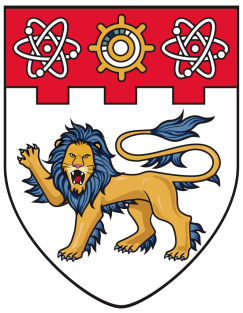


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Emulation Framework for AIoT Federated Learning

Brendan Ang Wei Jie

School of Computer Science and Engineering
2024

1 Abstract

Federated Learning (FL) allows a fleet of devices to collaborate towards a globally trained machine learning model. However, applications of FL for low-powered Artificial Intelligence Internet of Things (AIoT) devices continue to face key challenges in limited computing resources and network limitations. Although researchers have continued to produce novel algorithms to tackle these issues, measuring this performance through a realistic benchmark would require one to procure a large fleet of devices which is often infeasible and costly. The key role of software simulation is thus to provide the tools to assist in the process. Software systems can allow researchers to tweak FL system parameters such as the number of clients, as well as view metrics such as network costs and training time. This allows them to validate new ideas easily. However, simulation that is hardware-agnostic is not enough, especially for embedded devices where machine learning support is not standardized. Hence, this work introduces a software emulation framework called "zfl", utilizing QEMU and the Zephyr OS to streamline the process of building a fleet of clients and allow validation testing of an entire FL system lifecycle that is hardware dependent.

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 family and of course, my partner for her unwavering support and critical role as my editor.

Contents

List of Figures

3 Introduction

3.1 Federated Learning

Federated machine learning (FL) emerged as a method for solving key issues with the standard centralized learning approach, namely, it serves as a potential solution to the problem of input 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 or other model parameters are shared to form the global model, keeping the data local and private.

The key components and processes which make up the FL lifecycle and allow it to achieve the aforementioned advantages over centralized machine learning can be detailed as follows:

- Client Devices: these are the decentralized entities, such as smartphones or IoT devices, possessing local datasets and computational resources. Each client device conducts local model training on its data.
- Model Updates: instead of sharing raw data, client devices transmit model updates or gradients to the central server. These updates reflect the knowledge gained from local data during model training.
- Aggregation Mechanism: the model updates are aggregated and combined using techniques like federated averaging or secure aggregation, ensuring that the privacy of individual clients' data is preserved while constructing a robust global model.
- Deployment: the updated global model is then deployed back to the participating devices and the cycle repeats.

These properties makes FL highly relevant in the context of Artificial Intelligence of Things (AIoT) devices. By leveraging the vast amount of data generated by IoT devices, FL harnesses the collective intelligence of edge devices while respecting data sovereignty and privacy regulations. For example, Android utilizes FL training on user interactions for Smart Text Selection, which predicts the desired set of words around a user's tap[1].

While FL has been successful in these aspects, there are a few limitations to the approach which may not make FL as effective in some applications. In particular, areas which require low-powered devices face key challenges such as:

- 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 low-powered 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 by improving the convergence rate and increasing communication efficiency. In this front, researchers have come up with various methods such as utilising reinforcement learning to increase accuracy[2]. One study combined Federated Averaging with quantization of model updates to achieve reductions in communication cost while minimising impact on accuracy[3]. Another study made use of knowledge distillation[4] that can increase model accuracy while also occupying negligible network resources[5]. In this paper, we will not be focusing on how to optimize these factors, but rather focus on measuring these important metrics for FL research.

3.2 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 traditional data center machine learning 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 network packet loss, can also be important for proposed FL algorithms to handle.

3.3 Literary Review

There are a range of open software frameworks for federated learning research simulation which are available or in development. Some of them will be discussed here.

TensorFlow Federated(TFF)[6] offers a high level API for FL specifically targeting research use cases. In comparison to other frameworks which offer operations such as

send and receive as building blocks, TFF provides interfaces such as tff.federated_sum, tff.federated_reduce, or tff.federated_broadcast that encapsulate simple distributed protocols. TFF also offers large scale simulation capabilities such as GPU Accelerators as well as flexible orchestration with control over data sampling.

