | 1454 | 1.11 | 3256 | 3189 | 0.98 |
| GL Age-Weighted Avg. | 75% | 1314 | 1462 | 1.11 | 3322 | 3264 | 0.98 |
| GL Age-Weighted Avg. | 100% | 1311 | 1449 | 1.11 | 3276 | 3403 | 1.04 |
| FL FedAvg | 25% | 2835 | 1095 | 0.39 | 2828 | 1102 | 0.39 |
| FL FedAvg | 50% | 2530 | 806 | 0.32 | 2535 | 802 | 0.32 |
| FL FedAvg | 75% | 2728 | 1783 | 0.65 | 2740 | 1809 | 0.66 |
| FL FedAvg | 100% | 2460 | 618 | 0.25 | 2444 | 625 | 0.26 |
| Single | | 1510 | 2768 | 1.83 | 7544 | 9852 | 1.31 |
| Centralized | | 1322 | 1403 | 1.06 | 1324 | 701 | 0.53 |

TABLE V
MSE MEAN AND STANDARD DEVIATION WITH DIFFERENT NUMBERS OF MERGED MODELS

| N. Models | Sampling | Classical MMSE | Classical STD | Classical RSD | Generalization MMSE | Generalization STD | Generalization RSD |
|---|---|---|---|---|---|---|---|
| 1 | 100% | 1309 | 1454 | 1.11 | 3256 | 3189 | 0.98 |
| 3 | 100% | 1312 | 1464 | 1.12 | 2893 | 2601 | 0.90 |
| 1 | 20% | 1285 | 1417 | 1.10 | 5990 | 7345 | 1.23 |
| 3 | 20% | 1295 | 1439 | 1.11 | 4704 | 5446 | 1.16 |

number of the exchanged messages will be at least $N$ times higher than in the standard case ($N = 1$). In any case, note that an $N\times$ estimate may be too optimistic: in general, nodes do not receive *exactly* one message per update from each of its neighbors; instead, multiple messages are exchanged before collecting the $N$ required sets of model weights.

*5) Stopping criteria:* We run a set of experiments to investigate the impact of the two stopping criteria introduced in Section III-C4: *FixedUpdates* and *EarlyStopping* (ES). With respect to the latter we want to investigate whether, from a node perspective, its own validation loss history may be a valid indicator of the overall training progress. In particular, we designed a *conservative EarlyStopping* strategy that requires 5 updates without an improvement of at least 0.1 in validation loss, and an *aggressive EarlyStopping* one, which requires 5 epochs without an improvement of at least 1. We compare it with a *FixedUpdate* strategy where training is performed with 100 fixed updates per node.

As shown in Table VI, even though the *classical* metrics are roughly comparable, models trained with *EarlyStopping* show remarkably worse generalization performance than those trained for more epochs. Based on these results, it is reasonable to assume that validation loss is not a sufficient metric to define an intelligent stopping criterion as it does not take into account the generalization performance of models.

At the same time, we can observe that the generalization capabilities of GL models are still superior to those of a single-node-only model. Since *EarlyStopping*, limiting the number of updates per node helps reduce the network usage and the training time, a good trade-off can be found between the models' accuracy and the overall costs of their development.

## C. Scalability and network costs

Scalability is one of the most relevant properties for distributed approaches. We analyzed how the GL protocol scales

TABLE VI
MSE MEAN AND STD DEVIATION WITH DIFFERENT STOPPING CRITERIA

| Schedule | Classical | | | Generalization | | |
|---|---|---|---|---|---|---|
| | MMSE | STD | RSD | MMSE | STD | RSD |
| *FixedUpdates* | 1309 | 1454 | 1.11 | 3256 | 3189 | 0.98 |
| *Conservative ES* | 1328 | 1451 | 1.09 | 6267 | 8389 | 1.34 |
| *Aggressive ES* | 1334 | 1461 | 1.09 | 6392 | 8082 | 1.26 |

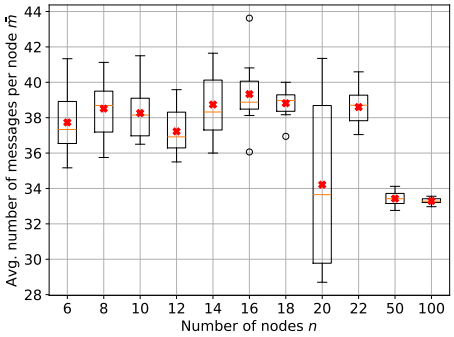


Fig. 7. Average number of messages per node with respect to the number of nodes in the network. Each blue data point represents the $\overline{m}$ value within a single simulation; red crosses represent the average over all the simulations.

to bigger P2P networks by considering as metric the *average number of messages* exchanged *per node* throughout the simulation, denoted by $\overline{m}$. In particular, Sections VI-C1–VI-C4 evaluate whether and how this is influenced by the protocol configuration parameters. Section VI-C5 discusses whether GL can limit, with respect to the *centralized* approach, the network load each node is subject to. All experiments except those in Section VI-C4 were run considering 25 updates per node, and assuming that 4 epochs are trained per update.

