ÉCOLE POLYTHECHNIQUE FÉDÉRALE DE LAUSANNE

UDP Communication Protocol for Decentralized Learning

Scalable Computing Systems Laboratory (SaCS)
Semester Project

Spring 2023

Professor: Dr. Anne-Marie Kermarrec

Project Supervisors: Rishi Sharma and Dr. Rafael Pires

Emna Fendri
School of Computer and Communication Sciences
Data Science

Contents

1 Introduction

Centralized machine learning applications require data to be stored and processed in a same central server, which can raise concerns about data privacy, security, and scalability. Decentralized learning, on the other hand, aims to distribute the learning process across multiple nodes. In fact, each node is responsible for learning a part of the model and for sharing it with its neighbors in order to aggregate and update each local model, which removes the need for a central server and allows training on larger datasets.
In this project, we will focus on the `DecentralizePy` framework written in Python3 used for running distributed Machine Learning applications. Initially, the nodes are using the Transmission Control Protocol (TCP) to communicate and send their models to their neighbors in a reliable way. In fact, in order to ensure a reliable delivery of data, TCP uses **acknowledgements** and **retransmission** mechanisms of lost or corrupted packets. However, this reliability comes at the cost of potential delays in case of packet losses. Our goal will be to study the performance of these machine learning applications when using the User Datagram Protocol (UDP) for communication in order to prioritize **low latency**.
The optimization of distributed learning systems involves finding a balance between computation and communication. This solution using UDP aims to address challenges in slower networks, more specifically, networks that present risks of packet loss - those can be networks with low bandwidth. Researches have been working on several solution to address these issues. We can mention for instance Communication Compression techniques that aim to reduce the size of the data being exchanged between decentralized learning nodes [3] . By compressing the data, the overall volume of data transmitted over the network can be reduced, which can be advantageous in scenarios with limited bandwidth.

In this report, we will discuss our implementation of the UDP module and evaluate its performance once integrated in the `DecentralizePy` framework. We will then compare it, under different settings, with the initial version that uses TCP for communication.
Mainly, we will try to answer the following questions: Is the presence of a retransmission mechanism essential for the machine learning model to learn effectively in the event of packet loss? More generally, in which cases would we use UDP over TCP ?

2 Background

2.1 Decentralized Machine Learning

Decentralized machine learning differs from centralized machine learning in two significant ways. Firstly, in Decentralized Learning each node will perform training on its own data partition. Therefore, the whole dataset does not need to be moved around. Secondly, in Decentralized Learning there is no need for a centralized coordinator like in a parameter-server architecture. Instead, the nodes communicate directly on their local data and work to optimize the machine learning model jointly by sharing their model with their neighbors.
In order to train the model in a decentralized setting, we will consider in this project the Decentralized Parallel Stochastic Gradient Descent (D-PSGD) algorithm . This algorithm is responsible for solving the optimization problem of finding minimum loss function by updating the parameters of the model. There are several variations that can be made to the D-PSGD. From a high level, we can say it consists of these two main steps:

For each iteration:

1. Each node computes the stochastic gradient using its local dataset and updates its local parameters.
2. Each node exchanges local parameters with its neighbors and average the local parameters it receives with its own local parameters.

We will get into the details of the implementation that we have used in this project in the following sections.

2.2 DecentralizePy

DecentralizePy [1] is a framework written in Python 3 used for running distributed Machine Learning applications and has been developed at EPFL's Scalable Computing Systems Laboratory (SaCS). It can run on several machines where each process within a machine represents a node in distributed machine learning. This framework consists of multiple modules, with each module being responsible for a specific part of the decentralized process. The file structure of the implementation in the `src` section is shown on the diagram below 1.
In the following, we will give an overview of the two modules that were used in the scope of this project, namely the `communication` and `sharing` classes and how it has been initially implemented. In the next section called **Implementation** we will explain what we have added to these modules.

2.2.1 Communication

The Communication class is an API for communication between processes in DecentralizePy, it exposes the communication functions to other parts of the framework. The interface consists of functions for establishing and closing communication links with neighbors, for sending and receiving messages, and for encrypting and decrypting data. Each instance of the `communication` class is initialized at the creation of each node with the specific information about that node such as its rank and machine ID.

TCP: The `TCP.py` file contains an implementation of the TCP communication protocol using the ZeroMQ messaging library [2]. The ZeroMQ library provides a high-level API for sending and receiving messages between applications or processes. This initial implementation uses a ZMQ ROUTER socket that we bind to each node, that will be responsible for receiving messages. It also uses a ZMQ DEALER socket per neighbor in order to connect to them and send messages. The main functions are the following:

- The `connect_neighbors` function that we find in the `Node` class establishes connections with all neighbors by initiating a connection with each one of them using the `init_connection` of the `TCP.py`. It then sends a Hello message. Similarly, the `disconnect_neighbors` method of the `Node` class terminates all connections with neighbors by calling `close_connection` of `TCP.py`.
- The `receive` method receives data from a connected neighbor, while the `send` method sends data to a specific neighbor.
- The `encrypt` method encodes data as a Python pickle object, while the `decrypt` method decodes the received pickle data.

The opening of a connection is confirmed by sending a Hello message. A node waits for Hello messages from all its neighbors before it finishes the connecting phase. It also waits for a Bye message from all neighbors when `disconnect_neighbors` is called.

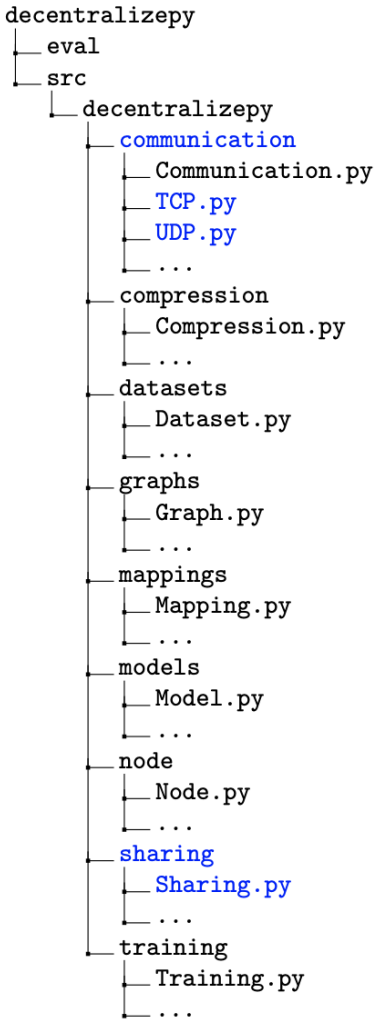


Figure 1: File structure of `DecentralizePy`

2.2.2 Sharing

The functions for serializing and deserializing the model and sharing model parameters with other nodes in the network are defined in the `sharing` class. The main functions are the following:

- `serialized_model()`: It serializes the model from a PyTorch state_dict to one single flattened vector.
- `get_data_to_send()`: It returns a Python dictionary containing all the data to be transmitted to the neighboring nodes.
- `_averaging()`: Averages the received model with the local model.
- `deserialized_model()`: Converts back the updated model to a PyTorch state_dict.

