EdgeTB: A Hybrid Testbed for Distributed Machine Learning at the Edge With High Fidelity

Lei Yang, Fulin Wen, Jiannong Cao, *Fellow, IEEE*, and Zhenyu Wang

Abstract—Distributed Machine Learning (DML) at the edge has become an essential topic for providing low-latency intelligence near the data sources. However, both the development and testing of DMLs lack sufficient support. Reusable libraries that abstract the general functionalities of DMLs are needed for rapid development. Moreover, existing physical testbeds are usually small and lack network flexibility, while virtual testbeds like simulators and emulators lack fidelity. This paper proposes a novel hybrid testbed EdgeTB, which provides numerous emulated nodes to generate large-scale and network-flexible test environments while incorporating physical nodes to guarantee fidelity. EdgeTB manages physical nodes and emulated nodes uniformly and supports arbitrary network topologies between nodes through dynamic configurations. Importantly, we propose Role-oriented development to support the rapid development of DMLs. Through case studies and experiments, we demonstrate that EdgeTB provides convenience for efficiently developing and testing DMLs in various structures with high fidelity and scalability.

Index Terms—Testbed, emulator, edge computing, distributed machine learning

1 Introduction

Recent years have witnessed an enormous increase in the number of mobile devices and IoT devices and the amount of data they generated. Transmitting this data to the cloud and processing puts tremendous pressure on the network and the cloud. Especially in latency-sensitive scenarios such as live streaming, many video data needs to be transmitted and processed. High latency will seriously degrade the user experience. Fog computing [1], also known as edge computing [2], is emerging to address this problem [3], [4]. It distributes numerous computing devices to the network edge, bringing computing and storage resources closer to the end-users. As a result, edge computing significantly reduces the latency for end-users to access network services, thereby improving the user experience.

Besides focusing on the improvements that edge computing brings to traditional applications [5], [6], researchers are also interested in deploying DMLs at the edge [7], [8]. In traditional centralized machine learning, the users need to upload their data to the cloud. Many robust and interconnected computing devices in the cloud efficiently train machine learning models based on these data. However, as people's awareness of privacy protection increases, they are unwilling to share their sensitive data. Therefore, deploying DMLs (e.g., Federated Learning [9], Gossip Learning [10], and E-Tree Learning [11]) at the edge, which can fully utilize data value while protecting user data privacy, has become a popular method. In Federated Learning, the user device always holds data and is also responsible for training tasks. Users only upload machine learning models, and the edge infrastructures will aggregate these models to speed up the training tasks. This is significantly different from traditional centralized machine learning because the performance of user devices is limited, and the network between user devices and edge infrastructures is unstable and slow. Therefore, the development and testing of DMLs used in edge computing scenarios should fully consider computing and network resources constraints.

Applications should be thoroughly tested before deployment to find potential problems as much as possible. The research community usually uses testbeds to provide test environments. There are mainly three types of testbeds, namely physical testbed, simulated testbed, and emulated testbed. To avoid ambiguity, we use "computing device" to refer to the physical hardware (e.g., PC, laptop, and smartphone) in the real world and use "node" (physical, virtual, simulated, and emulated) to refer to the entity in test environments.

The ideal testing method for high fidelity is to use the physical testbed composed of interconnected computing devices. In the physical testbed, each computing device directly acts as a physical node in the test environment. Moreover, computing devices are connected through networks (e.g., Ethernet, WiFi, and Bluetooth). Therefore, the physical testbed can provide test environments with high computing and network fidelity for applications. Some architectures or prototypes of fog computing physical testbed were proposed in [12], [13], [14]. Researchers may also use some Wireless Sensor Network (WSN) physical testbeds [15], [16], [17] to verify their IoT-based applications.

- *Lei Yang, Fulin Wen, and Zhenyu Wang are with the School of Software Engineering, South China University of Technology, Guangzhou 510006, China. E-mail: {sely, wangzy}@scut.edu.cn, 201921043987@mail.scut.edu.cn.*
- *Jiannong Cao is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China. E-mail: csjcao@comp.polyu.edu.cn.*

Manuscript received 5 July 2021; revised 22 Nov. 2021; accepted 18 Jan. 2022. Date of publication 25 Jan. 2022; date of current version 4 Apr. 2022.
This work was supported in part by the National Natural Science Foundation of China under Grant 61972161, in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2020A1515011496, in part by the Key Research and Development Program of Guangdong under Grant 2019B010154004, in part by Hong Kong RGC General Research Fund under Grants PolyU 152133/18 and PolyU 15217919, and in part by Hong Kong RGC Research Impact Fund 2019/20 under Grant R5060-19.
(Corresponding author: Lei Yang.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TPDS.2022.3144994

However, physical testbeds tend to be small in scale and expensive to deploy and maintain. In addition, the network topology between computing devices is fixed, making it difficult to provide a dynamic network environment similar to edge computing scenarios.

For lower cost and higher network flexibility, researchers can use the simulated testbed (simulator). The simulator abstractly models computing devices and networks, thereby providing test environments within a program (e.g., Java program and Python program). No simulator can fulfill the requirements of all types of test environments, so researchers have proposed various simulators for different research issues. CloudSim [18] is a well-known cloud computing simulator. Based on it, researchers have implemented many fog simulators, including iFogSim [19], MyiFogSim [20], and edgeCloudSim [21]. Some simulators are not based on the CloudSim, such as YAFS [22] and FogNetSim++ [23]. However, simulated nodes in the test environments are just instances in the program. Therefore, the simulator has not connected simulated nodes but only uses algorithms to simulate network link properties such as latency and packet loss. Moreover, since the simulator itself is a program, researchers cannot deploy the application to test it. They can only re-implement the main workflow of the application based on the simulator's abstract model. Due to the lack of computing and network fidelity, simulators are only suitable for roughly trying new ideas. Researchers still need to conduct more reliable verification.

The emulated testbed (emulator) achieves a good balance between the physical testbed and the simulated testbed by using virtualization technology (e.g., Virtual Machine (VM) technology and containerization technology). It creates multiple emulated nodes with isolated computing space and network space in a single computing device. Therefore, emulated nodes can run applications and communicate with each other through networks. EmuFog [24] used containerization technology to create multiple emulated nodes on a single computing device, while EmuEdge [25] used VM technology to achieve this goal. [26], [27], [28] use multiple computing devices as container emulators simultaneously to support larger scales. In these works, all computing devices are used as emulators to form a purely emulated testbed, regardless of the difference in computing and storage resources of the computing devices. However, only computing devices with sufficient computing and storage resources are suitable as emulators, while low-performance computing devices are unsuitable. Thus, although the emulator's computing and network fidelity are higher than the simulator's, there is still a big gap between the emulator and the physical testbed.

In the research of DMLs, the support for testing is insufficient, and the support for development is also lacking. On the one hand, DMLs are applications that need to be executed to obtain results (e.g., model accuracy and model converge time), so simulators which cannot deploy applications are not suitable for testing DMLs. Moreover, the existing physical testbeds and emulators are oriented to specific research fields other than DMLs or are general-purpose platforms and do not provide dedicated support for DMLs development. On the other hand, many parts in the DMLs applications for solving various research issues are reusable.

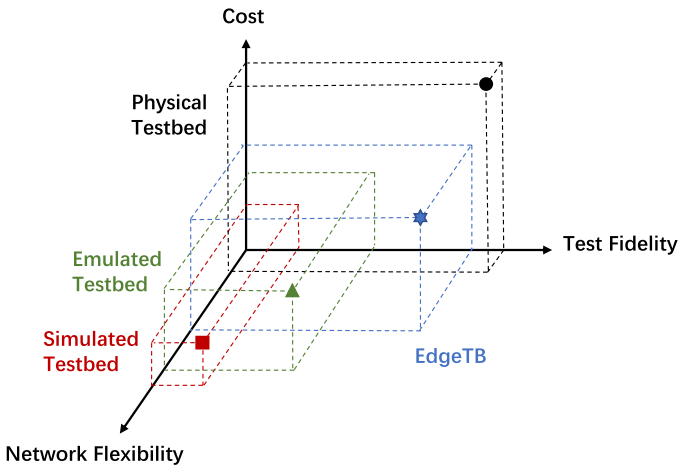


Fig. 1. A qualitative comparison among existing testbeds for edge computing.

For example, in Federated Learning, the research on neural network models and client selection can use the same model training, aggregation, and transmission function modules. However, there is currently no abstraction and encapsulation of these general function modules. As a result, researchers can only rely on more native machine learning libraries, such as *TensorFlow*, to develop all the function modules in DMLs. This dramatically increases the difficulty of developing DMLs and is not conducive to verifying DMLs and experiments among peers.

This paper proposes EdgeTB, a novel hybrid testbed for studying DMLs in edge computing scenarios. With EdgeTB, researchers can quickly develop DMLs and test them in high-fidelity, large-scale, and network-flexible test environments. In EdgeTB, a computing device can act as a physical node or an emulator to provide emulated nodes according to the computing device's performance or the researchers' requirements. On the one hand, the existence of physical nodes improves the computing and network fidelity of EdgeTB, making it close to the physical testbed. On the other hand, compared with the physical testbed, the adoption of emulators makes it easier for EdgeTB to generate large-scale and network-flexible test environments. Furthermore, to improve scalability, EdgeTB adopts Controller-Worker architecture and provides tools to manage physical and emulated nodes uniformly. Importantly, EdgeTB allows users to dynamically configure the network topology between nodes to generate non-fixed edge networks when running tests.

We make a qualitative comparison among existing testbeds for edge computing in Fig. 1. EdgeTB locates between the physical testbed and the emulated testbed. Since EdgeTB does not mandate which devices should be used as emulators or physical nodes, users can decide to strike a better balance between test fidelity, network flexibility, and cost. When all computing devices are used as physical nodes, EdgeTB will converge to a physical testbed. When all computing devices are used as emulators, EdgeTB will converge to an emulated testbed.