*1) Impact of the number of nodes in the network:* The first set of experiments explored the relationship between $\overline{m}$ and the number of nodes $n$, varied between 6 and 22, and two larger networks with 50 and 100 nodes. All experiments were conducted on 3-edge connected P2P networks. As expected, $\overline{m}$ is roughly constant with respect to $n$ (see Fig. 7). However, it is relevant to note that *the observed* value *of* $\overline{m}$ is always significantly higher than the theoretical minimum (which is 25, as the number of updates per node). This discrepancy arises because nodes typically exchange multiple messages before the communication process is effectively concluded.

Another key factor affecting $\overline{m}$ is the number of incoming messages a node discards, either because it is in the `TRAINING` state or because it has already begun the transition to the `STOPPED` state (see Section IV-A). In particular, the number of messages discarded during the transition to `STOPPED` is relatively small, and its impact diminishes, as the number of per-node updates increases. *In contrast, the number of messages discarded during the* `TRAINING` *state* is influenced by the average number of neighbors of each node (i.e., by the $k$-edge connectivity of the network), and by the *ratio* between the training and the model transfer times, as clarified in the next sections.

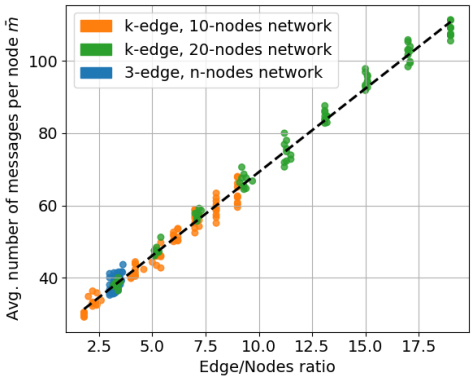


Fig. 8. Average number of messages per node for different network graphs.

*2) Impact of the number of P2P network links:* Since, as mentioned, the number of discarded messages is influenced by the average degree of a network node, it is interesting to see how network costs are impacted by an increase in the number of network links. For this reason, three sets of experiments were performed. We initially considered 10-nodes and 20-nodes networks, respectively, and tuned the $k$ parameter related to the $k$-edge connectivity by considering: *(i)* all values between 1 and 9 for the 10-nodes network; *(ii)* all odd values between 3 and 19 for the 20-nodes network. Then, we kept $k = 3$ fixed and varied $n$ between 5 and 20 as in the previous section.

Note that, as mentioned in Section V-A1, $k$-edge connectivity is guaranteed if each node has *at least* $k$ neighbors. However, the actual number of edges may be slightly different depending on the generated topology. Therefore, to better characterize the network graphs, we will report in the next figures their *edge/nodes ratio* instead of the prescribed $k$ value.

Figure 8 reveals a linear relationship between the ratio of edges to nodes (i.e., the average number of links per node) and $\overline{m}$ across all experimental settings. Moreover, the uniformity of the linear regression coefficients, combined with the analysis in the previous section, leads us to conclude that $\overline{m}$ is independent of the number of nodes in the network and depends solely on the node degree.

*3) Impact of the transfer time and target sampling probability:* We validated the observation that high values of $\overline{m}$ are related to a large number of message rejections by increasing by 10 times the model transfer time (i.e., *High transfer time*), under the assumption that, if this is significantly longer than the training time, the chance of receiving a message while in the `TRAINING` state is lower. Furthermore, since we observe fewer rejections when decreasing the target sampling probability $\pi$, we also run a set of simulations with $\pi = 0.25$ instead of 1 (i.e., *Low neighbor sampling probability*). We considered 20-nodes networks and varied $k$ as done previously.

Results are shown in Figure 9: we observe that the values of $\overline{m}$ decrease more significantly for more dense networks. In this setting, reducing the sample probability $\pi$ proves to be very effective in decreasing the network load.

*4) Impact of the number of model updates:* It is reasonable to assume that $\overline{m}$ grows proportionally to the number of per-node updates within a simulation. To validate this, we run a

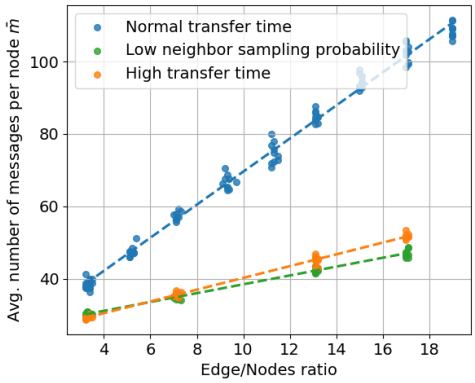


Fig. 9. Average number of messages per node for different transfer times and neighbor sampling probabilities.

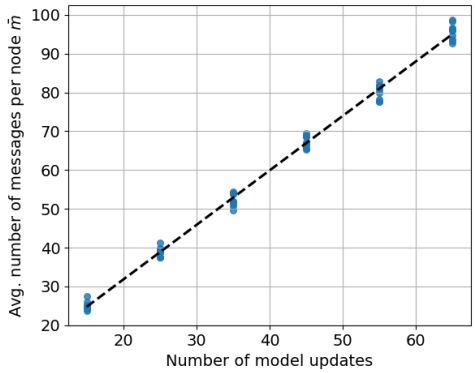


Fig. 10. Average number of messages per node varying the number of updates.

set of experiments with a variable number of updates on a 20-node 3-edge connected network (see Fig. 10). The results of Fig. 10 confirm our assumption.