2.3 TCP vs UDP

In this project, the aim is to compare the performance of decentralized learning when nodes communicate over TCP and UDP protocols. In this subsection we will revisit the fundamental concepts of these two protocols and explain the main differences.

TCP TCP stands for "Transmission Control Protocol" and is a connection-oriented protocol that provides reliable and ordered delivery of data packets over a network. It establishes a connection between two devices before transferring data, ensuring that data is received accurately and in the correct order. TCP includes mechanisms for error detection and retransmission of lost packets if necessary. It guarantees data integrity but the main drawback is that it introduces higher latency due to these mechanisms.

UDP UDP stands for "User Datagram Protocol" and is a connectionless protocol that offers a simpler and lightweight alternative to TCP. It provides a best-effort delivery mechanism without the same reliability guarantees as TCP. UDP does not establish a connection before sending data and does not provide features like error recovery or retransmission. It is often used in scenarios where real-time communication, low latency, and minimal overhead are prioritized, such as streaming media or online gaming.

3 Implementation

In this section, we will focus on the additions we made to the initial `DecentralizePy` Framework to incorporate the UDP communication module.
We created a new module called `UDP.py` in the `communication` folder with all the required functions:

- `send()`
- `receive()`

Additionally, we add 4 new functions in the `Sharing.py` module from the `sharing` folder :

- `serialized_model_UDP()`
- `get_data_to_send_UDP()`
- `_averaging_UDP()`
- `deserialized_model_UDP()`

3.1 Decentralized Parallel Stochastic Gradient Descent (D-PSGD)

The `DPSGDNode` class inherits from the `Node` and is used for decentralized parallel stochastic gradient descent. The `run` method of the `DPSGDNode` object, as we will show below, will be launched by each node to start the decentralized learning .

3.1.1 D-PSGD for TCP

The `run` method with the TCP communication module is shown in Algorithm 1. Within each iteration step, each node will perform one training iteration using the given dataset by calling the `train` function from the `Training` class (line 3). More specifically, by training over at most `rounds` mini-batches and will update the local model's parameters. After successfully initializing a connection with all the neighboring nodes (line 4), the next step is to prepare the data that will be sent to them by calling `get_data_to_send` (line 5). We iterate over the neighboring nodes and send the data. Once the data sent, we wait until we receive all the packets from the neighboring nodes. This is done in the While loop at lines 8 and 9 shown in blue. Finally, we perform the averaging step by calling the `sharing`'s `_averaging` method and passing to it the `received_models` (`DPSGDNode` attribute) and the current iteration (line 10).

3.1.2 D-PSGD for UDP

The modifications we made to the `run` method with the UDP communication module resides in how the receiving step is done, as shown in blue:

- Algorithm 2: In the first UDP implementation, we introduce at line 2 one receiving thread for each node that will be responsible for listening to incoming packets during the whole training and storing them, as we will explain shortly. For this, we use the `threading` library from Python.
- Algorithm 3: In the second UDP implementation, the steps are done sequentially. Before starting the next iteration, each node will keep receiving packets until it receives at least one packet from each one of its neighbors or a timeout occurs. This is done in the While loop at lines 8 and 9 in blue. The experiments have shown that there are rare cases where, at some iteration, a node would not receive packets from one of its neighbors, which explains the timeout. We will discuss in the following sections the value we have chose to set for the timeout.

We will now discuss in more details the implementation of these steps in the upcoming subsections.

Algorithm 1 D-PSGD for TCP

```
1: function RUN
2:     for iteration ← 0 to iterations do
3:         train(dataset)                                          ▷ One training iteration
4:         connect_to_neighbors()
5:         to_send ← get_data_to_send(degree = len(my_neighbors))
6:         for each neighbor in my_neighbors do
7:             communication.send(neighbor, to_send)
8:         while not received_from_all() do
9:             communication.receive()
10:        sharing._averaging(received_models, iteration)
11:    disconnect_neighbors()
```

Algorithm 2 D-PSGD for UDP with Receiving Thread

```
1: function RUN
2:     receiving_thread ← THREADING.THREAD(target=communication.receive)
3:     receiving_thread.start()
4:     for iteration ← 0 to iterations do
5:         train(dataset)                                          ▷ One training iteration
6:         my_neighbors ← get_neighbors()
7:         to_send ← get_data_to_send_UDP(degree = len(my_neighbors))
8:         for each neighbor in my_neighbors do
9:             communication.send(neighbor, to_send)
10:        sharing._averaging_UDP(received_models, iteration)
11:    Close all sockets and threads
```

Algorithm 3 D-PSGD for UDP with synchronization

```
1: function RUN
2:     for iteration ← 0 to iterations do
3:         train(dataset)                                          ▷ One training iteration
4:         my_neighbors ← get_neighbors()
5:         to_send ← get_data_to_send_UDP(degree = len(my_neighbors))
6:         for each neighbor in my_neighbors do
7:             communication.send(neighbor, to_send)
8:         while not all_nodes_received AND not timeout do
9:             communication.receive(received_models)
10:        sharing._averaging_UDP(received_models, iteration)
11:    Close all sockets
```

3.2 Preparing the data to send

We introduce the `get_data_to_send_UDP` (Algorithm 4) and the `serialized_model_UDP` (Algorithm 5) methods in the `Sharing` class. In PyTorch, the `model.state_dict()` method returns a dictionary containing the parameters of the model. The keys of this dictionary are the names of the model's parameters, and the values are the corresponding parameter tensors.

The `serialized_model_UDP` function takes the model's parameters, flattens them into 1D tensors, concatenates them into a single 1D tensor, and returns a dictionary `data` with the serialized model parameters converted to a NumPy array under the "params" key.
In `get_data_to_send_UDP` we add 2 more keys to this dictionary, storing the degree of the node and the current iteration. In fact, we will need to know the degree of parameter sender as well as the current iteration in the receiving and averaging steps.