FLSim[7] is standalone library that is written using PyTorch[**?**] and supports diverse domains and accommodates use cases such as computer vision and natural language processing. FLSim also supports cross-device FL and aims to provide a user friendly API by offering users a set of software components which can then be mixed and matched to create a simulator for their specific use case. Developers need only define the data, model and metrics reported, and FL system parameters can be altered in a JSON[8] configuration file. FLUTE[9] is another similar software framework but adds additional features by allowing users to gain access to cloud based compute and data.

As a whole, the above mentioned simulation solutions perform computations on the local system and 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.

On the other hand, there are some frameworks which also focus on testing, evaluation and deployment by attempting to provide on-device training for FL. FedML[10] is one such library which supports various algorithms and 3 platforms, on-device training for IoT and mobile devices, distributed computing and single-machine simulation. Simulation is offered through the FedML Parrot[11], which offers an accelerated simulation framework employing multiple optimizations to improve simulation speed. However, on-device training is supported only on Raspberry Pi 4 and NVIDIA Jetson Nano which limits hardware validation to these 2 devices.

Flower[12] is another FL framework which offers on-device training capabilities. Android support is done by leveraging TensorFlow Lite[13] which is a mobile library for deploying models on mobile, microcontrollers and other edge devices. In addition, Flower clients are also implemented in Python for Raspberry Pi and NVIDIA Jetson TX2, providing some embedded support similar to FedML.

FEDn[14] takes this a step further by providing a production-grade and framework-agnostic distributed implementation with strong scalability and resilience features sup-

porting both cross-silo and cross-device scenarios. To this end, their implementation utilizes hierarchical FL[**?**] and offers users distributed deployments with no code change. NVIDIA FLARE[15] also aims to achieve similar goals.

3.4 Research Approach

It can be observed that industry solutions to software for FL support is mainly focused in two areas:

1. Simulation: offer direct access to industry standard machine learning algorithms and aggregation techniques through a high level user API, ran entirely on the developer or cloud system. Validation testing is done via simulated modelling of the federated learning parameters, ignoring most aspects of the runtime system.

2. On-device: offer simple integrations and deployments to real edge devices. Evaluation is done by benchmarking physical fleets of clients.

Hence, there is a lack of frameworks suitable for users who wish to validate and test FL algorithms on different hardware architectures, but are unable to procure large fleets of physical devices. Moreover, most existing frameworks rely on existing machine learning libraries such as TensorFlow and Pytorch, inherently requiring devices to have a Python runtime capable of supporting these dependencies. To bridge this gap, one approach to acquiring hardware dependent results is to emulate the hardware environment. This can be achieved through software systems. One such system explored the combination of Quick Emulator (QEMU) and Zephyr, a real time operating system to emulate embedded devices in the machine learning context[16]. However, this approach only emulated machine learning inference rather than on-device training.

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

- Provide better insight into hardware specific support, especially for embedded IoT applications by running FL on emulated hardware through QEMU and Zephyr OS.

- Collect useful metrics during the FL lifecycle, especially on variables typically difficult to simulate.

- Support simulation of real word scenarios such as variable data and network packet loss.

3.4.1 QEMU

QEMU[17] 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.

3.4.2 Zephyr OS

To emulate the embedded IoT environment, it is important to make use of system runtimes used by those devices. Zephyr OS[18] 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

The framework implements the classical FedAvg algorithm[19] as part of its simulation, where each client sends its updated model weights to a central server, which then performs aggregation using simple averaging. Hence, the architecture of the framework is based on the assumption of multiple clients talking to a single server.

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. Note that the framework is implemented for Linux and all tests are run on a Debian based machine[1].

4.1 Configuration

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.

```
CONFIG_SRAM_SIZE=15360
CONFIG_MAIN_STACK_SIZE=8192
CONFIG_KERNEL_VM_SIZE=0
    x7000000
CONFIG_HEAP_MEM_POOL_SIZE
    =10485760
```

Listing 1: Zephyr prj.conf

```
#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 and their meanings.

```
./zfl client -c num_clients -e epochs -b batch_size
# -c num_clients: Number of clients to boot up.
```

[1] Source code available under a GPL license at https://github.com/bbawj/zfl

```
# -e epochs:      Number of epochs (training passes on the entire
    data set).
# -b batch_size:  Number of training labels used for a single update
    .

