Secure P2P Federated Learning

Project Final Report

Leonardo Randacio

17 settembre 2025

Contents

List of Figures

Project Final Report

Leonardo Randacio

Abstract

This project is an implementation of the algorithm proposed in [Wink et al., 2021].The goal is to simulate peer-to-peer Federated Learning implementing an n-out-of-n secret sharing schema and Secure Average Computation algorithm to let the peers collaborate in the training of a Neural Network, without the risk of direct or indirect training data theft, allowing the presence of semi-honest peers. The project will enable to simulate scenarios with a varing number of peers.

Goal

The project implements the Secure Average Computation (SAC) algorithm proposed in [Wink et al., 2021] to implement a distributed system with the following characteristics:

- Peer to peer: no central server, all members of the system have the same role
- Federated Learning: the peers collaborate in the training of a common machine learning model for which every peer contributes with part of the training dataset
- Data privacy: every peer has knowledge only on it's own dataset and does not acquire knowledge of the other peers' datasets

The deliverable enables the user to easily execute a simulation with a given number of peers and to compare the accuracy of the peer trained model with a centralizied counterpart.

Requirements Analysis

During project analysis the following requirements have been identified.

1. **Business Requirements**
    1. The system should demonstrate a practical application of advanced distributed systems and privacy-preserving computation concepts.
    2. The system should enable the comparison of a novel federated learning algorithm (SAC) against a traditional centralized baseline to evaluate its efficacy and overhead.
2. **Domain Requirements**

1. The implementation of the SAC protocol must adhere to algorithmic specifications outlined in [Wink et al., 2021].
2. The system must operate in a true peer-to-peer (P2P) topology, with no central server managing the learning process or aggregating model parameters.
3. The system must guarantee data privacy: a peer must not be able to derive the raw data or the class distribution of any other peer from the messages it receives.

3. **Functional Requirements**

    1. **User Functional Requirements**

        1. Main Usage

            1. Users (the experimenter) must be able to run the system to execute the control scenario (centralized training) and review the results.
            2. Users must be able to run the system to execute the simulation (SAC with IID data distribution across peers) and review the results.
            3. Users must be able to define the number of peers for a given simulation through command-line arguments.

        2. Results & Analysis

            1. Users must be able to view the final accuracy of the trained model for each executed scenario.
            2. Users must be able to view the number of training rounds executed by the distributed simulation.

    2. **System Functional Requirements**

        1. Centralized Training (Control Scenario)

            1. The system must implement a standard centralized training algorithm on the entire dataset to produce a baseline model.

        2. P2P Network Management

            1. The system must instantiate a configurable number of peer processes.
            2. Peers must be able to exchange messages directly with all other peers.

        3. Federated Learning with SAC

            1. Each peer must train a local model on its own subset of the data.
            2. Each peer must execute the SAC protocol to securely mask its model parameters before broadcasting them.
            3. Each peer must update its local model with the securely averaged global parameters.
            4. This process must repeat until a convergence criterion is met.

        4. Data Handling

    1. The system must partition a provided dataset among the peers in an IID manner.

4. **Non-Functional Requirements**

    1. The results of the SAC scenarios must be verifiable and reproducible across multiple runs with the same configuration.
    2. The process of configuring and running the scenarios must be well-documented and require minimal manual setup.
    3. The code must be modular, separating concerns like P2P communication, the SAC protocol, ML training, and data loading, to facilitate understanding and usage.
    4. The system should be designed to complete a experiment with a typical configuration (e.g., 5 peers) in a reasonable time frame on a single development machine. Performance optimization for a real distributed setting is not a primary goal.

5. **Implementation Requirements**

    1. The project must include a comprehensive README.md file with instructions for installation, dependency management, and usage.

Technologies

To configure a reproducible development environment Nix will be used.

The various cases will be implemented as docker projects and a python script will be implemented to enable the user to easily define the scenario that should be executed using command line parameters.

To configure the docker containers Nix will be used.

The de facto standard language for machine learning projects is python, so this is the language that will be used.

For python project management Uv will be used.

Just is used to simplify running cli commands.

