Swarm: Playground for Large-scale Decentralized Learning Simulations

Sangsu Lee*, Haoxiang Yu*, Xi Zheng†, and Christine Julien*
*Department of Electrical and Computer Engineering, University of Texas at Austin
{sethlee, hxyu, c.julien}@utexas.edu
†Department of Computing, Macquarie University, james.zheng@mq.edu.au

***Abstract*—Decentralized learning is an emerging field of research that opens doors for many novel pervasive computing applications. In decentralized learning, model training is offloaded to devices in the edge, and in some approaches, functions entirely without a central controller. SWARM is a tool for fast and large-scale simulations to test the performance of the practical implementations of decentralized learning algorithms that underlie many pervasive computing applications. In SWARM, developers can easily launch simulations for their algorithms by simply writing code that defines the behavior of a device when collaborating with others. The developer delegates to SWARM the emulation of the devices' encounters, given a pervasive computing scenario. By decoupling the encounter emulation and the learning algorithm execution, SWARM makes the configuration of diverse application scenarios easy and their simulations repeatable. Moreover, developers can evaluate the scalability of an algorithm in diverse and large-scale application contexts as SWARM can automatically deploy and manage multiple worker nodes. Finally, the SWARM Dashboard provides a visualization of the simulation progress and the algorithm performance.**

***Index Terms*—mobile computing, distributed machine learning, computer simulation**

I. INTRODUCTION

Recent advances in processing power of mobile devices enables machine learning tasks to be offloaded from central servers to mobile devices. This is becoming an increasingly favorable choice for pervasive computing applications as smartphones, sensors, and IoT devices collect massive amount of data. Rather than sending raw data to a centralized aggregator, the data can be processed on-device to save communication bandwidth. This can also be motivated by privacy-preserving machine learning, where private data is used to train a model locally, without directly sharing the data to a third party.

There has been significant recent attention to decentralized learning. Federated learning (FL) [4] is a technique that trains a model over decentralized data that resides in mobile devices. While this still requires a central server, approaches such as gossip learning [5], and fully decentralized federated learning [1] do not require central control. Opportunistic collaborative learning (OppCL) [3] performs egocentric learning to train a personalized model by selectively incorporating decentralized data encountered on neighboring devices.

Most of the existing efforts related to decentralized learning are theoretical, and the implementations associated with each approach are written in ways that are specific to a given approach. However, algorithm and application developers alike desire to test and compare decentralized learning approaches quickly, efficiently, and in a real world context, without the complexities of implementing and deploying their solutions on real-world devices. On the other hand, demands for large-scale simulations require resources beyond those provided by a single machine. This is especially true since training machine learning models is computationally intensive, and decentralized learning simulations could involve thousands of devices that cannot fit in memory on a single machine. Tensorflow Federated[1] is a framework for testing a federated learning algorithms. However, it does not support multi-machine simulation nor fully decentralized algorithms.

This paper presents SWARM, a tool for large-scale testing of decentralized deep learning algorithms. For algorithm developers, using SWARM is more favorable than implementing experiments from scratch for the following reasons:

- Developers can easily deploy repeatable simulations without dealing with complications associated with configuring the simulations. They can manage and monitor multiple simulations at once with SWARM.
- Developers can evaluate algorithms at a larger scale in a shorter time as SWARM scales the training task and handles the deployment and management of worker nodes by itself on behalf of the developer.
- SWARM is suitable for learning scenarios that involve complex encounter and collaboration patterns by relying on a configuration-time encounter trace to govern the interactions among simulated SWARM nodes.
- Developers can use SWARM to emulate devices by assigning the training task for each device to an isolated docker container that can be configured to emulate a variety of heterogeneous resource-constraints.

In short, SWARM is a highly configurable and removes the burden of evaluating complex decentralized learning options. It deals with deployment, provisioning, and management of multiple worker nodes and distributes workloads to them where stateless servers are running, which allows developers to easily and efficiently test algorithms in a larger scale.

II. OVERVIEW OF SWARM

Configuring Simulations. Fig. 1 depicts a workflow between SWARM and an algorithm developer and how the

[1] https://www.tensorflow.org/federated

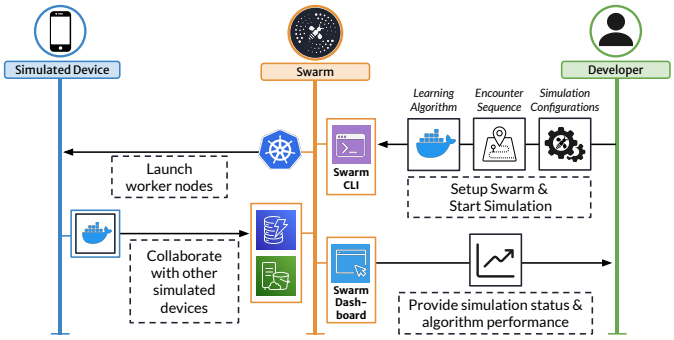

Fig. 1: Developers can deploy SWARM simulations via the CLI and monitor its status and performance on the Dashboard.