./zfl server -r num_rounds -c clients_per_round
# -r num_rounds:        Number of rounds of aggregation.
# -c clients_per_round: Number of clients that need to respond to
    start a round.
```

Listing 3: The main zfl binary

4.2 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 as shown in Listing 4.

```
ip link add $INTERFACE type bridge
ip addr add $IPV4_ADDR_1 dev $INTERFACE
ip link set enp61s0 master $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
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 stack trace 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 into 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 also able to send input to each instance, whose importance will be described in the next section. Listing 5 showcases how each feature is configured through the QEMU command line binary.

```
pid_t child = fork();
if (child < 0) {
    printf("ERROR: could not fork client %d: %s\n", i, strerror(
    errno));
    return 1;
}
...

// generate serial arguments
char serial_arg[80];
snprintf(serial_arg, 80, "pipe:%s", pipe_path);

// generate nic arguments
char nic_arg[100];
char *mac = generate_random_mac();
snprintf(nic_arg, sizeof(nic_arg), "bridge,model=e1000,mac=%s,br=zfl
    ", mac);

// start client as new process
execlp("qemu-system-i386", "qemu-system-i386",

       "-m", "15", "-cpu", "qemu32,+nx,+pae", "-machine", "q35",
       "-device", "isa-debug-exit,iobase=0xf4,iosize=0x04",

       "-no-reboot", "-nographic", "-no-acpi",

       "-serial", serial_arg,

       "-nic", nic_arg,

       "-kernel", "./zflclient/out/zephyr/zephyr.elf",

       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[20]:

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. This exposes a shell command that takes 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.

```
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

```
int run(const struct shell *sh, size_t argc, char **argv) {
    // ...
    char *addr_str = argv[1];
    LOG_INF("instance ipaddr is %s", addr_str);
    struct in_addr addr;
    zsock_inet_pton(AF_INET, addr_str, &addr);
    if (!net_if_ipv4_addr_add(net_if_get_default(), &addr,
    NET_ADDR_MANUAL, UINT32_MAX)) {
        LOG_ERR("failed to add %s to interface", addr_str);
        return -1;
    }
    // ...
}
```

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

4.3 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 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