Algorithm 4 Get Data to Send

1: **function** GET_DATA_TO_SEND_UDP(*degree*)
2: $data \leftarrow$ SERIALIZED_MODEL_UDP ▷ returns a dictionary
3: $data$["degree"] $\leftarrow$ *degree*
4: $data$["iteration"] $\leftarrow$ *iteration*
5: **return** *data*

Algorithm 5 Serialize the model parameters

1: **function** SERIALIZED_MODEL_UDP
2: $to\_cat \leftarrow [\,]$
3: **for** $\_, v$ **in** model.state_dict().items() **do**
4: $t \leftarrow v$.flatten() ▷ Flatten the tensor
5: to_cat.append(t) ▷ Append the flattened tensor
6: $flat \leftarrow$ torch.cat(to_cat) ▷ Concatenate everything into a single 1D tensor
7: $data \leftarrow$ dict()
8: $data$["params"] $\leftarrow$ flat.numpy() ▷ Store the flattened tensor as a NumPy array
9: **return** *data*

3.3 The `send` function of the UDP Module

We introduce the `send` and the `send_chunks` methods (Algorithm 6) in the `UDP` class. We pass to `send_chunks` the `neighbor` which corresponds to its uid and the `data_to_send` which is the dictionary returned by the `get_data_to_send_UDP` function. It then retrieves the specific values from it (from line 5 to 7), namely the flattened parameter array containing the trainable parameters, the degree of the sender, and the iteration. Given a `CHUNK_SIZE`, we chunk the 1D array into `CHUNK_SIZE` arrays that we store in a list `chunks` (line 8).
We initialize a `UDP` instance by creating a socket object for UDP communication over IPv4 using the Python's standard library `socket`: `socket.socket(socket.AF_INET, socket.SOCK_DGRAM)` that we bind to each node enabling it to receive messages from other nodes.
The goal of this function is to send the parameter chunks to the neighbors so it can be able to reconstruct the whole model. For this, each chunk will be sent along with its index, the uid of the sender, the degree of the sender and the iteration. These information are stored in a dictionary `to_send` than we pickle (line 17) and send to the neighbor given its IP address and Port number (line 20).

Algorithm 6 Send Chunks

```
1: function SEND(neighbor, data_to_send)
2:     SEND_CHUNKS(neighbor, data_to_send)
3: function SEND_CHUNKS(neighbor, data_to_send)
4:     dest_host, dest_port ← Get the IP and Port of neighbor
5:     flattened ← data_to_send["params"]          ▷ The params in a 1D Numpy array
6:     degree ← data_to_send["degree"]
7:     iteration← data_to_send["iteration"]
8:     chunks ← SPLIT_INTO_CHUNKS(flattened,CHUNK_SIZE)          ▷ Chunking
9:     index ← 0
10:    for all c in chunks do
11:        to_send ← dict()
12:        to_send["index"] ← index
13:        to_send["param_chunk"] ← c                          ▷ params
14:        to_send["uid_sender"] ← uid                         ▷ Node uid
15:        to_send["degree"] ← degree
16:        to_send["iteration"] ← iteration
17:        to_send_bytes ← PICKLE.DUMPS(to_send)
18:        data_size ← LEN(to_send_bytes)
19:        self.total_bytes ← self.total_bytes + data_size     ▷ Update total_bytes
20:        SOCKET.SENDTO(to_send_bytes, (dest_host, dest_port))   ▷ Send to dest node
21:        index ← index + 1
```

3.4 The `receive` function of the UDP Module

We introduce the `receive` method (Algorithm 7) in the `UDP` class that will be responsible for capturing and storing different packets from different neighbors for each node in its attribute dictionary `received_models`. The `received_models` has been initialized for each `DPSGDNode` instance as a 3-level nested dictionary:

```
received_models = defaultdict(lambda:defaultdict(dict))
```

After unpickling the received data, the variables `iteration`, `param_chunk`, `uid_sender` are extracted. The received parameter chunk is then stored in the appropriate location within the `received_models` data structure using the `iteration`, `uid_sender` and `index` as keys (lines 12 and 13).
The stopping condition of the While loop at line 2 will differ depending on the version of UDP that is used. In the case of the UDP version with receiving threads, this loop will always be running until we reach the end of the training i.e. until it finishes the iterations. For this we use an `Event()` object form the `threading` library that we set at the end of the training, this way allowing the thread to terminate gracefully. In case of the version of UDP that uses synchronization, we set a timeout of few milliseconds in order to be able to call `socket.recvfrom()` several times to intercept each packet.

3.5 The Averaging step

The averaging step is very important, this is where each node will finally deserialize the received models and average it with its local model using the metro-hastings weights. For this, we introduce the `_averaging_UDP` (Algorithm 8) and the `deserialized_model_UDP` (Algorithm 9) methods in the `sharing` class.

Algorithm 7 Receive

```
1: function RECEIVE(received_models)
2:     while condition do
3:         received_data ← RECEIVE_PACKET
4:         if received_data then
5:             data_bytes, _, _ ← received_data
6:             data ← PICKLE.LOADS(data_bytes)
7:             degree ← data["degree"]
8:             iteration ← data["iteration"]
9:             param_chunk ← data["param_chunk"]
10:            index ← data["index"]
11:            uid_sender ← data["uid_sender"]
12:            received_models[iteration][uid_sender][index] ← param_chunk
13:            received_models[iteration][uid_sender]["degree"] ← degree
14: procedure RECEIVE_PACKET
15:     data, addr ← SOCKET.RECVFROM          ▷ Returns (data, connection information)
16:     return data, addr[0], addr[1]          ▷ Returns data, IP_ofSender, port_ofSender
```

Until now, each node sent to its neighbors packets containing :

- Chunks of the model parameters
- The chunk index
- The `uid` of the sender
- An integer corresponding to the sender's degree (number of neighbors)
- The corresponding iteration

Each node will be able to retrieve from its `received_models` dictionary the packets associated to the current iteration with a simple lookup in $O(1)$. `received_models[curr_iteration]` will therefore map to a dictionary where the `uid_sender` is the key mapping to another dictionary that itself has the chunk `index` as key to finally map to the corresponding chunk of data.

