The simulator is implemented completely from scratch in Python 3.10.14 and is designed to replicate the behavior of a decentralized federated learning system based on a user-defined configuration file. This file allows the user to customize a wide range of parameters, enabling the simulation of diverse decentralized federated learning systems.
The simulator operates by creating a group of threads, each representing a node in the system. These threads are managed by a group of processes to enhance the parallelism inside the simulator. Each thread functions independently and communicates with others via socket messages to collaboratively execute the learning process.

JSON Configuration file

The user defines the simulation parameters through a JSON configuration file, which allows for extensive customization. The available options were inspired by state-of-the-art implementations and include the most commonly adopted configurations.

The most important configuration parameters are as follows:

- **Total number of rounds**
  Specifies the total number of simulation rounds. The simulation stops upon reaching that limit.
- **Model architecture**
  Specifies the file path containing the architecture of the global model to be trained. All trainers use this architecture for loading the global model's weights and performing training. The architecture remains unchanged during the simulation.
- **Starting model weights**
  Specifies the file path for the initial weights of the global model. These weights evolve throughout the simulation as the global model improves.
- **Node parameters**
  This sub-dictionary within the JSON configuration file defines some settings valid for all system nodes.
  - **Number of nodes**
    Specifies the total number of nodes in the simulated system.
  - **First node ID**
    Specifies the starting numeric ID for the first node. Subsequent nodes are assigned consecutive IDs. These IDs are used as nodes identifiers.
  - **First node port**
    Specifies the computer's port number for the first node. Subsequent nodes use consecutive port numbers. These ports are used to instantiate a socket server for each node.
- **Consensus algorithm parameters**
  This sub-dictionary defines the consensus algorithm and its parameters. The configuration varies depending on the selected algorithm.
  - For the **Proof-of-Work (PoW)** consensus algorithm:

    * **Type**
      Specifies the algorithm type, e.g., `pow`.
    * **Nodes computational power**
      A JSON list specifying each node's computational power by its ID. The various computational powers are used during each round to elect the winning miner responsible for generating the next block. Moreover, the computational power of each node does not change during the simulation.

  – For the **Proof-of-Stake (PoS)** consensus algorithm:
    * **Type**
      Specifies the algorithm type, e.g., `pos`.
    * **Number of validators**
      Specifies the size of the validator group for each round.
    * **Initial validator IDs**
      Specifies the IDs of nodes acting as validators in the first round. Validators for subsequent rounds are chosen randomly based on stake.
    * **Validation threshold**
      Specifies the minimum number of positive votes required to accept a model update. Each validator decides the vote to give to a model update based on the validation mechanism selected by the user.
    * **Stake reward amount**
      Specifies the stake amount rewarded at the end of each round to all the validators and all the trainers that submitted accepted model updates.

  – For the **Committee-based consensus algorithm**:
    * **Type**
      Specifies the algorithm type, e.g., `committee`.
    * **Number of validators**
      Specifies the size of the committee for each round.
    * **Initial validator IDs**
      Specifies the IDs of the initial committee. In subsequent rounds, validators are chosen based on the best updates submitted by trainers in the previous round.
    * **Validation threshold**
      Specifies the minimum number of positive votes required to accept a model update. Each validator decides the vote to give to a model update based on the validation mechanism selected by the user.

- **Validation algorithm parameters**
  This sub-dictionary defines the validation algorithm and its parameters. The configuration varies depending on the selected algorithm.

- **Pass-weights validation mechanism**
  Pass-weights means that every model update is automatically accepted without undergoing proper validation. It implies that the model updates must necessarily be in the form of weights. In fact, the exact form of the model updates cannot be directly specified; it depends solely on the validation and aggregation mechanisms used. The associated parameters are the following ones:
  - **Type**
    Specifies the validation mechanism, e.g., `pass-weights`.
  - **Minimum number of updates to start aggregation**
    Specifies the minimum number of accepted model updates required to initiate the countdown for triggering the aggregation mechanism.
  - **Maximum number of updates to start aggregation**
    Specifies the maximum number of accepted model updates required to terminate the countdown and force the aggregation.
  - **The number of seconds of the countdown**
    Specifies the countdown duration in seconds. The countdown is necessary to prevent the system from becoming stuck in a round while waiting for model updates that may never arrive, either because the maximum configured number of accepted updates is set too high or because many model updates have been discarded during the ongoing round.
- **Pass-gradients validation mechanism**
  Pass-gradients means that every model update is automatically accepted without undergoing proper validation. It implies that the model updates must necessarily be in the form of gradients. The associated parameters are the following ones:
  - **Type**
    Specifies the validation mechanism, e.g., `pass-gradients`.
  - **Minimum number of updates to start aggregation**
    Specifies the minimum number of accepted updates required to initiate the countdown for triggering the aggregation mechanism.
  - **Maximum number of updates to start aggregation**
    Specifies the maximum number of accepted updates required to terminate the countdown and force the aggregation.
  - **The number of seconds of the countdown**
    Specifies the countdown duration in seconds.
- **Global dataset validation mechanism**
  Global dataset validation means that every model update must be validated by each validator to determine whether it should be accepted. To validate a model update, each validator computes its accuracy on the global dataset and checks if the accuracy is below a certain threshold. If the accuracy is below the threshold, the validator broadcasts a negative vote, otherwise a positive