To support the rapid development of DMLs, we propose Role-oriented development, where the Role is a programming abstraction that extracts the general function modules such as model training and model aggregation from various

DMLs. EdgeTB encapsulates reusable libraries to support Role-oriented development. Users can use these libraries to develop DMLs quickly and replace the default function modules related to their research issues with custom ones.

We summarize our contributions as follows:

- We design and implement EdgeTB, a hybrid testbed that provides numerous emulated nodes to generate large-scale and network-flexible test environments while incorporating physical nodes to guarantee fidelity. It is the first hybrid edge computing testbed built across emulators and physical edge computing devices.
- We propose Role-oriented development and encapsulate reusable libraries to support the rapid development of DMLs. This is the first general library for developing arbitrary DMLs like Federated Learning, Gossip Learning, and E-Tree Learning.
- We conduct extensive case studies and experiments to demonstrate that researchers can use EdgeTB to develop DMLs and test them with high fidelity.

The remainder of this paper is organized as follows. Section 2 describes the architecture. Section 3 presents the user interface. Section 4 describes the implementation. Section 5 presents the case studies. Section 6 presents the experiment we conducted to evaluate EdgeTB. Section 7 presents the related work. Section 8 presents several discussion points and open issues. Section 9 concludes this paper.

2 EdgeTB Objectives and Architecture

2.1 Design Objectives

As shown in Fig. 1 above, edge computing testbeds have three main characteristics: *Test Fidelity*, *Network Flexibility*, and *Cost*. Considering the needs of studying DMLs in edge computing scenarios, we subdivide the *Test Fidelity* and *Cost* into more specific ones. We divide the *Test Fidelity* into *Computing Fidelity*, *Network Fidelity*, and *Execution Fidelity*. We divide the *Cost* into the cost of achieving *Resource Heterogeneity* in test environments, the cost of scaling the testbed (*Scalability*), and the cost of reproducing test environments (*Reproducibility*).

Computing Fidelity. In the real world, computing devices have proprietary hardware resources that are not shared with other devices. The testbed should isolate the resources of different virtual nodes to achieve high computing fidelity.

Network Fidelity. In the real world, computing devices communicate with each other through networks. Network messages are encapsulated in various network protocols (e.g., TCP and IP) for transmission. To achieve high network fidelity, all nodes in the testbed should communicate through networks by sending messages.

Execution Fidelity. Machine learning is an application that needs to be executed to obtain results, such as model accuracy and convergence time. Therefore, the testbed for machine learning should ensure that all nodes support commonly used machine learning libraries, such as *TensorFlow*.

Network Flexibility. In edge computing scenarios, the network topology between computing devices is unstable, and each network link has its bandwidth, delay, or other properties. The testbed should be able to generate various network topologies with different link properties between nodes. It is also essential to support dynamic modification of network topology and link properties to generate non-fixed edge networks when running tests.

Resource Heterogeneity. There are many heterogeneous computing devices in edge computing scenarios. Different computing and storage resources will affect the efficiency of performing machine learning. The testbed should be able to provide nodes with heterogeneous computing and storage resources at a low cost.

Scalability. It should support the use of multiple computing devices to generate large-scale test environments. It should be easy to add or remove computing devices in the testbed.

Reproducibility. It should be easy to reproduce the test environment, including node resources, network topology, and link properties, on other computing devices with sufficient resources. This allows researchers to reproduce and verify experiments conducted by other researchers quickly.

Existing testbeds can provide some characteristics, but at the expense of others. For example, the physical testbed can provide good test fidelity but lacks network flexibility, while the simulated testbed is just the opposite. We aim to provide a testbed that goes beyond the current state-of-the-art edge computing testbeds, enabling users to strike a better balance between test fidelity, network flexibility, and cost. Moreover, EdgeTB specifically encapsulates reusable libraries to support the development of DMLs, allowing users to develop and test DMLs quickly.

2.2 Architecture Overview

Fig. 2 depicts a high-level overview of the architecture of EdgeTB. The architecture comprises three layers: *DML Layer*, *Control Layer*, and *Execution Layer*.

The *DML Layer* locates at the top layer, where we provide *DML Lib* to support Role-oriented development. Users prepare *Role*, *Dataset*, and *Configuration File* for each node in this layer. The *Configuration File* is used to define the information of each node (e.g., the ID of the node and the dataset it owns) and the structure of DML (e.g., which nodes each node should communicate with and the frequency of communication). In addition, for some complex DML with multiple roles, users can implement a *DML Configuration* module to generate *Configuration File* for nodes.

Although we abstractly modeled DMLs, it does not mean that users can only develop DMLs based on our model, like using a simulator. On the contrary, users can develop DMLs according to their own needs in EdgeTB. For example, users can use *PyTorch* instead of *TensorFlow* as machine learning libraries. This means that users are not developing DMLs specifically for EdgeTB. Instead, they can deploy the DMLs on any computing device that provides a suitable environment, such as edge computing devices with *PyTorch*.

Below the DML Layer is the *Control Layer* and the *Execution Layer*. EdgeTB adopts the Controller-Worker architecture. One computing device runs as the *Controller* at the Control Layer, while the rest run as *Workers* at the Execution Layer. The *Controller* acts as a coordinator between users and Workers. The *Node Configuration* and *Network Configuration* of the Controller are responsible for receiving the input and parsing it into specific deployment commands. Then, the Controller

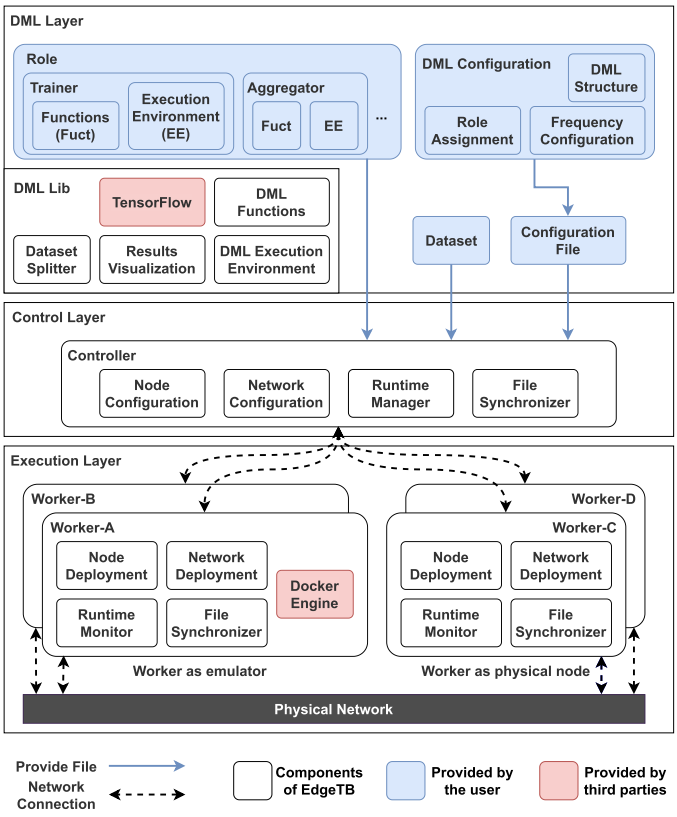


Fig. 2. The architecture of EdgeTB.

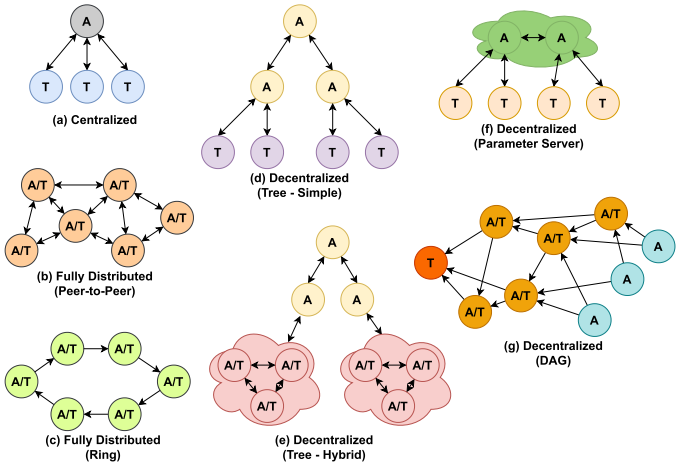


Fig. 3. Various DML architectures.

sends these commands to the corresponding Workers through networks. The *Worker* has *Node Deployment* and *Network Deployment* to execute these commands to generate the test environment. The hybrid design allows Workers to act as physical nodes or emulators. For Workers as emulators, they use the containerization technology supported by *Docker Engine* to provide Docker containers as emulated nodes.

Users may repeatedly modify their source codes and dataset distribution. The source codes and dataset are stored on the Controller. Both Controller and Worker have the *File Synchronizer* to transmit source codes and dataset between each other. This allows users to develop on the Controller, deploy the applications to each Worker for testing, and finally collect the test data to the Controller. Moreover, the *Runtime Monitor* in Worker and the *Runtime Manager* in Controller are used to detect and manage nodes' running status collaboratively. Using these modules, users can operate on the Controller to complete development and testing tasks. This speeds up the development and testing process.

2.3 Architecture Details

2.3.1 Role-Oriented Development

Although there are many nodes in various DMLs [29], [30], we can classify them according to the functions they perform. We propose Role-oriented development, where "Role" is a programming abstraction that extracts the general functions from various DMLs. We provide two basic roles, namely *Aggregator* and *Trainer*. As shown in Fig. 3, there are mainly three types of DML architectures, namely centralized, fully distributed, and decentralized. The centralized DMLs (Fig. 3a) are represented by Federated Learning, which has an *Aggregator* and multiple *Trainers*. The nodes in the fully distributed DMLs (Fig. 3b and 3c) can independently complete the model training and model aggregation functions. Therefore, these nodes can be regarded as an integration of *Aggregator* and *Trainer*. The decentralized DMLs have a variety of structures, including tree structure [11], [31] (Figs. 3d and 3e), parameter server structure (Fig. 3f), DAG structure [32] (Fig. 3g), etc. Different architectures may have different aggregation strategies or gradient upload strategies. However, all nodes in these decentralized DMLs can still be abstracted as *Trainer*, *Aggregator*, or their integration. Role-oriented development and the basic roles we provided can accelerate development.