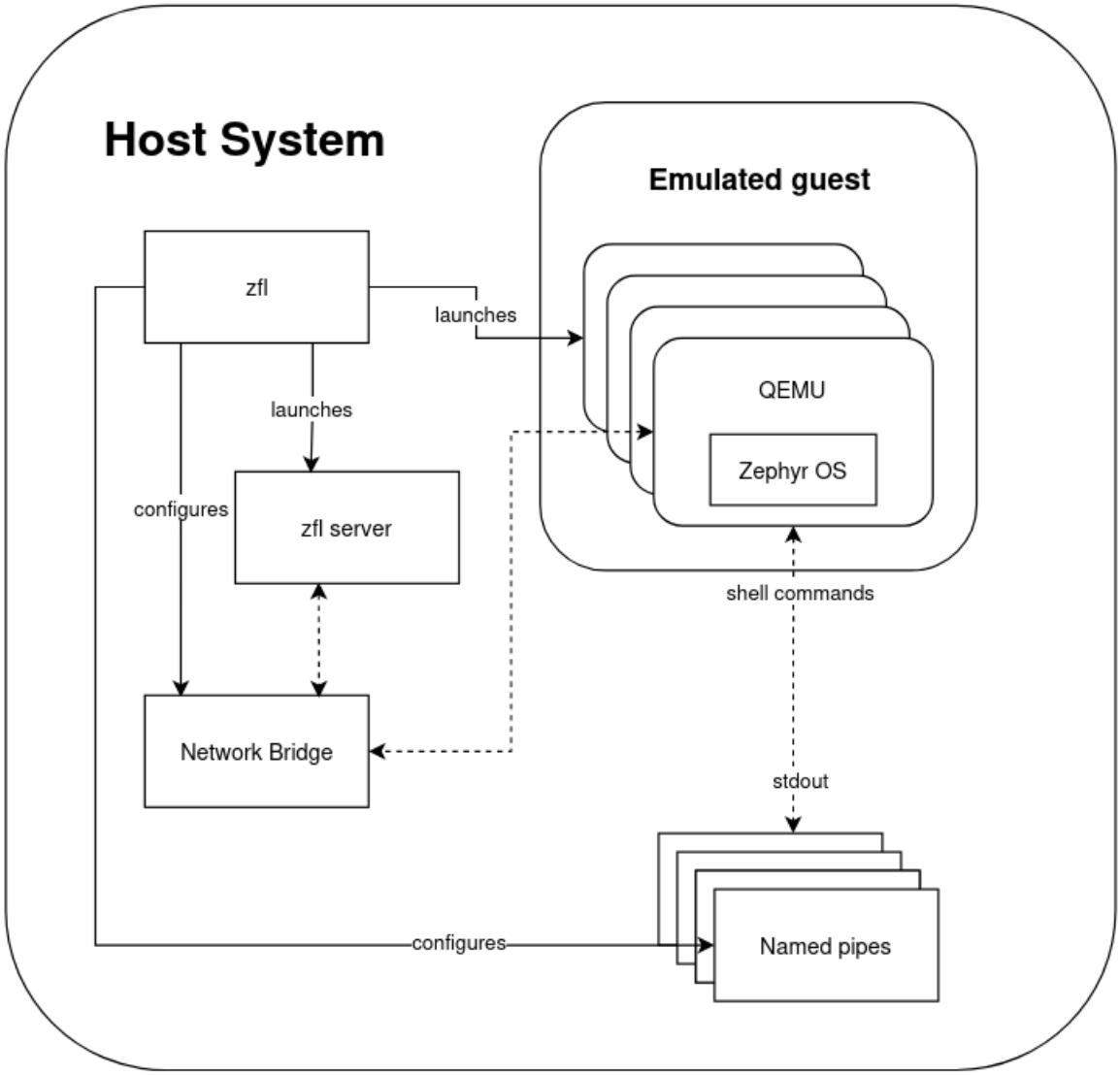


Figure 1: Host-guest emulation architecture

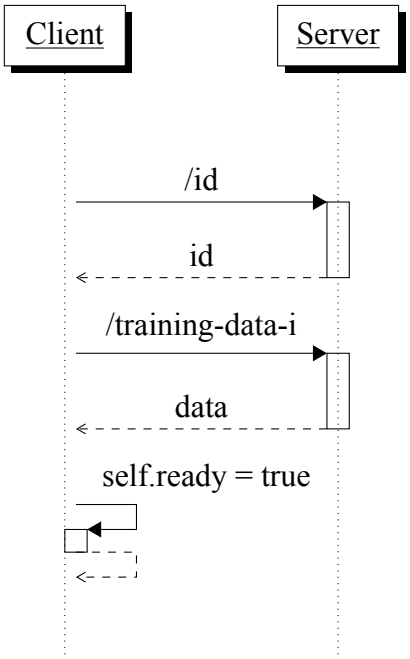


Figure 2: Client initialization sequence

client starts a HTTP server and marks itself as ready to begin the training round. The complete initialization process is illustrated in Figure 2.

4.3.1 Machine Learning

Once initialized, clients wait for a request to their `start` HTTP endpoint. This triggers the train function, which performs forwarding and backpropagation on the local data set. The underlying neural network implementation utilizes nn.h[21], an open source educational neural network C library. Porting the library for use included updating traditional POSIX syscalls to the Zephyr kernel syscalls. However, further modifications were made for specific use as described in section 5.

4.3.2 Metrics

The benefit of using emulated hardware is the ability to collect real statistics during the operation of the machine. The Zephyr OS exposes kernel APIs for user applications to query for network statistics such as bytes sent and received, as well as specific network L3 and L4 data including the number of IP and TCP packets dropped and retransmitted.

```
net_mgmt(NET_REQUEST_STATS_GET_TCP, NULL, &data, sizeof(data));
```

This data is queried by each client and sent together with their updated weights. 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.

4.4 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 `start` 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.