one. After each round, the threshold is deterministically adjusted based on the accuracies of the accepted model updates. In general, the threshold value is set to the accuracy of the model update located at the position representing the 75th percentile (three-quarters) of the list of accepted model updates, sorted in descending order by validation accuracy score. If the number of accepted model updates is too small, the threshold needs to be slightly relaxed. In such cases, the threshold value is set to the accuracy of the accepted model update with the lowest accuracy score, minus an additional 0.05. This mechanism implies that the model updates must necessarily be in the form of weights.

The associated parameters are the following ones:

* **Type**
  Specifies the validation mechanism, e.g., `global-dataset`.
* **Minimum number of updates to start aggregation**
  Specifies the minimum number of accepted updates required to initiate the countdown for triggering the aggregation mechanism.
* **Maximum number of updates to start aggregation**
  Specifies the maximum number of accepted model updates needed to stop the countdown and force the aggregation.
* **The number of seconds of the countdown**
  Specifies the countdown duration in seconds.
* **Starting value of the threshold**
  Specifies the starting threshold value used in the first round. The default value is 0.1. Subsequent thresholds adjust dynamically.

– **Local dataset validation mechanism**
Local dataset validation means that every model update must be validated by each validator to determine whether it should be accepted. To validate a model update, each validator computes its accuracy on his local dataset and checks if the accuracy is below a certain threshold. If the accuracy is below the threshold, the validator broadcasts a negative vote, otherwise a positive one. After each round, the threshold is deterministically adjusted based on the accuracies of the accepted model updates. The method for adjusting the threshold is the same as that described for the global dataset validation mechanism. This mechanism implies that the model updates must necessarily be in the form of weights.

The associated parameters are the following ones:

* **Type**
  Specifies the validation mechanism, e.g., `local-dataset`.
* **Minimum number of updates to start aggregation**
  Specifies the minimum number of accepted updates required to initiate the countdown for triggering the aggregation mechanism.
* **Maximum number of updates to start aggregation**

Specifies the maximum number of accepted model updates needed to stop the countdown and force the aggregation.

  * **The number of seconds of the countdown**
    Specifies the countdown duration in seconds.

  * **Starting value of the threshold**
    Specifies the starting threshold value used in the first round. Subsequent thresholds adjust dynamically.

- **Multi-Krum validation mechanism**
  In this scenario, Multi-Krum is not applied to each model update individually. Instead, each validator collects a group of model updates and applies the validation mechanism to that group collectively, deciding which ones to reject. The core idea behind Multi-Krum is to calculate the distance between each pair of model updates and, based on those measurements, discard the updates that differ the most from the others. The distance is computed using a distance function, such as the Euclidean distance. It implies that the model updates must necessarily be in the form of gradients.

  The associated parameters are the following ones:

  * **Type**
    Specifies the validation mechanism, e.g., `multi-krum`.

  * **Minimum number of updates to start validation**
    Specifies the minimum number of accepted updates required to initiate the countdown for triggering the validation and the aggregation mechanisms.

  * **Maximum number of updates to start validation**
    Specifies the maximum number of model updates needed to stop the countdown and force the validation and aggregation.

  * **The number of seconds of the countdown**
    Specifies the countdown duration in seconds.

  * **Number of model updates to discard**
    Specifies the number of updates discarded during validation.

  * **Distance function**
    Specifies the distance function used by the validation mechanism. The simulator can handle the following ones: euclidean, cityblock, chebyshev, mahalanobis, braycurtis, canberra and cosine.

- **Aggregation algorithm parameters**
  This sub-dictionary defines the aggregation algorithm.

  - **Federated Averaging (FedAvg)**
    Requires the model updates in the form of weights. The form of updates is determined by the selected validation and aggregation mechanisms.

    The associated parameter is the following one:

    * **Type**
      Specifies the aggregation mechanism, e.g., `fedavg`.