Each role has function modules and execution environments. We provide *DML Lib* in the *DML Layer*, which contains *DML Functions* and *DML Runtime Environment* to support the Role-oriented development. *DML Functions* contains commonly used function modules such as model training. *DML Runtime Environment* includes the basic package dependencies for the physical node and the *Docker Image* for the emulated node. We also provide *Dataset Splitter* and *Results Visualization* to help users conduct tests.

There are many research problems in DMLs, such as selecting a suitable neural network model, optimizing the synchronization of node collaboration, and selecting the appropriate trainers. Many function modules in the DMLs for solving these research issues are reusable. In EdgeTB, users can use the function modules in *DML Lib* to quickly develop DMLs and replace function modules related to their research issues. For example, Fig. 4 shows how to use Role-oriented development to study the trainer selection issue in Federated Learning. Users can use our function modules and environments to implement the *Trainer*. For the *Aggregator*, users should replace the default trainer selection module with a customized one. If necessary, users should also prepare a customized execution environment. Users can instantiate these roles as physical or emulated nodes after implementing the function modules and execution environments of roles in future tests.

2.3.2 Hybrid Test Environment

Unlike EmuEdge and other testbeds that can only use one computing device, EdgeTB supports using multiple

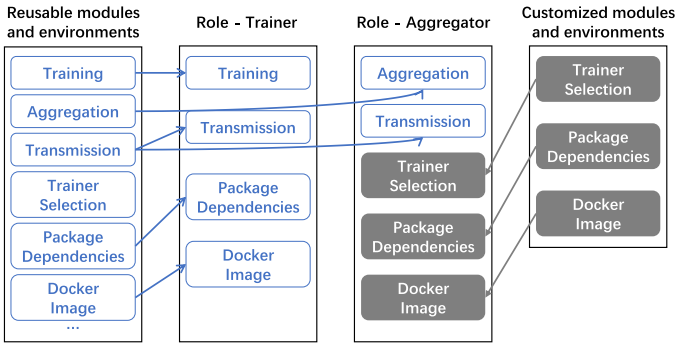


Fig. 4. An example of Role-oriented development.

computing devices to provide a large-scale hybrid test environment. To facilitate the deployment of testing and management of computing devices, EdgeTB uses Controller-Worker architecture. In EdgeTB, a dedicated computing device acts as the Controller, and the remaining computing devices act as Workers. The Controller is responsible for interacting with the user and controlling all the Workers, and the Workers are responsible for generating the test environment.

As shown in Fig. 5, users need to specify the details of the test environment by the Controller, including Worker information, node configuration, and network configuration. Although EdgeTB supports the hybrid test environment, one Worker can only act as a physical node or an emulator in a test, which users must explicitly specify. Then, users need to set the configuration of nodes. For a physical node, users need to identify which Worker it occupies. For an emulated node, users need to identify which Worker it is deployed and how much resources it occupies. Next, each node needs to be assigned a role, and the Controller will set up the execution environment of the role on the corresponding Worker. Users also need to configure the network topology between all nodes. EdgeTB currently supports end-to-end link resource (e.g., bandwidth) configuration between all nodes. In addition, EdgeTB also supports dynamic network adjustments when running tests. Network adjustments run silently without suspending or restarting nodes, which is similar to the network fluctuations of edge computing devices.

The Controller parses these configurations into deployment commands and sends them to the corresponding Workers through networks. The Workers deploy these commands to generate the target test environment. Since EdgeTB uses Docker containers as the emulated node, the Worker as the emulator should support *Docker Engine*. Even if a Worker supports *Docker Engine*, users can still use it as a physical node according to their requirements. This allows

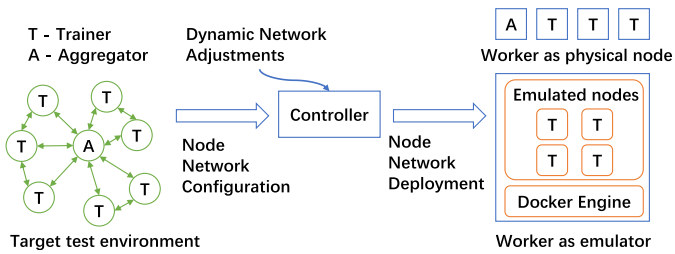


Fig. 5. An example of hybrid test environment.

```
# Trainer.py:
  def listen_at_train ():
    new_model = request.files.get ("model")  # obtain the model
    dml_utils.replace_model (nn.model, new_model)
    dml_utils.train (nn.model, dataset)
    dml_utils.send_model (nn.model, "/aggregate", aggregator)

# Aggregator.py:
  def start_next_round ():
    list = Trainer_Selection (trainers, number)  # select some Trainers
    dml_utils.send_model (nn.model, "/train", list)

  def listen_at_aggregate ():
    new_model = request.files.get ("model")
    dml_utils.store_model (received_models, new_model)
    if len (received_models) == number:
      dml_utils.aggregate (nn.model, received_models)
      acc = dml_utils.test (nn.model, dataset)
      start_next_round ()
```

Fig. 6. An example of implementing role's functions.

users to generate more diverse test environments. For example, users use two high-performance computing devices (A, B) and one low-performance computing device (C) to study Federated Learning. They can use (A) as a physical node to act as the *Aggregator*, (C) as a physical node to act as the *Trainer*, and (B) as an emulator to provide numerous emulated nodes to act as the *Trainer*. On the one hand, each role is played by physical nodes to improve test fidelity. On the other hand, there are numerous emulated nodes to increase the scale of the test environment.

Since EdgeTB is for experienced researchers, we allow them to make custom extensions to the Controller. The Controller includes a *Runtime Manager* for communicating with Workers, so users can register *Listeners* in the *Runtime Manager* to detect and manage the running status of their DMLs. The *Listener* can receive messages sent by nodes and reply to them through networks. A best practice is to register a listener named *Printer* to display the received messages on the screen. Nodes can send the messages generated during the test to the *Printer* in real-time, which helps users monitor the running status of each node.

3 User Interface

3.1 Role-Oriented Development

Fig. 6 shows how *DML developers* implement the main function modules of the two roles in Fig. 4 with Python. Developers can use the encapsulated function modules such as model training and model transmission to implement the *Trainer*. For *Aggregator*, developers can implement a custom *Trainer_Selection* function module to replace the default one if they want to study the trainer selection issue. For other function modules of *Aggregator*, such as model aggregation and model testing, developers can also use the encapsulated one.

If developers need to use additional Python packages, they need to prepare a new execution environment. EdgeTB uses the *Python requirements.txt* to specify the package dependencies for the physical node and uses the *Dockerfile* to specify the *Docker Image* for the emulated node. Thus, developers only need to update these two files, and the Controller will set the execution environment on the corresponding Worker. Fig. 7 shows an example of the *requirements.txt* and the *Dockerfile*. In the *requirements.txt*, we specify that 4 Python package dependencies are required, and the version of *TensorFlow*

```
# requirements.txt:
  flask
  numpy
  requests
  tensorflow>=2

# Dockerfile:
  FROM tensorflow/tensorflow:2-py3
  COPY requirements.txt /home
  WORKDIR /home
  RUN pip3 install -r requirements.txt
  CMD ["/bin/bash"]
```

Fig. 7. An example of preparing role's execution environment.

```
# Run.py:
  ctl = EdgeTB (ip_ctl)
  d1_emu = ctl.add_emu ("d1", ip_d1)
  n1 = d1_emu.add_node ("n1", "python3 Aggregator.py",
     image = "Aggregator", cpu = 4, memory = "5G")
  n2 = d1_emu.add_node ("n2", "python3 Trainer.py",
     image = "Trainer", cpu = 2, memory = "1G")
  d2_phy = ctl.add_phy ("d2", ip_d2)
  d2_phy.set_role ("Trainer", "python3 Trainer.py")
```

Fig. 8. An example of defining nodes.

```
# Run.py:
  ctl.symmetrical_link (n1, d2_phy, bw = "5mbps")
  # ctl.asymmetrical_link (n1, d2_phy, bw = "5mbps")
  # ctl.asymmetrical_link (d2_phy, n1, bw = "5mbps")
  ctl.asymmetrical_link (n1, n2, bw = "3mbps")
  ctl.asymmetrical_link (n2, n1, bw = "1mbps")
```

Fig. 9. An example of defining network using Python API.

should be at least 2. In the *Dockerfile*, we create a new image based on *tensorflow/tensorflow:2-py3* and install the packages declared by the *requirements.txt* in the new image.

3.2 Hybrid Test Environment

DML testers should configure nodes and networks to set up a test environment. EdgeTB provides Python API to configure the resources and roles of nodes. Fig. 8 declares a test environment and sets the computing device with the IP address "ip_ctl" as the Controller. First, we set the computing device "d1" with the IP address "ip_d1" as an emulator and deploy two emulated nodes in it, namely "n1" and "n2". Then, we allocate 4 CPU threads and 5 GB memory to "n1" and make it act as the *Aggregator*, and allocate 2 CPU threads and 1 GB memory to "n2" and make it act as the *Trainer*. Finally, we set the computing device "d2" with the IP address "ip_d2" as a physical node to act as the *Trainer*.

