

Emulation Framework for AIoT Federated Learning

Brendan Ang Wei Jie

January 28, 2024

1 Abstract

Federated Learning (FL) allows a fleet of devices to collaborate towards a globally trained machine learning model. Research has continued to produce novel algorithms to tackle different issues faced in FL such as data heterogeneity and network costs. The performance of these algorithms depend, in part, on the system parameters used such as the number of clients participating in each round as well as real world factors such as client drop off. However, measuring this performance through a realistic benchmark would require one to procure a large fleet of devices which is infeasible and costly. Therefore, the role of software simulation is to support researchers with the tools to do so. This work introduces a software emulation framework utilizing QEMU and the Zephyr OS to streamline the process of building a fleet of clients and allow validation testing of FL system parameters that is not hardware-agnostic.

2 Acknowledgements

I would like to thank Associate Prof Tan Rui, who provided me with this unique opportunity to work on such an interesting problem and for guiding me throughout the entire journey with new and refreshing ideas. Special thanks goes to my partner for her unwavering support and critical role as my proofreader.

Contents

1	Abstract	2
2	Acknowledgements	3
3	Introduction	5
3.1	Problem	5
3.2	Literary Review	6
3.3	QEMU	6
3.4	Zephyr OS	7
4	Implementation	7
4.1	Bootstrapping Process	8
4.2	Client	10
4.2.1	Machine Learning	12
4.3	Server	12
4.3.1	Metrics	13
5	Experiments	13
6	Limitations	14
6.1	Long training times	14
6.2	Limited algorithm support	15

List of Figures

1	Host-guest emulation architecture	11
2	Client initialization sequence	12
3	FL round	13
4	HTTP endpoints	13
5	GUI Implementation	14

3 Introduction

Federated Learning (FL) emerged as a method for solving key issues with the standard centralized learning approach, namely, it achieves the following properties:

- **Data Privacy:** a centralized training approach involves the need for the centralized machine performing the computation to have full access to the entire data set. With FL, data never leaves each individual client’s device. Instead, only the updated weights are shared to form the global model.
- **Scalability:** FL enables leveraging a network to perform computation in parallel.

While FL has been successful in these aspects, there are a few limitations to the approach which may not make FL practical in some applications. In particular, the distributed nature of FL poses a few key challenges:

- **Limited computing resources:** individual devices may face memory constraints, limiting the size of the local model it is able to train. Constraints on computing power also lead to longer training times. This is particularly so for Internet of Things (IoT) devices.
- **Network limitations:** communication speed can become the bottleneck for performance as IoT devices rely on unstable wireless communication networks. Furthermore, data constraints can exist such that the number of bytes sent may be a cost inducing factor.

Hence, the development of FL algorithms often seek to advance progress towards solving the aforementioned issues.

3.1 Problem

Prior to effectively deploying FL, different factors including the convergence rate and model accuracy needs to be well studied. This gives rise to the need for software simulation. However, the simulation of FL in the distributed setting involves dealing with issues which do not arise in datacenter ML research. These include running on different simulated devices, each with potentially varying amounts of data. Furthermore, metrics such as the number of bytes transferred by the device, as well as the ability to simulate real-world issues such as client drop-out, can also be important for proposed FL algorithms to handle.

3.2 Literary Review

FedML[1] is a research-oriented library which supports various algorithms and 3 platforms, on-device training for IoT and mobile devices, distributed computing and single-machine simulation. On-device training is supported only on Raspberry Pi 4 and NVIDIA Jetson Nano which limits hardware validation to these 2 devices. Simulation is offered through the FedML Parrot[2], which offers an accelerated simulation framework employing multiple optimizations to improve simulation speed.

FLSim[3] aims to provide a simulation framework for FL by offering its users a set of software components – which can then be mixed and matched to create a simulator for their use case. Developers need only define the data, model and metrics reported, and FL system parameters can be altered in a JSON configuration file. FLUTE[4] adds additional features by allowing users to gain access to cloud based compute and data. However, these simulation solutions are consequently hardware-agnostic. Without taking into account the specific platforms which FL is performed on, these simulators cannot provide insight into whether the system will work in the production environment.

Here, this work proposes a new emulation framework `zfl`, which aims to achieve the following properties:

- Provide better insight into platform specific support by running FL on emulated hardware through QEMU and Zephyr OS.
- Collect useful metrics during the FL lifecycle.
- Support implementation of real word issues such as variable data and client drop-off.

