Totoro$^+$: An Adaptive and Scalable Edge Federated Learning System

Cheng-Wei Ching, Xin Chen, Taehwan Kim, Jian-Jhih Kuo, *Member, IEEE*, Dilma Da Silva, and Liting Hu

***Abstract*—Federated Learning (FL) is an emerging distributed machine learning (ML) technique that enables in-situ model training and inference on decentralized edge devices. We propose Totoro$^+$, a novel scalable FL system that enables massive FL applications to run simultaneously on edge networks. The key insight is to explore a distributed hash table (DHT)-based peer-to-peer (P2P) model to re-architect the centralized FL system design into a fully decentralized one. In contrast to previous studies where many FL applications shared one centralized parameter server, Totoro$^+$ assigns a dedicated parameter server to each application. Any edge node can act as any application's coordinator, aggregator, client selector, worker (participant device), or any combination of the above, thereby radically improving scalability and adaptivity. Totoro$^+$ introduces three innovations to realize its design: a locality-aware P2P multi-ring structure, a publish/subscribe-based forest abstraction, and a game-theoretic path planning model with a guarantee of an ϵ-approximate Nash equilibrium. Real-world experiments on 500 Amazon EC2 servers show that Totoro$^+$ scales gracefully with the number of FL applications and N edge nodes speeds up the total training time by $1.2\times - 14.0\times$, achieves $\mathcal{O}(\log N)$ hops for model dissemination and gradient aggregation with millions of nodes, and efficiently adapts to the practical edge networks and churns.**

***Index Terms*—Distributed and parallel systems for machine learning, federated learning, game theory, and edge computing.**

Received 7 July 2025; revised 13 March 2026; accepted 22 May 2026. Date of publication 25 May 2026; date of current version 5 June 2026. This work was supported in part by the National Science Foundation under Grant NSF-OAC-2313738, Grant NSF-CAREER-2313737, and Grant NSF-CNS-2322919, and in part by the National Science and Technology Council under Grant 113-2221-E-194-040-MY3, and Grant 114-2628-E-194-002-MY3. An earlier version of this paper was presented in part at the Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24) [DOI: 10.1145/3627703.3629575]. Recommended for acceptance by A. C. Zhou. *(Corresponding author: Liting Hu.)*

Cheng-Wei Ching and Liting Hu are with the Department of Computer Science and Engineering, University of California, Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: cching1@ucsc.edu; liting@ucsc.edu).

Xin Chen is with the Department of Computer Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: xchen384@gatech.edu).

Taehwan Kim is with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: tk020@vt.edu).

Jian-Jhih Kuo is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Minhsiung 621301, Taiwan, and also with the Advanced Institute of Manufacturing with High-tech Innovations, National Chung Cheng University, Chiayi 621, Taiwan (e-mail: lajacky@cs.ccu.edu.tw).

Dilma Da Silva is with Texas A&M University, College Station, TX 77843 USA (e-mail: dilma@cse.tamu.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2026.3696917, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2026.3696917

I. Introduction

With the rise of 5G and the growth of connected devices, federated learning (FL) enables local data processing and machine learning at the network edge. This approach reduces data transmission and enhances privacy by eliminating the need for centralized servers. FL has been applied in various domains, including human activity prediction [2], [3], sentiment analysis [4], language processing [5], [6], and enterprise systems [7].

The problem: We consider a typical edge computing architecture where millions of edge devices (e.g., smart wearables, autonomous vehicle sensors) connect to the cloud through an edge layer. This layer comprises hundreds of thousands of server-grade machines, gateways, and routers—termed "edge nodes", managed by various providers across geographically distributed sites. Raw data is collected and stored on these edge nodes, enabling machine learning models to be trained locally without transferring raw data to a central server.

Federated learning (FL) tools and frameworks—such as TensorFlow Federated [8], LEAF [9], FLOAT [10], REFL [11], and PySyft [12]—have gained traction. However, building an efficient FL system on practical edge networks remains challenging due to dynamic edge environments and the high scalability demands of emerging FL applications.

Fig. 1 illustrates a use-case scenario that highlights these challenges. In future intelligent transportation systems such as the efforts currently funded by the US Department of Transportation [13], autonomous vehicles are interconnected and equipped with wide-area network access They continuously collect sensor and behavioral data such as speed, engine performance, driving patterns, and environmental conditions. On the back end, numerous FL applications will run concurrently on these vehicles or edge nodes, performing in-situ training using the collected data. Examples of such FL applications include driver behavior prediction, which trains Long-Short Term Memory (LSTM) networks [14] to anticipate lane change actions [15]; traffic pattern analysis, which learns traffic patterns to redirect optimal routes [16]; and anomaly detection, which employs federated learning to identify and respond to vehicle malfunctions or hazardous road conditions in real time [17].

Our solution:

Totoro$^+$: We present Totoro$^+$, an adaptive and scalable federated learning (FL) system designed to support large-scale concurrently running FL applications on edge networks and adapt to edge dynamics. Rather than modifying existing FL systems, we adopt a fully decentralized architecture. Unlike prior approaches where multiple FL applications rely on shared centralized parameter servers, Totoro$^+$ assigns each application

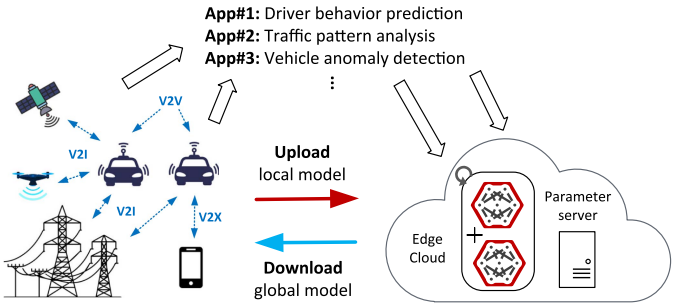


Fig. 1. An edge federated learning use case based on autonomous vehicles.

its own parameter server, preventing overload on any single edge node. Moreover, any edge node can dynamically serve as an application's coordinator, aggregator, client selector, worker, or any combination of these roles, significantly enhancing scalability and adaptivity.

The key idea is to leverage a distributed hash table (DHT)-based peer-to-peer (P2P) architecture to redesign FL systems. P2P models, widely used in file sharing (e.g., BitTorrent [18], Storj [19], Freenet [20]), peer-assisted CDNs [21], and blockchains [22], treat all nodes equally, without a central server, as they collaborate to perform tasks or deliver services. For instance, in BitTorrent [18], users both download and upload file pieces to/from peers. Inspired by this model, we enable all edge nodes to jointly handle FL training and testing for a large number of applications in parallel.

Totoro$^+$ introduces three innovations to realize its fully decentralized design: *a locality-aware P2P multi-ring structure*, *a publish/subscribe-based forest abstraction*, and *a game-theoretic path planning model*.

First, edge nodes self-organize into a scalable DHT-based P2P overlay. The *locality-aware P2P multi-ring structure* enforces administrative boundaries between edge sites, enabling efficient local FL execution and preserving site-specific control flows. Second, Totoro$^+$ builds *a publish/subscribe-based forest abstraction* on top of the locality-aware P2P multi-ring structure for model broadcasting and gradient aggregation. This decouples the centralized FL architecture into a "fully" decentralized architecture, where each FL application operates on an independent dataflow tree, significantly enhancing system scalability. Besides individual dataflow trees for FL applications, we introduce an advertise-discover (AD) tree that enables edge nodes to publish and locate FL applications within the overlay. Third, to address edge dynamics (e.g., link failures, stragglers, and bandwidth constraints over edge networks) Totoro$^+$ employs *a game-theoretic path planning model* that dynamically replans data transfer paths, ensuring robust and adaptive model broadcast and gradient aggregation.

Summary of results: We implement Totoro$^+$ using the open-source Pastry DHT [23] and PyTorch [24] frameworks, [1] and evaluate it on various FL tasks with real-world datasets across 500 Amazon EC2 servers. Compared to state-of-the-art FL systems [25], [26], Totoro$^+$ achieves superior scalability and load balancing, efficiently distributing masters across large-scale edge networks without overloading individual nodes. It accelerates total training time by $1.2\times$ to $14.0\times$, supports $\mathcal{O}(\log N)$ hops for model dissemination and gradient aggregation at million-node scales, and robustly handles network churn and unreliable edge conditions.

[1]https://github.com/UCSC-ELVES-Lab/Totoro-EuroSys-2024

TABLE I
COMPARISON BETWEEN TOTORO AND TOTORO$^+$, WHERE I_{KL} IS THE ITERATION COMPLEXITY OF SOLVING THE KULLBACK–LEIBLER DIVERGENCE-BASED CONVEX FEASIBILITY, AND *MATMUL* IS MATRIX MULTIPLICATION

Features	Totoro [1]	Totoro$^+$ (this work)
Application advertisement and discovery	Join-by-AppId	Advertise-discover tree
Path planning model	Bandit model without traffic congestion	Game-theoretic model with traffic congestion
Algorithmic complexity	$\mathcal{O}(\log N \cdot I_{KL})$	$\mathcal{O}(\log N \cdot Matmul)$
Theoretical guarantee	Heuristic	ϵ-approximate Nash equilibrium
Failure recovery	Worker	Worker and master

Contributions: This paper makes the following contributions:

- *Problem:* We analyze the software architecture of current FL systems and identify key challenges in applying FL to practical edge networks.
- *Key idea:* Totoro$^+$ explores the DHT-based P2P model to propose a novel, fully decentralized "many masters/many workers" architecture that significantly enhances scalability and adaptability.
- *Totoro$^+$ design & implementation:* Totoro$^+$ incorporates three key innovations: a locality-aware P2P multi-ring structure, a publish/subscribe-based forest abstraction, and a game-theoretic path planning model with theoretical guarantees of an ϵ-approximate Nash equilibrium.
- *Results:* Our evaluation demonstrates Totoro$^+$'s substantial improvements in scalability and adaptability over state-of-the-art FL systems.

A preliminary conference paper version [1] has been published in Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24). The summary of changes is presented in TABLE I. In this journal version, we provide the following three significant improvements: First, we introduce the advertise-discover (AD) tree to the publish/subscribe-based forest abstraction. The AD tree enables edge nodes to advertise and discover FL applications running over the DHT-based P2P overlay for FL application searching and crowdsourcing [27], [28] (see in the supplementary material for a detailed comparison with the conference version).

Secondly, we replace the bandit-based path planning model with the game-theoretic path planning model, which formalizes the path planning problem as a congestion game with bandit feedback and introduces a game-theoretic distributed hop-by-hop algorithm with the guarantee of an ϵ-approximate Nash equilibrium (see in the supplementary material for a detailed comparison with the conference version).

Thirdly, we provide respective failure recovery mechanisms for master and worker node failures. When a worker node fails, its child node sends a `JOIN` message using AppId as the key to find a new parent node to recover the dataflow tree. The master node in each communication round replicates the training state across k nodes in its neighborhood set. When the master node fails, its immediate node sends a `JOIN` message using AppId as the key to find an alternative master node. The new master node retrieves a state replica to proceed with the training process.

Finally, we demonstrate that the game-theoretic path planning model is a better solution for adapting to edge dynamics with

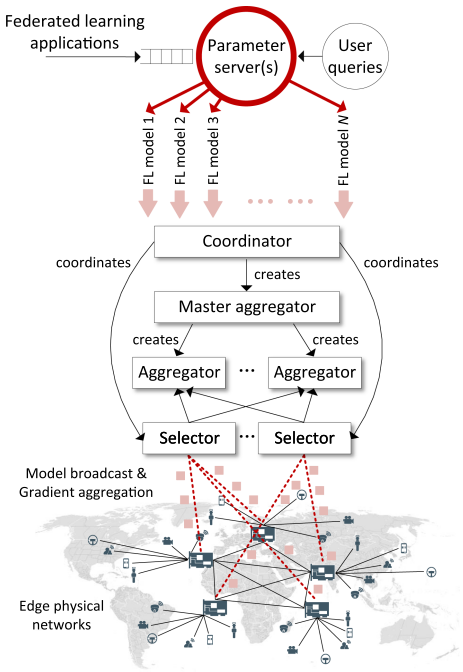


Fig. 2. The data pipeline in a typical FL framework.

experiments on packet latency, and Nash regret. Also, we provide more extensive experiment results to demonstrate further Totoro$^+$'s scalability to massive FL applications, fault tolerance to simultaneous node failures, and adaptivity to edge dynamics.

II. Background

We start with a quick primer on the software architecture used in state-of-the-art federated learning (FL) frameworks.

State-of-the-art FL frameworks (e.g., Meta's PAPAYA [29], LinkedIn's FLINT [30], Google's federated learning framework [31], IBM Federated Learning framework [7], and Apple's federated task processing system [32]) commonly adopt a centralized or hierarchical "single master/many workers" architecture. In this architecture, the master is typically deployed on a parameter server, acting as a coordinating service provider without data. The workers, on the other hand, connect to numerous edge devices and serve as both the data owners and beneficiaries of federated learning.

Fig. 2 illustrates the data pipeline between these components in a typical FL framework.

The high-level design involves two parts: the parameter server that runs the "master", and the end-user devices that run the "workers". The parameter server comprises three main components: Coordinator, Selector, and Aggregator. While the number of Selectors and Aggregators can scale elastically based on the workload demand, there is only one Coordinator. We summarize the functions of these components as follows.

Coordinator: The Coordinator performs three main functions:

- *Task Assignment*. Whenever a new FL application is submitted to the system, the Coordinator assigns the application to a single Aggregator based on the workload among Aggregators and the estimated workload of the application such as application concurrency and model size.
- *Client Assignment*. For each available client, the Coordinator constructs a list of eligible applications. The Coordinator assigns each client to an application.
- *Task Migration*. The Coordinator moves applications between Aggregators when it detects failed or overloaded Aggregators.