EdgeTB provides Python API and JSON API to define the network topology and link properties between nodes. We use the nodes of the above example to introduce. In Fig. 9, we use Python API to define the network topology and link properties. First, we add a symmetrical link between "n1" and "d2_phy" by calling the *symmetrical_link* API. We set the bandwidth from "n1" to "d2_phy" and from "d2_phy" to "n1" to "5mbps". However, asymmetrical links with downstream bandwidth higher than upstream bandwidth are more common in the real world. We make "n1" act as the *Aggregator*, which is usually performed by the edge infrastructure in the real world, so from "n1" to "n2" is the downstream link and from "n2" to "n1" is the upstream link. By calling the *asymmetrical_link* API twice, we can establish an asymmetrical link between two nodes. We set the bandwidth from "n1" to "n2" to "3mbps" and the bandwidth from "n2" to "n1" to "1mbps".

```
# Run.py:
  ctl.load_links ("links.json")

# links.json:
  {"link_list": [{"src": "n1", "dest": "d2_phy", "bw": "5mbps"},
       {"src": "d2_phy", "dest": "n1", "bw": "5mbps"},
       {"src": "n1", "dest": "n2", "bw" : "3mbps"},
       {"src": "n2", "dest": "n1", "bw" : "1mbps"}]}

# Update_network.py:
  url = "http://" + ip_ctl + "/update/network"
  requests.post (url, data = {"file": "new_links.json"})
```

Fig. 10. An example of defining network using JSON API.

```
# Run.py:
  @app.route ("/print", methods = ["POST"])
  def route_print ():
     print (request.form.get ("msg"))
     return ""

# Trainer.py:
  def send_print (msg):
     url = "http://" + ip_ctl + "/print"
     requests.post (url, data = {"msg": msg})
```

Fig. 11. An example of implementing a listener.

Besides calling the *symmetrical_link* and *asymmetrical_link* API, another way to define the network topology and link properties is to write a JSON file and load it by calling the *load_links* API. Fig. 10 shows how to define the same network as Fig. 9 through a JSON file. The JSON file contains an array of "links", and each "link" has three attributes: source (src), destination (dest), and bandwidth (bw). JSON files are also used to modify the network topology when running tests. The *Update_network.py* in Fig. 10 enables users to send a JSON file to the Controller through networks. If the Controller receives such a request, it will modify the network topology based on the specified JSON file.

Testers can register *listeners* in the *Runtime Manager*. Fig. 11 shows how to implement a *Printer* and how nodes interact with the *Printer*. We let the *Runtime Manager* listen for the *POST* requests sent to the */print* path. After receiving a request, it will take out the *msg* and display it on the screen. For any roles, they need to wrap the *msg* in a *POST* request and send it to the */print* path of the Controller.

4 Implementation

Computing Fidelity. EdgeTB contains physical nodes and emulated nodes, which significantly contribute to computing fidelity. On the one hand, users can directly set a Worker as a physical node. For example, if we set a Worker as a physical node to act as a *Trainer*, the Worker will install the Python packages declared by the *requirements.txt* in the user-space on the host OS. Then, the Worker will create a new process to run the node's execution program (i.e., the *Trainer.py*). This is the same as deploying DMLs on edge computing devices, so it has high computing fidelity.

On the other hand, users can deploy multiple emulated nodes (Docker containers) on one Worker. The *Docker Engine* provides sound isolation for emulated nodes. It offers a

```
# d1_emu.yml
  services:
    n1:
      cpuset: 0-3
      mem_limit: 5G
      ...
    n2:
      cpuset: 4-5
      mem_limit: 1G
      ...
```

Fig. 12. An example of emulated node deployment script.

dedicated namespace in the user space on the host OS for each emulated node so that each emulated node has its information, such as the hostname and process ID. Thus, an emulated node can be viewed as an independent host sharing the same kernel with the Worker. More importantly, EdgeTB uses the Linux Control Groups (*cgroups*) to restrict the processing capabilities of each emulated node. The *cgroups* is a Linux kernel mechanism used to manage the CPU and memory resource allocation of a group of processes and schedule these processes as a single entity.

In Fig. 8, we deploy two emulated nodes, "n1" and "n2". The Controller will generate an emulated node deployment script similar to Fig. 12. In this example, the processes of "n1" are scheduled to CPUs 0-3, while the processes of "n2" are scheduled to CPUs 4-5. EdgeTB ensures that each emulated node occupies different CPUs. Otherwise, it will throw an error of insufficient CPU resources to users. EdgeTB can also ensure that each emulated node only uses memory resources of a limited size. Otherwise, it will kill the process occupying memory in the emulated node, which is the same as how physical nodes handle errors. Thus, with the help of the physical nodes, *Docker Engine*, and *cgroups*, EdgeTB achieves good computing fidelity.

Network Fidelity. As shown in the left part of Fig. 13, the *Docker Engine* equips each emulated node with a virtual network interface, usually named "eth0". Therefore, each emulated node has its IP address. The *Docker Engine* creates a virtual bridge named "docker0" on the Worker. By default, all emulated nodes in this Worker are connected to the "docker0" bridge, so the emulated nodes can communicate with each other through IP addresses. The *Docker Engine* also implements Network Address Translation (NAT) for emulated nodes. Nodes on other Workers can access a specific emulated node by accessing the IP:port pair of the Worker which it is deployed. Therefore, all nodes can communicate through networks by sending messages, thereby ensuring network fidelity.

Execution Fidelity. For physical nodes, we can use the *Python requirements.txt* to install machine learning libraries for them. For emulated nodes, commonly used machine learning libraries such as *TensorFlow* and *PyTorch* have official *Docker Image*. For other needs, users can also use the *Dockerfile* to prepare a new *Docker Image*. In summary, EdgeTB can guarantee the execution fidelity of both physical nodes and emulated nodes.

Network Flexibility. Since all nodes are connected, EdgeTB can use the Linux Traffic Control (*tc*) to shape the link between any two nodes to form various network topologies. As shown in the right part of Fig. 13, the *tc* (colored box) creates Classful Queuing Disciplines (*qdisc*) to filter and redirect

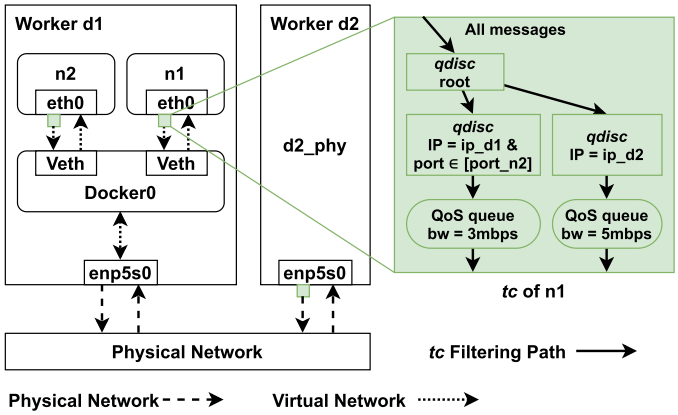


Fig. 13. EdgeTB's network model.

network messages to a particular Quality of Service (QoS) queue before sending them out. Each QoS queue has its own limit on bandwidth so that EdgeTB can limit the sending rate of network messages.

In this example, EdgeTB creates two *qdisc* and two QoS queues for "n1", which are used for "n2" and "d2_phy", respectively. For messages sent to "d2_phy", EdgeTB creates a *qdisc* to redirect messages with the destination IP address "ip_d2" to a QoS queue with "5mbps" bandwidth. We assume that "n2" listens on some ports mapped to "[port_n2]" (an array) of "d1" through NAT, respectively. Since multiple emulated nodes may be deployed in a Worker simultaneously, we cannot distinguish different emulated nodes in the same Worker only by destination IP address. Therefore, for messages sent to "n2", the *qdisc* matches both IP address and port. Moreover, all messages sent to "n2" share the same bandwidth limit, which is the same as in the real world. Thus, using *tc* to shape the link between every two nodes, EdgeTB achieves good network flexibility.

Resource Heterogeneity. As mentioned above, EdgeTB can use *cgroups* to easily create emulated nodes with different CPU resources and memory resources. Using high-performance computing devices as emulators, users can flexibly generate test environments with many heterogeneous emulated nodes to achieve resource heterogeneity.

Scalability. EdgeTB adopts the Controller-Worker architecture, which is easy to scale horizontally. If users want to add a new Worker, they need to connect it to the Controller and other Workers through networks. Then run the EdgeTB program on this new Worker. After that, users can use this new Worker in the Controller.

Reproducibility. The role definitions, node definitions, and network definitions of EdgeTB are all portable files. To reproduce a test environment on other computing devices with sufficient resources, users only need to slightly modify some files, such as update the IP address of Workers.

Although the emulated nodes' absolute performance may change in the new test environment, EdgeTB can still guarantee that the relative performance between emulated nodes on the same Worker remains unchanged. Thus, compared with purchasing the same hardware to reproduce a test environment, users can use their hardware to reproduce the test environment as similar as possible to reduce costs in EdgeTB.

TABLE 1
Details of Computing Devices

Name	CPU	Memory	Performance
Controller	2 threads	4 GB	/
Worker 1	4 threads	1 GB	Low
Worker 2	4 threads	1 GB	Low
Worker 3	4 threads	1 GB	Low
Worker 4	16 threads	64 GB	High
Worker 5	128 threads	256 GB	High

5 Case Study

In this section, we show how to use EdgeTB to develop and test DMLs. As shown in Table 1, we have 6 heterogeneous computing devices, one of which is used as the Controller, and the rest are used as Workers. Workers 1-3 are Raspberry Pies with a low-performance ARM processor, while Workers 4-5 are high-performance desktop computers. Workers 1-3 are only used as physical nodes, while Worker 4-5 can be used as physical nodes or emulators.

First, we take Federated Learning as an example to show how to use Role-oriented development to support the research of the trainer selection issue. Then, we take Gossip Learning as an example to use our Workers to generate a test environment as large as possible. Finally, we take E-Tree Learning as an example to show how to develop and test more complex DML on EdgeTB.