3.3 QEMU

QEMU[5] is an open source machine emulator and virtualizer. It enables system emulation, creating a virtual model of an entire machine (CPU, memory and emulated devices) to run a guest OS. In this mode, the CPU may be fully emulated, or it may work with a hypervisor to allow the guest to run directly on the host CPU. In `zfl`, QEMU with full CPU emulation is used without a hypervisor as part of its software architecture to emulate hardware. Note that by default, the

framework is built for QEMU x86 32-bit, which entails the use of the qemu-i386 binary in section 4.

3.4 Zephyr OS

To emulate the issue of limited computing resources, it is important to make use of system runtimes used by those devices. Zephyr OS[6] is one such operating system. It is based on a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications. Furthermore, Zephyr is highly configurable, allowing the user to choose only the specific kernel services required, and also delve into lower level memory mapping of the system SRAM and DRAM. Most importantly, Zephyr supports a wide range of CPU architectures including ARM, RISC-V and x86. Zephyr also provides built-in capabilities for building for the QEMU environment, and implements key features such as networking through the QEMU built-in Ethernet adapter. In zfl, client code will be written to work in the Zephyr OS environment.

4 Implementation

To run on Zephyr OS, the entire software stack is developed in the C programming language. Although Zephyr also supports applications written in C++, the C programming language would make the framework more suitable to run on embedded devices.

In order to run machine learning training on each client, some configuration is required to ensure that we have sufficient memory to allocate for the neural network model and training data. Depending on the architecture of the model and the size of the data, these options may need to be further tweaked to prevent any runtime memory allocation crashes. Listing 1 shows the options made to `prj.conf` to create a runtime capable of up to 10MB of heap memory. Listing 2 shows the configuration made to the QEMU board under `qemu_x86.dts` for a total DRAM size of 15MB. This is in excess of the heap memory size as additional memory is reserved for the kernel image and stack memory.

```

1 CONFIG_SRAM_SIZE=15360
2 CONFIG_MAIN_STACK_SIZE=8192
3 CONFIG_KERNEL_VM_SIZE=0
  x7000000
4 CONFIG_HEAP_MEM_POOL_SIZE
  =10485760

```

Listing 1: Zephyr prj.conf

```

1 #define DT_DRAM_SIZE
  DT_SIZE_K(15360)

```

Listing 2: QEMU board.dts

The main zfl binary is used to bootstrap either the client or server aspects of FL simulation. Listing 3 shows the different options available for configuring FL parameters.

```

1 ./zfl client <num_clients> <epochs> <batch_size>
2
3 ./zfl server <num_rounds> <clients_per_round>

```

Listing 3: The main zfl binary

4.1 Bootstrapping Process

Next the framework needed a method for spawning an arbitrary number of QEMU instances, and allowing these instances to communicate back to the server. When called in client mode (Listing 5), zfl accomplishes this by making use of the “fork” and “exec” pattern with the desired number of clients.

However, each instance of QEMU persists in an isolated network different from the host PC, and is not able to communicate. We can bypass this limitation using a network bridge to act as a virtual network device forwarding packets between connected network devices. The network bridge is set up on the host under the name “zfl” with a set of network parameters using the `ip` command line utility.

```

1 ip link add $INTERFACE type bridge
2 ip addr add $IPV4_ADDR_1 dev $INTERFACE
3 ip link set enp61s0 master $INTERFACE
4 ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
5 ip link set dev $INTERFACE up

```

Listing 4: Network bridge setup

To tell QEMU to use it, we pass the name of the bridge along with a randomly generated MAC address as arguments to the `-nic` flag

```
-nic bridge,model=e1000,mac=%s,br=zfl
```