Aggregator: The Aggregators are *persistent and stateful* to avoid a substantial cold start overhead for a new application. Each Aggregator is responsible for one single FL application and carries out three main responsibilities.

- *Gradient Aggregation:* Once a client completes training, it uploads the trained serialized gradient update to the server. This update is then pushed into an in-memory queue on the Aggregator, which aggregates client gradient updates to produce new versions of the server model of an application.
- *Client Guidance:* The Aggregator guides clients into running the client protocol, such as downloading, uploading, and training configurations, by responding clients requests from the Selector.
- *Client Tracking:* The Aggregator is responsible for tracking (i) if clients are satisfied with their assigned applications, and (ii) if clients are still eligible for their assigned applications. The tracking information will be used by the Coordinator to run client assignments.

Selector: The Selector is the only component that directly communicates with clients and plays two roles:

- *Task Advertisement:* The clients check in with the Selector and report their eligibility for available applications. The Selector summarizes client availability for the Coordinator.
- *Request Forwarding:* When a client is assigned an eligible application. The Selector forwards the client to the Aggregator responsible for that application. The Selector also routes clients' requests to the corresponding Aggregator, such as model broadcasting, and reporting client status and gradient updates.

III. Challenges

We highlight the challenges that we face when applying FL to real-world edge networks in this section.

Challenge #1: Scaling gracefully with the number of diverse FL applications and edge nodes.

As emerging edge applications and edge devices grow in quantity and complexity, the number of FL applications submitted to the edge will likely become *huge*. As shown in Fig. 1, different FL applications may require training of various FL models for different driver profiles (e.g., speeds, distances, passengers), vehicle conditions, and environmental factors (e.g., intersections, pedestrians, moving objects) simultaneously based on the same raw data. This results in the generation of a vast number of FL tasks. Therefore, key considerations in addressing this challenge include:

Distributed task management: As shown in Fig. 2, existing production FL systems typically rely on a single instance of Coordinator to direct Aggregators and Selectors across all FL training and testing tasks. While this hierarchical design scales well in datacenter environments, it struggles in edge systems with millions of nodes and many concurrent FL applications. The challenge is amplified by the presence of multiple edge providers [33], each managing its own set of edge nodes, resulting in no global view of workloads and resources. Without this global view, existing FL systems cannot balance FL tasks effectively across the edge, hindering their ability to scale with

new nodes and applications and increasing overall training time. Besides longer training time, existing FL systems may fail to advertise the existing/new FL tasks to new/existing clients to further strengthen model accuracy without a global view of running applications.

Application-specific customization: As FL applications diversify with emerging use cases, flexible designs for participant selection [34], [35], [36], compression [37], [38], and communication protocols—synchronous [39], semi-synchronous [40], or asynchronous [41]—are increasingly needed. However, existing FL systems typically share a single parameter server across applications and enforce fixed FL policies, limiting their ability to support diverse requirements. Achieving application-specific customization often requires using multiple FL frameworks, which compromises efficiency, maintainability, and simplicity.

Challenge #2: Adapt to practical edge network conditions such as varying bandwidths, unreliable links, high churn, and workload surges.

The second challenge arises from the first one. To scale with massive FL applications, cloud administrators usually partition all nodes into many sets and assign a parameter server per each set of nodes in a static manner (e.g., one parameter server per rack) [10], [11], [26]. While this assignment approach may work well in datacenters, it lacks the agility to adapt quickly to edge platforms that have millions of resource-constrained nodes.

Edge environments are characterized by the following unique difficulties: (1) edge network link delays are unpredictable and vary stochastically due to unreliable links and random access protocols (e.g., in wireless networks [40], [42]), client mobility (e.g., in mobile ad-hoc networks [27], [43]), and randomness of demand (e.g., workload surges [44], [45]) in arbitrary (and dynamic) geographical edge locations, and (2) edge nodes fail or lag unexpectedly (e.g., due to signal attenuation, interference, and wireless channel contention [40], [42]). However, unlike datacenter servers, edge nodes have limited computing resources (few-core processors, little memory, and little permanent storage [44]) and no backpressure [46]. As such, there is little room to adapt to the edge dynamics or handle stragglers by over-provisioning resources or replicating links like previous studies. Moreover, edge nodes may belong to different owners who do not share operation or application information and have to compete for communication resources over edge networks [47], [48]. Therefore, efforts should be made to replan the data transfer paths *dynamically* and *autonomously* to adapt to the edge dynamics and constrained network bandwidth.

IV. TOTORO+ DESIGN

Totoro+ proposes a novel scalable FL system for practical edge networks. To achieve this, Totoro+ incorporates several key techniques and components, which we describe in detail.

A. Overview

Totoro+'s system goals are:

- *Scalability:* Totoro+ can scale to process a vast number of FL applications' tasks simultaneously on millions of edge nodes without introducing any centralized bottleneck.
- *Adaptivity:* Totoro+ can quickly adapt to the practical edge networks characterized by varying bandwidths, unreliable links, high churn (nodes join and leave), and workload surges in arbitrary geographical edge locations.
- *Good FL performance:* When handling a vast number of FL applications, Totoro+ can speed up the training process for each of them.

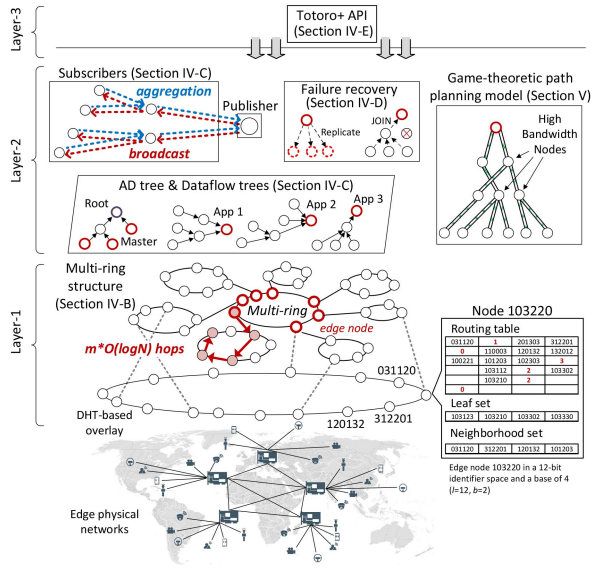


Fig. 3. The Totoro+ system overview.

As shown in Fig. 3, Totoro+ has three layers: a locality-aware P2P multi-ring structure, a publish/subscribe-based forest abstraction, and a high-level API.

Layer 1. locality-aware P2P multi-ring structure: All distributed edge nodes are self-organized into a DHT-based P2P overlay. Each node has a unique 128-bit NodeId in a very large circular NodeId space. In an edge network with N nodes, the DHT-based P2P overlay guarantees that, no matter where the source node is, any FL data (e.g., model or gradient) can be routed to any destination node within $\mathcal{O}(\log N)$ hops. Compared to existing DHTs studies [49], [50], [51], our innovations are: (1) Totoro+ divides the original single P2P ring structure into many smaller, more manageable locality-aware P2P multi-ring structures that enable locality-aware FL processing; and (2) Totoro+ designs a new boundary-aware two-level routing table that ensures administrative isolation for privacy concerns.

Layer 2. publish/subscribe-based forest abstraction: Built upon Layer 1's locality-ware P2P multi-ring structure, Totoro+ introduces a new publish/subscribe-based forest abstraction that manages a vast number of FL applications in a scalable manner. Each FL application is assigned a dynamically structured dataflow tree that operates with maximum independence and is responsible for disseminating the model from the master to the workers and aggregating the gradients from the workers to the master. In addition to dataflow trees, the masters of all dataflow trees construct an advertise-discover (AD) tree to advertise and discover FL applications running over the overlay. These trees together form a "forest". Our innovations are (1) a fully decentralized architecture—unlike other federated learning systems, our system does not have a static assignment of parameter servers. Instead, any edge node can be automatically promoted as a parameter server (master) when workload surges, which significantly improves load balancing and scalability. (2) Advertising and discovering FL applications—new edge nodes

that just joined the overlay can easily locate and subscribe the FL applications running over the overlay. (3) DHT-based routing—the time complexity of model/AppId broadcast, and gradient/AppId aggregation is limited to $\mathcal{O}(\log N)$ hops.

Layer 3. high-level API: We provide a high-level API to abstract away the complexities of P2P overlay construction, dynamic-structured dataflow tree construction, model/AppId dissemination, and gradient/AppId aggregation. Totoro$^+$ supports application-specific customization, allowing application owners to set their own FL policies.

B. Layer 1: Locality-Aware P2P Multi-Ring Structure

Many times, geographic diversity or location matters for training FL applications. For example, a road traffic detection application may require nodes with varying weather condition information in different geographic locations [15], [16], [17]. Training a model on a medical disease prevalent in a certain region may require information from a specific location [52], [53].

Therefore, we organize distributed edge nodes into a locality-aware P2P multi-ring structure to enable locality-aware FL processing.

First, we organize distributed edge nodes into a DHT-based P2P ring overlay, which is similar to the BitTorrent nodes that use the Kademila DHT [54] for "trackerless" torrents. Each edge node is assigned a unique 128-bit NodeId in a very large circular NodeId space (e.g., $0 \sim 2^{128}$). NodeIds are used to identify edge nodes and route FL data (e.g., model or gradient) over large-scale edge networks.

To do that, each node needs to maintain three data structures: a routing table, a leaf set, and a neighborhood set.

- *Routing table* is used for routing FL data. It consists of node characteristics organized in rows by the length of the common prefix. The routing works based on prefix-based matching. Every node knows m other nodes in the ring and the distance of the nodes it knows increases exponentially. The routing jumps closer and closer to the destination, like a greedy algorithm, within $\lceil \log_{2^b} N - 1 \rceil$ hops, where $2^b - 1$ is the routing table's entry size.
- *Leaf set* is used for rebuilding the routing tables upon failures.
- *Neighborhood set* contains a fixed number of nodes that are "physically" closest to that node for maintaining the locality properties.

Second, we divide the original large P2P ring into m smaller, more manageable, locality-aware rings, which we call "multi-rings", using Ratnasamy and Shenker's distributed binning algorithm [55] (m is a configurable parameter). Each ring is an "edge zone" and is characterized by a maximum desired network round-trip time (RTT), called *diameter*.

Third, we design a new routing table to enable administrative isolation. The challenge is, *how to achieve path convergence to enable administrative isolation, i.e., data paths from different nodes in an edge site should converge at a node in that edge site?* Existing DHTs [49], [50], [51], [54] do not guarantee path convergence as those systems try to optimize the search path to reduce response latency. To route a packet to an arbitrary destination key, the packet will be routed to the destination node in another site as long as it has a longer NodeId prefix matching the key. To address this challenge, we make the following changes to existing routing tables: (1) each NodeId now has $(m+n)$-bit, where the m-bit prefix presents the zone Id and the n-bit suffix represents the NodeId within a zone. Let P denote the prefix of NodeId, i.e., $P_1 \ldots P_n$. Let S denote the suffix of NodeId, i.e., $S_1 \ldots S_n$. Then the NodeId equals $D = P * 2^n + S$; correspondingly (2) each node's routing table will have two levels: the level 1 routing table and the level 2 routing table. The i_{th} entry in the level 1 routing table with m entries at peer x equals to $(P_x + 2^{i-1}) \bmod 2^m * 2^n$. The i_{th} entry in the level 2 routing table with n entries at peer y equals to $(S_y + 2^{i-1}) \bmod 2^n$.

To achieve administration isolation, the administrator of an edge site can leverage the level 1 routing table to control the data flow among different edge zones. For example, when an FL application should be running only within an edge site, any packet generated by that FL application should only travel within this edge site. Administrators can check the destination of packets. If the packet's destination shares a different prefix with the administrator's zone Id, the administrator can block the packet before routing it outside the edge zone.

C. Layer 2: Publish/subscribe-Based Forest Abstraction

Built upon Layer 1, Totoro$^+$ introduces a new publish/subscribe-based "forest" abstraction for managing a vast number of FL applications in a scalable manner. Specifically, our goal is to achieve a balanced distribution of masters for hundreds of thousands of FL applications, ensuring that they are not concentrated on a few overloaded nodes. The key innovation is leveraging DHTs to decompose the FL system architecture from *1:n* to *m:n*, where each FL application can be assigned a dynamic-structured dataflow tree that operates with maximum independence, thereby radically improving load balance and scalability. A DHT is a hash table that partitions the key space and distributes the parts across a set of nodes, providing a lookup service similar to a hash table. The trick with DHTs is that the node that gets to store a particular key is found by hashing that key, so in effect, the hash-table buckets are now independent nodes in a network.

We utilize the fully decentralized nature of DHT to process FL applications' tasks at an extreme scale: unlike other FL systems, Totoro$^+$ does not have a static assignment of a parameter server. Instead, the parameter server is broken down into many components, such as the coordinator, client selector, and aggregator. Any edge node can act as any FL application's coordinator, aggregator, client selector, worker (participating edge device), or any combination of the above, thereby radically improving load balance and scalability.

Fig. 4 lays out the construction of the publish/subscribe-based "forest". It has the following steps.