4.4.1 Scenario Emulation

zfl takes advantage of the emulated architecture to introduce hooks into tools which can emulate interesting scenarios. Passing the "–loss" flag into zflserver will enable packet loss emulation using the Linux Traffic Control[22] tool on the network interface between the server and clients. The flag translates to this command which adds 10% packet loss to our zfl network interface:

```
sudo tc qdisc add dev zfl root netem loss 10%
```

Other options can also be ported in this similar fashion to provide a wide variety of network emulation capabilities during the federated learning lifecycle.

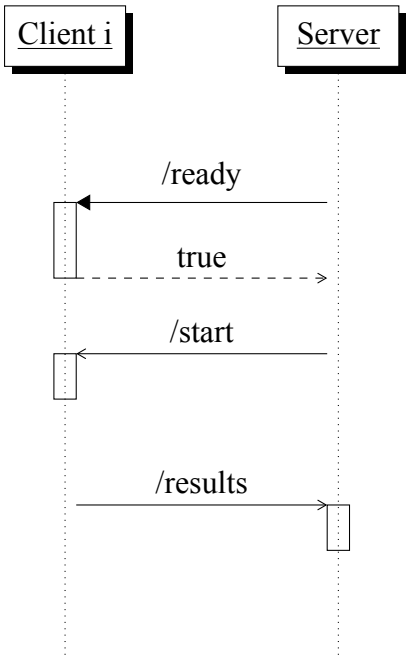


Figure 3: FL round

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

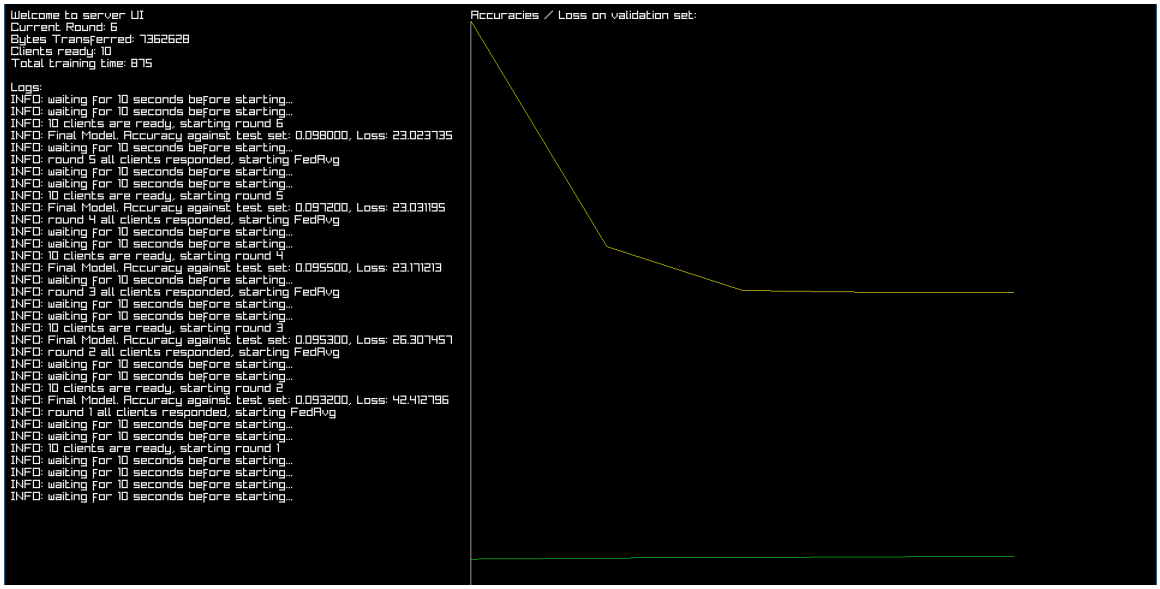


Figure 5: GUI Implementation

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 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.

5.1 Validation of zfl training accuracy

As mentioned earlier, factors affecting convergence rate and thus the number of global rounds is an active research area out of the scope of this study. However, to demonstrate the correctness of the implementation, the data is first shuffled and split equally into 50 sets of 1200 labels. zfl was then run with the following FL parameters:

- Server: 20 rounds and 50 clients per round
- Clients: 5 epochs and a batch size of 600.

The result in Figure 6 shows the validation set accuracy at the end of each round. In particular, the experiment manages to achieve the baseline centralized performance of 86%.