Although each QEMU client is now able to communicate with the host via the NIC adapter, we still needed a way to monitor the output of each instance. One method is to transmit output and logs over the network. However, this would not allow important crash logs and stacktrace information to be transmitted as the application software would have shutdown. To overcome this, the host creates a named or FIFO pipe for each client, which then is passed in to QEMU through the `-serial` flag. Now, all standard output goes through the named pipe, ready to be read by the host. Another outcome of this is that the host is now also able to send input to each instance, whose importance will be described in the next section. Listing 5 showcases how each feature configured through the QEMU command line binary.

```

1 pid_t child = fork();
2 if (child < 0) {
3     printf("ERROR: could not fork client %d: %s\n", i, strerror(
4         errno));
5     return 1;
6 }
7 ...
8 // generate serial arguments
9 char serial_arg[80];
10 snprintf(serial_arg, 80, "pipe:%s", pipe_path);
11
12 // generate nic arguments
13 char nic_arg[100];
14 char *mac = generate_random_mac();
15 snprintf(nic_arg, sizeof(nic_arg), "bridge,model=e1000,mac=%s,br=
16     zfl", mac);
17
18 // start client as new process
19
20 execlp("qemu-system-i386", "qemu-system-i386",
21     "-m", "15", "-cpu", "qemu32,+nx,+pae", "-machine", "q35",
22     "-device", "isa-debug-exit,iobase=0xf4,iosize=0x04",
23     "-no-reboot", "-nographic", "-no-acpi",
24     "-serial", serial_arg,
25     "-nic", nic_arg,
26     "-kernel", "./zflclient/out/zephyr/zephyr.elf",
27
28
29
30

```

```
NULL);
```

Listing 5: Client forking process

In addition to the NIC adapter configuration, each client needed a unique Internet Protocol Version 4 (IPv4) address in order to establish TCP based connections with the central server. Samples provided by Zephyr OS describe a way to achieve this by setting a compile time configuration flag `CONFIG_NET_CONFIG_MY_IPV4_ADDR`. However, this is impractical to scale to a large number of clients, since each client would need a separately compiled binary. One solution to this problem is to assign the IPv4 address dynamically using the built-in Zephyr network function[7]:

```
net_if_ipv4_addr_add
```

To achieve this, each client starts off as a Zephyr shell instance and a user-defined command `run` is registered, taking in a command line argument which defines the desired IP address of the instance (Listing 6). Finally, this command triggers the `run` function (Listing 7) as a way to start the main program.

Overall, the complete emulation architecture is illustrated in figure 1, describing the interactions between the host system and guest instances.

```
1 SHELL_CMD_ARG_REGISTER(run, NULL, "Run with IPv4 address", run,
   4, 0);
```

Listing 6: Registering user defined command “run” to the function pointer `run`

```
1 char *addr_str = argv[1];
2 LOG_INF("instance ipaddr is %s", addr_str);
3 struct in_addr addr;
4 zsock_inet_pton(AF_INET, addr_str, &addr);
5 if (!net_if_ipv4_addr_add(net_if_get_default(), &addr,
   NET_ADDR_MANUAL, UINT32_MAX)) {
6     LOG_ERR("failed to add %s to interface", addr_str);
7     return -1;
8 }
```

Listing 7: The “run” function which performs IP address assignment at runtime

4.2 Client

The first step within the internal implementation of each client is to establish TCP socket connection to the central server and obtain their assigned ID. This is then