3.5.1 Deserialization

We process the data parameters sent by each neighbor by looping over each `received_models[curr_iteration][uid_sender]` entry (line 5 of Algorithm 8) and pass it to the `deserialized_model_UDP`. In this procedure, we will reconstruct the model parameters in a PyTorch state dictionary format (with the name of the layer as key and the actual parameter tensor as value). For this, we will be able to get the shape of each tensor for each layer by going through the local state dictionary. We will then transform the local parameters to one 1D array that we will chunk and store in a temporary dictionary `temp_dict` with the chunk index as key (line 15 of Algorithm 9) . This operation will take $O(N)$ where N is the size of the model.
We can now go through the received chunks of the current sender. This is done at line 19, we loop over the keys and values of the `temp_dict` to fill it with the neighbors chunk at index k if received, otherwise we keep the local parameter chunk. In other words, if a chunk index has not been received, it will be replaced by the local parameter chunk. After going through all the chunk indexes, in an increasing order, we append each chunk to the `merged_flattened` list that we next concatenate. We reshape this 1D array and return the reconstructed state dictionary `state_dict` at line 26.

It is worth mentioning that for memory optimization purposes, every entry that have been processed will be deleted from the `received_models` dictionary, this operation takes O(1).

3.5.2 Averaging

The next step is the actual averaging which consists in merging the local model with its neighbors' through a weighted average. For that, we use Metropolis-Hastings weights, where each model parameter p_i^t of node i at iteration t is computed by attributing weights to the contributions of its d_i neighbors and to its own contribution using the following formula:

$$p_i^{t+1} = w_i p_i^t + \sum_{n=1}^{d} w_n p_n^t,$$

with

$$w_n = \frac{1}{1 + \max(d_n, d)}$$

and

$$w_i = 1 - \sum_{n=1}^{N} w_n$$

Note that d_n corresponds to the degree of neighbor n that was sent along with the parameter chunk.And d is the number of neighbors from which the node i received at least one chunk for that current iteration. Finally, at line 16 we update the node's model.

Algorithm 8 Averaging (UDP)

```
1:  function _AVERAGING_UDP(received_models, curr_iteration)
2:      if received_models[curr_iteration] then
3:          total ← {}                                          ▷ State dict storing model
4:          weight_total ← 0                                    ▷ Total weight of models
5:          for all (i, uid_sender) in enumerate(received_models[curr_iteration]) do
6:              if LEN(received_models[curr_iteration][uid_sender]) > 0 then
7:                  data ← received_models[curr_iteration][uid_sender]
8:                  degree ← data["degree"]
9:                  data ← DESERIALIZED_MODEL_UDP(data)         ▷ Deserialize and merge models
10:                 weight ← 1 / (max(LEN(received_models[curr_iteration]), degree) + 1)
11:                 weight_total ← weight_total + weight
12:                 for all (key, value) in data.items() do
13:                     total[key] ← total[key] + value × weight
14:         for all (key, value) in MODEL.STATE_DICT.items() do
15:             total[key] ← total[key] + (1 - weight_total) × value
16:         MODEL.LOAD_STATE_DICT(total)                        ▷ Update local model
```

Algorithm 9 Deserialized Model (UDP)

```
function DESERIALIZED_MODEL_UDP(m)
    state_dict ← MODEL.STATE_DICT()
    shapes ← [ ]
    lens ← [ ]
    tensors_to_cat ← [ ]
    for all (_, v) in state_dict do
        shapes.APPEND(v.shape)
        t ← v.flatten()
        lens.APPEND(t.shape[0])
        tensors_to_cat.APPEND(t)
    T ← TORCH.CAT(tensors_to_cat, dim=0).numpy()          ▷ Local model serialized in 1D array
    chunks ← SPLIT_INTO_CHUNKS(T,CHUNK_SIZE)              ▷ Chunking local model
    temp_dict ← {}
    index ← 0
    for all c in chunks do                ▷ Fill in the temp dict with chunks of local params
        temp_dict[index] ← c
        index ← index + 1
    merged_flattened ← [ ]                                ▷ To be concatenated and reshaped
    for all (k, v) in temp_dict do                        ▷ Merge local with received params
        if k in m then
            merged_flattened.APPEND(m[k])
        else
            merged_flattened.APPEND(v)
    merged_flattened_concatenated ← CONCATENATE(merged_flattened)
    start_index ← 0
    for all (i, key) in enumerate(state_dict) do          ▷ Start reconstructing state dict
        end_index ← start_index + lens[i]
        state_dict[key]←merged_flattened_concatenated[start_index : end_index].reshape(shapes[i])
        start_index ← end_index
    return state_dict
```

4 Experiments and Evaluation

In this section we conduct several experiments to test our implementation in different settings. We will first compare our two versions of UDP (Threading vs Synchronised) and discuss relatively to the initial TCP implementation. Secondly, we will conduct a final evaluation using the `TC` library to introduce network delay and packet loss.
In each experiment, we describe its setup, followed by the results and corresponding assessment.

4.1 Dataset and Model

We use the CIFAR-10 datasets to test our implementation and evaluate the algorithms.

CIFAR-10: The CIFAR-10 dataset is a widely used dataset for image classification. It contains $60,000$ 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is balanced as there are 6,000 images of each class. The training and test datasets are loaded using the `torchvision.datasets.CIFAR10()` function and contain 50,000 and 10,000 images respectively.
The model architecture that we are considering is called `LeNet` and is inspired by the original `LeNet` network for MNIST (Figure 2). It consists of a convolutional neural network (CNN) with 3 convolutional layers, each followed by a max pooling layer and a group normalization layer. The activation function used here is the `ReLU`. The first convolutional layer has 32 filters of kernel 5 and takes as input the 3-channel images and uses "same" padding to keep the spatial dimensions unchanged. The output is then passed through a max pooling layer, which downsamples the output by taking the maximum value within a 2x2 window and moving by a stride of 2. And finally a group normalization layer, which normalizes the activations. The second and third convolutional layers have a similar architecture as the first - the number of filters is however of 32 and 64, respectively. Finally the output of the third convolutional layer is passed through another max pooling layer and a group normalization layer before being flattened to a 1×1024 1D array and fed into a fully connected layer with 10 output units, corresponding to the 10 classes in CIFAR-10. The output of the fully connected layer is the final classification logits of the model.
There are $89,578$ trainable parameters in total, including biases:

1. Conv1 layer : $3 \times 32 \times 5^2 + 32 = 2,432$
2. Conv2 layer: $32 \times 32 \times 5^2 + 32 = 25,632$
3. Conv3 layer: $64 \times 32 \times 5^2 + 64 = 51,264$
4. FC layer: $64 \times 4^2 \times 10 + 10 = 10,250$