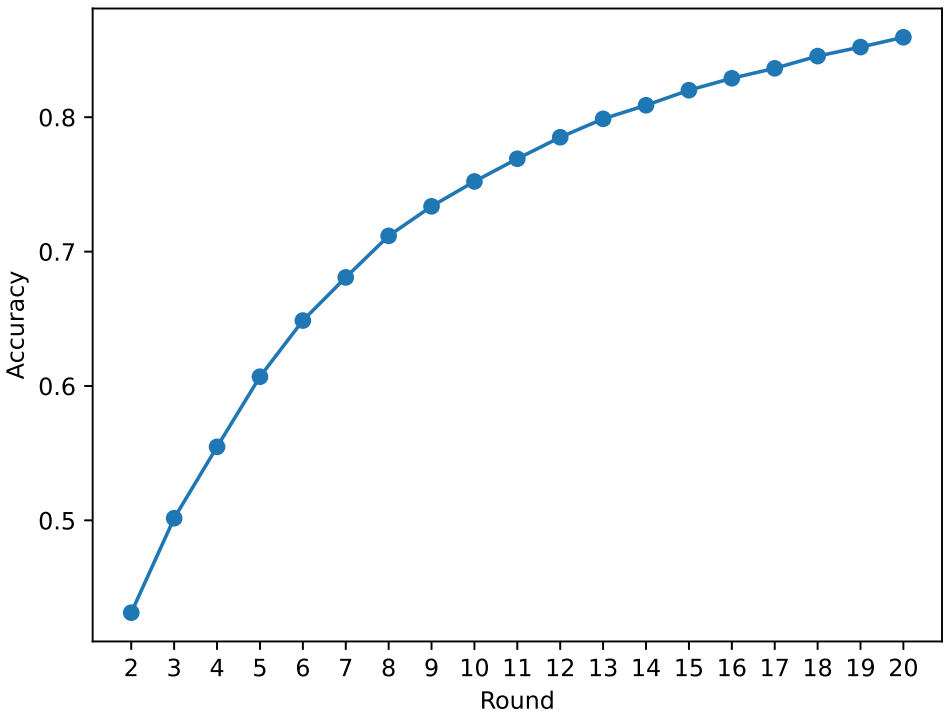


Figure 6: Validation set accuracy

5.2 Emulated packet loss

In addition, the same experiment is repeated with 10% packet loss emulation enabled. As described in section 4.3.2, Figure 8a, 8b, 8c, 8d plots the network statistics. As expected, packet loss results in greater number of TCP retransmissions. Wireshark[23] can also be used to capture packet information going through the network interface. Figure 8e depicts a plot of TCP packets per second over the time of the experiment using a logarithmic scale. The brown lines represent all good TCP packets while the red bars represent TCP errors. Figure 7 also depicts the training time.