*5) Comparison with the centralized approach:* We compared the traffic generated by the GL protocol and the *centralized* approach considering a *star* network architecture, where the master node (i.e., centralized server) is directly linked with all the participants. In terms of network costs, this can be considered the best-case scenario for centralized training, as all the data transmissions between master and clients have only to traverse the single link connecting them.

In a centralized scenario, the incoming network cost for a master node connected to $n$ clients is $nD$, where $D$ is the dataset dimension, whereas the outgoing network traffic is $nM$, where $M$ is the model size. Therefore, the overall network cost can be defined as:

$$C_C = C_C^{in} + C_C^{out} = nD + nM = n(\alpha M + M) = n(1+\alpha)M,$$

where $\alpha$ is a scaling factor between model and dataset size (in our experiments, $\alpha$ is $\sim 4.7$).

On the other hand, the average network load for a node participating in GL is given by:

$$\overline{C}_{GL} = \overline{C}_{GL}^{in} + \overline{C}_{GL}^{out} = r\overline{m}M + r\overline{m}M = 2r\overline{m}M,$$

where $r \in (0, 1]$ denotes the compression rate if weights sampling is applied.

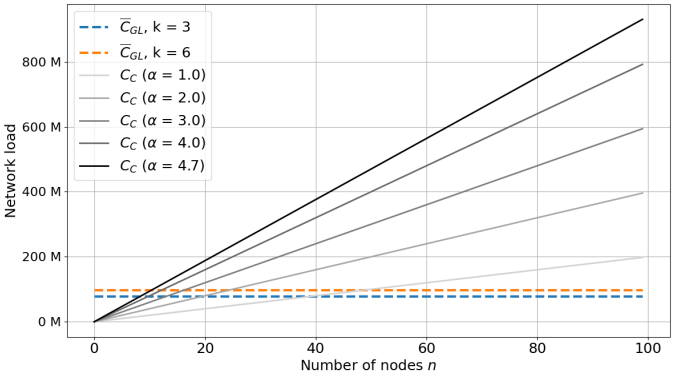


Fig. 11. Comparison between the network load with GL and the centralized approach, expressed as a function of the model size $M$.

As discussed in Section VI-C1, $\overline{m}$ and, consequently, $\overline{C}_{GL}$, remain invariant with respect to the number of nodes $n$. This invariance implies that, irrespective of the GL protocol parameters and associated network costs, there always exists a sufficiently large $n$ for which $\overline{C}_{GL} < C_C$. An example is reported in Fig. 11, where we report the network load computed for: *(i)* a 3-edge connected network (with $r = 1$), *(ii)* a 6-edge connected network (with $r = 1$), and *(iii)* a centralized master node varying the value of $\alpha$.

For both values of $k$ and all $\alpha$, we observe that $C_C$ becomes higher than $\overline{C}_{GL}$ as the number of nodes reaches a given threshold. We can conclude that the GL protocol offers advantages over the centralized model in terms of network load, particularly when dealing with large graphs. While the GL protocol may generate more traffic in smaller networks, it distributes communications more evenly across the network. This reduces the likelihood of bottlenecks, which are more common when all data must be sent from clients to a centralized server.

### D. Node failure resiliency

We assessed the resiliency of our approach to node failures by varying the failure probability. In each simulation, at fixed intervals of $i = 60$, the simulator probabilistically determines whether to trigger a node failure based on the configured probability $p \in \{0.0025, 0.005, 0.01\}$. When a failure occurs, the recovery time is sampled from a normal distribution with mean $\mu = 125$ and standard deviation $\sigma = 5$.

The parameters $i$, $\mu$, and $\sigma$ were chosen by considering the network size (10 nodes), the overall simulated training duration at each round (5 time units), and the model weights transmission time (i.e., $t_w \in [25, 35]$). For instance, with a failure probability of $p = 0.01$, the simulation produces an average of 4 node failures, with each failure causing a node to miss approximately 3–4 training rounds before recovering.

In our experiments, runs with node failures occurred as follows: for $p = 0.0025$, 6 out of 10 runs experienced failures (10 failures in total, averaging 1.0 failure per run); for $p = 0.005$, 9 out of 10 runs were affected (18 failures in total, averaging 1.8 failures per run); and for $p = 0.01$, all 10 runs exhibited failures (53 failures in total, averaging 5.3 failures per run).