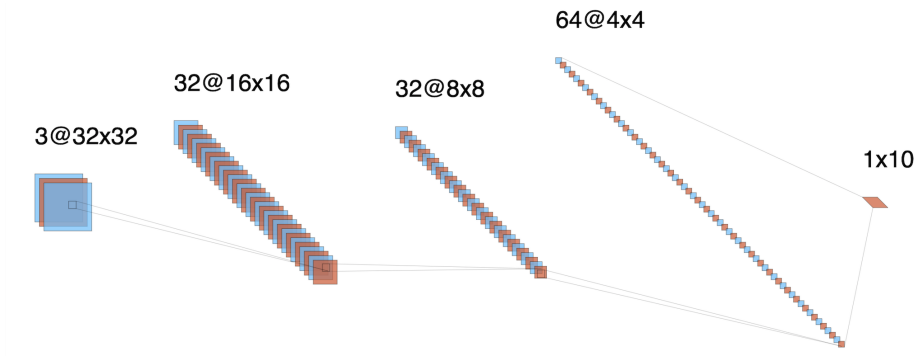


Figure 2: The LeNet architecture for CIFAR-10

4.2 Training Configuration

In the following sections, the models have been trained using the D-PSGD method on the IC cluster. In order to properly compare the results, we use the same dataset, hyperparameters and seeds at every evaluation:

- Dataset: CIFAR-10
- Model: LeNet
- Random seed: 90
- Shards: 4
- Learning rate: 0.01
- SGD batch size: 8
- rounds: 9
- Loss function: Cross entropy loss
- Machines: 1
- Total nodes: 16
- Procs per machine: 16
- Iterations: 1000
- Test after: every 20 iterations
- Initial topology: 3-regular graph with 16 vertices
- CHUNK_SIZE: 4000 parameters

The CHUNK_SIZE argument represents the number of parameters that will be contained in one packet. In this case, we chose to send 4,000 parameters at a time. The total size of the model (89,578 parameters) after pickeling is about 363,216 Bytes. Note that pickeling does not just convert the data to bytes but it also stores metadata about it. Therefore, there will be 23 packets, each of size 16,220 Bytes roughly.

4.3 Comparing two UDP implementations

We start our evaluation by running our two versions of UDP compared to the initial framework that uses TCP for 1000 iterations on the CIFAR10 dataset. During the training, the nodes are exchanging their models after each iteration. Every 20 iterations, each node evaluates its current model on the test set. At the end of the training we aggregate the results of the evaluations over all nodes by computing the mean and the standard deviations.
Regarding the notations in the following plots:

- `TCP`: The framework that uses the initial TCP module.
- `UDP`: The framework that uses UDP with synchronization (400ms timeout).
- `UDP_threading`: The framework that uses the version of UDP based on receiving threads.

4.3.1 Analysing results: Issues with the receiving thread

We report on the graphs below the results of the training and testing loss with respect to communication rounds (Figure 4) as well as the testing accuracy with respect to communication rounds and with respect to time (Figure 5).
We notice from the graphs showing the loss, that the framework using TCP and our implementation of UDP perform very similarly. However the UDP version that uses the receiving thread, referred to as `UDP_threading`, is performing very poorly. If we take a look at the testing loss, we can see that, firstly, the loss is not decreasing and secondly, the values of the losses of the nodes along the training are highly dispersed around the mean compared to the TCP and UDP versions i.e. we notice a high variance.

Analysis: After looking at the log files showing the timestamps of the communication between nodes, we notice that the nodes do not follow the same speed of execution. With the receiving thread setting, the nodes do not wait for neighbors to receive their models in order to begin the next iteration. In other words, whenever a node finishes an iteration, it directly sends its model to its neighbors regardless of their training stage, and pursues training. The 'fastest' nodes will never get the updates from their neighbors, and will therefore train the model based on their local updates uniquely, leading to overfitting, thus to a relatively high testing loss.
Additionally, with looking into more details in the topology of the graph, we identified an independent set (i.e. a set of nodes that are not adjacent to each other) of maximum size of 5 nodes within this 3-regular graph (Figure 3) that always receive the models of their 3 neighbors. The issue for these 'slow' nodes comes from the fact that they are receiving overfitted models from neighbors that are faster. Slow nodes will update their local model and produce models with a high training loss thus a high testing loss.
As a result, all of the nodes will perform poorly overall, which explains the behavior of the training and testing loss. We are showing in Figure 3 below the 3-regular graph with 16 nodes that we used to run this experiment. The slowest nodes are colored in red. At the end of the 1000 iterations, we computed the difference in time at which they finished the training with respect to their 3 neighbors. These are the statistics of the aggregated results:

- **mean**: 14.29 sec
- **std**: 13.95 sec
- **median**: 8.21 sec
- **max**: 44.05 sec / **min**: 2.47 sec

Based on these observations, it can be concluded that the varying rates at which the nodes operate have a detrimental effect on the overall training process. This is an issue that we should consider using UDP. To mitigate this, we introduced a synchronisation mechanism between the nodes in our second implementation of UDP.