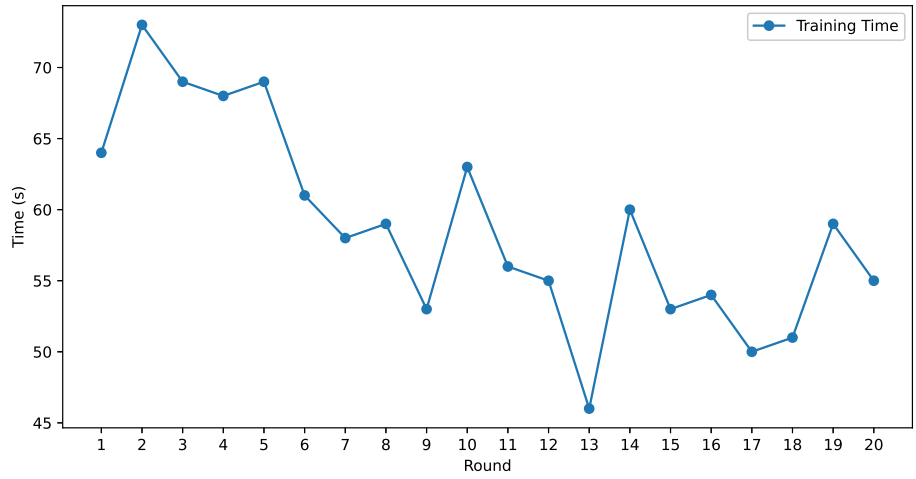


Figure 7: Average Training Time

6 Limitations and Future Work

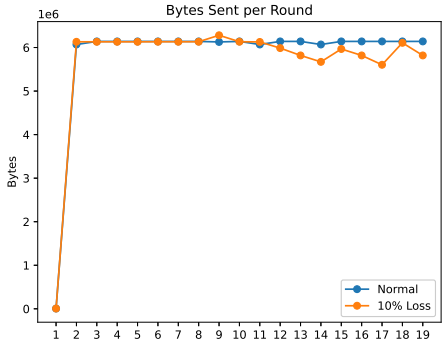
6.1 Long training times

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 and suitable for use in Zephyr OS could not be found. Traditional runtimes such as Tensorflow[24] rely on the availability of a Python interpreter on the device. Furthermore, these runtimes are usually not developed or optimized for resource constrained environments, incurring a large binary size and high memory usage. However, there is progress towards integrating Tensorflow Lite Micro into Zephyr OS through external modules and has examples for running pre-trained neural network on their platform. However, on-device training is still not supported.

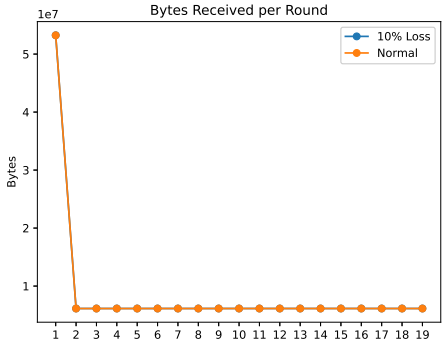
6.2 Algorithm support

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[25].

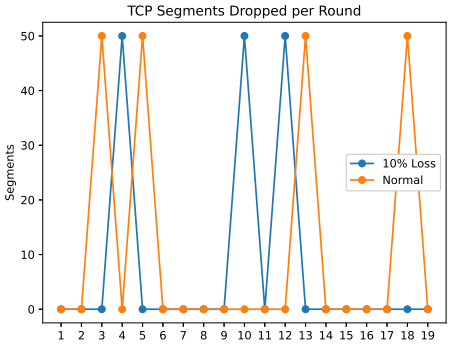
Furthermore, future work could also look into implementing other FL algorithms