Built on top of Layer 1, all edge nodes are structured into a DHT-based P2P overlay. Here we use one ring as an example. The DHT-based P2P overlay guarantees that: *given a message and a key, no matter where the source node is, the message can be reliably routed to the node whose NodeId is numerically closest to that key, within* $\lceil \log_{2^b} N - 1 \rceil$ *hops, where* $2^b - 1$ *is the routing table's entry size.*

The first step is constructing application-based logical "trees" of nodes and ensuring that these trees are well balanced over the large-scale edge topologies (Fig. 4 left).

a) When any new FL application is launched, we calculate the application's AppId, which equals the cryptographic hash of the application's textual name, the creator's public

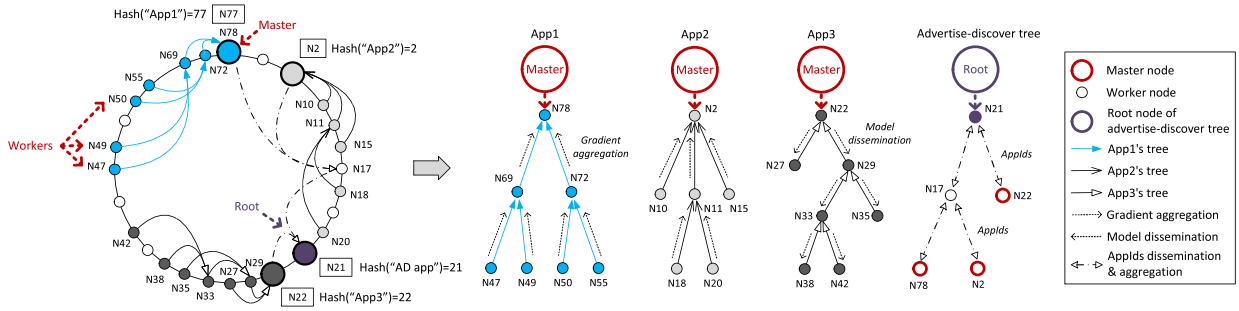


Fig. 4. The workflow of Totoro^{+}'s publish/subscribe-based "forest" abstraction with three dataflow trees for three FL applications and an advertise-discover tree for application advertise and discovery.

key, and a random salt, *AppId=hash("FL application")*. The hash is computed using the collision resistant SHA-1 hash function, ensuring a uniform distribution of AppIds.

b) Then, the edge node processing the application's data routes a `JOIN` message using AppId as the key. Since all nodes belonging to the same application use the same key, their `JOIN` messages eventually arrive at a rendezvous node, with NodeId numerically close to AppId. The rendezvous node is set as the root of this application's tree.

c) The unions of all `JOIN` messages' paths are registered to construct the tree, in which the internal node, as the forwarder, maintains a children table for the group containing an entry (IP address and AppId) for each child. All of the trees together form a "forest" abstraction.

d) For each application's tree, we designate the root node as the master, the internal nodes as the coordinator, aggregator, and client selector components, and the leaf nodes as the workers (participating edge devices).

Rationale: (1) Since different applications have different AppIds, the paths and the rendezvous nodes of their spanning trees will also differ, resulting in an even distribution of trees across all edge nodes. (2) Because all nodes are equal, each node can serve as a leaf, internal, or root node for different applications, thus removing the scalability bottleneck without overburdening any single node.

The second step is implementing a topic-based publish/ subscribe messaging protocol within the dataflow tree. Each FL application has a "topic". The master is the "publisher", and the workers are the "subscribers".

a) *Model broadcast:* The master disseminates the FL model to the workers along the tree. Then each worker independently trains a local model and computes the model updates (e.g., gradients and weights) on the local data.

b) *Gradient aggregation:* Once the workers have completed the computation, the master aggregates model updates from the workers, in which each level of the tree progressively aggregates the updates from tree leaves to the root. To meet the diverse needs of different applications, owners can specify different aggregation functions in their trees. For instance, FedAvg [39] works well in most situations, while FedProx [56] demonstrates superior performance in highly heterogeneous settings for more stable and accurate convergence.

Rationale: (1) Due to the loosely coupled interaction between the publisher and subscribers, Totoro^{+} can simultaneously support a large number of dataflow trees with a wide range of tree sizes and a high rate of membership turnover. (2) The use of DHTs enables the efficient construction of aggregation trees and multicast services, as their converging properties guarantee model broadcast or gradient aggregation to be fulfilled within only $\mathcal{O}(\log N)$ hops, which places an upper bound on worst-case latency for data transfers.

The third step is constructing an advertise-discover (AD) tree that enables edge nodes to advertise and discover FL applications running over the overlay (Fig. 4 right).

a) Similar to the first step, each master of various applications' trees routes a `JOIN` message using *AppId=hash("AD application")* as the key. Since all masters use the same key, their `JOIN` messages eventually arrive at a rendezvous node whose NodeId is numerically closest to AppId. The rendezvous node becomes the root of the AD tree.

b) The unions of all `JOIN` messages' paths are registered to construct the AD tree. In addition to children tables for FL applications, the internal node in the tree works as a forwarder that maintains a children table consisting of the IP addresses of each child and AppIds of FL applications each child is hosting.

c) The root of the AD tree is the "publisher", and other nodes are the "subscribers". The publisher disseminates AppIds of FL applications to each node along the tree, whereas the subscribers in different levels of the tree progressively aggregate AppIds of FL applications to the root.

d) Whenever a new edge node that joins the DHT-based P2P overlay wants to discover FL applications running over the overlay or an existing edge node intends to switch to a different FL application running on the overlay, it can route a `JOIN` message using the same AppId to join the AD tree as a subscriber and receive the AppIds of interest. After that, it can route a `LEAVE` message using the same AppId to leave the AD tree.

Rationale: (1) Since edge nodes may join and leave the overlay frequently and suddenly, joining nodes can subscribe to the AD tree first and receive the information of current FL applications instead of sending broadcast messages to all other nodes, making Totoro^{+} adaptive to practical edge networks. (2) Because edge nodes can leave the AD tree immediately after they receive the necessary information, the tree may have M nodes plus some intermediate nodes N' most of the time and $N' << N$, where M denotes the number of current FL applications and N denotes the number of nodes in the overlay. Therefore, with the use of DHTs, the AppId broadcast and aggregation can also be fulfilled within $\mathcal{O}(\log(M + N'))$ hops.

A comparison of Totoro and Totoro$^+$ in terms of FL application discovery and advertisement is presented in the supplementary material.

D. Failure Recovery

In the case of node failures, we use a parallel recovery approach to repairing the dynamic-structured dataflow trees. A dataflow tree has a master and several workers, and the parallel recovery approach leverages two respective ways to deal with the failure of a worker and the failure of a worker.

Worker node fails: Periodically, each internal node in the tree sends a keep-alive message to its child nodes. A child node suspects its parent fails when it cannot receive keep-alive messages from its parent node. In such a scenario, the child node routes a `JOIN` message using AppId as the key. The overlay network will route the message to a new parent, create an alternative route, and repair the dataflow tree.

Master node fails: In each training round, we replicate the state associated with each application's master across the k nodes in its neighborhood set ($k = 2$, by default), which consists of a fixed number of nodes that are "physically" closest to the master node. If the master node fails, its immediate child node will detect it, route a `JOIN` message using AppId as the key, and identify a new master node to take over the role of the failed master. According to Layer 2: publish/subscribe-based forest abstraction, the new master node's NodeId will be numerically closest to the AppId, and it will first contact the nodes that have state replicas to recover the state and inform workers to continue the training tasks like aggregation, global model update, and broadcast.

The tree repair process scales well: failure detection is done by sending messages to child nodes only. Failure recovery is also local. Only a small number of nodes ($\mathcal{O}(\log_{2^b} N)$) is involved, in which 2^b is the fanout of the tree. In addition, the replicas of states are typically transmitted over local networks with plentiful bandwidth; hence, the communication overhead is negligible.

E. Layer 3: High-Level API

We abstract key components of Totoro$^+$ to provide an easy-to-use API (see TABLE II). Totoro$^+$ supports application-specific customization, allowing application owners to set their own FL policies. The Pastry DHT and Scribe multicast infrastructure are written in Java, with end-user functionality encapsulated in a Python API. This is done so that users do not have to deal with two libraries in separate languages. Thus Totoro$^+$ can be easily integrated with popular FL frameworks such as PyTorch [24], PySyft [12], and TensorFlow Federated [8].

Application-level customization: Totoro$^+$'s APIs support application-level customization. For example, to prevent potential leakage of model weights to other nodes, application owners can specify various privacy techniques in `Aggregate(app_id, object)`, such as differential privacy [57], secure aggregation [29], and homomorphic encryption [58]. Nodes that subscribe to an FL application adhere to the privacy technique specified by that application during the FL process. Take differential privacy as an example, if an application owner launches an FL application and specifies the use of differential privacy with Gaussian noise to secure weights, the edge nodes, as the master, coordinator, aggregator, and client selector, will operate following the privacy technique. Similarly, the leaf nodes, serving as workers, will apply Gaussian noise to local training. Moreover, to realize client selection techniques, application owners can rule out edge nodes that do not meet specific requirements (e.g., data distributions, battery, network conditions, etc.) when receiving `JOIN` messages from edge nodes.

TABLE II
TOTORO$^+$ API

Join(IP, port, site) Edge node joins the DHT-based P2P overlay network.
CreateTree(app_id) Application owner creates a dynamic-structured dataflow tree and configures the parameters (e.g., fanout).
Subscribe(app_id) Edge node sends a `JOIN` message to subscribe to a dynamically-structured dataflow tree with the topic equal to app_id. Application owner can specify her client selection function in the API.
Unsubscribe(app_id) Edge node sends a `LEAVE` message to unsubscribe to a dynamically-structured dataflow tree with the topic equal to app_id.
Broadcast(app_id, object) Application's master disseminates the model or AppIds to workers along its dynamically-structured dataflow tree. Application owner can specify her compression function in the API.
onBroadcast(app_id, object) Callback. Invoked when the worker receives any model, updates, or AppIds from the application's master.
Aggregate(app_id, object) Application's master aggregates updates or AppIds from the workers to the root. Application owner can specify her aggregation function in the API.
onAggregate(app_id, object) Callback. Invoked when any internal node receives updates or AppIds from a child node.
onTimer(app_id) Callback. Invoked periodically. Application's master uses it to get the information about the progress of training (e.g., round_num, accuracy, straggler_id) and inference.

Multi-rings: Application owners can specify whether their applications span multiple zones. For example, a road traffic detection application may require nodes with different weather condition information in different geographical locations. If an application needs to ingest data sources across multiple zones, it will traverse multiple zones (at most m) to build the dataflow tree, resulting in $m \times \mathcal{O}(\log N)$ routing hops.

V. Game-Theoretic Path Planning Model

Totoro$^+$ introduces a new game-theoretic path planning model that can replan the data transfer paths in dynamically structured dataflow trees to adapt to unreliable edge networks.

In most real-world edge networks, link delays are unpredictable and vary stochastically [59]. When a link becomes slow, it can disrupt communication with its child and parent nodes, thereby affecting the entire path from the leaves to the root passing through this link in dataflow trees.

The challenge is that, in many cases, we don't know the quality of the network links in advance, such as the probability of successfully transmitting packets in wireless sensor networks [59]. This information is often obtained by actually sending packets and observing the outcomes. Furthermore, the network links have limited capacities [59]. When a link carries too much traffic, its overall data rate dramatically decreases. Therefore, we face two dilemmas when planning the paths for model dissemination and gradient aggregation.

One is *whether to explore new or unknown links or exploit well-known ones*. If we only rely on the known links, we may miss out on finding a better one with a higher success rate or data rate. However, exploring too many new links may result in

more packet losses and higher communication delays. The other is *whether to prioritize link quality or link capacity.* Maximizing link quality avoids packet loss and errors but may overload these paths, leading to congestion and increased latencies. However, prioritizing link capacity may increase the risk of transmission failures instead.

This is where Multi-Armed Bandit (MAB) algorithms [60], [61] and game theory [48], [62], [63] come into play. Imagine someone at a bustling amusement park, surrounded by various facilities like roller coasters, merry-go-rounds, and Ferris wheels. Each facility differs in its entertainment type and its queue length. Initially, she explores different facilities randomly, collecting data on two key aspects: the joy each ride brings and the length of the waiting time. Her goal is to maximize her day by finding the facilities that offer the best balance of fun and short waiting times. However, her decision-making process influences the decisions of others. The presence of other tourists, who also choose facilities based on their own experiences and preferences, affects the queues at each location. The collective decisions of all tourists shape these queues, each acting on their assessments of enjoyment and waiting times. As time passes, some facilities may become busier, while others may see reduced waiting times. She must constantly adapt to these changing conditions, balancing her preferences for fun with the evolving waiting times.

Therefore, the path-planning problem can be formulated as a general-sum congestion game with bandit feedback [64], [65]: we follow a policy to explore different paths to learn about their rewards (e.g., link quality and link capacity) or exploit the paths that offer the highest rewards and update the policy based on the rewards. And the rewards are influenced by collective decisions on paths. We gradually find out the optimal policy to select paths over edge networks.

The formal definitions and theoretical concepts underlying this formulation are provided in the supplementary material. Specifically, we review general-sum matrix games, policy representations, Nash equilibrium and Nash regret, as well as potential and congestion games, together with different feedback models. In the following, we build upon these preliminaries to formulate a routing problem, define the corresponding optimization objective, and develop a game-theoretic algorithm with theoretical performance guarantees. In addition, we present a comparison of the mathematical models underlying Totoro and Totoro$^+$ in the supplementary material.

A. Problem Formulation

A dataflow tree in Totoro$^+$ consists of at most N nodes, and there are at most P paths to reach the root of the tree. Since each path $p \in [P]$ has different success rate, and capacity, we let $\mathcal{R}^p(\cdot|k,\theta_p) \in [0,1]$ denote the reward distribution for path p with mean $r^p(k,\theta_p)$ given that k nodes select path p with mean success rate being θ_p.

The higher the mean success rates, the higher the rewards. In contrast, the more nodes select the same path, the less the rewards they receive as the success rate and data rate drop.