Design

Structure

A high level architecture is shown in Figure 1.

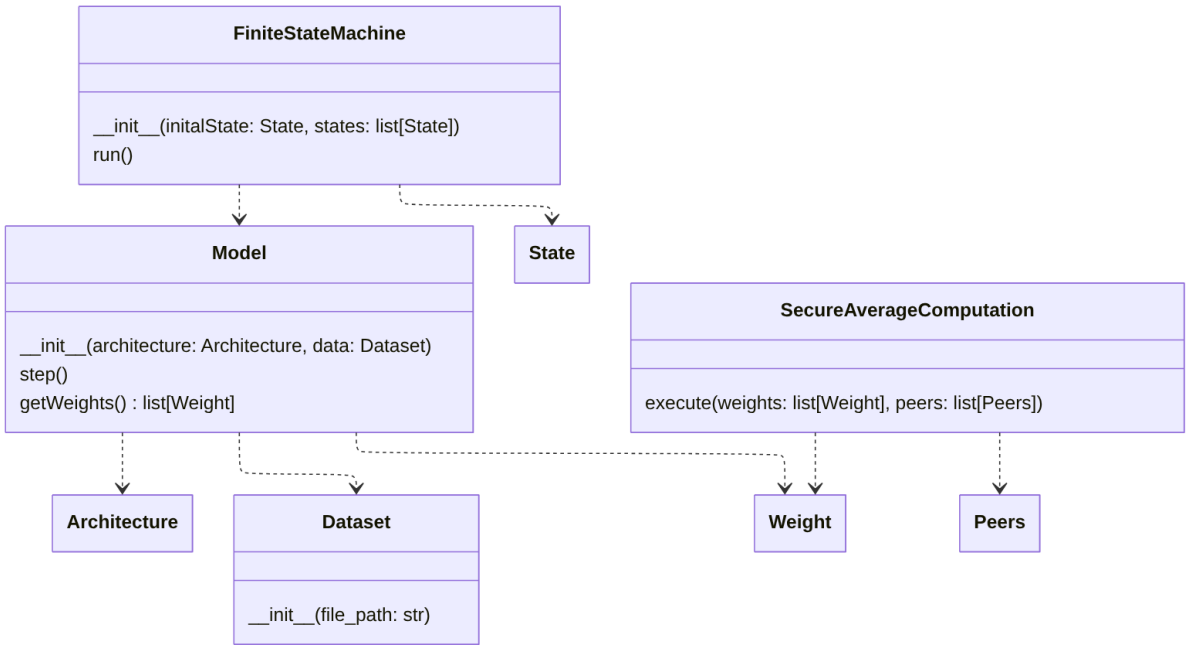

Figure 1: High Level Design

The full class diagram is shown in Figure 2, although some typings have been removed from longer method signatures for rappresentational purpose. To better navigate the image it is recommended to checkout the png on github directly.