  – **Mean aggregation mechanism**
    Supports model updates in both gradient and weight forms. Consequently, in this case, the form of the model updates depends only on the selected validation mechanism.

    The associated parameter is the following one:

    * **Type**
      Specifies the aggregation mechanism, e.g., `mean`.

  – **Median aggregation mechanism**
    Supports model updates in both gradient and weight forms.

    The associated parameter is the following one:

    * **Type**
      Specifies the aggregation mechanism, e.g., `median`.

- **Malicious nodes parameters**
  This sub-dictionary specifies the nodes that should act maliciously and defines their behaviors. The configuration varies by attack type.

  – **Label flipping**
    A malicious node performing the label flipping attack manipulates his local dataset by replacing all the labels of one or more targeted label classes with labels from the corresponding wrong label classes. These nodes then perform honest training on the manipulated datasets.

    The associated parameters are:

    * **Type**
      Specifies the malicious behavior type, e.g., `label-flipping`.

    * **Malicious Node IDs**
      Specifies the IDs of nodes exhibiting this behavior.

    * **Selected label classes**
      Specifies the label classes to be overwritten with incorrect labels.

    * **Replacing label classes**
      Specifies the incorrect labels used to overwrite the targeted ones.

    * **Starting round for the malicious behavior**
      Specifies the round from which the nodes begin behaving maliciously.

  – **Targeted data poisoning**
    A malicious node performing the targeted data poisoning attack manipulates his local dataset by adding a blue square in the center of the images of a specific label class. These nodes then train honestly on the manipulated data.

    The associated parameters are:

    * **Type**
      Specifies the malicious behavior type, e.g., `targeted-poisoning`.
    * **Malicious node IDs**
      Specifies the IDs of nodes exhibiting this behavior.
    * **Target label class**
      Specifies the label class of images to be manipulated.
    * **Square size**
      Specifies the size of the blue square added to images.
    * **Starting round for the malicious behavior**
      Specifies the round from which the nodes begin behaving maliciously.
  - **Additive noise attack**
    A malicious node performing the additive noise attack manipulates his honestly computed model updates by adding some noise. The noise magnitude is defined by a sigma value. To execute the additive noise attack, the malicious node begins by performing an honest training session. Next, it determines the minimum and maximum values of the resulting weights or gradients and multiplies them by the sigma parameter to calculate the range of noise values. Finally, for each weight or gradient, a random noise value is selected from this range and added to the original value.

    The associated parameters are:

    * **Type**
      Specifies the malicious behavior type, e.g., `additive-noise`.
    * **Malicious Node IDs**
      Specifies the IDs of nodes exhibiting this behavior.
    * **Sigma**
      Specifies the magnitude of noise added to model updates.
    * **Starting round for the malicious behavior**
      Specifies the round from which the nodes begin behaving maliciously.

- **Dataset parameters**
  This sub-dictionary specifies the dataset used and how it is partitioned among nodes.
  - **Dataset**
    Specifies the dataset used during the simulation. The simulator is able to handle the following datasets: MNIST [4] and CIFAR-10 [3].
  - **Number of quanta**
    Specifies the number of partitions in which the dataset must be partitioned.
  - **Percentage of IID quanta**
    Specifies the percentage of IID partitions in which the dataset must be partitioned. The remaining partitions are N-IID.

  - **Nodes composite datasets**
    Specifies the number of IID and N-IID partitions assigned to each node. Consequently, the specific partitions are selected randomly.

- **Archive parameters**
  This sub-dictionary defines the settings valid for the archive node. The archive node is a special node responsible for simulating the blockchain and and managing the storage of global model weights, thereby reducing the simulator's overall workload in modeling the entire system. Whenever a node needs to interact with the blockchain or retrieve the weights of global models, it must communicate directly with the archive node through socket messages.

  - **Archive port**
    Specifies the computer's port number for the archive node. This port is used to instantiate its socket server.