5.1 Federated Learning

This case study builds two roles (*Aggregator* and *Trainer*) of Federated Learning similar to Fig. 6. First, we build a baseline by using the default *dml_utils.random_selection* function module. Then, we add function modules for both *Aggregator* and *Trainer* so that the *Aggregator* can obtain *Trainer*'s model training time and model transmission time to adjust the candidate probability of each *Trainer*. The longer the total time consumed (training time plus transmission time), the lower the candidate probability is.

As shown in Fig. 14, the target test environment contains 16 nodes, of which 1 node acts as the *Aggregator*, and the remaining nodes act as *Trainers*. Since the *Aggregator* is usually performed by the robust edge infrastructure in the real world, we set Worker 4 as a physical node to act as the *Aggregator*, denoted as p4 (physical node 4). We set Workers 1-3 as physical nodes to act as *Trainers*, denoted as p1 to p3. We set Worker 5 as an emulator and deploy 12 emulated nodes with heterogeneous CPU resources in it, denoted as e1 to e12 (emulated node 1-12). These emulated nodes occupy 1, 2, 4, 6 CPU threads and 5 GB memory, respectively. It is worth noting that the performance of the emulated nodes is much higher than that of the physical nodes, even if the emulated

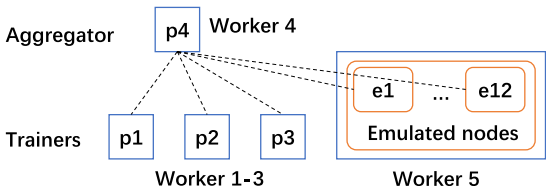


Fig. 14. Target test environment of federated learning.

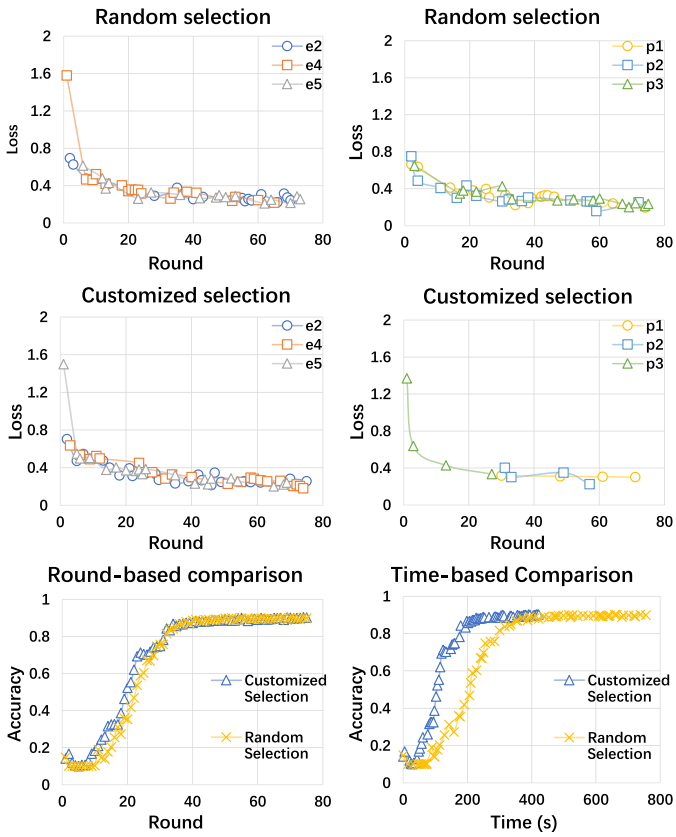


Fig. 15. Test results of federated learning.

nodes only occupy 1 CPU thread. We set the bandwidth from the *Aggregator* to each *Trainer* to "10mbps", and the bandwidth from each *Trainer* to the *Aggregator* to a random value between "200kbps" to "500kbps".

We use the Fashion-MNIST dataset, and each *Trainer* has the same amount of i.i.d. data. The *Aggregator* selects 3 *Trainers* each round and performs 75 rounds of aggregation, which is enough for the model to converge. The total time consumed in a round of emulated nodes is 3s to 6s, while physical nodes are 10s to 15s. One goal of customizing the trainer selection function module is to reduce the candidate probability of slow *Trainers*, thereby accelerating model convergence.

We record the test results of all nodes, and Fig. 15 shows some of them, including 3 emulated *Trainers* (e2, e4, and e5 as representatives), 3 physical *Trainers* (p1, p2, and p3), and the *Aggregator*. In Random Selection, the emulated *Trainers* and the physical *Trainers* were selected almost the same number of times. It is worth mentioning that since not all *Trainers* are selected initially, the loss of the selected *Trainers* in the early period will be higher, while the loss of the selected *Trainers* in the later period will be lower. In Customized Selection, the number of times that the physical *Trainers* are selected has been dramatically reduced. This is because the candidate probability of these slow *Trainers* is reduced in Customized Selection. When considering rounds, there is almost no difference in the convergence speed of the two trainer selection strategies, but when considering time, the time of Customized Selection is significantly shortened. This case study shows that users can quickly build well-known DMLs to support their research in EdgeTB.

```
# G-peer.py:
  def listen_at_gossip ():
    new_model = request.files.get ("model")  # obtain the model
    # just aggregate this two models
    dml_utils.aggregate (nn.model, [nn.model, new_model])
    dml_utils.train (nn.model, dataset)
    gossip ()

  def gossip ():
    peer = dml_utils.random_selection (peers, 1)  # select 1 peer
    dml_utils.send_model (nn.model, "/gossip", peer)
```

Fig. 16. G-peer role of gossip learning.

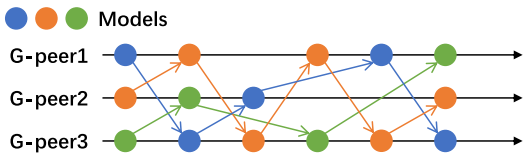


Fig. 17. Models walk between G-peers.

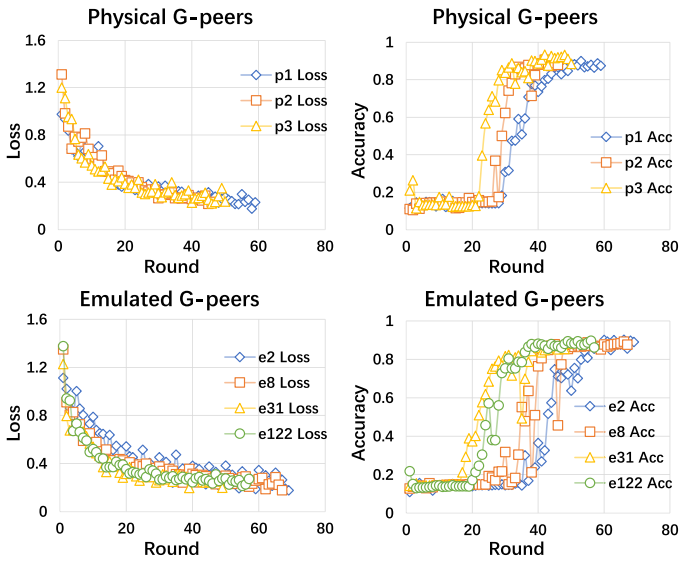


Fig. 18. Test results of gossip learning.

5.2 Gossip Learning

Gossip Learning [10] is a method for learning models from fully distributed data without central control. There is only one type of node in Gossip Learning, namely *G-peer*. The primary function modules of *G-peer* are shown in Fig. 16. *G-peers* contain both model training and model aggregation function modules. Each *G-peer* initializes a local model and sends it to another random *G-peer* at the beginning. When a *G-peer* receives a model, it immediately aggregates it with its local model, conducts a round of training based on this new model, and finally sends it out to another random *G-peer*. As shown in Fig. 17, this mechanism results in the model taking random walks between *G-peers* and being updated when visiting a *G-peer*.

We set Workers 1-3 as physical nodes, denoted as p1 to p3, and set Workers 4-5 as emulators. We deploy 16 emulated nodes in Worker 4, denoted as e1 to e16, and deploy 128 emulated nodes in Worker 5, denoted as e17 to e144. Each emulated node occupies 1 CPU thread and 1 GB of memory. Each *G-peer* randomly connects to about 20 *G-peers*, and the bandwidth is between "200kbps" to "500kbps".

We use the same dataset as in the case study of Federated Learning. We set that each *G-peer* will no longer *gossip* after 75 rounds of training, which is enough for the model to converge, but it can still receive models from others for training. Some *G-peers* may perform more than 75 rounds of training, while others may not.

We record the test results of all nodes, and Fig. 18 shows some of them, including 3 physical *G-peers* (p1, p2, and p3), 2 emulated *G-peers* in Worker 4 (e2 and e8 as representatives), and 2 emulated *G-peers* in Worker 5 (e31 and e122 as representatives). The model accuracy fluctuates more obviously compared with Federated Learning. Since *G-peers* constantly train those randomly arriving models, their records are not the accuracy of a particular model after continuous training. However, we can still observe the overall accuracy improvement of all models in the network through the records of *G-peers*. In this case study, we show how to use EdgeTB to generate hybrid and large-scale test environments.