Suppose the joint paths chosen by all nodes are denoted by $\mathbf{p} = [p_1, p_2, \ldots, p_N]$, where p_n denotes the path selected by node n, then we let $n^p(\mathbf{p})$ denote the number of nodes selecting path p given the joint action $\mathbf{p}$. A node n selects path p and gets the reward $r_n(p)$ drawn from the reward distribution $\mathcal{R}^p(\cdot|n^p(\mathbf{p}),\theta_p)$ with mean $r^p(n^p(\mathbf{p}),\theta_p)$.

Algorithm 1: Game-Theoretic Distributed Hop-By-Hop Routing Algorithm.

Input: mixture weights $\alpha, \beta \in [0,1]$; initial policy π_n^1 for all $n \in N$.

1: **for** episode $k = 1, 2, \cdots$ **do**
2: **for** packet $t = 1, \ldots, \tau$ **do**
3: Each node n in parallel selects hop $p_n^{k,t} \sim \pi_n^k$ to send a packet, observes reward $r_n^{k,t}$.
4: **for** node $n = 1, \ldots, N$ **in parallel**
5: $\rho_n^k \leftarrow \arg\min_{\lambda \in \Delta(P_n)} \det(M(\lambda))$.
6: $\widehat{\nabla}_n^k \Phi(p) \leftarrow \frac{1}{\tau} \sum_{t=1}^{\tau} \psi(p)^\top M(\pi_n^k)^{-1} \psi(p_n^{k,t}) r_n^{k,t}$, for all $p \in P_n$.
7: $\tilde{\pi}_n^{k+1} \leftarrow \arg\max_{\lambda \in \Delta(P_n)} \langle \lambda, \widehat{\nabla}_n^k \Phi \rangle$.
8: $\pi_n^{k+1} \leftarrow \alpha \underbrace{[\pi_n^k + \beta(\tilde{\pi}_n^{k+1} - \pi_n^k)]}_{\text{Frank-Wolfe update}} + \underbrace{(1-\alpha)\rho_n^k}_{\text{Exploration}}$.

Optimization objective: We aim to efficiently route T packets (such as gradients in many rounds) from a worker node to a master node, minimizing the overall time taken. In other words, our goal is to maximize total rewards. Also, since collective actions taken by all nodes influence the rewards, we resort to see how those actions affect others' decisions.

First, we define policy π_n followed by node n to select paths, where π_n is from the probability simplex over node n's action space P_n, denoted by $\Delta(P_n)$, where $P_n \subseteq [P]$ is the set of paths available for node n's selection. Similarly, we can define a general policy $\pi = [\pi_1, \pi_2, \ldots, \pi_N]$ as a tuple in the joint space $\Delta(P_1) \times \cdots \times \Delta(P_N)$. Hence, we can have $\mathbf{p} = (p_1, p_2, \ldots, p_N) \sim \pi$ and $p_n \overset{i.i.d}{\sim} \pi_n$.

Then, since the policy influences the rewards the nodes receive, we use the rewards to represent *the value of a policy*. We define the value of policy π for node n as $V_n^\pi = \mathbb{E}_{\mathbf{p}\sim\pi}[r_n(\mathbf{p})]$, where $r_n(\mathbf{p}) = r^{p_n}(n^{p_n}(\mathbf{p}), \theta_{p_n})$. Let π_{-n} be the marginal joint policy of nodes $1, \ldots, n-1, n+1, \ldots, N$. We refer to [48], [63] to define the *Nash Regret* after T packets as

$$\text{Nash-Regret}(T) = \sum_{t=1}^{T} \max_{n \in [N]} (V_n^{\pi_n^+, \pi_{-n}^t} - V_n^{\pi^t}), \tag{1}$$

where π^t is the policy for packet t, $V_n^{\pi^t}$ is the value of policy π^t for node n, $\pi_n^+ = \arg\max_{\mu \in \Delta(P_n)} V_n^{\mu, \pi_{-n}^t}$ represents the best response of node n under policy π, and $V_n^{\pi_n^+, \pi_{-n}^t}$ is the value of the policy (π_n^+, π_{-n}^t). The Nash Regret indicates the value difference between when a node unilaterally switches its policy and when it follows the same policy. The optimization objective is to learn a general policy π^* over T packets such that

$$\text{Nash-Regret}(T) \leq \epsilon, \tag{2}$$

where $\epsilon > 0$ is a given tolerance threshold.

B. Our Algorithm

We propose a game-theoretic distributed hop-by-hop routing algorithm based on the bandit feedback model [48], [60], [62], [66], [67]. The pseudo-code of the game-theoretic distributed hop selection algorithm is presented in Algorithm 1. The notations used in the algorithm are summarized in the supplementary material.

When node n receives a packet t in episode k from a previous node, it follows the current policy π_n^k to select the next hop $p_n^{k,t}$, sends the packet, and observes the corresponding reward $r_n^{k,t}$ (line 3). Whenever node n has sent τ packets and collected τ rewards, it updates the policy π_n^k based on the τ rewards collected.

First, we identify an exploratory policy with its policy correlation matrix:

$$M(\lambda) = \sum_{p \in \lambda} \lambda(p)\psi(p)\psi(p)^\top, \tag{3}$$

which has the minimum determinant among all possible policies in the probability simplex $\Delta(P_n)$ over node n's action space P_n (line 5), where $\lambda(p)$ denotes the probability of selecting the next hop p, and $\psi(p)$ represents the vector-valued function mapping hop p to a one-hot vector.

Next, we use linear regression with τ rewards to estimate the gradient of the mapping from policy π_n^k to potential rewards for node n (line 6), where $M(\pi_n^k)^{-1}$ represents the inverse of $M(\pi_n^k)$. We yield $|P_n|$ gradient estimators, which form a $|P_n|$-dimension vector $\widehat{\nabla}_n^k \Phi = [\widehat{\nabla}_n^k \Phi(p_1), \ldots, \widehat{\nabla}_n^k \Phi(p_{|P_n|})]$. With the gradient estimator for all possible hops, we calculate the dot product of the estimator and all possible policies $\lambda \in \Delta(P_n)$ and obtain the optimal policy $\tilde{\pi}_n^{k+1}$ that yields the maximum dot product (line 7).

Lastly, we leverage the Frank-Wolfe method [68] to linear combine current policy π_n^k with optimal policy $\tilde{\pi}_n^{k+1}$:

$$\pi_n^k + \beta(\tilde{\pi}_n^{k+1} - \pi_n^k), \tag{4}$$

where β is in the range [0,1]. The Frank-Wolfe update in (4) guarantees that the results of the linear combination fall in the probability simplex $\Delta(P_n)$. In addition, we add exploratory policy ρ_n^k to the Frank-Wolfe update (line 8):

$$\alpha\left[\pi_n^k + \beta(\tilde{\pi}_n^{k+1} - \pi_n^k)\right] + (1-\alpha)\rho_n^k, \tag{5}$$

where α is also in the range [0,1] and determines the extent to explore different policies. Node n follows the final policy π_n^{k+1} by (5) to route packet $\tau+1$ to $\tau+\tau$ in episode $k+1$, and hence each episode k has individual policy π_n^k to route τ packets. We provide a numerical example to illustrate the workflow of Algorithm 1 in the supplementary material, discuss how local congestion is observed and inferred at the application level in the supplementary material, and analyze the implementation and adaptivity of Algorithm 1 in the supplementary material.

C. Theoretical Analysis

We conduct a theoretical analysis on Algorithm 1 in two aspects: the *Nash regret bound* and *time complexity*.

Theorem 1. Let $T = k\tau$ and assume that each policy in $\Delta(P_n)$ has no zero element for all n. By running Algorithm 1 with gradient estimator $\widehat{\nabla}_n^k \Phi(p)$ defined in line 6 and exploratory policy ρ_n^k defined in line 5, if $k \geq \frac{2}{N}$, then with probability $1-\delta$, we have

$$\text{Nash-Regret}(T) \leq \tilde{\mathcal{O}}(N^2 T^{5/6} \log N).$$

The proof details of Theorem 1 are given in the supplementary material. The results in Theorem 1 demonstrate a *sublinear* Nash regret with polynomial dependence on N, the number of nodes. Following Definition 1 in [48] and Section 3 in [63], we obtain the following corollary.

Corollary 1: Algorithm 1 reaches an ϵ-approximate Nash equilibrium.

Corollary 1 tells us that nodes that run Algorithm 1 to update their policies can find a general policy such that a node receives at most ϵ more rewards by unilaterally changing its policy, implying reaching an ϵ-approximate Nash equilibrium. In addition, we provide the Nash regret guarantee under bounded asynchrony in the following corollary.

Corollary 2: Under bounded asynchrony, the Nash regret guarantee in Theorem 1 continues to hold, up to an additional sublinear regret term induced by asynchrony.

The proof details of Corollay 2 are given in the supplementary material. Finally, we analyze the time complexity of Algorithm 1 in the following theorem.

Theorem 2: The time complexity of Algorithm 1 in each episode is $\mathcal{O}(\tau \log^3 N + |\Delta(P_n)| \log^3 N)$, where τ represents the number of packets sent in each episode, and $|\Delta(P_n)|$ denotes the number of policies node n can adopt.

The proof details of Theorem 2 are given in the supplementary material.

VI. Implementation

We implement Totoro$^+$ on top of the Pastry (v.2.1) [23] and PyTorch (v.2.10.0) [69] software stacks. Such implementation choice is motivated by the following considerations: (1) Pastry [49] is a widely used overlay and routing network for the implementation of a DHT similar to Chord [50]. Instead of implementing another distributed system core, we can leverage Pastry's excellent routing substrate (e.g., $\mathcal{O}(\log N)$ node lookup), self-repairing routing table, message transportation layer, and scalable application-level multicast infrastructure (Scribe [70]). These features greatly simplify the development process. (2) PyTorch [24] provides a flexible and intuitive API for building neural networks. It also offers a rich ecosystem of libraries (e.g., TorchVision [71] and HuggingFace Transformers [72]) that support many modern neural network models.

We made the following major modifications: (1) We changed the original single P2P ring structure to a new locality-aware P2P multi-ring structure. (2) We implemented a fully decentralized "many masters/many workers" architecture by utilizing DHTs and adding several data structures: a list of operations for tracking routing paths, selecting masters and workers, and constructing dynamically structured training trees. (3) We implemented a publish/subscribe messaging pattern for scalable model propagation and gradient aggregation. We introduced a serialization mechanism to convert trained models into binary arrays for low-cost communication over edge networks. (4) We introduced an advertise-discover mechanism to enable nodes to advertise and discover FL applications running over the overlay. We implemented a parallel recovery approach to dealing with the failure of masters and the failure of workers. (5) We implemented a game-theoretic path planning model to "autonomously" replan or repair the dynamically structured dataflow tree by collecting feedback, detecting node stragglers or failures, balancing network traffic, and creating alternative routes. We discuss the adaptivity of Totoro$^+$ to edge heterogeneity in the supplementary material.

VII. Evaluation

We evaluate Totoro$^+$'s performance in a real-world distributed environment that includes 500 Amazon EC2 nodes and

uses real-world computer vision (CV) and natural language processing (NLP) datasets at different scales. We have the following key results.

- Totoro$^+$ scales the number of masters to handle a varying workload (from 125 to 2000 concurrently running FL applications) in real-world edge topologies (Section VII-B).
- Totoro$^+$ achieves $\mathcal{O}(\log N)$ hops for model dissemination and gradient aggregation (Section VII-C).
- Totoro$^+$ outperforms state-of-the-art FL systems (OpenFL [25] and FedScale [26]) by speeding up the training time 1.2×-14.0× to reach the equivalent model accuracy (Section VII-D).
- Totoro$^+$ efficiently adapts to edge networks with constrained bandwidth capacity (Section VII-E).
- Totoro$^+$ recovers from node failures to adapt to node churns (Section VII-F).

A. Methodology

Experimental setup: Totoro$^+$ is designed to operate in large deployments with millions of edge nodes. However, such a deployment is prohibitively expensive and impractical in an academic environment. As such, we resort to emulating a real-world edge setting on 500 AWS EC2 `t2.medium` nodes, each of which has 2 vCPUs and 4GB of RAM, and 100 GB of disk: (1) we use one JVM to represent one edge node and emulate up to $100k$ edge nodes on the testbed; (2) we divide the $100k$ edge nodes into geo-distributed zones based on the real-world EUA dataset [73], which consists of 95,271 edge nodes distributed across 12 Australian states and regions. To emphasize the system-level benefits of Totoro$^+$, we conduct experiments using homogeneous nodes in the main paper. Experimental results under heterogeneous node settings are reported in the supplementary material.

Baselines: We use OpenFL (v.1.3) [25] and FedScale (v.0.5) [26] as the baseline. FedScale is a scalable and extensible open-source FL system and benchmarking suite developed by Symbiotic Lab [74], which provides high-level and flexible API to implement, evaluate, and deploy FL algorithms easily in both standalone (single CPU/GPU) and distributed (multiple machines) settings. OpenFL is an open-source framework developed by Intel that runs FL in a single-machine setting.

Parameters: Totoro$^+$'s Pastry DHT is configured with a leaf set of 24, max open sockets of 5000, and a transport buffer size of 6 MB. We configure different fanouts 8 (2^3), 16 (2^4), and 32 (2^5) for Totoro$^+$'s dataflow trees by changing the DHT routing table base bit values to 3, 4, and 5, respectively. The minibatch size of each node is 20 in image classification and speech recognition tasks. The initial learning rate for the ShuffleNet V2 model is 0.05 and 0.1 for the ResNet-34 model.