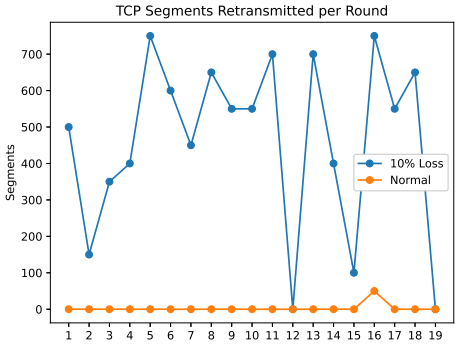
(a) Total Bytes Sent



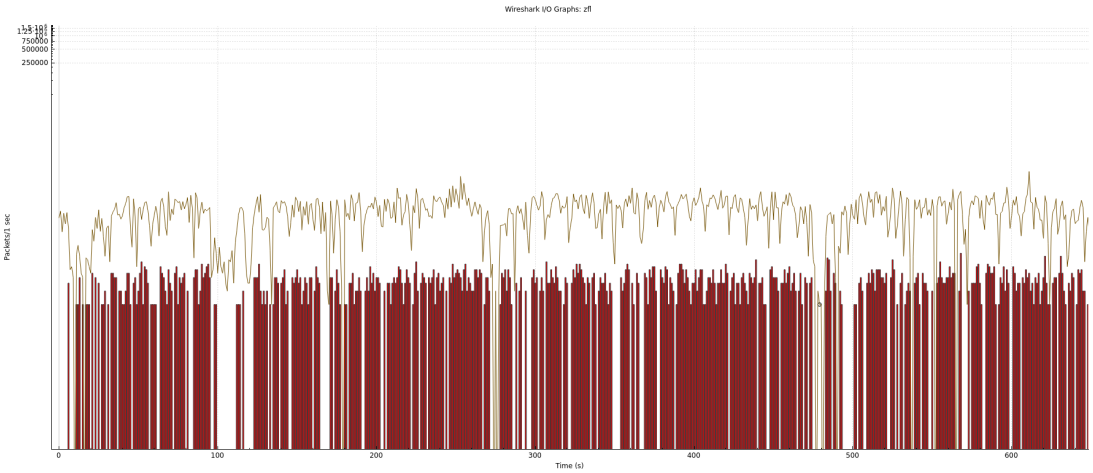
(b) Total Bytes Received



(c) TCP Segments Dropped



(d) TCP Segments Retransmitted



(e) Wireshark

in a bid to improve the usability of the framework. These algorithms should aim to achieve higher model accuracy without compromising on the requirements needed to function in an embedded low-powered device environment. One example could be Distillation Based Federated Learning[5].

6.3 Cross-Platform support

The implementation of zfl is dependent on a few POSIX standard APIs such as `pthreads` which makes it incompatible for running on Windows based machines. Future work could implement abstractions to call different platform APIs depending on compile time definitions to overcome this barrier.

7 Conclusion

In this paper, we describe the design and implementation of a hardware emulated FL framework zfl which manages to perform machine learning on the Zephyr operating system supported on a range of embedded platforms and architectures. The framework also collects valuable metrics such as network statistics including variables typically difficult to simulate such as TCP packet drops. We also demonstrate how this architecture allows emulating other factors such as network packet loss easily with existing tools.

References

[1] F. Hartmann, "Predicting text selections with federated learning," Google Research, 2021. [Online]. Available: https://blog.research.google/2021/11/predicting-text-selections-with.html

[2] H. Wang, Z. Kaplan, D. Niu, and B. Li, "Optimizing federated learning on non-iid data with reinforcement learning," *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pp. 1698–1707, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220903131

[3] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2017.

[4] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

[5] T. Liu, Z. Ling, J. Xia, X. Fu, S. Yu, and M. Chen, "Efficient federated learning for aiot applications using knowledge distillation," 2022.

[6] "Tensorflow federated: Machine learning on decentralized data," TensorFlow, 12. [Online]. Available: https://www.tensorflow.org/federated

[7] L. Li, J. Wang, and C. Xu, "Flsim: An extensible and reusable simulation framework for federated learning," pp. 350–369, 01 2021.

[8] "Json," ECMA, 12 2017. [Online]. Available: https://ecma-international.org/publications-and-standards/standards/ecma-404/

[9] M. H. Garcia, A. Manoel, D. M. Diaz, F. Mireshghallah, 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

[10] 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.

[11] 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.

[12] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, H. L. Kwing, T. Parcollet, P. P. d. Gusmão, and N. D. Lane, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[13] "Deploy machine learning models on mobile and edge devices," TensorFlow, 12. [Online]. Available: https://www.tensorflow.org/lite

[14] M. Ekmefjord, A. Ait-Mlouk, S. Alawadi, M. Åkesson, D. Stoyanova, O. Spjuth, S. Toor, and A. Hellander, "Scalable federated machine learning with fedn," *arXiv preprint arXiv:2103.00148*, 2021.

[15] "Nvidia federated learning application runtime environment," NVIDIA. [Online]. Available: https://github.com/NVIDIA/NVFlare

[16] M. Shafiq, "Building an aiot system emulator with zephyr and qemu," 01 2023.

[17] T. Q. P. Developers, "About qemu — qemu documentation," www.qemu.org, 2023. [Online]. Available: https://www.qemu.org

[18] "Introduction — zephyr project documentation," Zephyrproject.org, 2015. [Online]. Available: https://docs.zephyrproject.org/latest/introduction/index.html

[19] 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.

[20] 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.

[21] Tsoding, "tsoding/nn.h," GitHub, 01 2024. [Online]. Available: https://github.com/tsoding/nn.h

[22] "Traffic-control-howto," The Linux Documentation Project, 01 2024. [Online]. Available: https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

[23] "Wireshark," Wireshark Foundation, 01 2024. [Online]. Available: https://www.wireshark.org/

[24] 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/

[25] 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.