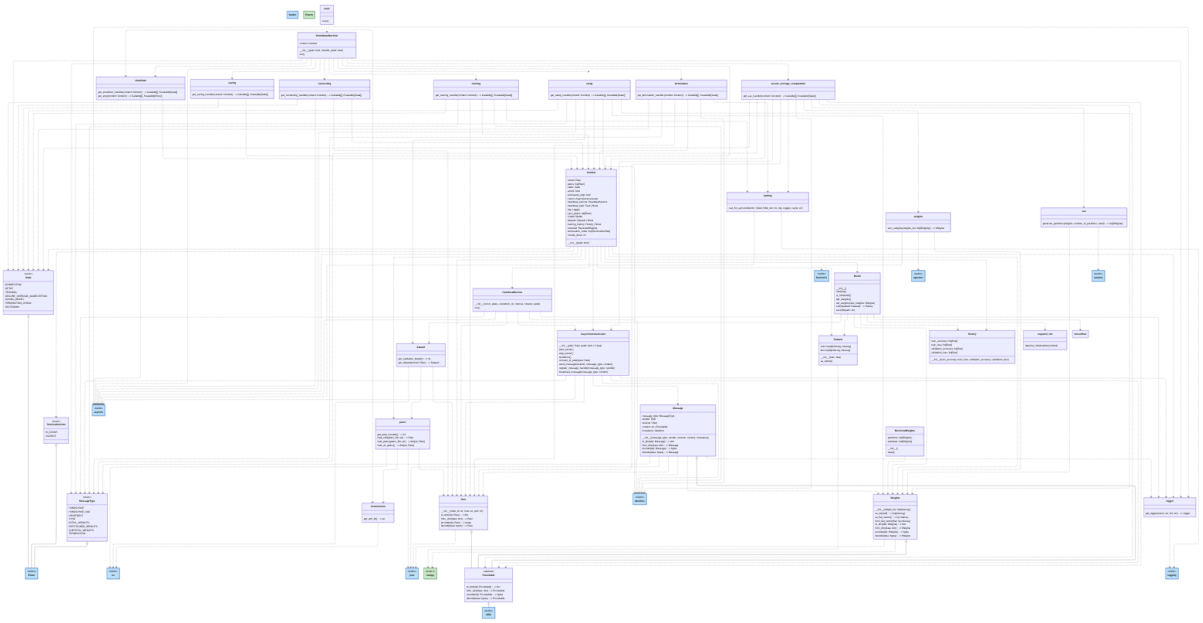

Figure 2: Full Class Diagram

Behaviour

The beheviour of the peers can be easily represented with the state diagram in Figure 3.