SWARM controller and simulated devices on worker nodes interact with each other. First, SWARM expects developers to represent a decentralized learning scenario with three components to configure a simulation; (i) the learning algorithm, (ii) an encounter sequence, and (iii) simulation configurations.

The learning algorithm shared with SWARM defines a simulated device's behavior when it is within collaboration range of other devices. This includes (i) the logic that determines whether the devices collaborate on a learning task and (ii) the specific actions that each partner in the collaboration takes. For instance, in a classic decentralized FL algorithm, two devices that encounter one another may simply exchange their model gradients, to be averaged into the locally constructed model. In OppCL, one of the devices shares its personalized model gradients, and the other performs a round of training to generate an update for those gradients based on the private local data. Algorithm developers construct a Python class that extends a base `Device` class and implements the individual device behavior. This functionality is then packaged as a docker container and given to SWARM.

In decentralized learning, devices encounter one another opportunistically, collaborate on a learning task, then move apart. The performance of a decentralized learning algorithm depends heavily on the order in which devices encounter one another. For a device moving in a physical world, this order is determined by its mobility trace. In SWARM, the order is decoupled from the simulation of the learning task. Rather, it is fed to SWARM as an *encounter sequence* – a time-stamped sequence of device pairs that captures which devices meet which other devices and for how long they are connected. Every encounter is labeled with an *encounter index*, which is a monotonically increasing integer from 0 that allows worker nodes to track the progress of the simulation.

The additional simulation parameters include (i) training settings, (ii) environment settings, and (iii) simulation settings. Training settings refer to the configurations and hyperparameters related to model training and learning algorithm, e.g., model, dataset, optimizer, and hyperparameters associated with them. These parameters are used to seed the python implementations of the learning algorithms that are encapsulated in the docker containers. Developers can also set parameters that are related to the computational or communication capabilities of

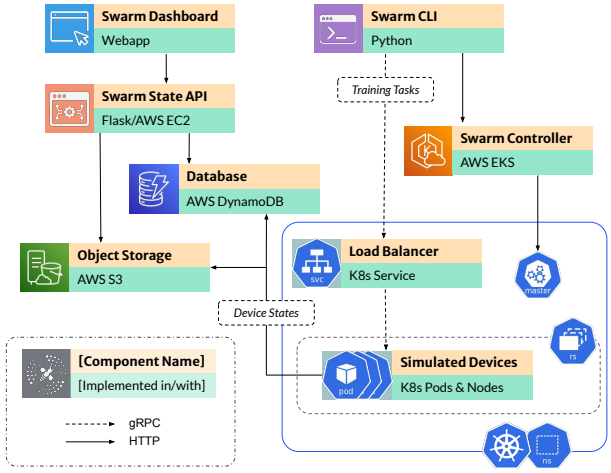

Fig. 2: SWARM system design

devices. For instance, the latency of collaboration between devices can be set. Third, examples of simulation settings include a random seed for reproducing the simulation or the number of worker nodes. These configurations are stored in json format and can be used to reproduce previous experiments.

Running Simulations. Fig. 2 shows the system design of SWARM. When a simulation is configured, the developer enters a command on the SWARM CLI to start it. SWARM CLI asks the SWARM controller to automatically start worker nodes. The current implementation of SWARM relies on AWS Elastic Kubernetes Service (EKS)[2] and Kubernetes (K8s)[3] to deploy and manage worker nodes. A group of worker nodes assigned to a simulation is deployed as a K8s *ReplicaSet*; K8s ensures that a fixed number of *Pods* are running at any time. Each pod has one docker container that runs a stateless server. Deploying multiple pods yields faster running time for a simulation and enables developers to deploy large-scale experiments.

Next, *training tasks* for the simulation are distributed to the Pods through K8s load balancer, in a form of remote procedure call implemented with gRPC. A training task is a sequence of collaborations between two devices. While any pod can simulate any training task, it is generally advantageous to associate a pod with a simulated device.