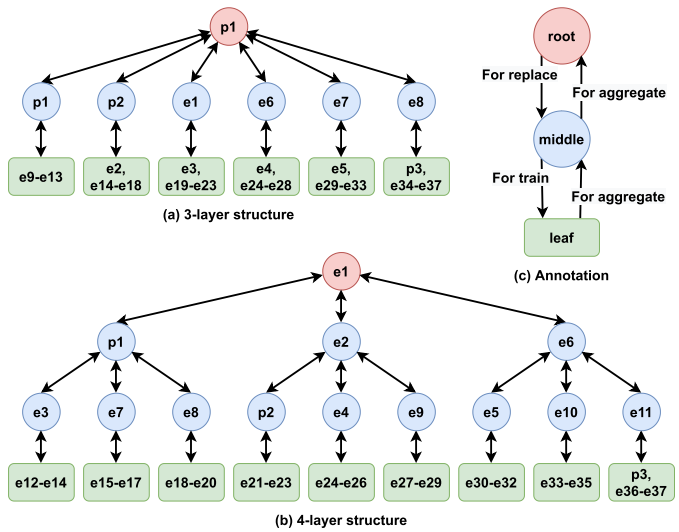


Fig. 19. Target test environments of E-tree learning.

5.3 E-Tree Learning

E-Tree Learning [11] is a decentralized model learning framework dedicated to DML at the edge. It organizes nodes into a tree structure, where the leaf nodes represent the trainer and the non-leaf nodes represent aggregators. As shown in (c) of Fig. 19, leaf nodes send their local models to the middle nodes after training, and the middle nodes continuously aggregate models and send them to the upper nodes until the models are sent to the root node. Since E-Tree Learning allows a node to be in multiple layers of the tree simultaneously, it may not be easy to define roles based on the locations of different nodes. As shown in Fig. 20, we only implement one role, namely *E-peer*, which contains all the function modules used in different layers. We rely on the configuration files to determine each node, such as location, the upper node of each layer, and the lower nodes of each layer.

We set Workers 1-3 as physical nodes, denoted as p1 to p3, and set Workers 4-5 as emulators. We deploy 5 emulated nodes in Worker 4, denoted as e1 to e5, and deploy 32

```
# E-peer.py:
  def listen_at_train ():  # as leaf node
    # same as Trainer in Federated Learning

  def send_model_to_lower ():
    if self_layer == 2:
      dml_utils.send_model (nn.model, "/train", lower_nodes)
    elif self_layer > 2:
      dml_utils.send_model (nn.model, "/replace", lower_nodes)

  def listen_at_replace ():  # as middle node
    new_model = request.files.get ("model")
    dml_utils.replace_model (nn.model, new_model)
    send_model_to_lower ()

  def listen_at_aggregate ():  # as middle node or root node
    new_model = request.files.get ("model")
    dml_utils.store_model (received_models, new_model)
    if len (received_models) == len (lower_nodes):
      dml_utils.aggregate (nn.model, received_models)
      acc = dml_utils.test (nn.model, dataset)
      if upper_node != null:
        dml_utils.send_model (nn.model, "/aggregate", upper_node)
      else:
        send_model_to_lower ()
```

Fig. 20. E-peer role of E-tree learning.

emulated nodes in Worker 5, denoted as e16 to e37. These emulated nodes occupy 1, 2, 4, 6 CPU threads and 5 GB of memory, respectively. The bandwidth between connected nodes is between "200kbps" to "500kbps". Since E-Tree Learning supports building various tree structures, we designed a 3-layer structure and a 4-layer structure as shown in (a), (b) of Fig. 19.

We use the same dataset as in the case study of Federated Learning. In the 3-layer structure, we set the root aggregator to perform 25 rounds of aggregation and each middle aggregator to perform 100 rounds of aggregation. In the 4-layer structure, these numbers are 10, 50, and 100. Each leaf node will perform 100 rounds of training in these two structures, but the aggregation rounds of non-leaf nodes are different.

We record the test results of all nodes, and Fig. 21 shows some of them, including one node in each layer of both the 3-layer structure and the 4-layer structure. Since E-Tree Learning is similar to Federated Learning, which continuously

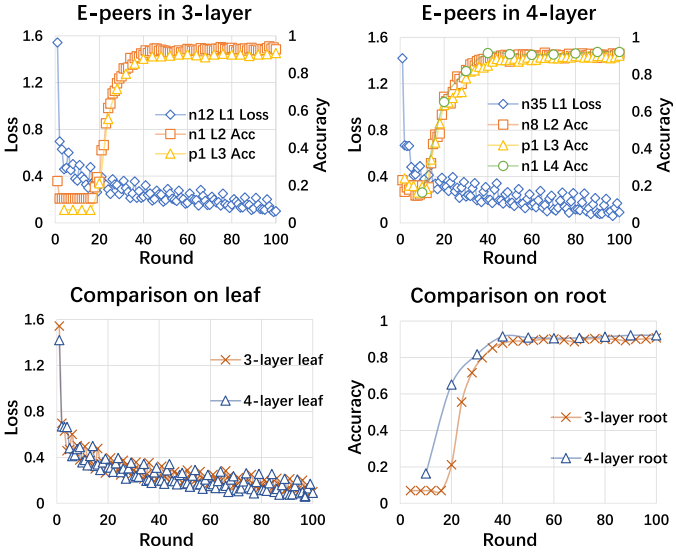


Fig. 21. Test results of E-tree learning.

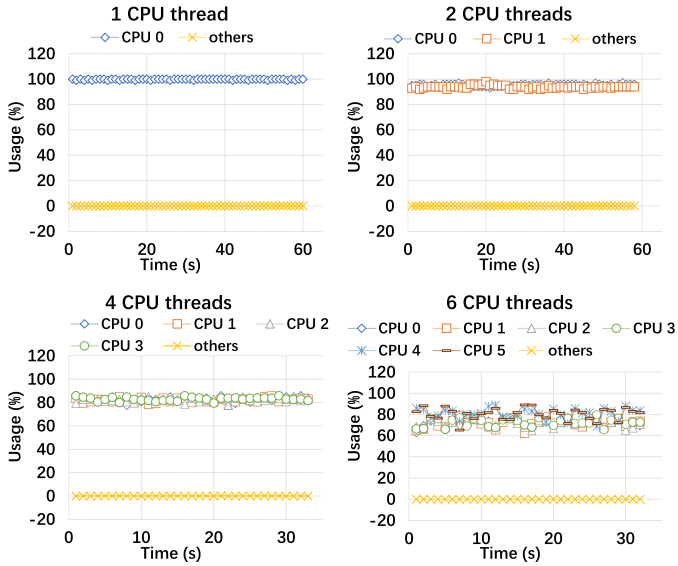


Fig. 22. Computing isolation between allocated CPU threads and others.

trains a model, the model accuracy fluctuates less. Different structures in E-Tree Learning may lead to different model accuracy and convergence time. We can quickly generate E-Tree Learning with different structures in a test environment with well-designed roles and configuration files. This case study shows that EdgeTB supports the development and testing of arbitrary and complex DMLs.

6 Experiment

6.1 Computing Isolation

In this section, we evaluated the computing fidelity of EdgeTB. We focus on the computing isolation of emulated nodes. As discussed before, testbeds with emulated nodes should isolate the computing resource of different emulated nodes. We deployed an emulated node occupying 1, 2, 4, 6 CPU threads on Worker 5 and performing the same model training tasks. Fig. 22 shows the CPU usage during the training. We can see that only the CPU threads assigned to the emulated node are in a high usage state, while other CPU threads are in an idle state. This reflects the excellent computing isolation of EdgeTB. It is worth mentioning that as CPU threads increase, the model training time is shortened, but the usage of each thread is declining. Model training is not a completely CPU-intensive task, and it also has I/O tasks for reading data. For more complex structures with bypass links and multiple branches fan-out such as ResNet and Inceptions, memory is also a bottleneck because those intermediate layers require storing previous results. Therefore, when CPU threads increase, I/O speed and memory may become a bottleneck that affects the usage of CPU threads.

We also tested the impact of Worker's CPU usage on emulated nodes' performance. We deployed 1, 8, 16, 32 emulated nodes, each occupying 4 CPU threads, so the CPU usage of Worker 5 is 3%, 25%, 50%, and 100%. As shown in the boxplot (a) of Fig. 23, the batch training time of nodes is similar, reflecting the similar performance of the nodes. When the Worker's CPU is under high usage, the performance of nodes drops obviously, but this is unavoidable.

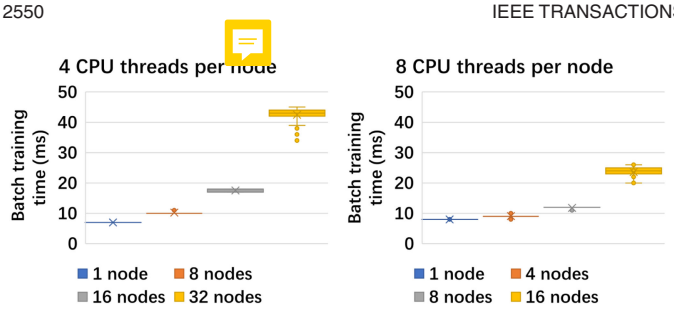


Fig. 23. Computing isolation between emulated nodes.

However, in other cases, we think that the magnitude of node performance degradation is acceptable. Moreover, the bottleneck of I/O speed also affects the performance of nodes. Therefore, we halve the number of nodes and make each node occupy 8 CPU threads so that the CPU usage of Worker 5 is almost unchanged while reducing the I/O tasks for reading data by half. As a result, the boxplot (b) of Fig. 23 shows less performance degradation for each node.

6.2 Network Link Shaping

In this section, we evaluated the network fidelity of EdgeTB. We focus on the network link shaping of both physical nodes and emulated nodes. We recorded the time consumed in model transmission by each node in the case study of Gossip learning. Fig. 24 shows some of them, including 2 emulated nodes in Worker 4 (e1 and e9 as representatives), 2 emulated nodes in Worker 5 (e37 and e48 as representatives), and 3 physical nodes (p1, p2, and p3).

We categorize the links into "e-e-s" (emulated nodes to emulated nodes on the same Worker), "e-e-d" (emulated nodes to emulated nodes on different Workers), "e-p" (emulated nodes to physical nodes), and "p-all" (physical nodes to all other nodes). All the actual transmission time is slightly larger than the theoretical time. This is because the network messages have passed through the physical network after network link shaping, so it takes a bit of time, which is acceptable.