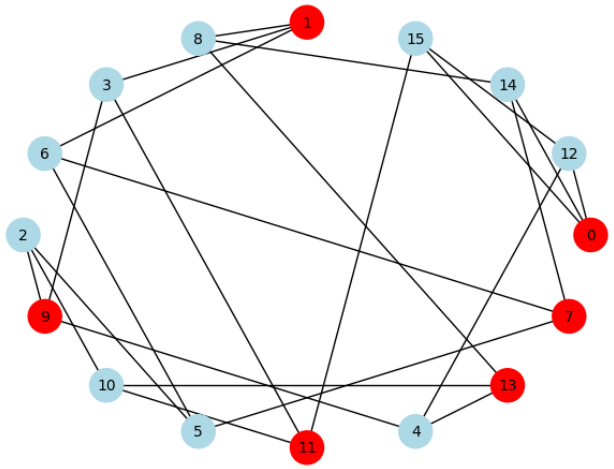
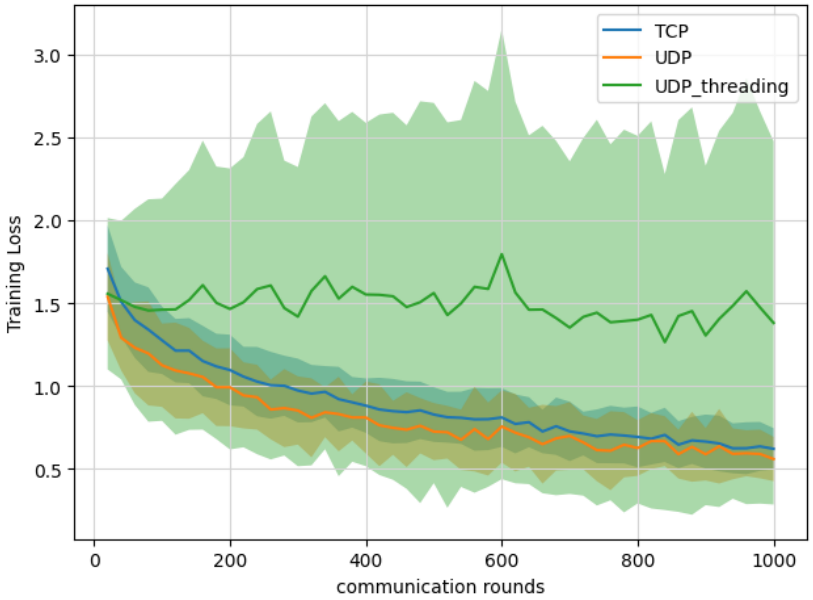
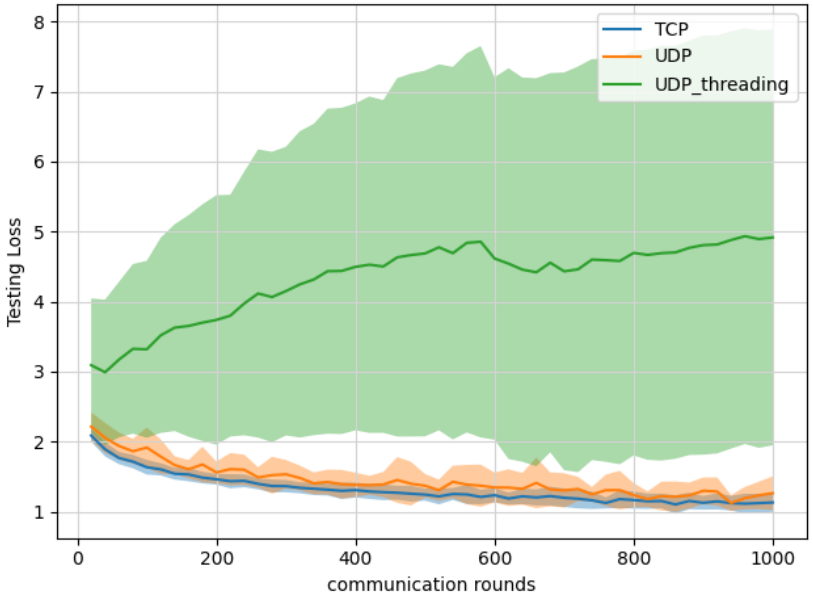


Figure 3: 3-regular Graph with 16 nodes. Slowest nodes are colored in red.

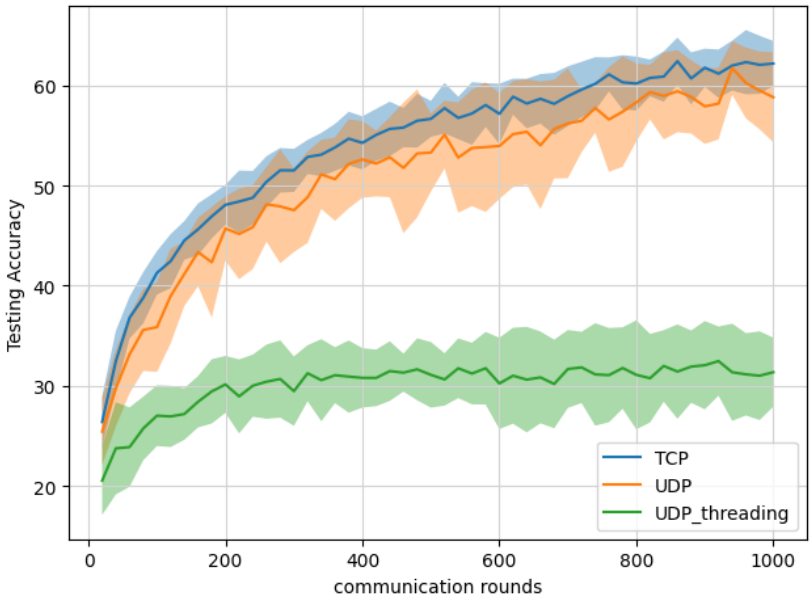


(a) Training Loss with respect to communication rounds

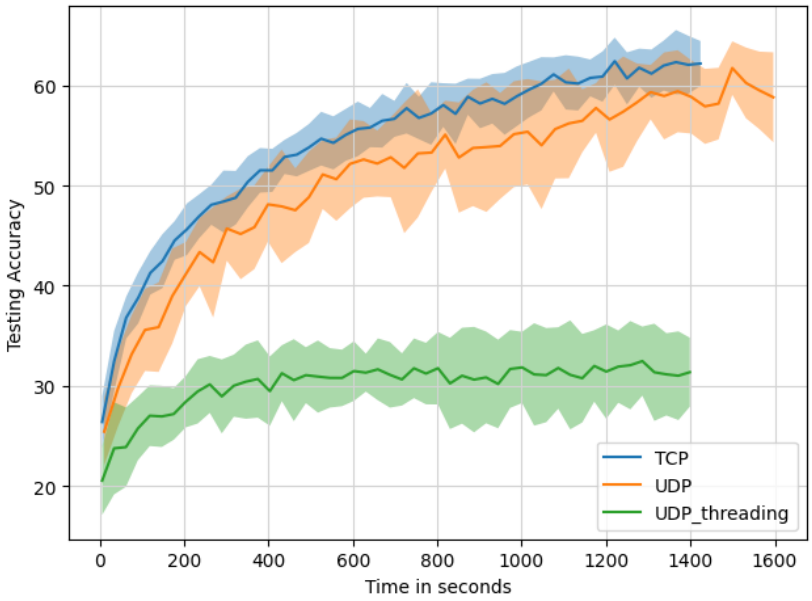


(b) Testing Loss with respect to communication rounds

Figure 4: 1000 iterations on CIFAR-10



(a) Testing Accuracy with respect to communication rounds



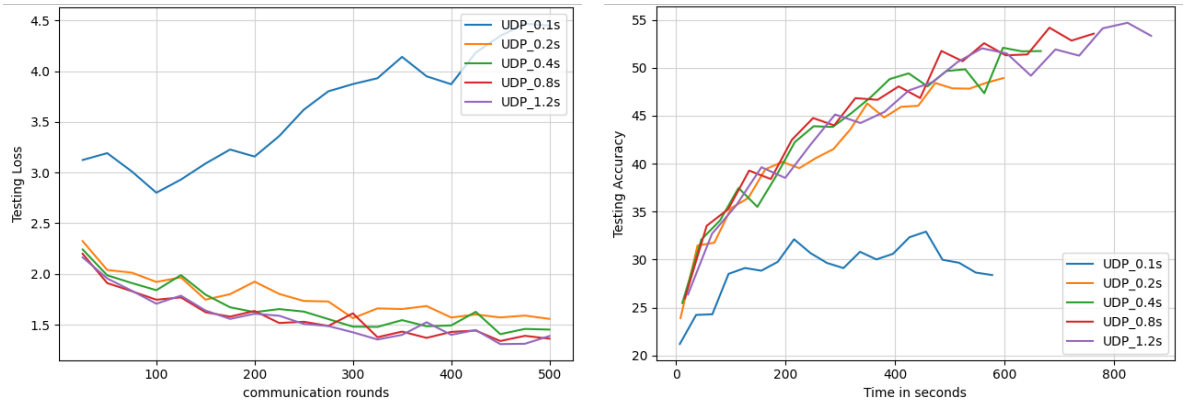
(b) Testing Accuracy with respect to time