TABLE VII
MEAN MSE, STD, AND RSD FOR THE CLASSICAL METRIC FOR ALL THE NODES IN THE NETWORK, NON-FAILED NODES, AND FAILED NODES UNDER DIFFERENT NODE FAILURE PROBABILITIES

| Failure prob. | All nodes | | | Non-failed nodes | | | Failed nodes | | |
|---|---|---|---|---|---|---|---|---|---|
| | MMSE | STD | RSD | MMSE | STD | RSD | MMSE | STD | RSD |
| 0.0025 | 1306 | 90 | 0.07 | 1277 | 206 | 0.16 | 1593 | 1876 | 1.18 |
| 0.005 | 1307 | 87 | 0.07 | 1265 | 314 | 0.25 | 1169 | 646 | 0.55 |
| 0.01 | 1312 | 90 | 0.07 | 1523 | 554 | 0.36 | 1196 | 268 | 0.22 |

TABLE VIII
MEAN MSE, STD, AND RSD FOR THE GENERALIZATION METRIC FOR ALL THE NODES IN THE NETWORK, NON-FAILED NODES, AND FAILED NODES UNDER DIFFERENT NODE FAILURE PROBABILITIES

| Failure prob. | All nodes | | | Non-failed nodes | | | Failed nodes | | |
|---|---|---|---|---|---|---|---|---|---|
| | MMSE | STD | RSD | MMSE | STD | RSD | MMSE | STD | RSD |
| 0.0025 | 2958 | 962 | 0.33 | 2813 | 812 | 0.29 | 5108 | 4835 | 0.95 |
| 0.005 | 3150 | 1074 | 0.34 | 3072 | 1049 | 0.34 | 3332 | 2127 | 0.64 |
| 0.01 | 3227 | 1170 | 0.36 | 3019 | 1538 | 0.51 | 3580 | 1446 | 0.40 |

Tables VII and VIII summarize the performance results using the *classical* and *generalization* metrics, respectively. For each metric, we report aggregated values of the MMSE, STD, and RSD computed across all nodes, non-failed nodes, and failed nodes.

The results are consistent with our initial comparisons with benchmark approaches (see Fig. 4), as there is no significant performance difference between failed and non-failed nodes—demonstrating the notable resilience of our approach. An exception was observed for $p = 0.0025$, where the failed nodes exhibited an unexpected increase in MMSE (and a high STD) for the *generalization* metric. Further investigation revealed that, in one run, the sole failed node produced a substandard model (with a *generalization* MSE of 14299), which skewed the aggregated metrics. Additionally, one of its three neighboring nodes also concluded the simulation with poor performance (a *generalization* MSE of 9829), while the remaining neighbors performed comparably to the non-failed nodes. We hypothesize that this deviation is attributable to the interplay between network topology and random peer selection; however, further experiments are required to validate this hypothesis.

### E. Validation with real function traces

Following the same approach described in Section V-C, we trained GL models on the datasets built upon Microsoft Azure functions traces [50] as outlined in Section V-A. We considered ten randomly-generated 10-node, 3-edge-connected networks and adopted the configuration parameters described in Table II, and the Age-Weighted Average merge strategy.

The results achieved by GL, single-node, and centralized models, both in terms of classical and generalized metrics, are reported in Table IX. The values of *classical* and *generalized* metrics confirm the pattern observed with synthetic traces, with the GL protocol improving the generalization capabilities of the models with respect to the single-node case.

TABLE IX
MSE MEAN AND STANDARD DEVIATION FOR GL, SINGLE-NODE AND CENTRALIZED MODELS TRAINED AND TESTED ON THE AZURE FUNCTIONS DATASET

| Training | Classical | | | Generalization | | |
|---|---|---|---|---|---|---|
| | MMSE | STD | RSD | MMSE | STD | RSD |
| GL | 27188 | 3348 | 0.12 | 37132 | 8693 | 0.23 |
| Single | 26843 | 2229 | 0.08 | 40572 | 9512 | 0.23 |
| Centralized | 24038 | 1991 | 0.08 | 25212 | 958 | 0.04 |

TABLE X
ENERGY CONSUMPTION IN DIFFERENT PHASES OF THE GL PROTOCOL

| Event | Model fit | | | | `SendModel` | | | | `ReceiveModel` | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | mean | std | 25% | 75% | mean | std | 25% | 75% | mean | std | 25% | 75% |
| **Power (W)** | 9.69 | 0.52 | 9.37 | 9.92 | 10.70 | 2.45 | 10.92 | 11.88 | 10.69 | 0.45 | 10.48 | 10.96 |
| **Energy (J)** | 1.93 | 3.68 | 0.67 | 0.75 | 0.16 | 0.07 | 0.14 | 0.19 | 20.74 | 1.55 | 19.88 | 21.69 |
| **Runtime (s)** | 0.19 | 0.34 | 0.07 | 0.08 | 0.01 | 0.01 | 0.01 | 0.02 | 1.93 | 0.12 | 1.86 | 1.97 |

Additionally, Figure 12 reports the real and predicted workload values for a sample experiment, obtained by one node participating in the GL protocol. The predictions align with both the training and test traces, as well as with the *common test trace* used to evaluate the generalized metrics. Although peak values are not correctly identified by the model in most cases, the predictions capture well the moving average of the distributions, which is represented in the figure by the green line, and closely follow the workload pattern. The same behavior can be observed in Figure 13, which reports the real and predicted traces for the common test set obtained with the single-node model and the centralized model.