6.3 Scalability

In this section, we evaluated the scalability of EdgeTB. We focus on the node deployment time and network deployment time. We test the time taken by EdgeTB from receiving user input to completing deployment under different numbers of nodes. We assume that the execution environments of roles are ready because the time to build execution environments is mainly spent in downloading Python packages and basic *Docker images*, which depends on the user's Internet speed.

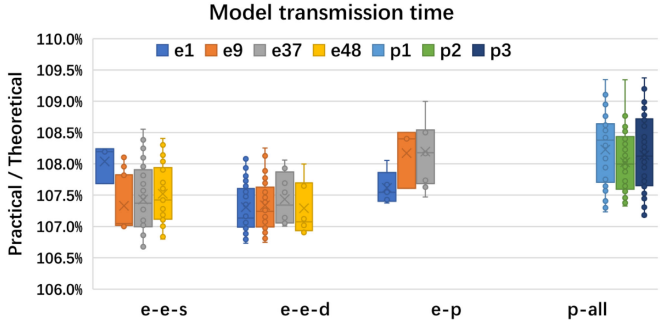


Fig. 24. Model transmission time in gossip learning case study.

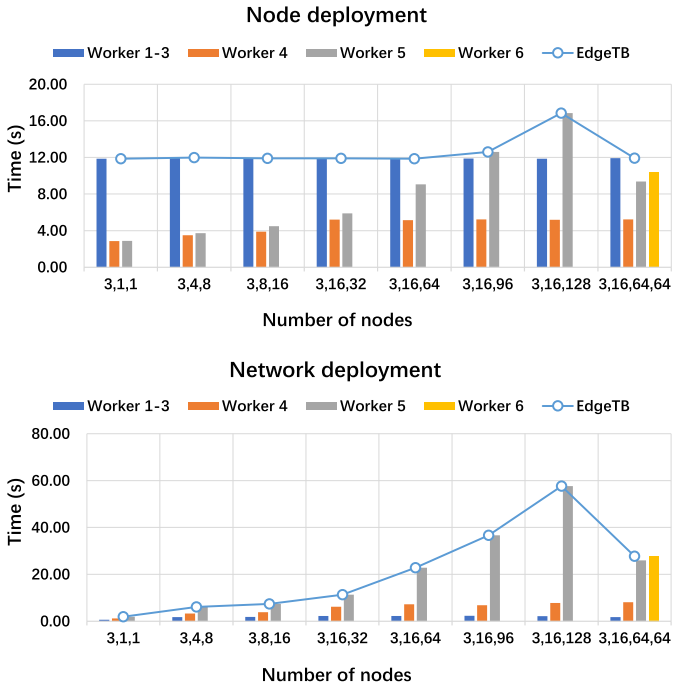


Fig. 25. Deployment time.

We consider the node deployment to be successful when all nodes can communicate with others through networks. For the network deployment time, we test the time consumed by dynamically adjusting the network while the test environment is running. In the network deployment, we set each node to connect with about 20 nodes. When there are less than 20 nodes, we set all nodes to connect with each other. We consider the network deployment to be successful when all nodes successfully execute the *tc* commands.

We set Workers 1-3 as physical nodes and set Workers 4-5 as emulators. Fig. 25 shows the deployment time, where the horizontal axis represents the number of nodes. For example, (3,4,8) means 3 physical nodes (Worker 1-3), 4 emulated nodes in Worker 4, and 8 emulated nodes in Worker 5. The Controller converts user input into each Worker's node deployment and network deployment commands and sends them to the corresponding Workers. Each Worker executes these deployment commands independently and in parallel. Thus, the overall deployment time of EdgeTB depends on the slowest Worker. In the node deployment, Worker 5 becomes the slowest one after the number of nodes in Worker 5 reaches 96. In the network deployment, Worker 5 is always the slowest one.

Since it is easy to add Workers in EdgeTB, we added Worker 6 with the same performance as Worker 5 to share the workload of Worker 5. As shown in (3,16,64,64) in Fig. 25, the node deployment and network deployment time are significantly reduced. With more hardware resources, EdgeTB can provide shorter deployment times.

EdgeTB requires each emulated node to occupy at least one physical CPU thread. Therefore, the number of nodes supported by EdgeTB is related to the hardware resources of workers. With more hardware resources, EdgeTB can provide larger network scales. This reflects the excellent scalability of EdgeTB.

7 Related Work

To the best of our knowledge, there is no testbed dedicated to studying DML at the edge. Therefore, we study various testbeds that can be used for edge computing, including physical testbed, simulated testbed, and emulated testbed.

Physical Testbed. [12] proposed a proof-of-concept prototype of Cumulus, which is a private physical testbed for edge cloud computational offloading. The authors implemented the prototype using 10 low-performance computing devices (e.g., smartphones and Raspberry Pies) and 3 PCs. [14] processed a 4-layer (IoT-device, edge, fog, and cloud) physical testbed architecture for streaming IoT-based applications. In this paper, the Edge layer comprises edge gateways, while the Fog layer comprises computing devices. Researchers also focus on how to build a shared physical testbed to reduce costs. [13] processed a generic shared physical testbed architecture for fog-based systems. Although the architectures of [13], [14] support numerous nodes, deploying such a physical testbed is so expensive that they only use several PCs and low-performance computing devices to verify their architectures. Some shared WSN physical testbeds may help verify lightweight edge computing applications. FIT IoT-LAB [15] is a shared testbed containing 2728 low-power wireless nodes and 117 mobile robots, of which 550 nodes support the Linux system. These Linux-empowered nodes have the potential to run various edge computing experiments. Indriya2 [16] and LinkLab [17] are shared testbeds containing hundreds of IoT devices, which support multiple IoT platforms, including Arduino, AliOS-Things, and Contiki. These testbeds may be helpful to test some IoT-based applications. However, further modifications are required when using these WSN testbeds. In addition, it is usually challenging to customize and reproduce errors when using shared testbeds.

Simulated Testbed. EdgeCloudSim [21], which is forking from CloudSim [18], mainly focuses on the performance evaluation of edge computing. It covers network and computational modeling and supports many edge computing features, such as node mobility and computing offload. iFogSim [19], also forking from CloudSim, provides a toolkit to model IoT applications in Cloud/fog environments. It can measure the impact of resource management techniques in latency, network congestion, energy consumption, and cost. iFogSim does not support node mobility, but MyiFogSim [20] noticed this problem and extended iFogSim to support node mobility. YAFS [22] is a fog computing simulator that helps researchers study issues like the placement of application modules, workload location, path routing and, scheduling of services. FogNetSim++ [23] is a fog computing simulator that enables researchers to incorporate customized mobility models and fog node scheduling algorithms and manage handover mechanisms. [33] compared several fog computing simulators by publicly available information and experience with their practical use, while [34] conducted a more detailed comparative analysis of several simulators. However, the inability to execute applications and transmit messages over networks makes the simulator lack fidelity.

Emulated Testbed. [26] proposed a shared emulated testbed that uses multiple Raspberry Pies as emulators. However, sharing the low-performance Raspberry Pies makes it challenging to carry out complex experiments. [27] uses heterogeneous devices as emulators, including Raspberry Pi, Nvidia Jetson, and Intel NUC. These devices use containerization technology to provide emulated nodes. However, these are low-performance computing devices that are not suitable for use as emulators. Moreover, these works have not considered the network topology between emulated nodes. EmuFog [24] is an extensible emulation framework built on top of MaxiNet [35] for fog computing scenarios. It uses containers as emulated nodes and allows users to define the network topology between containers. However, It only considers the latency of network links. EmuEdge [25] is an emulator for edge computing experiments. It uses both containers and VMs as emulated nodes and uses the Open vSwitch to configure the network topology between emulated nodes. EmuEdge uses *Linux netns* to provide containers, which cannot isolate computing resources but can only limit the CPU time of each container. Therefore, EmuEdge mainly focuses on VMs. However, EmuEdge only uses one computer as the emulator, making it challenging to deploy many VMs and limiting its scalability. Fogify [28] is an emulator easing the modeling, deployment, and large-scale experimentation of fog and edge testbeds. It supports using multiple computers as emulators to provide containers, but it does not isolate computing resources between containers. Moreover, only computers with sufficient computing and storage resources are suitable as emulators.

Unlike the above works, EdgeTB uses a hybrid design to fully utilize heterogeneous computing devices to provide high-fidelity, large-scale and network-flexible test environments. Moreover, we specially designed EdgeTB for studying DMLs and proposed Role-oriented development to support the rapid development of DMLs.

8 Discussions and Future Work

This section discusses some open issues and answers some questions that may appeal to readers.

Open Source. EdgeTB components are implemented in Python (v3.6) with about 3000 lines of code and are available at https://github.com/Lin-1997/Edge-TB.

CPU-Only Emulation. The emulated nodes in EdgeTB only use CPU resources because there is currently insufficient support for GPU resources isolation. With CPU resources isolation, EdgeTB can schedule the workload of different emulated nodes to different physical CPU threads so that the emulated nodes have exclusive CPU resources (relatively speaking). Suppose emulated nodes use GPU resources. In that case, they need to wait for the scheduling of the GPU driver to compete for the use of GPU resources. This is contrary to the fact that each device has its own hardware resources in the real world.

Data-Parallel and Model-Parallel Architecture. The architectures mentioned in Section 2.3.1 are Data-Parallel Architectures (DPAs). In DPAs, each node maintains a local copy of the model and trains on its data while periodically synchronizing the model with other nodes. The basic roles *Aggregator* and *Trainer* are suitable for implementing DPAs. However, another architecture called Model-Parallel Architecture (MPA) partitions the DNN and assigns subsets of layers to each node. The intermediate outputs (forward propagation) and corresponding gradients (backward propagation) of the

DNN are transmitted between nodes. Therefore, the *Aggregator* and *Trainer* are not suitable for implementing MPAs. In future work, we will explore suitable roles for implementing MPAs.