Metrics: We focus on Totoro$^+$'s scalability, adaptivity, and FL effectiveness. To evaluate scalability, we measure how numerous applications' dataflow trees are distributed over large-scale edge topologies. We also measure how Totoro$^+$ scales with the number of nodes in terms of *model broadcast time* and *gradient aggregation time*. To evaluate the FL effectiveness, we measure *time-to-accuracy* performance, which is the duration of model training tasks on the testing set to achieve the target accuracy. To evaluate adaptivity, we measure *cumulative packet latency*, *Nash regret*, *node selection frequencies*, and *failure recovery time*. We also measure Totoro$^+$'s *runtime overhead*.

B. Scalability Analysis

Fig. 5(a) shows the real-world edge zones generated from the EUA dataset [73]. Australian Communications and Media Authority publishes the EUA dataset [73], which contains the geographical locations of 95,271 cellular base stations in 12 states of Australia (`ACT`: 931, `ANT`: 15, `EXT`: 8, `ISL`: 36, `NSW`: 24574, `NT`: 3137, `QLD`: 21576, `SA`: 7682, `TAS`: 3213, `VIC`: 18163, `WA`: 15933, `WLD`: 3). We estimate the maximum round-trip time based on the distances between nodes in the dataset, and then use Ratnasamy and Shenker's distributed binning algorithm [55] to divide them into different zones.

Totoro$^+$ fairly distributes masters: Fig. 5(b) shows the normal probability plot of the number of masters mapped on each node in a 1000-node edge zone under stress testing. The results show that when we create a large number of dataflow trees like 500, 99.5% of the nodes are the roots of 3 trees or less. The results illustrate a good load balance among participating devices when performing a large number of FL applications' tasks simultaneously. Fig. 5(c) shows the distribution of Totoro$^+$'s masters over different edge zones that have different workloads. We assume densely populated topologies have heavy workloads and sparsely populated topologies have light workloads. The results show that Totoro$^+$ automatically scales the number of masters to handle the varying workload.

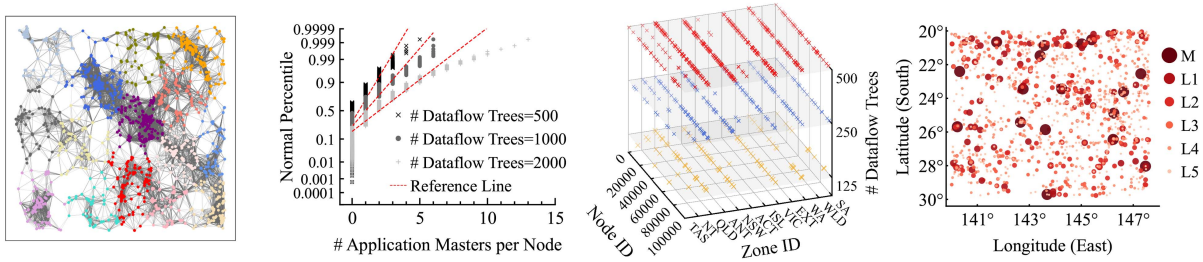
Totoro$^+$ excels at load balancing: Fig. 5(d) shows the distribution of all branches in 17 Totoro$^+$ dataflow trees on 1946 edge nodes over the 3 most popular topologies. Each tree has a fanout of 8 (2^3) and a random number of depths (from level 1 to level 6). Different colors represent different levels of nodes in a tree, with the darkest color representing the root node. The results show that these dataflow trees are well balanced across different topologies, not only the root nodes but also the forwarder nodes and the leaf nodes, demonstrating Totoro$^+$'s attractive scalability and load balancing properties.

C. Model Dissemination and Gradient Aggregation

Totoro$^+$ scales with #nodes: Fig. 6(a) and (b) show Totoro$^+$'s model dissemination and gradient aggregation times for an exponentially increasing number of edge nodes in a single training tree. When the number of nodes grows *exponentially* (from 20 to 5120), the dissemination time and aggregation time only increase *linearly*. This is because the dissemination time and aggregation time are limited by the dataflow tree depth ($\mathcal{O}(\log N)$) by using the DHT-based P2P overlay. Therefore, when the number of edge devices grows to the scale of millions or even billions, Totoro$^+$ guarantees that only a few extra hops are needed for them to receive updated FL models or send updated gradients.

Totoro$^+$ offers flexible tree fanouts: Fig. 6(c) and (d) show the model dissemination time and gradient aggregation time of different tree fanouts for the ResNet-34 model. We can observe that a larger fanout leads to less model dissemination and gradient aggregation times because the dataflow trees become less deep. However, if an internal node fails or leaves, the new one needs to rebuild many connections between the child nodes and the parent nodes to rebuild the tree, and the new node may become an I/O or communication bottleneck.

Totoro$^+$ reduces communication cost: Fig. 7 shows the comparison of the traffic per node of Totoro$^+$ and the baseline FL systems. Communication is a critical bottleneck in FL systems. We can observe that the additional network traffic imposed by



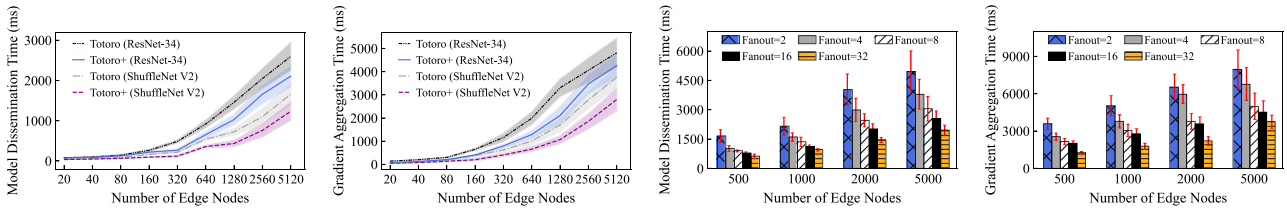
(a) Real-world edge topologies generated from the EUA dataset [73].

(b) Normal probability plot of the number of masters mapped per node.

(c) Totoro$^+$ scales with #masters to handle the varying workload.

(d) The distribution of Totoro's dataflow trees over edge topologies.

Fig. 5. Totoro$^+$ excels at scalability by fairly distributing many dynamic-structured dataflow trees across large-scale edge topologies.



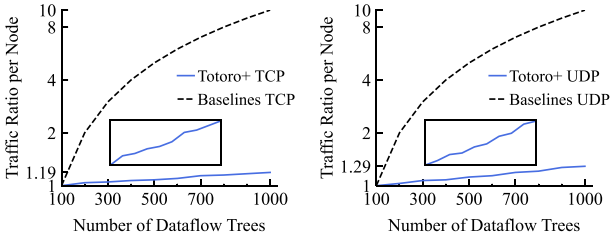
(a) Model dissemination time.

(b) Gradient aggregation time.

(c) Model dissemination time of different fanouts.

(d) Gradient Aggregation time of different fanouts.

Fig. 6. Totoro$^+$ scales with an exponentially increasing number of edge nodes for model dissemination and gradient aggregation.



(a) TCP traffic cost.

(b) UDP traffic cost.

Fig. 7. Totoro$^+$ reduces communication cost.

Totoro$^+$ is small. The network traffic is increased by only 1.19× for TCP and 1.29× for UDP when the number of dataflow trees is increased by 10×. This is because when a new training tree is created, it merely routes `JOIN` messages toward the root of the tree using the overlay, adding overlay links to reconstruct the tree without establishing a new connection. Then the overhead for creating new dataflow trees can be amortized over the overhead of the overlay.

D. Federated Learning Effectiveness

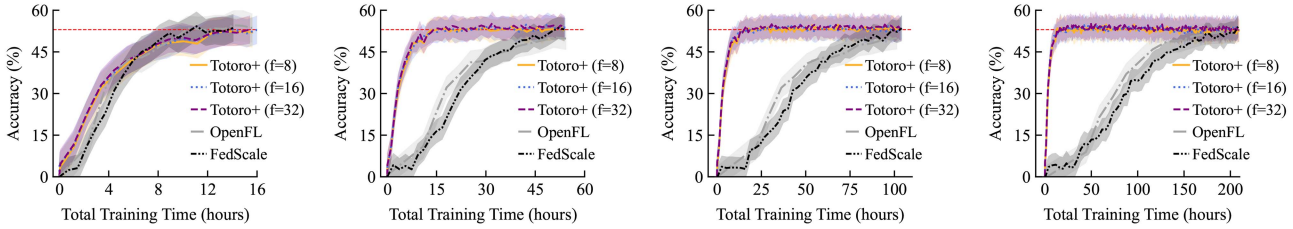
In this section, we evaluate Totoro$^+$'s performance on model training when an increasing number of models (1 to 20) are simultaneously trained on large-scale edge topologies, and compare it with OpenFL [25] and FedScale [26]. Both FedScale and OpenFL rely on a server-client architecture and leverage a logically central coordinator to spawn aggregators and orchestrate many distributed clients to collaboratively train a model. For both the Google Speech [75] and FEMNIST [9] datasets, we evenly partitioned the data across the edge nodes such that each node contains samples from all classes. This IID setting allows us to isolate and compare pure system-level performance differences among Totoro$^+$, OpenFL, and FedScale under the same controlled conditions. All compared systems follow the same data partitioning strategy.

Totoro$^+$ reduces the total model training time: TABLE III summarizes the results of time-to-accuracy comparison of Totoro$^+$, OpenFL and FedScale. When 5∼20 models are simultaneously trained on the same platform, we notice that Totoro$^+$ speeds up the total training time 3.0×-14.0× to reach the equivalent model accuracy on the middle-scale Google Speech dataset; speedup on the large-scale FEMNIST dataset is 1.2×-11.5×. The speedup gap increases as the number of concurrently running applications' tasks increases. This is because both OpenFL and FedScale rely on a server-client architecture where a logically central coordinator orchestrates many distributed clients to collaboratively train a model. When there are a massive number of models that need to be trained simultaneously, the central coordinator needs to handle them one by one on a first-come, first-served basis, which causes large queuing delays. By contrast, Totoro$^+$'s distributed masters can train many models in parallel, thus precluding them from being stuck in a central coordinator. To further evaluate the effectiveness of Totoro$^+$, we compare it with decentralized federated learning baselines on additional models and datasets. The results are presented in the supplementary material.

Totoro$^+$ scales with #applications: Figs. 8 and 9 show the time-to-accuracy performance comparison of Totoro$^+$, OpenFL, and FedScale. The shaded regions represent the standard deviations, and the red dotted lines represent the target accuracies for Google Speech (53.0%) and FEMNIST (75.5%). The results show that Totoro$^+$ scales well with a large number of applications' training tasks. Totoro$^+$ takes almost the same total training time to reach model convergence for training 1 model (15.41 hours), 5 models (15.43 hours), 10 models (15.44 hours),

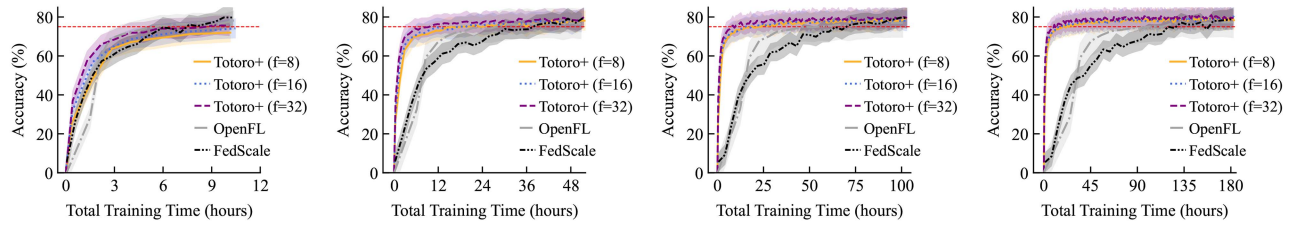
TABLE III
SUMMARY OF TIME-TO-ACCURACY COMPARISON OF TOTORO+, OPENFL, AND FEDSCALE. WE SET THREE DIFFERENT FANOUTS FOR TOTORO+'S DATAFLOW TREE: 8, 16, AND 32. TOTORO+ OUTPERFORMS STATE-OF-THE-ART FL SYSTEMS BY SPEEDING UP THE TRAINING TIME AT DIFFERENT SCALES. THE SPEEDUP GAP INCREASES AS THE NUMBER OF CONCURRENTLY RUNNING APPLICATIONS' TASKS INCREASES.

Task	Dataset	Accuracy Target	Model	Number of Applications	Speedup for OpenFL [25]			Speedup for FedScale [26]		
					Fan.=8	Fan.=16	Fan.=32	Fan.=8	Fan.=16	Fan.=32
Speech Recognition	Google Speech [75]	53.0%	ResNet-34 [76]	5	3.7×	3.1×	3.5×	3.6×	3.0×	3.4×
				10	6.4×	6.2×	6.9×	6.2×	6.0×	6.7×
				20	13.0×	11.8×	14.0×	12.4×	11.3×	13.5×
Image Classification	FEMNIST [9]	75.5%	ShuffleNet V2 [77]	5	1.2×	1.4×	3.1×	1.4×	1.6×	3.4×
				10	2.4×	3.2×	5.6×	2.7×	3.5×	6.1×
				20	5.0×	5.5×	10.3×	5.6×	6.1×	11.5×



(a) 1 application's model (b) 5 models trained simultaneously (c) 10 models trained simultaneously (d) 20 models trained simultaneously

Fig. 8. Time-to-accuracy comparison of Totoro+, OpenFL, and FedScale. When 5∼20 applications' models are simultaneously trained, Totoro+ speeds up the total training time 3.0×-14.0× to reach the equivalent model accuracy on the middle-scale Google Speech dataset.