### F. Energy and resource profiling

We leveraged *carbontracker* [48] to monitor the power (in Watt) and energy (in Joule) required by specific node operations throughout the GL simulation. In particular, we focused on the model training, collecting metrics for each epoch, and on the `SendModel` and `ReceiveModel` events (see Subsection IV-B), gathering the power and energy consumption related to each event processing. We run simulations considering 30 updates per node, and running 10 training epochs per update. We chose to increase the number of epochs per update with respect to the experiments reported in Subsection VI-E to ensure that the time spent by nodes in the training and model update phases is large enough to guarantee stable measurements. We trained the models considering the dataset based on Azure functions traces, running on a MacBook Pro M3 with 11 cores and 18Gb RAM.

The average, standard deviation and quantiles of the collected metrics per each model fit, send model and receive model call are reported in Table X. As in the previous sections, they were obtained by considering 10 randomly-generated networks. Throughout the overall simulation, which corresponds to 300 training epochs and around 30 communication rounds, each node consumes (on average) $2906.29W$ for model training, $310.30W$ to send the model to peers, and $320.75W$ in the receive model routine. These can be used to estimate the overall $CO_2$ emissions by considering the average carbon intensity where simulations were conducted

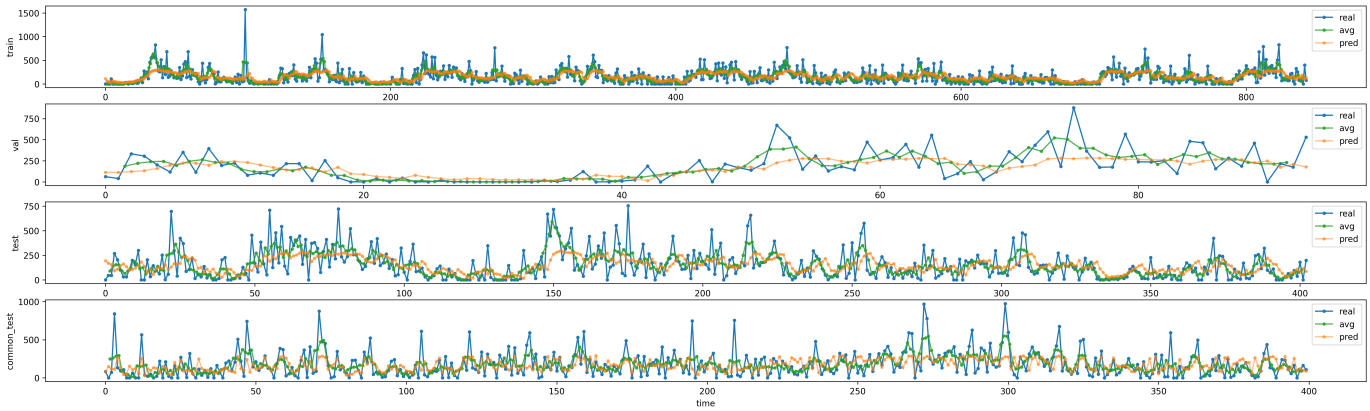


Fig. 12. Real and predicted workload values for a sample node participating in the GL protocol, considering the training, validation and test trace, and the common test trace used to evaluate the generalized metrics. The green line represents a moving average of the real values over 4-step windows.

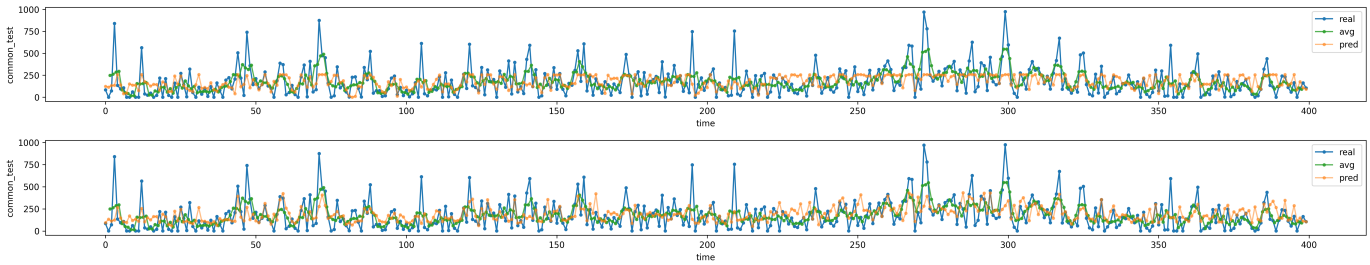


Fig. 13. Real and predicted workload values for the single-node (above) and the centralized model (below), considering the common test trace used to evaluate the generalized metrics. The green line represents a moving average of the real values over 4-step windows.

(in our case, Monza, Lombardy, IT, had an average intensity of $330.72 gCO_2/kWh$).

Throughout the simulations, we have also gathered information on the CPU and RAM consumption, leveraging the *scalene* [49] profiler. In particular, we have observed that nodes spend most of their time in conducting the model training phase (almost 90% of the overall runtime). This is characterized by an average CPU utilization of $79.48\%$ (with a standard deviation of $13.70$) and consumes on average $27.82 Mb$ of RAM (with a standard deviation of $0.85$); the peak RAM consumption is of $41.45 Mb$ on average (standard deviation: $3.78$), which guarantees that the training process is tractable even for small-scale edge devices.