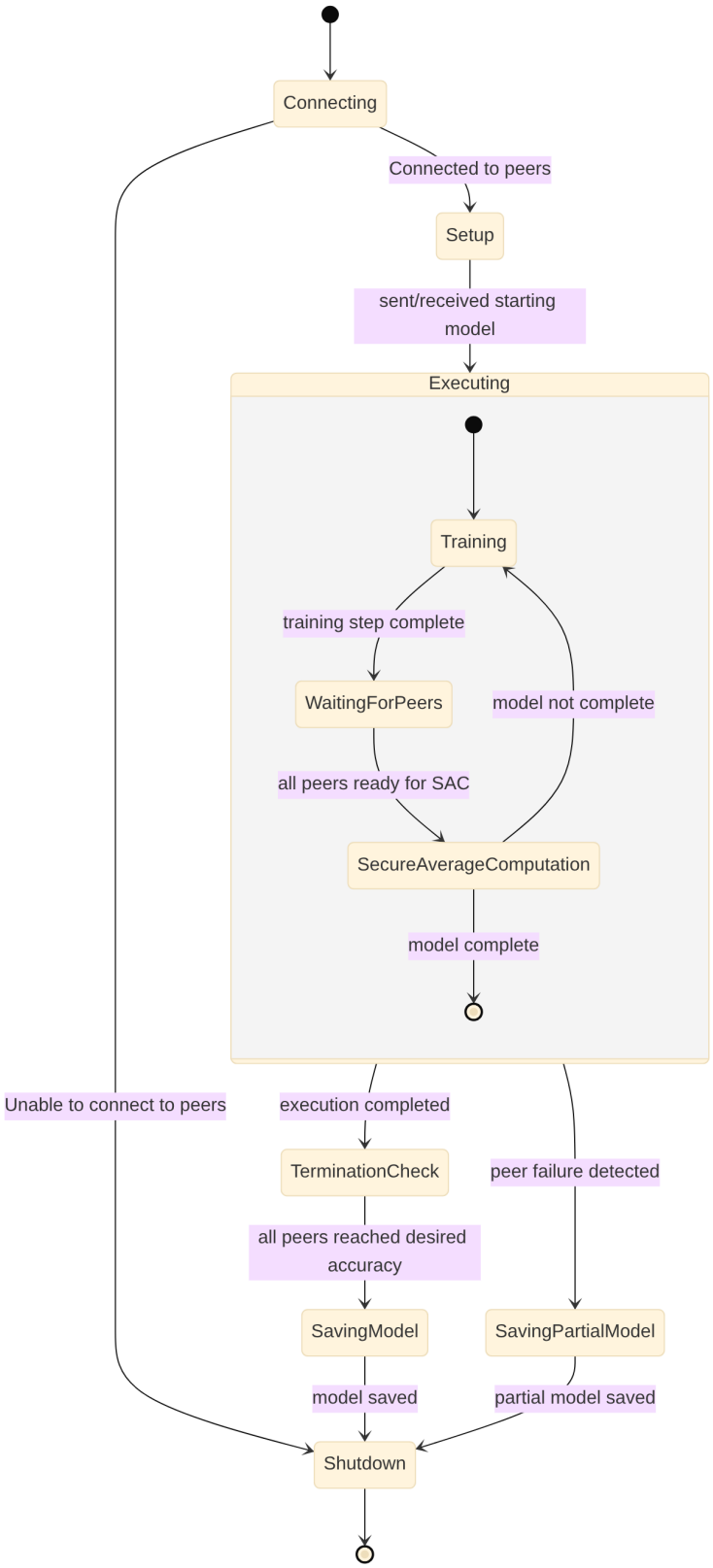

Figure 3: State Diagram

Interaction

The sequence diagram in Figure 4 shows the interactions that take place between peers in a 3 peer system example. The diagram omits the trivial initial connection step. In this example all nodes agree to terminate the algorithm with the current step, but if one node would have not agreed then all nodes would have started a new iteration. This is repeated untill all nodes agree that the algorithm can terminate.

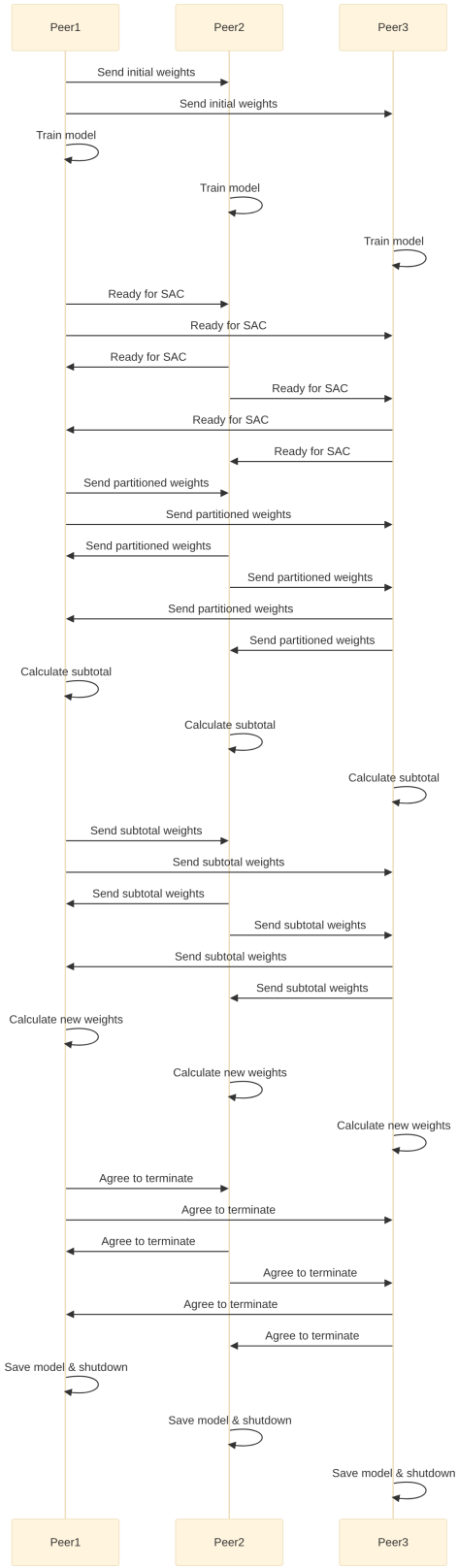

Figure 4: Sequence Diagram

All peers keep track of eachother's liveliness through heartbeat messages. Every peer broadcasts to all other peers a heartbeat message every given interval proving it is still running. If a peer does not receive a heartbeat message from another peer in a given timeframe than the other peer is considered dead. Because of the computationally intense operations involved in the algorithm and because the failure of a peer puts in jeopardy the entire algorithms execution, if this happens all other peers save their partially trained models and shutdown. Ideally the algorithm could be restarted, after ensuring the failing peer is back online, by using the partially trained models as a new starting point.