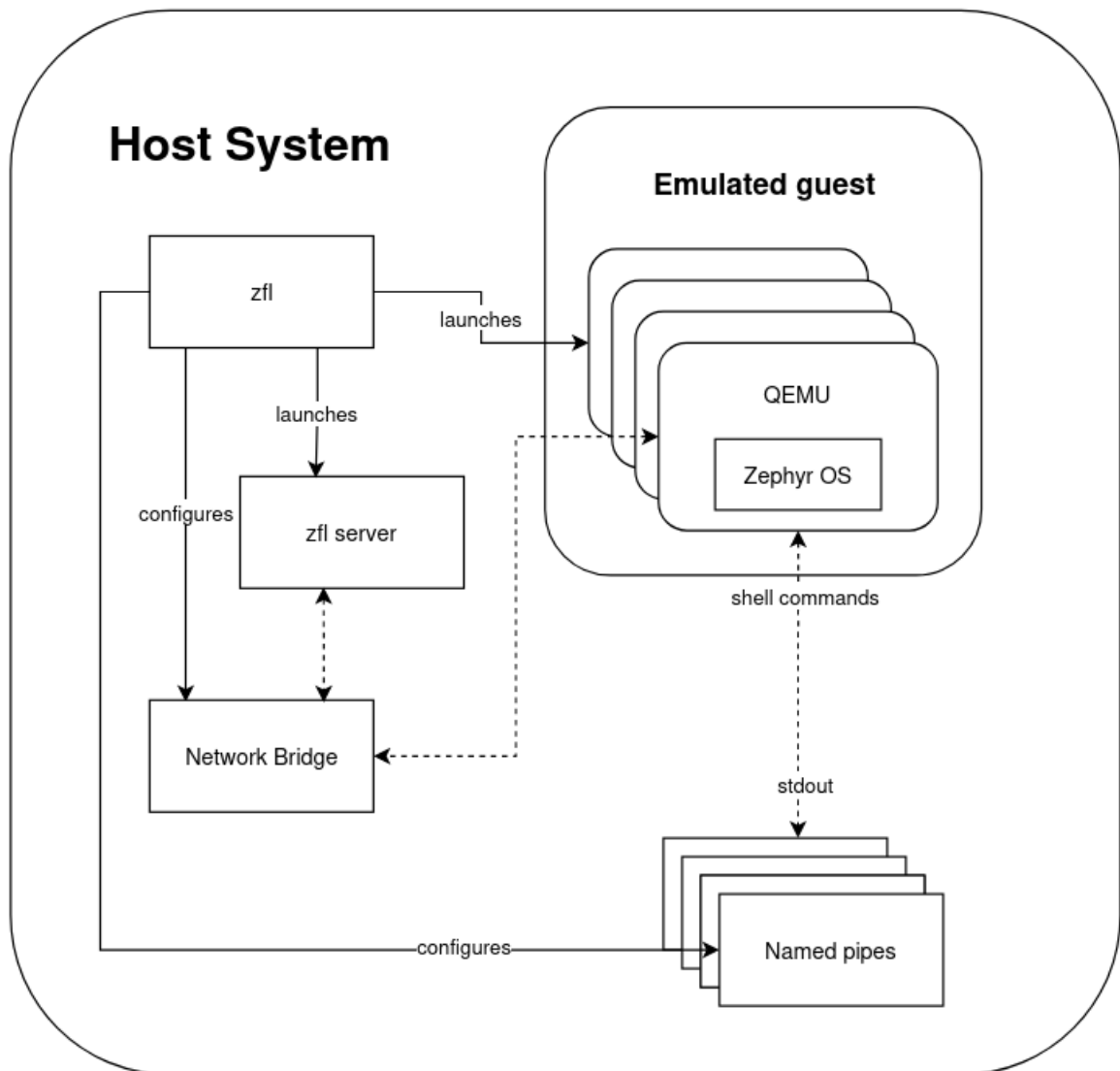


Figure 1: Host-guest emulation architecture

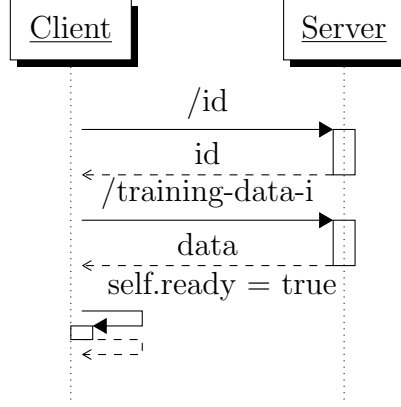


Figure 2: Client initialization sequence

used to obtain the training data from the server which corresponds to that ID. Thus, this process allows for training data to be dynamically assigned to different instances. Finally, each client starts a HTTP server and marks itself as ready to begin the training round. The complete initialization process is illustrated in Figure 2.

4.2.1 Machine Learning

The framework implements the FedAvg algorithm[8] as part of its simulation. The underlying neural network implementation utilizes `nn.h`[9], an open source educational neural network C library, modified for specific use as described in section 5.

4.3 Server

The central server runs on the host machine without needing additional configuration. Its primary purpose is to aggregate individual client weights every round. Implemented HTTP endpoints are described in figure 4. After assigning each connecting client their ID and training data, it performs the function `start_round` at an interval of 10 seconds. Each time, the server pings all previously connected clients to check if they are ready to start the training round. When enough clients are ready, it sends a HTTP POST request to the `train` endpoint of the client, which then triggers the training function. Once the client has completed its training, it sends the resulting weights of the local model to the HTTP POST endpoint `results` of the server, which then performs the aggregation. The sequence of events for a single round is illustrated in Figure 3.