## VII. Related Work

This section recalls the main related work in the context of time series forecasting and edge workload forecasting, with main focus on decentralized learning.

### A. Time-series analysis and prediction

The field of edge workload forecasting is based on the principles of time-series analysis and prediction, a subject that has been extensively researched and documented in the academic literature for a considerable period of time [55]. In general, classical approaches (e.g., ARIMA [56]) for time-series prediction aim to forecast a single time series by studying linear relationships between observations collected in the past and future observations. Typically, this linear approach yields suboptimal predictions [55]. In recent times, there has been a notable increase in the use of neural network models, largely due to their capacity to discern non-linear relationships. For example, Recurrent Neural Network (RNN) models, such as LSTM [57] and Gated Recurrent Unit (GRU) [58], are widely used. However, these ML models often require greater time and hardware resources for training purposes [55]. In this paper we adopted LSTM given the good performance obtained in some preliminary experiments.

### B. ML-based workload forecasting at the Edge

Recently, research efforts have been made on the use of ML techniques specifically for workload forecasting in edge environments. For instance, Miao *et al.* [59] present an approach based on Graph Neural Networks to capture the interconnected topology of edge servers, assuming that servers in close proximity often exhibit similar workload patterns. Ma *et al.* [55] propose an edge-cloud collaboration as a means of capturing inter-site correlations while simultaneously advancing a distributed approach wherein the majority of the computation is conducted at the Edge. Some recent work has also adopted the Federated Learning paradigm as a means to collaboratively train a model for workload forecasting at the Edge without sharing local data. For instance, Que *et al.* [60] focus on this aspect in the context of Metro Optical Networks. However, these studies still either perform model training in a single location or need a central coordination, as in the case of Federated Learning. Our approach completely eliminates

the need for a central location and fully decentralizes model training at the Edge.

### C. Decentralized training solutions

When dealing with the need for decentralized training, some studies investigate the possibility of adapting FL to a decentralized setting [16], [61], [62]. Wink *et al.* [62] propose a P2P variant of FL that can enhance data confidentiality and privacy protection, particularly in systems that cater to smaller groups of collaborating data owners. In the proposed approach, peers do not exchange model weights; rather, they collaboratively perform Secure Average Computation to obtain the merged models. Kalra *et al.* [61] propose ProxyFL, a proxy-based FL method for decentralized collaboration that facilitates the training of robust and high-performance models while maintaining data privacy and communication efficiency. Zhou *et al.* [16] present PPAFL, a P2P-based privacy-perceiving asynchronous FL framework for the decentralized training of secure and resilient mobile robotic systems. These works built on top of a seminal paper by Hegedus *et al.* [15], which proposes Gossip Learning as a promising decentralized alternative to FL [43]. Starting from this body of research, our work aims at adopting decentralized learning [14], with particular focus on GL, in the context of workload forecasting at the Edge, while incorporating tailored adaptations to this use case.

### D. Gossip Learning adaptations

Our proposed extensions and adaptations to GL are related to four main aspects: *(i)* how the received models are merged, *(ii)* how the peers are selected for sending model updates, *(iii)* how the model can be compressed to reduce communication overhead and *(iv)* when the protocol should stop its execution.

In literature, some different merging strategies have been proposed, being either parameter-based [38] [15], or gradient-based [39]. In our paper we adopt the merging strategies designed in [15], and we propose a novel strategy, i.e., *Age-Weighted Overwrite*, that incentivize more stable models. In addition, inspired by [23], we enable the simultaneous merging of multiple models. Different peer selection strategies have also been proposed, based e.g. on topology-aware selection and node prioritization [35]–[37]. In this paper we instead adopt an *active-peer random selection* strategy, which better suits to dynamic network conditions as those occurring in the considered scenario. Model compression has been investigated in some papers [15], [43], [47]: building upon them, we propose *subsampling* as a mean to achieve model compression. Finally, no previous work on GL focuses on the problem of protocol execution's stopping to prevent overfitting. In this paper we propose two strategies, i.e., *FixedUpdates* and *EarlyStopping*, with the latter adapted from [44].

### E. Time series forecasting with Gossip Learning

The application of GL to the forecasting of time-series data has been investigated by few works in literature. Dinati *et al.* [63] focus on vehicle trajectory prediction. Their scenario is similar to the one considered in this work, as it accounts for dynamic network topologies. However, our study introduces the extensions reported in Section VII-D and extends the evaluation to aspects related to scalability and network costs. Palmieri *et al.* [64] investigate the impact of different network topologies on the capacity of nodes to incorporate knowledge derived from data patterns observed in other nodes into their local models. Their work provides a valuable complement to our experiments on the impact of network size, focusing on the impact of network topology variation on GL.