Implementation Details

Some interesting implementation details are reported in the following.

FiniteStateMachine

A simple `FiniteStateMachine` class is used to implement the beheviour showed in Figure 3.

The following simplifications have been made to the states for ease of implementation:

- the 'Executing' super-state was not implemented, using it's substates 'Training' and 'SecureAverageComputation' directly instead.
- the 'SavingModel' and 'SavingPartialModel' states where fused in a single 'SavingModel' state as the implementation of each has no significant difference.

Closures for fsm state handlers

Defining all of the finite state machine implementation in a single file would create a massive file, unideal for maintenence and debugging.

The execution logic behind every state is encapsulated in functions that return the new state the fsm would be in. Ideally these 'handlers' should be implemented in separate files. But handlers must access the context of the fsm so a classic function would not be sufficient.

The solution is to use higher order functions and closures. Handler defining modules provide a method that takes a context and returns a handler that has the context captured.

In simpler words each state handler is implemented as a closure: a function returned by a factory function that captures the FSM context. This lets handlers access state without explicitly passing context around in every call.

The following is a simplified example:

```python
from typing import Callable, Awaitable

from fsm.state import State
from fsm.context import Context

def get_handler(context: Context) -> Callable[[], Awaitable[State]]:
    async def handler() -> State:
        # execute logic
        return State.NEWSTATE
    return waiting_handler

# simplified fsm loop implementation
async def loop(self):
    while True:
        handler = self.handlers[self.context.state]
        self.context.state = await handler()
```

Another way to implement this is to modify the loop function of the fsm to always execute the state handler passing the context:

```python
from fsm.state import State
from fsm.context import Context

async def handler(context: Context) -> State:
    # execute logic
    return State.NEWSTATE

# simplified fsm loop implementation
async def loop(self):
    while True:
        handler = self.handlers[self.context.state]
        self.context.state = await handler(self.context)
```

The closure solution was choosen because deemed more elegant and readable.

This also works well when defining ‘message handlers’ alongside fsm state handlers in the same module, coupling code responsible for the same beheviour. A good example of this is located at `src/main/fsm/handler/setup.py`

Message type

The need for a ‘message type’ arose when designing the communication system. Messages must be typed so that different handlers can be defined for different message types, helping with separation of concerns, as modules are usually able to define handlers only for some, usually only one, message type.

Various solutions where considered for the implementation of the ‘message type’.

The easiest solution is to use a simple string. This was quickly discarded as using strings would be too error proned, as typos would be hard do debug.

A simple solution is to use an enumeration, as follows:

```
class MessageType(str, Enum):
    EXAMPLE_TYPE1 = "exampletype1"
    EXAMPLE_TYPE2 = "exampletype2"
    EXAMPLE_TYPE3 = "exampletype3"

# usage

message = Message(MessageType.EXAMPLE_TYPE1, contents)
```

Although this solution is typo resistant, it forces all types to be defined in a single file, centralizing the definition. Also any module that creates messages would have to import the entire enum, even though it would probably only a limited number of it's cases. Still, this solution has the advantage of a very easy and straightforward usage and serialization/deserialization (by inheriting from `str`).

Another solution considered was an abstract class 'MessageType' that would be extended by empty concrete classes, replacing the Enum members, defined by the modules that would use them. This solution provides good separation of concerns and follows the open/closed principle, but it loses straightforward usage and usage and serialization/deserialization.

Given the project's small scope and prototyping nature, the 'Enum' solution has been implemented, preferring simplicity over advanced software principles.

Model weights

Machine learning model weights are implemented as a wrapper class. Since weights implementation might vary greatly based on the machine learning library used, the usage of a wrapper class ensures that replacing the machine learning library should be an easy operation.

Having the weights wrapper class implemement a collection of single 'Weight' objects was considered, but due to the non trivial implementation of weights (e.g. in keras the weights are provided as a list of numpy arrays, two for each layer, separating weights and biases in different arrays) in machine learning libraries a full wrapper class ensures minimal code rewriting when changing ML library.