SWARM maintains an AWS DynamoDB as a central aggregator for all simulated devices' training progress. SWARM tracks training progress using the encounter index and stores the history of model accuracy corresponding to previous encounter indices. Moreover, the database keeps unique identifiers of all training data items allocated as a device's local data and all other mutable parameters associated with the learning algorithm. Because the state of a device changes over time, a worker node fetches the current state of the simulated devices by querying the database. In particular, they can fetch the state of a simulated device at a specific point of the simulation by looking at the encounter index. For example, when a worker node is simulating an encounter between device 0 and 1 with encounter index 10, it first checks whether all the encounters

[2] https://aws.amazon.com/eks/ [3] https://kubernetes.io/

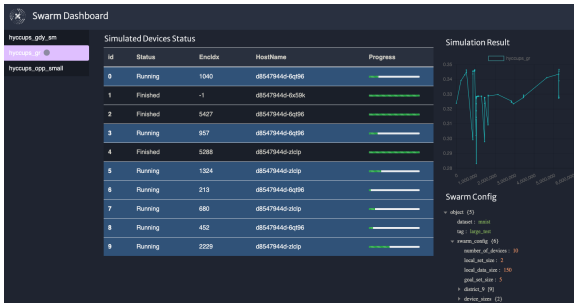

Fig. 3: SWARM Dashboard

prior to encounter index have been processed for devices 0 and 1. If not, the simulation on the worker node is blocked until it is, and when it is, worker node gets the device states for these two devices at encounter index 10 from the database and proceeds with the simulation.

An object storage (AWS S3) is used to store large objects such as model parameters and datasets. In simulations, model parameters change frequently. Therefore, a simulated device caches others' models in local storage and only downloads models from the object storage if there has been any update, which can be checked by querying the database whenever the device needs to run a training task.

SWARM Dashboard. The states of the simulated devices are made available via the Swarm State API, which is hosted by a web server. SWARM Dashboard is a web application that uses the Swarm State API to display the progress of the simulation in real time. Fig. 3 shows screenshot of the web application. Developers can check the progress of any running simulations on the dashboard. The table in the middle shows the progress of simulated devices and which pod the training task of the device is allocated to. In the screenshot, the first row of the table shows that training tasks associated with the simulated device 0 are currently running. Also, we can see that the last encounter index processed was 1040, and the pod name in K8s which has simulated the device. The dashboard also visualizes the performance of the model (top right), and displays the configuration of the simulation (bottom right).

III. CASE STUDY: OPPCL

In this section, we discuss a case study of SWARM running a simulation for OppCL algorithms [3]. First, a developer downloads a docker image from docker hub, launches the image, and writes a new `Device` class overriding the method that is called when a neighbor device is encountered. In particular, in OppCL, the developer writes the code that determines whether asking a neighbor device to perform a round of training on its local data would be useful. The developer also implements the training round performed on the neighbor's local data using the device's personalized model and the merging of the updated gradients back into the personalized model. Once complete, the developer repackages the resulting code in a new docker image and pushes it to the docker hub. The URL to the docker image is later specified in SWARM configurations.

Next, the developer creates a `csv` file specifying the encounter sequences. The source of the encounter sequences is

```
{
    "dataset": "mnist",
    "tag": "hyccups",
    "swarm_config": { "number_of_devices": 10 },
    "device_config": {
        "encounter_config": { "computation_time":
    0.24 },
        "train_config": { "optimizer": "adam" },
    },
}
```

Listing 1: Example configuration for OppCL simulation

up to the developer, for example, it could be created manually, sampled from random a random walk, or generated from an existing contact dataset. Each line in the encounter sequence file contains (1) an encounter index; (2) the device ids of the two devices; and (3) the duration of the encounter.

Finally, the developer sets the remaining parameters for the SWARM configuration. Listing 1 shows an abbreviated example for a simulation that trains using the MNIST [2] dataset on 10 devices, tagged as "hyccups". The developer sets the computation time for a single round of OppCL, which affects the execution of the learning algorithm in the simulation. Finally, the developer starts the simulation using the SWARM CLI and monitors its progress using the Dashboard.

IV. DEMONSTRATION AND TECHNICAL REQUIREMENTS

SWARM is a cloud-based service, and we will demonstrate creation and deployment of simulations on SWARM CLI using a laptop. Visitors may view the SWARM Dashboard page via their own devices, including laptops and mobile devices with an Internet connection. We will also have some introductory tutorials that visitors can explore to create their own simulations based on some templates. They will also be able to navigate the webpage to check the status of simulations. We will provide our own laptops for the demonstration and require only access to power and Internet.

V. FUTURE WORK

We plan to release SWARM as an open source programming environment that will be made available in advance of the conference (with a link to the artifact within the camera ready paper). Future planned enhancements to the tool include the ability to use SWARM as a library, rather than having to package algorithms manually to a docker container. Finally, SWARM can be extended to emulate resource-constrained devices by running docker containers with limited communication bandwidth and computational capabilities.

REFERENCES

[1] A. Lalitha et al. Fully decentralized federated learning. In *3rd Workshop on Bayesian Deep Learning (NeurIPS)*, 2018.

[2] Y. LeCun et al. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[3] S. Lee et al. Opportunistic federated learning: An exploration of egocentric collaboration for pervasive computing applications. In *Proc. of PerCom*, pages 1–8, 2021.

[4] B. McMahan et al. Communication-efficient learning of deep networks from decentralized data. In *Proc. of AISTATS*, 2017.

[5] R. Ormándi et al. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.