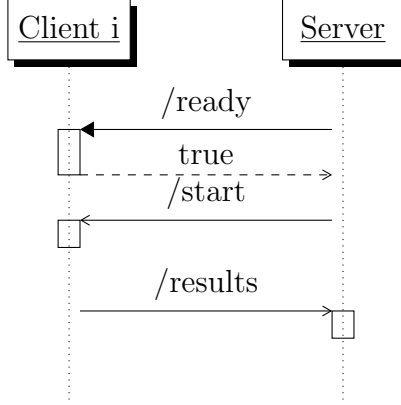


Figure 3: FL round

Client		
Endpoint	HTTP Method	Description
/start	POST	Initiates training round using updated weights with JSON body: {weights: int}
/ready	GET	Gets client ready status
Server		
/results?id=	POST	Client id submits results with JSON body: {round: int, weights: string}
/id	GET	Obtain an id for round participation
/training-data?id=	GET	Obtain training data for id
/training-labels?id=	GET	Obtain training labels for id

Figure 4: HTTP endpoints

4.3.1 Metrics

As part of the framework’s aim to track useful metrics, the server keeps track of the total number of bytes transferred between the clients and itself during the entire FL lifecycle. The total training time in seconds is also tallied. This is displayed along with additional details such as the current round, number of ready clients, as well as a graph of the validation set loss and accuracy on a simple graphical user interface shown in Figure 5.

5 Experiments

The framework is tested using FedAvg over the MNIST digit recognition data set of 60000 labels. Due to memory constraints, clients are configured with a

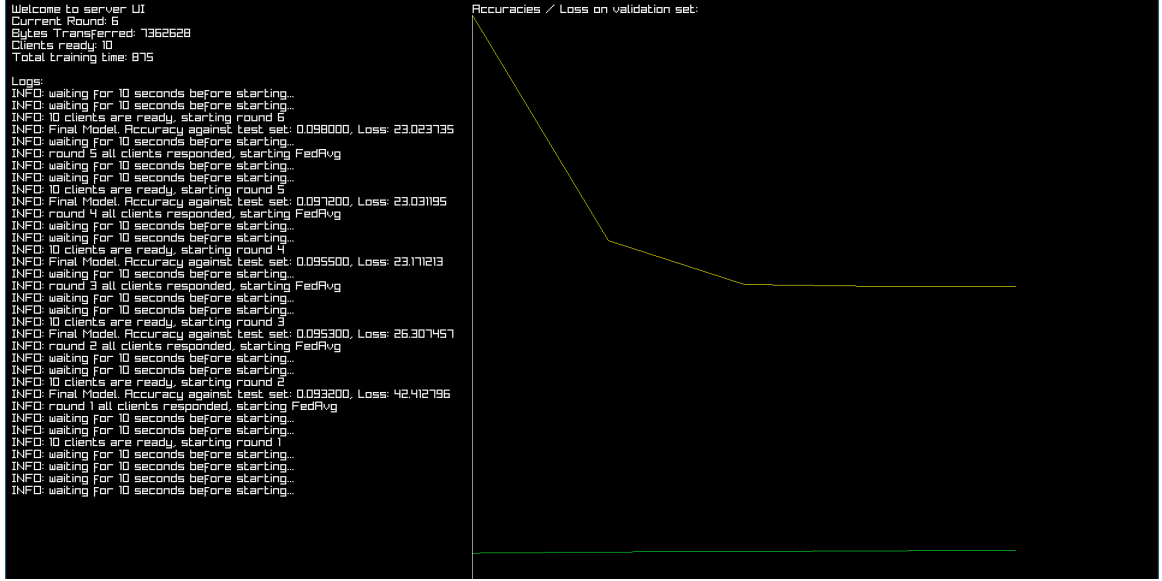


Figure 5: GUI Implementation

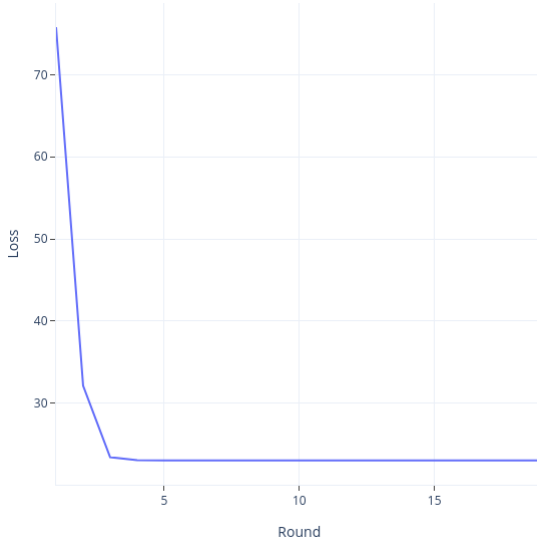
neural network architecture with 1 hidden layer of 16 nodes. Additionally, implementations for the softmax activation and cross entropy were made to ensure suitability for machine learning with MNIST categorical data.