Figure 5: 1000 iterations on CIFAR-10

4.3.2 Analysing results: Better results with the synchronized version of UDP

In our second implementation for UDP, we removed the receiving thread, and introduced instead a while loop that takes on the responsibility of waiting for the neighboring nodes to send their model.

Choosing the Timeout Value: By running some experiments, we noticed that there are rare cases where a given node would not receive packets from one of its neighbors at some point during the training. We therefore introduced a timeout for that matter. As a reminder, as explained in the D-PSGD for UDP with synchronization (Algorithm 3), the main stopping condition of the while loop is when a node receives packets from all its neighbors.
In the graphs below (Figure 6), we run 500 iterations on the CIFAR-10 dataset with different timeout values for UDP in order to see which value gives the best results. The 5 timeout values that we considered are the following: 1200, 800, 400, 200 and 100 Milliseconds. We referred to the duration of 1 iteration over TCP that is on average 263 ms. In terms of notation, `UDP_Xs` refers to the experiment with a timeout of `X` seconds.



(a) Testing Loss with respect to communication rounds

(b) Testing Accuracy with respect to time

Figure 6: 500 iterations on CIFAR-10

Analysis: As we can see from the two graphs, The testing loss is decreasing for all timeout values except for the timeout of 100 milliseconds. Which is predictable as this timeout value is less than 1 TCP iteration step. In other words, we are not giving enough time for the nodes to receive the models from their neighbors, which is also reflected in terms of accuracy - which is below 30%. On the other hand, the other experiments behave similarly overall. More specifically, with both timeout values of 1200ms and 800ms the model reaches a level of accuracy of 54%. A 400ms timeout induces a 52% accuracy. A 200ms timeout however performs significantly less better with a 48% accuracy.
More importantly, the difference that we are interested in lies in the runtime. We will keep a **400ms timeout** for the following experiments as it performs well and faster.

The `UDP` experiment that we showed in the previous subsection uses, as we mentioned, the 400 ms timeout version. We logged for each node the number of timeout events that occurred during the 1000 iteration training on CIFAR-10. As a reminder, a timeout occurs when a given node, at a given iteration didn't receive packets from one of his three neighbors. Each of the 16 nodes, on average, encounters approximately 51.2 timeouts out of a total of 1000 iterations, which corresponds to a relatively low rate of 5.12%.

Comparing to TCP: Now that we have discussed the details of the two UDP versions, we will now proceed to evaluate and compare them relative to TCP.
As expected, the testing loss and accuracy (Figures 4 & 5) obtained from the evaluation of UDP closely resemble the model's performance when operating over TCP. However, we can still notice that TCP demonstrates better overall performance in terms of accuracy and testing loss. This can be explained by the fact that running over this UDP implementation still presents risks for packet loss.

Additionally, the main goal of this project was to see if we can train faster with UDP. **From these observations, we conclude that the TCP implementation performs faster and slightly better than UDP in the case where the network is not prone to packet loss.** In fact, we can see from Figure 5, as we plot the testing accuracy with respect to time, that the TCP implementation reaches an accuracy of 63% within 23 minutes, while the UDP implementation takes about 26 minutes to reach a 60% accuracy.

For each of the communication protocols, we run for 100 iterations the CIFAR10 model and compute the statistics of the duration of 1 iteration. The measure of 1 iteration includes the steps from preparing the data, sending the data to each neighbor, receiving and storing the received models and averaging. On Figure 7 we show the box plots of the measurements for each of the three protocols and we report on the table below the corresponding statistics. The `TCP` and `UDP_threading` implementations behave similarly as 1 iterations takes 263 ms and 272 ms respectively. The UDP implementation with a 400ms timeout shows an average duration of 487 seconds, which is consistent with our setting.

Communication Module	**Mean**	**Median**	**Standard Deviation**
`TCP`	263 ms	258 ms	37 ms
`UDP_threading`	272 ms	272 ms	12 ms
`UDP`	487 ms	423 ms	201 ms

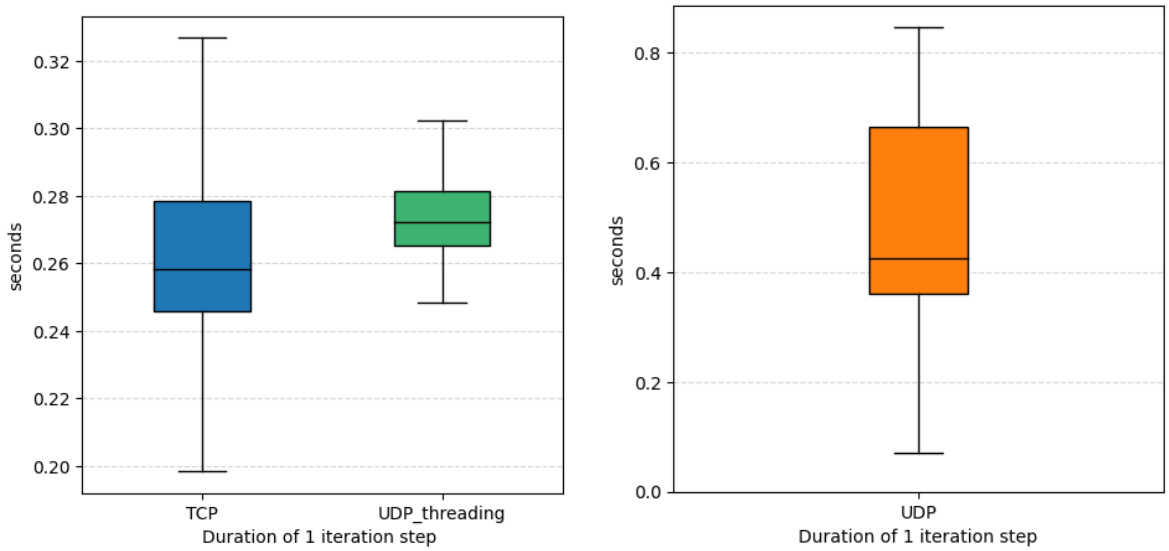


Figure 7: Box plots for the duration of 1 iteration step over TCP, UDP and UDP_threading. Measured by running 100 iterations on CIFAR10.