(a) 1 application's model (b) 5 models trained simultaneously (c) 10 models trained simultaneously (d) 20 models trained simultaneously

Fig. 9. Time-to-accuracy comparison of Totoro+, OpenFL, and FedScale. When 5∼20 applications' models are simultaneously trained, Totoro+ speeds up the total training time 1.2×-11.5× to reach the equivalent model accuracy on the large-scale FEMNIST dataset.

and 20 models (15.47 hours) when fanout is equal to 32. The rationale behind the results lies in that (1) Totoro+ decomposes the conventional "*1:n*" architecture into an "*m:n*" architecture, which ensures that every peer can participate in the process of model training, gradients calculation, and aggregation; and (2) ideally, any peer in the system can act as a worker, a master, a forwarder, or any combination of the above. This helps avoid the scalability bottleneck caused by any single node, auto-scale as needed, balance the workload, and improve the scalability.

E. Adaptivity Analysis

We evaluate Totoro+'s game-theoretic distributed hop-by-hop routing algorithm using a real-world distribution of edge servers and end devices in the EUA dataset [73]. We randomly extracted 900 end devices and 100 edge servers in Melbourne, Australia, from coordinate −37.820 to −37.808 in latitude and 144.955 to 144.975 in longitude, as shown in Fig. 10.

These 1,000 nodes establish a dataflow tree, where the 900 end devices are worker nodes, an edge server serves as a master, and the remaining 99 edge servers are internal nodes. The worker

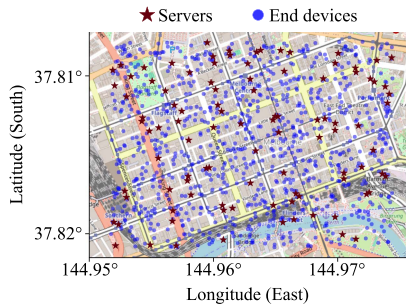


Fig. 10. 100 edge servers and 900 end devices in Melbourne, Australia, from the EUA dataset [73].

and internal nodes run three algorithms to select the next hops for model updates.

- Totoro+: The game-theoretic distributed hop-by-hop routing algorithm in Algorithm 1.
- Totoro [1]: The bandit-based distributed hop-by-hop algorithm that does not consider bandwidth capacities.

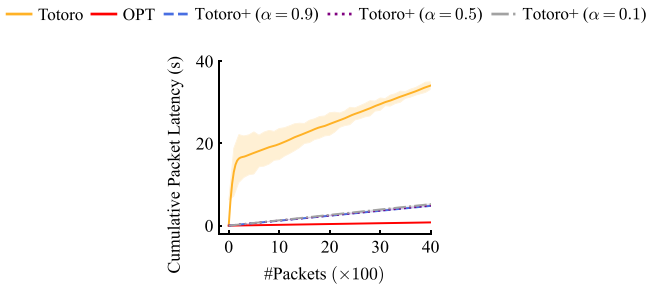


Fig. 11. Cumulative packet latency of different algorithms.

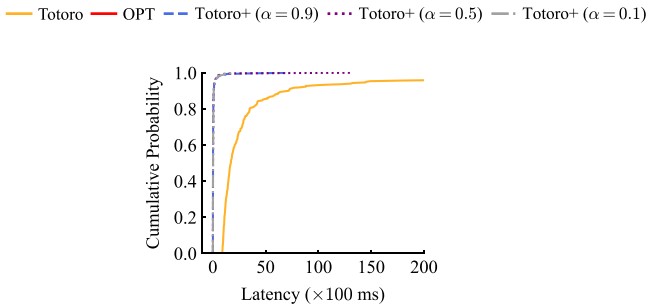


Fig. 12. Cumulative probabilities of different algorithms over latency.

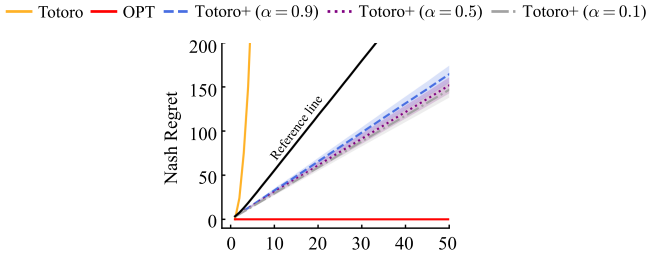


Fig. 13. Nash regret of different algorithms.

- OPT: The optimal algorithm that always selects the optimal next hop and considers bandwidth capacities.

We refer to [59], [78] to set the constrained bandwidth of each node in the range between 20 Mbps and 100 Mbps. Each worker and internal node has constrained bandwidth. The more packets it forwards simultaneously, the slower its data rate becomes. For example, if a node with 100 Mbps bandwidth needs to forward model updates from four nodes, the data rate is $100/4 = 25$. Also, the higher the data rate, the higher the corresponding reward.

Totoro$^+$ adapts to edge networks with constrained bandwidth capacity: Fig. 11 shows the cumulative packet latency of three algorithms. We can see that Totoro$^+$ achieves a cumulative packet latency that grows slowly. In contrast, Bandit suffers a higher cumulative packet latency. This is because Totoro$^+$ considers that nodes' bandwidth decreases proportionally if they have to forward many packets simultaneously. Hence, Totoro$^+$ tends to distribute network traffic more evenly over all nodes.

Fig. 12 shows cumulative probabilities of different algorithms over latency. Totoro$^+$ consistently outperforms Totoro in latency, regardless of the choice of α, delivering nearly all responses under 2000ms. The results explain why Totoro$^+$ can achieve lower cumulative packet latency in Fig. 11.

Totoro$^+$ achieves an ϵ-approximate Nash equilibrium: Fig. 13 shows the Nash regret comparison of Totoro$^+$, OPT, and a reference line. We refer to the solution from [48] as the reference line, where the solution is proved to achieve sublinear Nash regret with the G-optimal design. The results show that Totoro$^+$ achieves a sublinear Nash regret closer to OPT. Therefore, according to Corollary 1, Totoro$^+$ achieves an ϵ-approximate Nash equilibrium. In contrast, Totoro does not consider bandwidth capacities but selects the optimal next hop so far. Therefore, when too many nodes select the same next hop, the overall communication time increases, which significantly increases Nash regret.

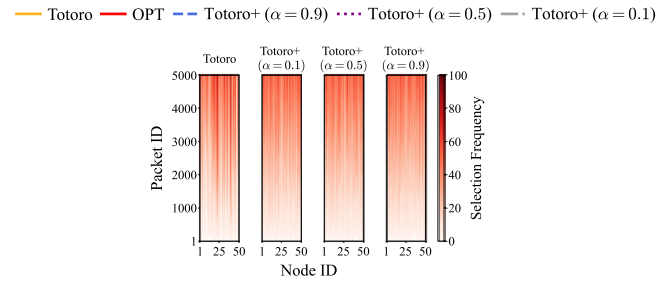


Fig. 14. Node selection frequencies generated by different algorithms.

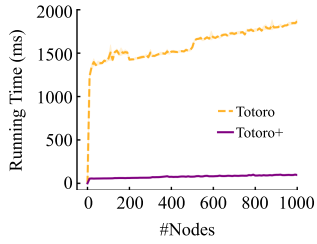


Fig. 15. Running time comparison.

Fig. 14 shows the node selection frequencies generated by different algorithms, in which each color grid shows the frequencies of selecting the x_{th} node for each packet. The X-axis represents the sequential order of next-hop nodes. The Y-axis represents the sequential order of packets. Although Totoro relies on a distributed hop-by-hop routing algorithm to send packets, it does not distribute packets evenly across all possible next hops. Instead, Totoro$^+$'s improved game-theoretic routing algorithm considers bandwidth capacities across nodes and distributes packets more evenly across all possible next hops, thereby leading to lower Nash regret.

Totoro$^+$ forwards packets rapidly: Fig. 15 shows the running time comparison of Totoro and Totoro$^+$. We can see that Totoro$^+$ maintains consistently low running time ($\approx$ 50 ms) regardless of the number of nodes, whereas Totoro's running time sharply increases up to around 1500 ms when the number of nodes increases from 0 to 100. This is because a node has to run the convex optimization to evaluate the empirical transmission cost for each possible hop. Instead, Totoro$^+$'s operations can be implemented by parallel matrix multiplications.

Fig. 16 shows the Totoro$^+$'s running time breakdown. Lines 5, 6, 7, and 8 represent 4 lines in Algorithm 1. We can see that the proportion of Line 5 increases significantly when the number of nodes increases. This is because Line 5 requires calculating determinant among all possible policies, which is consistent with the results in Theorem 2.

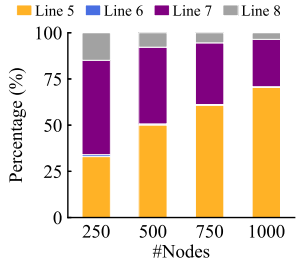


Fig. 16. Totoro+'s running time breakdown.

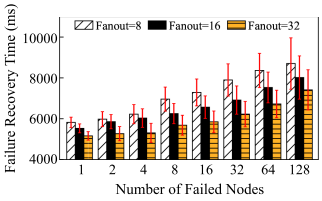


Fig. 17. Totoro+ tolerates many simultaneous node failures in a tree.

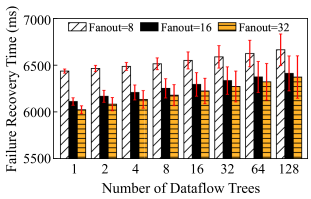


Fig. 18. Totoro+ achieves a stable recovery time for many simultaneous node failures in many trees.

F. Failure Recovery Analysis

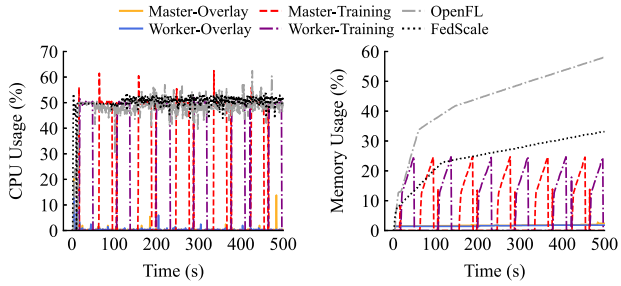
In this section, we evaluate how long it takes Totoro+ to recover from node failures in a single and many dataflow trees.

Totoro+ tolerates simultaneous node failures in a dataflow tree: Fig. 17 shows the failure recovery time for recovering a single dataflow tree with an exponentially increasing number of failed nodes. The tree has 1,000 nodes at the beginning, and then a varying number of random nodes fail or leave the tree simultaneously. The results show that the failure recovery time increases linearly when the number of failed nodes grows exponentially (from 1 to 128). This is because only $\mathcal{O}(\log_{2^b} N)$ nodes are involved in recovering the tree. Even though the root fails, its immediate child nodes can locate a new root, and it can recover the state from the replicas.

Totoro+ achieves a stable recovery time for many simultaneous trees' failures: Fig. 18 shows the failure recovery time for an exponentially increasing number of dataflow trees. Each tree has at most 1,000 nodes, and 5% of nodes fail or leave simultaneously. The results show that Totoro+ achieves a stable recovery time for many simultaneous trees' failures. This is because each failed node can be quickly detected and recovered by its neighbors through keep-alive messages without having to talk to a central coordinator, so many simultaneous failures can be repaired in parallel.

G. Overhead Analysis

We train a feedforward model for text classification with a single Totoro+'s dataflow tree of 10 nodes and compare the overhead with OpenFL and FedScale.



(a) The CPU overhead. (b) The memory overhead.

Fig. 19. Overhead comparison of Totoro+, OpenFL, and FedScale.

CPU overhead: Fig. 19(a) shows the CPU overhead comparison of Totoro+, OpenFL, and FedScale. For a fair comparison, we divide the CPU overhead into two parts: "Overlay" and "Training", where Overlay represents DHT-related tasks(including constructing P2P overlay, overlay maintenance, building dataflow trees, replanning paths, etc.), whereas Training includes any FL-related tasks (e.g., model training, model validation, model aggregation, etc.). Also, "Master" and "Worker" represent resource usage on the master node and worker node in a Totoro+'s dataflow tree, respectively. The results show that the master and worker nodes collaboratively run the FL-related tasks, so the CPU usage on these two nodes takes turns reaching the peak. In contrast, OpenFL and FedScale maintain high CPU usage all the time. For DHT-related tasks, Totoro+ only adds negligible CPU overhead, demonstrating Totoro+'s portability to resource-constrained edge nodes.

Memory overhead: Fig. 19(b) shows the memory overhead comparison of Totoro+, OpenFL, and FedScale. The initial increase in the memory overhead of Overlay is because of the construction of the P2P overlay, routing tables, neighborhood sets, leaf sets, and dataflow trees. After the DHT starts functioning, no additional memory overhead is added in either the master (root node) or the leaf nodes. For FL-related tasks, the memory usage of Training on the master and worker nodes matches the trend of CPU usage. OpenFL and FedScale demand more and more memory as they allocate and cache models, training data, and other necessary metadata in memory all the time.

VIII. Related Work

This section discusses existing FL system designs and their shortcomings for deployment in edge platforms (TABLE IV).