To test the sanity of the framework, the data is first shuffled and split equally into 100 sets of 6000 labels. The server is started for 20 rounds and 100 clients per round and clients were started with 5 epochs and a batch size of 600. Figure 6a shows the validation set loss and Figure 6b shows the validation set accuracy at the end of each round.

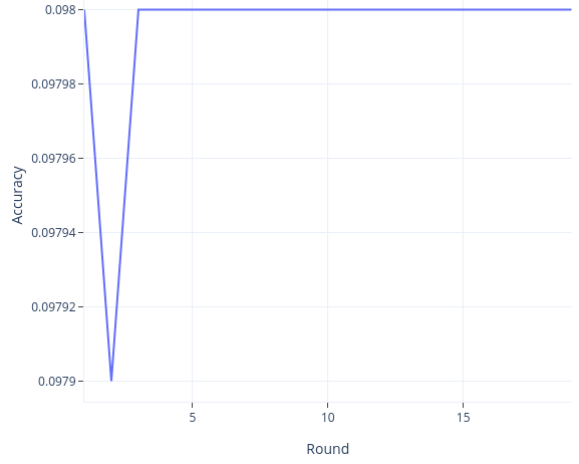
6 Limitations

6.1 Long training times

Unoptimized client training implementations has resulted in long training times required per FL round. This is because standard implementations for machine learning algorithms that can be easily ported for use in Zephyr OS could not be found. For example, Zephyr OS provides external modules for supporting TensorFlow Lite Micro[10] and has examples for running pre-trained neural network on their platform. However, on-device training is not supported.



(a) 20 round validation set loss



(b) 20 round validation set accuracy

6.2 Limited algorithm support

The framework currently only implements training algorithms for FedAvg. The architecture of zfl also inherently assumes the use of a single server instance used for aggregation. However, this limitation can be bypassed by porting specific HTTP endpoints from the server to the client. For example, the implementation of the `results` endpoint, would effectively move aggregation into each client instance. This could potentially allow the framework to support hierarchical FL algorithms[11].

References

- [1] C. He, S. Li, J. So, Z. Mi, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, “Fedml: A research library and benchmark for federated machine learning,” *arXiv (Cornell University)*, 07 2020.
- [2] Z. Tang, X. Chu, R. Y. Ran, S. Lee, S. Shi, Y. Zhang, Y. Wang, A. Q. Liang, S. Avestimehr, and C. He, “Fedml parrot: A scalable federated learning system via heterogeneity-aware scheduling on sequential and hierarchical training,” *arXiv (Cornell University)*, 03 2023.
- [3] L. Li, J. Wang, and C. Xu, “Flsim: An extensible and reusable simulation framework for federated learning,” pp. 350–369, 01 2021.
- [4] M. H. Garcia, A. Manoel, D. M. Diaz, F. Miresghallah, R. Sim, and D. Dimitriadis, “Flute: A scalable, extensible framework for high-performance federated learning simulations,” *arXiv.org*, 11 2022. [Online]. Available: <https://arxiv.org/abs/2203.13789>
- [5] T. Q. P. D. , “About qemu — qemu documentation,” *www.qemu.org*, 2023. [Online]. Available: <https://www.qemu.org/docs/master/about/index.html>
- [6] “Introduction — zephyr project documentation,” *Zephyrproject.org*, 2015. [Online]. Available: <https://docs.zephyrproject.org/latest/introduction/index.html>
- [7] B. Schirrmeister, F. Geyer, and S. Späthe, “Simulation and benchmarking of iot device usage scenarios using zephyr and qemu,” *SMART 2020*, pp. 6–11, 09 2020.
- [8] M. H. Brendan, E. Moore, D. Ramage, S. Hampson, and Blaise, “Communication-efficient learning of deep networks from decentralized data,” *arXiv (Cornell University)*, 02 2016.
- [9] T. , “tsoding/nm.h,” *GitHub*, 01 2024. [Online]. Available: <https://github.com/tsoding/nm.h>
- [10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray,

- C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [11] O. F. Rana, T. Spyridopoulos, N. Hudson, M. Baughman, K. Chard, I. Foster, and Aftab, “Hierarchical and decentralised federated learning,” *arXiv (Cornell University)*, 04 2023.