Self-assessment

Through the help of github actions simulation from 2 to 7 peers have been executed. All simulation reached the accuracy of the centralized model.

Clearly the 2 peer example has no practical meaning, as the partitioning doesn't actually take place, but it serves as baseline examples.

The number of training rounds necessary to reach the required accuracy is reported in Figure 5.

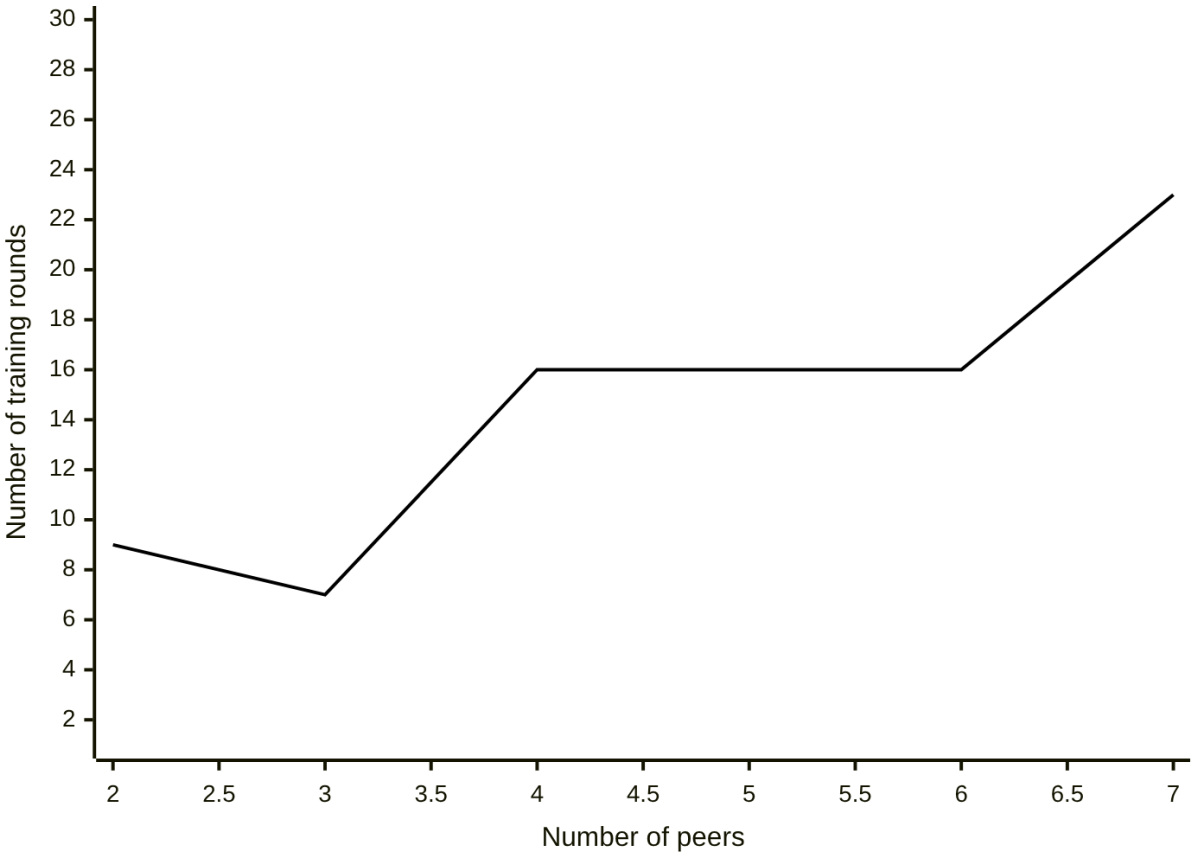

Figure 5: Number of training rounds per peer in the simulation

The following table summarises the results from every simulation

Number of peers	Number of rounds	Test accuracy	Execution time
2	9	0.88	4m 53s
3	7	0.88	5m 5s
4	16	0.88	7m 50s
5	16	0.88	8m 45s
6	16	0.88	9m 11s

Number of peers	Number of rounds	Test accuracy	Execution time
7	23	0.88	12m 23s

The following is a list of links to the full execution logs, through the github actions feature:

- 2 peers
- 3 peers
- 4 peers
- 5 peers
- 6 peers
- 7 peers

Deployment Instructions

The only project dependency necessary to run the simulations are Docker and Nix.