Centralized FL systems: The centralized server-client architecture is widely used in state-of-the-art FL systems such as Oort [35], FedAT [79], PyramidFL [80], TiFL [34], and FEDRECON [103]. In these systems, a powerful centralized parameter server and sufficient node-to-server bandwidth are provided to maintain the communications between the parameter server and clients. These systems focus on optimizing participant selection strategies [34], [35], [79], [80], improving communication efficiency [39], [104], tackling data heterogeneity [56], [83], [105], or ensuring privacy [57], [106]. For example, many algorithms have been proposed to reduce the communication overhead concentrated on the central server by reducing communication rounds with local updates allowance [107], [108], employing compression techniques to reduce the transmitted bits [109], [110], and sampling a subset of clients [34], [35],

TABLE IV
OVERVIEW OF STATE-OF-THE-ART FEDERATED LEARNING SYSTEM DESIGNS COMPARED TO TOTORO$^+$

Federated learning systems	Architecture	Central parameter server	Computing environment	Master/worker communication structure	Type of learned model	Scale to massive diverse applications	Adapt to edge networks
FedAT [79], TiFL [34], Oort [35], PyramidFL [80]	Centralized	✓	Datacenter	Hub-and-spoke	Global consensus	✗	✗
FLOAT [10], REFL [11]	Centralized	✓	Cloud	Hub-and-spoke	Global consensus	✗	✓
AdaFL [81], FeS [82], DoFed-SPP [83]	Centralized	✓	Edge	Hub-and-spoke	Global consensus	✗	✓
Client-edge-cloud [84], Edge-DemLearn [45]	Hierarchical	✓	Edge	Hub-and-spoke	Global consensus	✗	✗
HiFlash [85], Hwamei [86], ShapeFL [87], FedUC [88], AHFL [89]	Hierarchical	✓	Edge	Dynamically structured tree	Global consensus	✗	✓
D^2 [90], BrainTorrent [91], Lalitha et al. [92]	Decentralized	✗	Datacenter	P2P (one-hop neighbor)	Global consensus	✗	✗
ByRDiE [93], BRIDGE [94], Gupta et al. [95]	Decentralized	✗	Datacenter	P2P (one-hop neighbor)	Global consensus	✗	✗
Bellet et al. [96], Vanhaesebrouck et al. [97]	Decentralized	✗	Datacenter	P2P (one-hop neighbor)	Personalized	✗	✗
AsyDFL [98], YOGA [99], FRACTAL [100], DFLStar [101], FedHP [102]	Decentralized	✗	Edge	P2P (one-hop neighbor)	Personalized	✗	✓
Totoro$^+$	Decentralized	✗	Edge	Dynamically structured tree	Global consensus	✓	✓

[79]. Some systems harness workers' resource strengths with feedback integration [10], [11], [81], [82]. FLOAT [10] adopts a multi-objective reinforcement learning agent that autonomously selects optimization techniques and configurations to meet individual client resource conditions. AdaFL [81] aims at NLP federated learning and introduces an online configurator that automatically adjusts the tuning depth and width based on training accuracy, per-round training time, and network time for each communication round. While these systems work well in resource-rich datacenters, the centralized control plane with a static assignment of parameter servers may not adapt well to resource-constrained edge settings with millions of nodes, numerous FL applications, and unreliable network links.

Hierarchical FL systems: In hierarchical FL systems, an intermediate layer (e.g., edge servers) is inserted between the central server and client devices to perform partial aggregations on distributed local models before the global aggregation. The intermediate layer helps mitigate the non-independent and identically distributed (non-IID) effects of client data [45], control the staleness of model updates [85] from client devices with heterogeneous resources, synchronize edge and cloud aggregation frequencies [86], associate client devices with edge servers for training and communication efficiency [87], [88], reduce the communication burden on the central server [84], [111], and offload the computation burden from client devices to edge servers [89]. Abad [111] employ small cell-base stations to orchestrate federated learning among mobile users and periodically exchange model updates with the macro-base station for global model learning. AHFL [89] proposes to offload the training overhead from end devices to edge servers and inject local noise to secure data privacy while the cloud is responsible for model aggregation only. Although this structure enhances scalability with additional edge aggregators, they may become points of failure, causing disconnection between the central server and client devices and interrupting the training process. Some studies like HiFlash [85], FedUC [88], and AHFL [89] leverage asynchronous aggregation between edge aggregators and cloud aggregators to avoid points of failure, but additional efforts to balance cloud aggregation and edge aggregation frequencies.

Peer-to-peer FL (P2PFL) systems: P2PFL is a distributed learning protocol without a central parameter server. The key idea is to leverage P2P communication between individual clients to exchange model updates. Model aggregation and updates are performed locally on client nodes with the acquired gradients from their one-hop neighbors. Previous P2PFL research considered design from different angles: adversarial models (non-Byzantine vs. Byzantine settings) and learned models (global consensus vs. personalized). Non-Byzantine algorithms [90], [91], [92] focus on improving convergence speed. Conversely, Byzantine algorithms [93], [94], [95] aim to ensure that the trained model remains close to the model learned without adversaries. Global consensus learning [90], [91], [92], [93], [94], [95] aims to generate a single global model at the end of training, while personalized learning [96], [97] allows each peer to train a unique model. Some studies aim to improve the training efficiency of P2PFL [98], [99], [100], [102]. AsyDFL [98] proposes to push a subset of gradient updates to peer to trade-off communication cost and training accuracy and leverage asynchronous model aggregation to mitigate the straggler effect. FedHP [102] integrates adaptive control of both local updating frequency and network topology to speed up convergence and improve model accuracy. These efforts mostly focus on novel algorithm development for P2P model aggregation and communication topology construction without considering cross-layer support of implementation. Totoro$^+$ backs up the P2PFL work with a back-end architecture for task management, application-specific customization, and failure recovery.

IX. CONCLUSION

We present Totoro$^+$, a novel scalable federated learning system for edge networks. It presents three design innovations: a locality-aware P2P multi-ring structure, a publish/subscribe-based forest abstraction, and a game-theoretic path planning model. We evaluate Totoro$^+$ on 500 Amazon EC2 nodes using real-world CV and NLP datasets at different scales. We compare Totoro$^+$ with the state-of-the-art and demonstrate substantial advancements in scalability and load balancing while significantly speeding up the total training time for many concurrently running applications, reducing the communication overhead, and efficiently adapting to unreliable edge networks and churns. We provide a theoretical guarantee that the game-theoretic path planning model achieves an ϵ-approximate Nash equilibrium. We present the discussions and future work in the supplementary material.

ACKNOWLEDGMENT

The authors would like to thank the Editor-in-Chief, Prof. Xian-He Sun; the Associate Editor, Prof. Amelie Chi Zhou; and the anonymous reviewers for their valuable comments and constructive suggestions, which significantly improved the quality of this article. The authors also thank Yue Lin and Shih-Jhi Liang for their assistance with the experimental evaluation and validation of the proposed routing algorithm.

REFERENCES

[1] C.-W. Ching et al., "Totoro: A scalable federated learning engine for the edge," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 182–199, doi: 10.1145/3627703.3629575.

[2] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous online federated learning for edge devices with non-IID data," in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 15–24.

[3] Y. Chen, X. Sun, and Y. Jin, "Communication-efficient federated deep learning with layerwise asynchronous model update and temporally weighted aggregation," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 10, pp. 4229–4238, Oct. 2020.

[4] V. Smith, C.-K. Chiang, M. Sanjabi, and A. S. Talwalkar, "Federated multi-task learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4427–4437.

[5] A. Hard et al., "Federated learning for mobile keyboard prediction," 2018, *arXiv:1811.03604*.

[6] T. Yang et al., "Applied federated learning: Improving Google keyboard query suggestions," 2018, *arXiv:1812.02903*.

[7] H. Ludwig et al., "IBM federated learning: An enterprise framework white paper v0. 1," 2020, *arXiv:2007.10987*.

[8] "Tensorflow federated: Machine learning on decentralized data," 2026. [Online]. Available: https://www.tensorflow.org/federated

[9] S. Caldas et al., "LEAF: A benchmark for federated settings," 2018, *arXiv:1812.01097*.

[10] A. F. Khan, A. A. Khan, A. M. Abdelmoniem, S. Fountain, A. R. Butt, and A. Anwar, "FLOAT: Federated learning optimizations with automated tuning," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 200–218.

[11] A. M. Abdelmoniem, A. N. Sahu, M. Canini, and S. A. Fahmy, "REFL: Resource-efficient federated learning," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 215–232.

[12] "PySyft: A library for easy federated learning," 2025. [Online]. Available: https://github.com/OpenMined/PySyft

[13] "United states department of transportation annual modal research plans," 2023. [Online]. Available: https://www.transportation.gov/sites/dot.gov/files/2024-08/AMRP\%20FY2024\%20-\%202025\%20ITSJPO.pdf

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

[15] R. Du, K. Han, R. Gupta, S. Chen, S. Labi, and Z. Wang, "Driver monitoring-based lane-change prediction: A personalized federated learning framework," in *Proc. IEEE Intell. Veh. Symp.*, 2023, pp. 1–7.

[16] N. Wang, X. Li, Z. Guan, and S. Yuan, "FedStream: A federated learning framework on heterogeneous streaming data for next-generation traffic analysis," *IEEE Trans. Netw. Sci. Eng.*, vol. 11, no. 3, pp. 2485–2496, May/Jun. 2024.

[17] X. Wang, W. Liu, H. Lin, J. Hu, K. Kaur, and M. S. Hossain, "AI-empowered trajectory anomaly detection for intelligent transportation systems: A hierarchical federated learning approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 24, no. 4, pp. 4631–4640, Apr. 2023.

[18] "Bittorrent," 2026. [Online]. Available: http://www.bittorrent.com/

[19] "Storj.io," 2026. [Online]. Available: http://storj.io/

[20] "FreeNet: The free network," 2026. [Online]. Available: https://freenet.org/

[21] M. Zhao et al., "Peer-assisted content distribution in Akamai netsession," in *Proc. Conf. Internet Meas. Conf.*, 2013, pp. 31–42.

[22] K. Croman et al., "On scaling decentralized blockchains: (A position paper)," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2016, pp. 106–125.

[23] "Pastry," 2026. [Online]. Available: https://www.freepastry.org/FreePastry/

[24] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[25] "OpenFL - An open-source framework for federated learning," 2026. [Online]. Available: https://openfl.readthedocs.io/en/latest/

[26] F. Lai et al., "FedScale: Benchmarking model and system performance of federated learning at scale," in *Proc. 39th Int. Conf. Mach. Learn.*, 2022, pp. 11814–11827. [Online]. Available: https://proceedings.mlr.press/v162/lai22a.html

[27] S. R. Pandey, N. H. Tran, M. Bennis, Y. K. Tun, A. Manzoor, and C. S. Hong, "A crowdsourcing framework for on-device federated learning," *IEEE Trans. Wireless Commun.*, vol. 19, no. 5, pp. 3241–3256, May 2020.

[28] R. Xiong et al., "CoPiFL: A collusion-resistant and privacy-preserving federated learning crowdsourcing scheme using blockchain and homomorphic encryption," *Future Gener. Comput. Syst.*, vol. 156, pp. 95–104, 2024.

[29] D. Huba et al., "PAPAYA: Practical, private, and scalable federated learning," *Proc. Mach. Learn. Syst.*, vol. 4, pp. 814–832, 2022.

[30] E. Wang, B. Chen, M. Chowdhury, A. Kannan, and F. Liang, "FLINT: A platform for federated learning integration," *Proc. Mach. Learn. Syst.*, vol. 5, pp. 21–34, 2023.

[31] K. Bonawitz et al., "Towards federated learning at scale: System design," *Proc. Mach. Learn. Syst.*, vol. 1, pp. 374–388, 2019.

[32] M. Paulik et al., "Federated evaluation and tuning for on-device personalization: System design & applications," 2021, *arXiv:2102.08503*.

[33] R. Pietrantuono, M. Ficco, and F. Palmieri, "Testing the resilience of MEC-based IoT applications against resource exhaustion attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 804–818, Mar./Apr. 2024.

[34] Z. Chai et al., "TiFL: A tier-based federated learning system," in *Proc. 29th Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2020, pp. 125–136.

[35] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "OORT: Efficient federated learning via guided participant selection," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 19–35.

[36] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *Proc. IEEE Int. Conf. Commun.*, 2019, pp. 1–7.

[37] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1707–1718.

[38] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar, "signSGD with majority vote is communication efficient and fault tolerant," in *Proc. Int. Conf. Learn. Representations*, 2019.

[39] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2017, pp. 1273–1282.

[40] D. Stripelis, P. M. Thompson, and J. L. Ambite, "Semi-synchronous federated learning for energy-efficient training and accelerated convergence in cross-silo settings," *ACM Trans. Intell. Syst. Technol.*, vol. 13, no. 5, pp. 1–29, 2022.

[41] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," in *Proc. Adv. Neural Inf. Process. Syst. Workshop Optim. Mach. Learn.*, 2020.

[42] S. Wang et al., "Adaptive federated learning in resource constrained edge computing systems," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1205–1221, Jun. 2019.

[43] L. U. Khan et al., "Federated learning for edge networks: Resource optimization and incentive mechanism," *IEEE Commun. Mag.*, vol. 58, no. 10, pp. 88–93, Oct. 2020.

[44] M. S. Aslanpour et al., "Serverless edge computing: Vision and challenges," in *Proc. Australas. Comput. Sci. Week Multiconference*, 2021, pp. 1–10.

[45] S. R. Pandey et al., "Edge-assisted democratized learning toward federated analytics," *IEEE Internet Things J.*, vol. 9, no. 1, pp. 572–588, Jan. 2022.

[46] C.-W. Ching et al., "AgileDART: An agile and scalable edge stream processing engine," *IEEE Trans. Mobile Comput.*, vol. 24, no. 5, pp. 4510–4528, May 2025.

[47] Z. Fu, J. Ren, D. Zhang, Y. Zhou, and Y. Zhang, "Kalmia: A heterogeneous QoS-aware scheduling framework for DNN tasks on edge servers," in *Proc. Conf. Comput. Commun.*, 2022, pp. 780–789.