Dataset Management

The management of datasets in the simulator is a critical aspect that ensures each node is provided with data that is both representative of the global dataset and tailored to simulate real world conditions. In this simulator, the dataset management process begins with the acquisition of a global dataset, sourced from the Hugging Face repository [5]. This dataset is partitioned into two distinct types: independently and identically distributed (IID) data and non-IID (NIID) data using a similar approach to what is done in the Flower framework [2]. These partitions represent the varying data distributions that nodes might experience in a federated learning scenario. To simulate an environment where nodes may have heterogeneous data distributions, the global dataset is first divided into IID or NIID quanta. The IID partitions ensure that each node receives a balanced and representative portion of the global data, promoting fair and consistent training. On the other hand, the NIID partitions introduce data heterogeneity, mimicking real world situations where data may be biased or imbalanced. This partitioning method allows the simulator to reflect the complexities of federated learning, where nodes might have unequal access to data or where data from different sources may not align perfectly with the global distribution. Once the dataset is partitioned, the next step involves distributing multiple IID or NIID quanta to each node which then aggregates them to create his local dataset. This distribution is designed to ensure that each node receives a unique combination of data, thereby preventing identical datasets across nodes. By doing so, the system encourages diverse learning experiences and avoids the risk of overfitting, as each node is exposed to different subsets of the dataset with varying levels of data quality and distribution. In this way it is possible to simulate a more realistic federated learning environment where nodes have access to both high-quality and degraded data. The inclusion of NIID quanta introduces variability in the local datasets, allowing the simulation to explore how data heterogeneity impacts model training and performance.

The way the simulator combines IID and N-IID quanta to create the datasets for the various nodes is defined by the user in the JSON configuration file. This dataset management framework thus facilitates a comprehensive analysis of how decentralized federated

learning performs under varying conditions of data quality and availability.

Archive node

Every simulation includes a special node called the archive, regardless of the number of nodes specified by the user. The archive is executed by an additional thread among the ones used to managed the nodes of the system. The archive is responsible for managing the blockchain. This approach simplifies the system by reducing the complexity and effort required to maintain the blockchain across multiple nodes. By having a single archive node to manage a centralized instance of the blockchain, we eliminate the need for nodes to synchronize their local instances or resolve potential forks that could arise during updates. This simplification is justified, as the goal of this thesis is not to study the impact of forks on decentralized federated learning systems. Moreover, the archive is responsible for managing the storage of global model weights. Instead of storing the weights directly within the blockchain blocks, they are persistently saved in separate files. This approach reduces the size of the blocks, making them more lightweight. Each block contains a reference to the corresponding global model weights, allowing nodes to download the weights from the archive using that reference.
The archive node does not participate in the system's learning process. Instead, its primary function is to run a socket server to handle requests from other nodes via socket messages. These messages allow nodes to read or append blocks to the blockchain managed by the archive and to download or upload weights of global models. For example, at the end of each round, the aggregator node uploads the weights of the new global model and the related block to the archive. The aggregator then broadcasts only the hash of the new block to the other nodes, allowing them to retrieve the full block directly from the archive. Using the reference included in the block, the nodes can subsequently download the weights of the new global model from the archive.

Blockchain structure

At the start of the simulation, the simulator creates the archive and initializes the blockchain with a genesis block. This block is then downloaded by all nodes to ensure they are aware of the system's configuration.
The genesis block includes the following information:

- Global model architecture
- Reference to the initial global model weights
- Total number of rounds
- Parameters of the validation mechanism defined by the user
- Parameters of the aggregation mechanism defined by the user
- Parameters of the consensus algorithm defined by the user

Subsequent blocks in the blockchain, referred to as model blocks, contain the new global model computed at the end of each system round. The information included in each

model block depends partially on the consensus algorithm used.
Each model block contains the following information:

- Timestamp
- Hash of the preceding block
- Reference to the weights of the new global model
- Current round number
- Node IDs of the trainers who submitted accepted model updates
- Node IDs of the current validators or the ID of the winning miner
- Current node stakes (only for Proof-of-Stake)

Operations of a single round

Each simulation round follows a fixed series of steps, with specific details determined by the user-defined configuration. While parameters such as the dataset, validation mechanism, aggregation mechanism, global model architecture and malicious behavior of nodes do not alter the steps of a system round, the consensus algorithm significantly impacts the process. This is because the consensus algorithm determines the number of validators, their election, and their interactions for accepting model updates. In contrast, the aggregator is always elected randomly among the validators and operates independently to create the new model block by applying the aggregation mechanism.
We assume validators and aggregators behave honestly, as the simulator's primary purpose is to analyze the impact of malicious trainers in decentralized federated learning systems. Consequently, the simulator does not allow users to define malicious behavior for nodes acting as validators or aggregators.
The simulator implements aggregation mechanisms, validation mechanisms and malicious trainer behaviors based on literature. However, simplified versions of consensus algorithms are used to reduce computational overhead. For example, in Proof-of-Work (PoW), miners are elected randomly based on computational power, avoiding complex competition mechanics.
The first simulation round differs slightly from the others as nodes obtain initial information partially from the genesis block downloaded from the archive and partially from the parameters used by the simulator during the startup process. Instead, subsequent rounds rely on the latest model block.