## VIII. Conclusion

This paper addressed the workload forecasting problem in a fully-decentralized edge environment, leveraging the Gossip Learning paradigm to mitigate challenges arising from geographical distribution, workload pattern heterogeneity, and the dynamic nature of nodes, which join and leave the network due to unstable links and failures. Building on existing peer-to-peer learning protocols, we enhanced the foundational Gossip Learning algorithm by introducing a performance-driven selection mechanism that prioritizes models received from neighboring nodes based on their quality. Additionally, we proposed a novel merging strategy and integrated two distinct stopping criteria into the protocol to mitigate overfitting. Extensive experimental validation, conducted using the discrete-event simulator developed in this work, demonstrated the effectiveness of Gossip Learning and its extensions in a representative P2P edge environment. Our results show that Gossip Learning not only achieves competitive performance against benchmark methods (i.e., centralized, single-node learning, and Federated Learning) but also exhibits robustness and scalability across diverse protocol and network conditions.

## References

[1] X. Kong, Y. Wu, H. Wang, and F. Xia, "Edge computing for internet of everything: A survey," *IEEE Internet of Things Journal*, vol. 9, no. 23, pp. 23 472–23 485, 2022.

[2] C. N. L. Tan, C. Klein, and E. Elmroth, "Multivariate LSTM-Based Location-Aware Workload Prediction for Edge Data Centers," in *IEEE/ACM CCGRID*, 2019.

[3] C. Cicconetti, M. Conti, and A. Passarella, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 2166–2180, 2021.

[4] I. Wang, E. Liri, and K. K. Ramakrishnan, "Supporting iot applications with serverless edge clouds," in *IEEE CloudNet*, 2020.

[5] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. D. de Assunção, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *ACSW*, 2021.

[6] E. Paraskevoulakou and D. Kyriazis, "Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum," in *ICIN*, 2021.

[7] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *SIGMOD*, 2021.

[8] P. G. Sarroca and M. S. Artigas, "Mlless: Achieving cost efficiency in serverless machine learning training," *J. Parallel Distributed Comput.*, vol. 183, p. 104764, 2024.

[9] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will serverless computing revolutionize nfv?" *Proc. IEEE*, vol. 107, no. 4, pp. 667–678, 2019.

[10] M. Savi, A. Banfi, A. Tundo, and M. Ciavotta, "Serverless computing for NFV: is it worth it? A performance comparison analysis," in *IEEE PerCom Workshops*, 2022.

[11] A. Sabbioni, A. Garbugli, L. Foschini, A. Corradi, and P. Bellavista, "Serverless computing for qos-effective NFV in the cloud edge," *IEEE Commun. Mag.*, vol. 62, no. 4, pp. 40–46, 2024.

[12] V. Karagiannis, A. Venito, R. Coelho, M. Borkowski, and G. Fohler, "Edge computing with peer to peer interactions: use cases and impact," in *IoT-Fog*, 2019.

[13] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Clust. Comput.*, vol. 23, no. 4, pp. 2399–2424, 2020.

[14] L. Yuan, Z. Wang, L. Sun, P. S. Yu, and C. G. Brinton, "Decentralized federated learning: A survey and perspective," *IEEE Internet of Things Journal*, vol. 11, no. 21, pp. 34 617–34 638, 2024.

[15] I. Hegedűs, G. Danner, and M. Jelasity, "Gossip learning as a decentralized alternative to federated learning," in *DAIS*, 2019.

[16] X. Zhou, W. Liang, K. I. Wang, Z. Yan, L. T. Yang, W. Wei, J. Ma, and Q. Jin, "Decentralized P2P federated learning for privacy-preserving and resilient mobile robotic systems," *IEEE Wirel. Commun.*, vol. 30, no. 2, pp. 82–89, 2023.

[17] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *AISTATS*, A. Singh and X. J. Zhu, Eds., vol. 54, 2017, pp. 1273–1282.

[18] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, "Dfaas: Decentralized function-as-a-service for federated edge computing," in *IEEE CloudNet*, 2021.

[19] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella, "Toward distributed computing environments with serverless solutions in edge systems," *IEEE Commun. Mag.*, vol. 58, no. 3, pp. 40–46, 2020.

[20] C. Cicconetti, M. Conti, and A. Passarella, "Architecture and performance evaluation of distributed computation offloading in edge computing," *Simul. Model. Pract. Theory*, vol. 101, p. 102007, 2020.

[21] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *ICFC*, 2020.

[22] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Springer, 2017.

[23] N. Onoszko, G. Karlsson, O. Mogren, and E. L. Zec, "Decentralized federated learning of deep neural networks on non-iid data," *arXiv preprint arXiv:2107.08517*, 2021.

[24] R. Ormándi, I. Hegedűs, and M. Jelasity, "Asynchronous peer-to-peer data mining with stochastic gradient descent," in *Euro-Par*, 2011.

[25] M. Polato, "gossipy documentation," 2024, accessed 27.07.2024. [Online]. Available: https://makgyver.github.io/gossipy/index.html

[26] A. Tundo, F. Filippini, F. Regonesi, M. Ciavotta, and M. Savi, "Github repository," 2024, accessed 13.05.2025. [Online]. Available: https://github.com/unimib-datAI/gl-forecasting-edge

[27] ——, "Zenodo repository," 2024, accessed 13.05.2025. [Online]. Available: https://doi.org/10.5281/zenodo.15393791

[28] M. Savi and F. Olivadese, "Short-Term Energy Consumption Forecasting at the Edge: A Federated Learning Approach," *IEEE Access*, vol. 9, pp. 95 949–95 969, 2021.