[48] Q. Cui, Z. Xiong, M. Fazel, and S. S. Du, "Learning in congestion games with bandit feedback," in *Proc. Adv. Neural Inf. Process. Syst.*, 2022, pp. 11009–11022.

[49] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM Int. Conf. Distrib. Syst. Platforms*, 2001, pp. 329–350.

[50] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001.

[51] B. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J. Sel. Areas Commun.*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[52] Y. Chen, X. Qin, J. Wang, C. Yu, and W. Gao, "FedHealth: A federated transfer learning framework for wearable healthcare," *IEEE Intell. Syst.*, vol. 35, no. 4, pp. 83–93, Jul./Aug. 2020.

[53] D. Y. Zhang, Z. Kou, and D. Wang, "FedSens: A federated learning approach for smart health sensing with class imbalance in resource constrained edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.

[54] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 53–65.

[55] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *Proc. 21st Annu. Joint Conf. IEEE Comput. Commun. Societies*, 2002, pp. 1190–1199.

[56] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proc. Mach. Learn. Syst.*, vol. 2, pp. 429–450, 2020.

[57] R. C. Geyer, T. Klein, and M. Nabi, "Differentially private federated learning: A client level perspective," in *Proc. Adv. Neural Inf. Process. Syst. Workshop: Mach. Learn. Phone Other Consum. Devices*, 2017.

[58] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 493–506.

[59] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, and M. Xiao, "Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 257–266.

[60] M. S. Talebi, Z. Zou, R. Combes, A. Proutiere, and M. Johansson, "Stochastic online shortest path routing: The value of feedback," *IEEE Trans. Autom. Control*, vol. 63, no. 4, pp. 915–930, Apr. 2018.

[61] S. Bubeck et al., "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations Trends Mach. Learn.*, vol. 5, no. 1, pp. 1–122, 2012.

[62] T. Roughgarden, "Algorithmic game theory," *Commun. ACM*, vol. 53, no. 7, pp. 78–86, 2010.

[63] D. Ding, C.-Y. Wei, K. Zhang, and M. Jovanovic, "Independent policy gradient for large-scale Markov potential games: Sharper rates, function approximation, and game-agnostic convergence," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 5166–5220.

[64] J. W. Crandall and M. A. Goodrich, "Learning to compete, compromise, and cooperate in repeated general-sum games," in *Proc. 22nd Int. Conf. Mach. Learn.*, 2005, pp. 161–168, doi: 10.1145/1102351.1102372.

[65] B. Hambly, R. Xu, and H. Yang, "Policy gradient methods find the nash equilibrium in n-player general-sum linear-quadratic games," *J. Mach. Learn. Res.*, vol. 24, no. 139, pp. 1–56, 2023. [Online]. Available: http://jmlr.org/papers/v24/21-0842.html

[66] T. Lattimore and C. Szepesvári, *Bandit Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2020.

[67] Z. Wen, B. Kveton, M. Valko, and S. Vaswani, "Online influence maximization under independent cascade model with semi-bandit feedback," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 3026–3036.

[68] M. Frank et al., "An algorithm for quadratic programming," *Nav. Res. Logistics Quart.*, vol. 3, no. 1/2, pp. 95–110, 1956.

[69] "PyTorch," 2026. [Online]. Available: https://pytorch.org/

[70] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 8, pp. 1489–1499, Oct. 2002.

[71] "Torchvision," 2026. [Online]. Available: https://docs.pytorch.org/vision/stable/index.html

[72] "Hugging face transformers," 2026. [Online]. Available: https://huggingface.co/docs/transformers/en/index

[73] P. Lai et al., "Optimal edge user allocation in edge computing with variable sized vector bin packing," in *Proc. 16th Int. Conf. Service-Oriented Comput.*, 2018, pp. 230–245.

[74] "Symbiotic lab," 2026. [Online]. Available: https://symbioticlab.org/

[75] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.

[76] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[77] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical guidelines for efficient CNN architecture design," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 122–138.

[78] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "AWStream: Adaptive wide-area streaming analytics," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 236–252.

[79] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala, "FedAT: A high-performance and communication-efficient federated learning system with asynchronous tiers," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–16.

[80] C. Li, X. Zeng, M. Zhang, and Z. Cao, "PyramidFL: A fine-grained client selection framework for efficient federated learning," in *Proc. 28th Annu. Int. Conf. Mobile Comput. Netw.*, 2022, pp. 158–171.

[81] D. Cai, Y. Wu, S. Wang, F. X. Lin, and M. Xu, "Efficient federated learning for modern NLP," in *Proc. 29th Annu. Int. Conf. Mobile Comput. Netw.*, 2023, pp. 1–16.

[82] D. Cai, S. Wang, Y. Wu, F. X. Lin, and M. Xu, "Federated few-shot learning for mobile NLP," in *Proc. 29th Annu. Int. Conf. Mobile Comput. Netw.*, 2023, pp. 1–17.

[83] C.-W. Ching, J.-M. Chang, J.-J. Kuo, and C.-Y. Wang, "Dual-objective personalized federated service system with partially-labeled data over wireless networks," *IEEE Trans. Serv. Comput.*, vol. 16, no. 5, pp. 3265–3279, Sep./Oct. 2023.

[84] L. Liu, J. Zhang, S. Song, and K. B. Letaief, "Client-edge-cloud hierarchical federated learning," in *Proc. IEEE Int. Conf. Commun.*, 2020, pp. 1–6.

[85] Q. Wu et al., "HiFlash: Communication-efficient hierarchical federated learning with adaptive staleness control and heterogeneity-aware client-edge association," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1560–1579, May 2023.

[86] T. Qi, Y. Zhan, P. Li, J. Guo, and Y. Xia, "Hwamei: A learning-based synchronization scheme for hierarchical federated learning," in *Proc. IEEE 43rd Int. Conf. Distrib. Comput. Syst.*, 2023, pp. 534–544.

[87] Y. Deng et al., "A communication-efficient hierarchical federated learning framework via shaping data distribution at edge," *IEEE/ACM Trans. Netw.*, vol. 32, no. 3, pp. 2600–2615, Jun. 2024.

[88] Q. Ma, Y. Xu, H. Xu, J. Liu, and L. Huang, "FedUC: A unified clustering approach for hierarchical federated learning," *IEEE Trans. Mobile Comput.*, vol. 23, no. 10, pp. 9737–9756, Oct. 2024.

[89] Y. Guo, F. Liu, T. Zhou, Z. Cai, and N. Xiao, "Privacy vs. efficiency: Achieving both through adaptive hierarchical federated learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1331–1342, Apr. 2023.

[90] H. Tang, X. Lian, M. Yan, C. Zhang, and J. Liu, "D^2: Decentralized training over decentralized data," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 4848–4856.

[91] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger, "BrainTorrent: A peer-to-peer environment for decentralized federated learning," 2019, *arXiv:1905.06731*.

[92] A. Lalitha, S. Shekhar, T. Javidi, and F. Koushanfar, "Fully decentralized federated learning," in *Proc. 3rd Workshop Bayesian Deep Learn.*, 2018, pp. 1–9.

[93] Z. Yang and W. U. Bajwa, "ByRDiE: Byzantine-resilient distributed coordinate descent for decentralized learning," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 5, no. 4, pp. 611–627, Dec. 2019.

[94] C. Fang, Z. Yang, and W. U. Bajwa, "BRIDGE: Byzantine-resilient decentralized gradient descent," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 8, pp. 610–626, 2022.

[95] N. Gupta and N. H. Vaidya, "Byzantine fault-tolerance in peer-to-peer distributed gradient-descent," 2021, *arXiv:2101.12316*.

[96] A. Bellet, R. Guerraoui, M. Taziki, and M. Tommasi, "Personalized and private peer-to-peer machine learning," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2018, pp. 473–481.

[97] P. Vanhaesebrouck, A. Bellet, and M. Tommasi, "Decentralized collaborative learning of personalized models over networks," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2017, pp. 509–517.

[98] Y. Liao, Y. Xu, H. Xu, M. Chen, L. Wang, and C. Qiao, "Asynchronous decentralized federated learning for heterogeneous devices," *IEEE/ACM Trans. Netw.*, vol. 32, no. 5, pp. 4535–4550, Oct. 2024.

[99] J. Liu, J. Liu, H. Xu, Y. Liao, Z. Wang, and Q. Ma, "YOGA: Adaptive layer-wise model aggregation for decentralized federated learning," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 1768–1780, Apr. 2024.

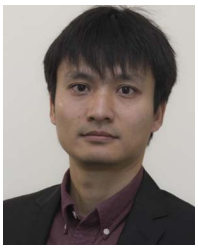
[100] Q. Ma, J. Liu, H. Xu, Q. Jia, and R. Xie, "FRACTAL: Data-aware clustering and communication optimization for decentralized federated learning," *IEEE Trans. Big Data*, vol. 11, no. 5, pp. 2102–2118, Oct. 2025.

[101] B. Soltani, V. Haghighi, Y. Zhou, Q. Z. Sheng, and L. Yao, "DFLStar: A decentralized federated learning framework with self-knowledge distillation and participant selection," in *Proc. 33rd ACM Int. Conf. Inf. Knowl. Manage.*, 2024, pp. 2108–2117.

[102] Y. Liao, Y. Xu, H. Xu, L. Wang, C. Qian, and C. Qiao, "Decentralized federated learning with adaptive configuration for heterogeneous participants," *IEEE Trans. Mobile Comput.*, vol. 23, no. 6, pp. 7453–7469, Jun. 2024.

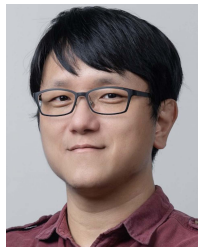
[103] K. Singhal, H. Sidahmed, Z. Garrett, S. Wu, J. Rush, and S. Prakash, "Federated reconstruction: Partially local federated learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 11220–11232.
[104] Z. Li, D. Kovalev, X. Qian, and P. Richtarik, "Acceleration for compressed gradient descent in distributed and federated optimization," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 5895–5904.
[105] S. J. Reddi et al., "Adaptive federated optimization," in *Proc. Int. Conf. Learn. Representations*, 2021.
[106] A. Girgis, D. Data, S. Diggavi, P. Kairouz, and A. T. Suresh, "Shuffled model of differential privacy in federated learning," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2021, pp. 2521–2529.
[107] T. Lin, S. U. Stich, K. K. Patel, and M. Jaggi, "Don't use large mini-batches, use local SGD," in *Proc. Int. Conf. Learn. Representations*, 2020.
[108] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD," *Proc. Mach. Learn. Syst.*, vol. 1, pp. 212–229, 2019.
[109] D. Basu, D. Data, C. Karakus, and S. N. Diggavi, "Qsparse-Local-SGD: Distributed SGD with quantization, sparsification, and local computations," *IEEE J. Sel. Areas Inf. Theory*, vol. 1, no. 1, pp. 217–226, May 2020.
[110] D. Rothchild et al., "FetchSGD: Communication-efficient federated learning with sketching," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 8253–8265.
[111] M. S. H. Abad, E. Ozfatura, D. GUndUz, and O. Ercetin, "Hierarchical federated learning across heterogeneous cellular networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2020, pp. 8866–8870.

Cheng-Wei Ching received the MS degree in computer science and information engineering from National Chung Cheng University, Taiwan, in 2021. He is currently working toward the PhD degree in computer science and engineering with the University of California Santa Cruz, USA. His research interests include distributed systems for machine learning, approximation algorithms and their applications, mobile edge computing.



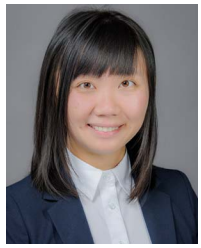
Xin Chen received the BS degree in computer science from Shandong University, China, in 2009, the MS degree in software engineering from Tsinghua University, China, in 2012, and the PhD degree in computer science from the Georgia Institute of Technology, USA, in 2020. His research interests include computer systems and machine learning systems, including solving path planning and collision detection problems.

Taehwan Kim received the BS degree from Bucknell University in 2020 and the MS degree from Virginia Tech in 2022. He is currently a software engineer with Google, where he focuses on Gemini post-training infrastructure.



Jian-Jhih Kuo (Member, IEEE) received the B.S. and Ph.D. degrees in computer science from National Chung Cheng University (CCU), Taiwan, in 2008, and National Tsing Hua University, Taiwan, in 2014, respectively. He was a Postdoctoral Fellow in the Institute of Information Science, Academia Sinica, Taiwan. He joined the Department of Computer Science and Information Engineering, CCU, Taiwan, in 2018, where he is currently an associate professor. His research interests include computer networking, quantum networking, traffic engineering, and cloud and edge computing. He was a recipient of the Ta-You Wu Memorial Award from National Science and Technology Council (NSTC), Taiwan, in 2025, NSTC Project for Excellent Junior Research Investigators in 2022 and 2025, and CCU Young Faculty Award in 2021.

Dilma Da Silva received the PhD degree in computer science from the Georgia Institute of Technology, USA, in 1997. She is a Regents Professor and holder of the Ford Design Professorship II in the Department of Computer Science and Engineering at Texas A&M University. From 2022 to 2026, she served in several leadership roles at the U.S. National Science Foundation. Her primary research interests include distributed systems, high-performance computing, operating systems, computer science education, and quantum computing.



Liting Hu received the BS degree in computer science from the Huazhong University of Science and Technology, China, in 2007, and the PhD degree in computer science from the Georgia Institute of Technology, USA, in 2016. She conducts AI-driven experimental computer systems research in stream processing systems, cloud and edge computing, distributed systems, and systems virtualization. Currently she is an associate professor at the University of California Santa Cruz.