Emulated Node Deployment. Currently, users need to arrange the deployment of emulated nodes. However, emulated nodes on different physical devices may use bandwidth beyond the Network Interface Card (NIC) capabilities. In future work, we will develop algorithms to deploy emulated nodes with large bandwidth requirements on the same physical device. As a result, the network communication between them does not pass through the NIC.

Why Not Kubernetes. Kubernetes does not support using computing devices as physical nodes, while we need physical nodes to enhance test fidelity. Since Kubernetes is for production-grade container orchestration, its network model focuses on service discovery and load balance. However, this makes it challenging to customize the link properties between emulated nodes. In future work, we will integrate modern SDN software (such as Open vSwitch) to enrich EdgeTB's network model, which also conflicts with Kubernetes' network model.

9 Conclusion

Deploying DMLs at the edge has huge potential to make full use of the value of data. However, the existing platforms lack support in both the development and testing of DMLs. This paper presents EdgeTB, a hybrid testbed that fully utilizes physical nodes and emulated nodes to provide high-fidelity, large-scale, and network-flexible test environments. Moreover, we propose Role-oriented development and encapsulate reusable libraries to support the rapid development of DMLs. We conducted case studies and experiments to prove that EdgeTB can help develop and test DMLs.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st Ed. MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[2] P. Garcia Lopez *et al.*, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, p. 37–42, Sep. 2015.

[3] W. Yu *et al.*, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[4] N. K. Giang, R. Lea, M. Blackstock, and V. C. M. Leung, "Fog at the edge: Experiences building an edge computing platform," in *Proc. IEEE Int. Con. Edge Comput.*, 2018, pp. 9–16.

[5] Y. Li, P. A. Frangoudis, Y. Hadjadj-Aoul, and P. Bertin, "A mobile edge computing-assisted video delivery architecture for wireless heterogeneous networks," in *Proc. IEEE Symp. Comput. Commun.*, 2017, pp. 534–539.

[6] Z. Chang, X. Zhou, Z. Wang, H. Li *et al.*, "Edge-assisted adaptive video streaming with deep learning in mobile edge networks," in *Proc. IEEE Wireless Commun. Netw. Conf.*, 2019, pp. 1–6.

[7] F. Liang, W. Yu, X. Liu, D. Griffith, and N. Golmie, "Toward edge-based deep learning in industrial Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4329–4341, May 2020.

[8] J. Mills, J. Hu, and G. Min, "Communication-efficient federated learning for wireless edge intelligence in IoT," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 5986–5994, Jul. 2020.

[9] H. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. Agüeray Arcas, "Communication-efficient learning of deep networks from decentralized data," *CoRR*, vol. abs/1602.05629, 2016. [Online]. Available: http://arxiv.org/abs/1602.05629

[10] I. Hegedűs, G. Danner, and M. Jelasity, "Gossip learning as a decentralized alternative to federated learning," in *Proc. 19th IFIP Int. Conf. Distrib. Appl. Interoperable Syst.*, 2019, pp. 74–90.

[11] L. Yang, Y. Lu, J. Cao, J. Huang and M. Zhang, "E-tree learning: A novel decentralized model learning framework for edge ai," *IEEE Internet Things J.*, vol. 8, no. 14, pp. 11290–11304, Jul. 2021.

[12] H. Gedawy, S. Tariq, A. Mtibaa and K. Harras, "Cumulus: A distributed and flexible computing testbed for edge cloud computational offloading," in *Proc. Cloudification Internet Things*, 2016, pp. 1–6.

[13] Y. Karim and R. Hasan, "FogTestBed: A generic architecture for testbed for fog-based systems," in *Proc. SoutheastCon*, 2020, pp. 1–7.

[14] S. Nguyen, Z. Salcic, X. Zhang and A. Bisht, "A low-cost two-tier fog computing testbed for streaming IoT-based applications," *IEEE Internet Things J.*, vol. 8, no. 8, pp. 6928–6939, Apr. 2021.

[15] C. Adjih *et al.*, "FIT IoT-LAB: A large scale open experimental iot testbed," in *Proc. IEEE 2nd World Forum Internet Things*, 2015, pp. 459–464.

[16] P. Appavoo, E. K. William, M. C. Chan, and M. Mohammad, "Indriya2: A heterogeneous wireless sensor network (WSN) testbed," in *Testbeds and Research Infrastructures for the Development of Networks and Communities*, H. Gao, Y. Yin, X. Yang, and H. Miao, Eds. Cham, Switzerland: Springer, 2019, pp. 3–19.

[17] Y. Gao, J. Zhang, G. Guan and W. Dong, "LinkLab: A scalable and heterogeneous testbed for remotely developing and experimenting IoT applications," in *Proc. IEEE/ACM 5th Int. Conf. Internet-Things Des. Implementation*, 2020, pp. 176–188.

[18] R. N. Calheiros, R. Ranjan, A. Beloglazov, and C. A. F. De Rose, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, p. 23–50, Jan. 2011.

[19] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Softw.: Pract. Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[20] M. M. Lopes, W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "MyiFogSim: A simulator for virtual machine migration in fog computing," in *Proc. Companion Proc. 10th Int. Conf. Utility Cloud Comput.*, 2017, pp. 47–52.

[21] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," in *Proc. 2nd Int. Conf. Fog Mobile Edge Comput.*, 2017, pp. 39–44.

[22] I. Lera, C. Guerrero, and C. Juiz, "YAFS: A simulator for IoT scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019.

[23] T. Qayyum, A. W. Malik, M. A. Khan Khattak , O. Khalid and S. U. Khan, "FogNetSim++: A toolkit for modeling and simulation of distributed fog environment," *IEEE Access*, vol. 6, pp. 63 570–63 583, 2018.

[24] R. Mayer, L. Graser, H. Gupta, E. Saurez and U. Ramachandran, "EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures," in *Proc. IEEE Fog World Congr.*, 2017, pp. 1–6.

[25] Y. Zeng, M. Chao, and R. Stoleru, "EmuEdge: A hybrid emulator for reproducible and realistic edge computing experiments," in *Proc. IEEE Int. Conf. Fog Comput.*, 2019, pp. 153–164.

[26] Q. Xu and J. Zhang, "piFogBed: A fog computing testbed based on raspberry Pi," in *Proc. IEEE 38th Int. Perform. Comput. Commun. Conf.*, 2019, pp. 1–8.

[27] B. Diao *et al.*, "A scalable testbed for task offloading and deployment of heterogeneous edge computing," in *Proc. IEEE Int. Conf. Ubiquitous Comput. Commun. Data Sci. Comput. Intell. Smart Comput. Netw. Serv.*, 2019, pp. 586–591.

[28] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *Proc. 5th ACM/IEEE Symp. Edge Comput.*, 2020, pp. 42–54.

[29] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–33, Mar. 2020.

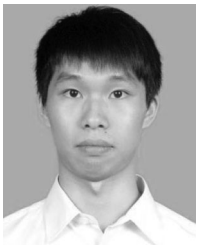
[30] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.

[31] Y. Li, J. Park *et al.*, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 175–188.

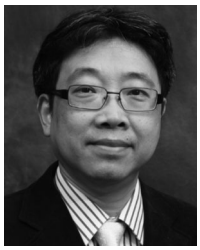
[32] B. Jossekin, P. Bjarne, S. Robert, G. Paul, A. Bert, and P. Andreas, "Implicit model specialization through dag-based decentralized federated learning," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 310–322.

[33] C. Kunde and Z. A. Mann, "Comparison of simulators for fog computing," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, 2020, pp. 1792–1795.
[34] D. Perez Abreu , K. Velasquez, M. Curado, and E. Monteiro, "A comparative analysis of simulators for the cloud to fog continuum," *Simul. Model. Pract. Theory*, vol. 101, 2020, Art. no. 102029.
[35] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee and H. Karl, "MaxiNet: Distributed emulation of software-defined networks," in *Proc. IFIP Netw. Conf.*, 2014, pp. 1–9.

Lei Yang received the BSc degree from Wuhan University, in 2007, the MSc degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2010, and the PhD degree from the Department of Computing, Hong Kong Polytechnic University, in 2014. He is currently an associate professor with the School of Software Engineering, South China University of Technology, China. He has been a visiting scholar with Technique University Darmstadt, Germany from November 2012 to March 2013. His research interests include edge and cloud computing, distributed machine learning, and scheduling and optimization theories and techniques.



Fulin Wen received the BEng degree in software engineering from the South China University of Technology, China, in Jun. 2019. He is working toward the postgraduate degree with the School of Software Engineering, South China University of Technology, China. His research interests include edge computing and distributed machine learning.



Jiannong Cao (Fellow, IEEE) received the BSc degree in computer science from Nanjing University, China, in 1982, and the MSc and PhD degrees in computer science from Washington State University, USA, in 1986 and 1990 respectively. He is currently a chair professor of distributed and mobile computing with the Department of Computing, The Hong Kong Polytechnic University. He is also the director of Internet and Mobile Computing Lab in the department and the Director of University Research Facility in Big Data Analytics. His research interests include parallel and distributed computing, wireless networks and mobile computing, big data and cloud computing, pervasive computing, and fault tolerant computing. He has co-authored five books in Mobile Computing and Wireless Sensor Networks, co-edited nine books, and published more than 500 papers in major international journals and conference proceedings.

Zhenyu Wang received the BSc degree in computer science degree from Xiamen University, China, in 1987, and the MSc and PhD degrees in computer science from the Harbin Institute of Technology, China, in 1990 and 1993. He is currently a professor and the dean with the School of Software Engineering, South China University of Technology, China. His research interests include cloud and edge computing, social computing, and blockchain.