Unfortunately Docker's installation is not provided by Nix development environment functionalities, as it seems there is currently no good mechanism for configuring services on non-NixOS hosts. This means that docker must be installed 'manually'.

The easiest way to get Docker (and it's many products) installed is by installing Docker Desktop.

Usage Examples

For executing a simulation the only dependencies required to be installed are Nix and Docker.

The easiest way to get Docker (and it's many products) installed is by installing Docker Desktop.

To install all other dependencies (defined in a `flake.nix` file) run the following command:

```
1 nix develop
```

Just is used to simplify running cli commands.

To see available commands:

```
1 just
```

The following command starts a simulation:

```
1 just run <number_of_peers>
```

Where <number_of_peers> is the number of peers involved in the simulation.

Conclusion

Future works

Unit tests and CI Before any additonal work would be done on the project, a unit test and consequent CI infrastructure would have to be instantiated, as the project exceedes it's proof of concept goal on it's way to becoming a production ready artifact.

Python Package A non trivial resetructure of the high level implementations of the project could be done to turn the project into a python package. This would entail the use of a simple and understandable API to empower users to quickly setup peers and have them start the distributed algorithm.

Although a costly improvement it would make the adoption of the project much easier for the users.

Leader election In the first part of the algorithm execution the peers must elect a 'leader' that generates the model's starting weights and passes then to the other peers, to ensure all peers start the first training round using the same model.

In this project the peer named "Peer1" is always selected as this temprorary leader. The protocol could be improved by using an election mechanism to remove the necessity to select the peer to cover this role beforehand.

It could be argued that the ML model architecture is also something the peers should agree on through the algorithm, but immagining diverce companies collaborating through SAC some preliminary meetings would probably have the responsability of deciding the model's architecture, and possibly the startging model weights, so in a practical application these problems would not arise.

Still an algorithm for a leader election might improve the algorithms generality.

Improved heartbeat system The current implementation of the heartbeat pattern is very rudimentary.

A more advanced version could implement a gossip-based heartbeat to limit the number of messages sent through network and to improve the scaling capabilities of the algorithm.

Also the current pulse message contains a dummy string (`"I'm alive!"`). The message could be improved to containing usefull information, such as surrent finite state machine status or uptime.

Dinamic members and recovery The current implementation implies knowledge on the number of peers involved and their info (as in port and host) in advance. Also it does not implement any recovery

plans for temporary failed peers neither does it provide a way to add a peer to an already started run of the algorithm.

Distributed system design could be imployed to provide these desirable functionalities, allowing the project prototype to be closer to production ready code.

Improve communication protocol The current implementation of the low level communication protocol could be improved significantly.

Some possible improvements could be:

- Automaitcally attempt to reconnect on peer disconnection.
- Replace newline delimited json messages with length-prefixed framing, to enable the sending of arbitrarily large messages.
- Improve general error detection and recovery.

Simulate bad actors It could be interesting to see the effects that a malevolent actor would have on the convergence rate of the trained model.

A second parameter could be implemented for the simulation, identifing the number of peers that send random weights to try and slow down the training progress.

What I learned

This project was an very interesting endevour. Not finding a peer to peer python library that sadisfied my needs, without providing too many unhelpfull features, the decision to implement the comunication system directly with the simple `asyncio` python builtin library has been a very good didactical exercise. The project managed to stimulate me in applying the teoretical notions on distributed systems touched on during the course, often diving deeper than before, and providing practical insights on the discipline.

It has been interesting to use Docker in combination with Nix, providing full reporoducibility to the projects code artifacts, enabeling me to quickly setup the development environment for the project on any machine with no effort.

Often during development new functionalities came to mind, as is noticeble from the size of the Future works section. With more time available this project is defenetly one I will rivisit in the future.