The steps of a generic round are as follows:

1. **Nodes download the latest model block**
   At the start of each round, all nodes download the latest model block from the archive and use the reference included inside it to download the global model's weights for their next training session. In Proof-of-Stake (PoS), nodes also retrieve the updated node stakes from the model block.

2. **Election of validators and aggregator**

Based on the consensus algorithm and the latest model block, nodes determine who is responsible for the validation and aggregation:

- **Proof-of-Work (PoW):** Nodes randomly select a winning miner based on computational power. Nodes with higher computational power have a greater likelihood of selection. The winning miner validates and aggregates model updates. Moreover, the winning miner is not properly a validator and so he can still participate to the round as trainer. Additionally, due to the fact that there is not any competition between multiple different miners as the winning miner is elected randomly, the winning miner does not need to find any nonce to create the new model block. All the nodes are able to obtain the same random result because before performing that operation they configure the same seed. The seed changes deterministically after each round.
- **Proof-of-Stake (PoS):** Validators are randomly selected based on stake, with higher stakes increasing the likelihood of selection. The aggregator is chosen randomly from the validators. All nodes use the same seed to ensure consistent results. The seed changes deterministically after each round.
- **Committee:** Validators are deterministically selected as the trainers who submitted the highest-scoring model updates in the previous round. Scores are determined by validators in the preceding round.

3. **Trainers create model updates**
   The Python library used by us to develop the operations needed to let the nodes create and train models is Tensorflow 2.15.0 [1]. Trainers use the global model weights from the latest model block to perform training sessions in parallel. When one trainer finishes his training session, sends his model update to validators the or winning miner via socket messages without waiting for the others. Depending on the model updates form required by the system, the training session can be done in two different ways:
   - **Gradient-based model updates:** Trainers train their local models on a single batch of 32 samples from their local dataset. Gradients representing the weight changes are used to create model updates.
   - **Weight-based model updates:** Trainers train their local models on their entire local dataset for two epochs. The new weights are used to create model updates.

   Additionally, malicious trainers modify their training sessions according to the user-defined malicious behavior.

4. **Validators validate model updates**
   Validators validate model updates sent by trainers up to user-defined limits. When the minimum number of accepted updates is reached, a countdown starts. Upon countdown expiration or reaching the maximum number of updates, validation stops, and the aggregator begins aggregation.
   Validation involves applying the user-specified validation mechanism, producing a list of results with votes (positive/negative) and scores:

- **Pass-gradients and pass-weights mechanisms:** Due to the fact that model updates are accepted by default, scores are always 1.
- **Local and global dataset mechanisms:** Scores are based on accuracy.
- **Multi-Krum mechanism:** Scores are inversely proportional to total distance. Higher distances result in lower scores.

Model updates are accepted if the required number of positive votes is reached. Unlike Committee and Proof-of-Stake, in Proof-of-Work validation is performed solely by the winning miner.

5. **Aggregator creates the new model block**
The aggregator (or winning miner in Proof-of-Work) collects all accepted model updates and applies the aggregation mechanism to generate the weights of the new global model. First, the new weights are uploaded to the archive, and subsequently, the new model block is created and uploaded to the archive. Finally, the aggregator broadcasts the block's hash to all nodes so that they can refer to it when downloading the model block from the archive. In Proof-of-Stake (PoS), before creating the new model block, the aggregator is also responsible for adding the user-defined reward to the stake of all validators and trainers who submitted accepted model updates.

Bibliography

[1] 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. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] 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.

[3] U. o. T. Department of Computer Science. The cifar-10 dataset. URL `https://www.cs.toronto.edu/%7Ekriz/cifar.html`.

[4] Y. LeCun. The mnist database of handwritten digits. URL `https://yann.lecun.com/exdb/mnist`.

[5] Q. Lhoest, A. V. del Moral, P. von Platen, T. Wolf, M. Šaško, Y. Jernite, A. Thakur, L. Tunstall, S. Patil, M. Drame, J. Chaumond, J. Plu, J. Davison, S. Brandeis, V. Sanh, T. L. Scao, K. C. Xu, N. Patry, S. Liu, A. McMillan-Major, P. Schmid, S. Gugger, N. Raw, S. Lesage, A. Lozhkov, M. Carrigan, T. Matussière, L. von Werra, L. Debut, S. Bekman, and C. Delangue. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184. Association for Computational Linguistics, nov 2021. doi: 10.5281/zenodo.4817768. URL `https://aclanthology.org/2021.emnlp-demo.21`. arXiv:2109.02846.