[29] V. Perifanis, N. Pavlidis, R.-A. Koutsiamanis, and P. S. Efraimidis, "Federated learning for 5G base station traffic forecasting," *Computer Networks*, vol. 235, 2023.

[30] H. G. Abreha, M. Hayajneh, and M. A. Serhani, "Federated learning in edge computing: A systematic survey," *Sensors*, vol. 22, no. 2, 2022.

[31] R. Ormándi, I. Hegedűs, and M. Jelasity, "Gossip learning with linear models on fully distributed data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[32] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8–es, 2007.

[33] M. Assran, J. Romoff, N. Ballas, J. Pineau, and M. Rabbat, "Gossip-based actor-learner architectures for deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[34] D. Kempe, J. Kleinberg, and A. Demers, "Spatial gossip and resource location protocols," *Journal of the ACM*, vol. 51, no. 6, pp. 943–967, 2004.

[35] J. Zhang, L. Zhao, and N. Lin, "Similarity-based gossip learning for generative adversarial networks," in *ICCSN*, 2023.

[36] L. Giaretta and Š. Girdzijauskas, "Gossip learning: Off the beaten path," in *IEEE Big Data*, 2019.

[37] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya, "Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent," *arXiv preprint arXiv:1803.05880*, 2018.

[38] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," in *MLSys*, 2020.

[39] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *ICML*, 2018.

[40] Y. Wang, T. Yu, and K. Sakaguchi, "Context-based MEC platform for augmented-reality services in 5g networks," in *IEEE VTC*, 2021.

[41] I. Lujic, V. D. Maio, K. Pollhammer, I. Bodrozic, J. Lasic, and I. Brandic, "Increasing traffic safety with real-time edge analytics and 5g," in *EdgeSys*, 2021.

[42] Y. Gou, S. Weng, M. A. Imran, and L. Zhang, "Voting consensus-based decentralized federated learning," *IEEE Internet of Things Journal*, vol. 11, no. 9, pp. 16 267–16 278, 2024.

[43] I. Hegedüs, G. Danner, and M. Jelasity, "Decentralized learning works: An empirical comparison of gossip learning and federated learning," *J. Parallel Distributed Comput.*, vol. 148, pp. 109–124, 2021.

[44] T. Zhang, T. Zhu, K. Gao, W. Zhou, and P. S. Yu, "Balancing learning model privacy, fairness, and accuracy with early stopping criteria," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 9, pp. 5557–5569, 2023.

[45] H. B. McMahan, F. Yu, P. Richtarik, A. Suresh, D. Bacon *et al.*, "Federated learning: Strategies for improving communication efficiency," in *NeurIPS*, 2016.

[46] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in *AISTATS*, 2020.

[47] H. Wang, S. Guo, Z. Qu, R. Li, and Z. Liu, "Error-compensated sparsification for communication-efficient decentralized training in edge environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 14–25, 2022.

[48] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," in *ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems*, 2020.

[49] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with Scalene," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/berger

[50] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020.

[51] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How does it function? characterizing long-term trends in production serverless workloads," in *SoCC*, 2023.

[52] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, 1997.

[53] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are transformers effective for time series forecasting?" in *AAAI*, 2023.

[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[55] X. Ma, M. Xu, Q. Li, Y. Li, A. Zhou, and S. Wang, "Edge workload prediction based on deep learning," in *5G Edge Computing: Technologies, Applications and Future Visions*, 2024, pp. 45–61.

[56] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[57] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, 11 1997.

[58] R. Zhao, D. Wang, R. Yan, K. Mao, F. Shen, and J. Wang, "Machine health monitoring using local feature-based gated recurrent unit networks," *IEEE Trans. Ind. Electron.*, vol. 65, no. 2, pp. 1539–1548, 2018.

[59] W. Miao, Z. Zeng, M. Zhang, S. Quan, Z. Zhang, S. Li, L. Zhang, and Q. Sun, "Workload prediction in edge computing based on graph neural network," in *IEEE ISPA/BDCloud/SocialCom/SustainCom*, 2021.

[60] C. Que and F. N. Khan, "A scalable federated learning-based approach for accurate traffic prediction in edge computing-enable metro optical network," *Computers & Industrial Engineering*, p. 111004, 2025.

[61] S. Kalra, J. Wen, J. C. Cresswell, M. Volkovs, and H. R. Tizhoosh, "Decentralized federated learning through proxy model sharing," *Nature communications*, vol. 14, no. 1, p. 2899, 2023.

[62] T. Wink and Z. Nochta, "An approach for peer-to-peer federated learning," in *IEEE/IFIP DSN Workshops*, 2021.

[63] M. A. Dinani, A. Holzer, H. Nguyen, M. A. Marsan, and G. Rizzo, "Gossip learning of personalized models for vehicle trajectory prediction," in *IEEE WCNC Workshops*, 2021.

[64] L. Palmieri, L. Valerio, C. Boldrini, and A. Passarella, "The effect of network topologies on fully decentralized learning: a preliminary investigation," in *NetAISys*, 2023.