4.4 Evaluating with TC Library

In this section, we evaluate our model using the Linux traffic control `TC` to introduce network delay and packet loss. We will run the experiments with different delay and loss configurations:

- **Delay**: For all of the experiments, we set the average delay to 200ms (close to the duration of one TCP iteration step) with a standard deviation of 50ms .
- **Loss**: We use different values for the loss rate, with a fixed correlation between consecutive packet losses of 25%. In terms of notation, `TCP_TCX%` refers to an experiment run over TCP with `X%` of loss rate.

The results of the experiments are reported on Figure 8. We train on the CIFAR10 over 1000 iterations. We make two main observations from this experiment:

- **UDP implementation:**
  We run this experiment with a packet loss rate of 10%, 20%, 40% and 70%. A 10% packet loss gives naturally less better results than no packet loss, it induces a drop in the accuracy from 59% to 56% . A 40% packet loss induces a 6% drop of accuracy while a 70% packet loss induces a 15% drop of accuracy. A 20% packet loss performs similarly as the 10% packet loss.
- **TCP implementation with respect to UDP's performance:**
  Packet loss over TCP makes the whole process slower as it uses a retransmission mechanism. `TCP_TC10%` and `TCP_TC20%` finished 1000 iterations within 40 minutes and 1h 36 minutes respectively. A 20% packet loss over TCP have caused a significant delay of approximatively 1 hour to finish 1000 iterations with respect to `UDP`.
  `TCP_TC20%` reached an accuracy of 62% within 1h 36 minutes while `UDP` reached 59% accuracy within 26 minutes only. It is also worth noting that `UDP_TC40%` still behaves better than `TCP_TC20%`.

We report on Figure 9 a box plot of the measurements of the duration of 1 iteration step of `TCP_TC20%`. The average duration is of 4.185 seconds with a standard deviation of 3.702 seconds. In the scenario of a network configuration with 20% packet loss, the use of TCP makes the training slower than UDP with a factor of $\frac{4185ms}{487ms} = \mathbf{8.4}$ for an improvement in the accuracy of just 6% (`TCP_TC20%` with 62% vs `UDP_TC20%` with 56% accuracy). This makes the UDP implementation significantly faster.

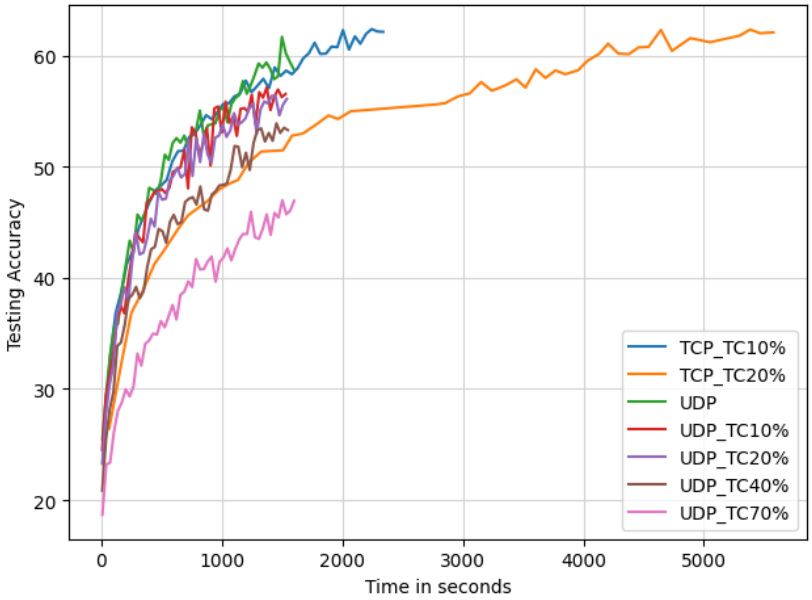


Figure 8: Testing accuracy with respect to time - 1000 iterations on CIFAR-10 using TC

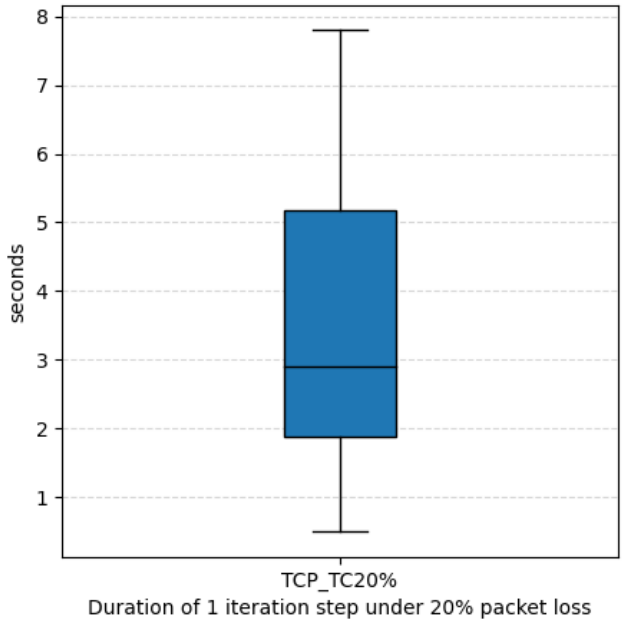


Figure 9: Box Plot for the duration of 1 iteration step over TCP under 20% packet loss - 20 iterations on CIFAR10

5 Conclusion

The aim of this project was to implement the UDP communication protocol and to integrate it into the existing `DecetralizePy` framework, which initially uses TCP. Our goal was to investigate if it would be possible for Machine Learning models to effectively learn in a decentralized setting while communicating over an unreliable protocol like UDP. This way, prioritizing low latency.
The first observation we made was that in a decentralized setting, nodes operate at different rates. Therefore training in a fully asynchronous fashion can have a detrimental effect on the overall training process as we saw earlier. This issue should be taken into account in the UDP implementation and is in favour of TCP.
After addressing this issue, we were able to compare the performances of these two communication protocols. Our findings indicate that in scenarios with no packet loss, TCP performs slightly better and faster. However, in networks where there is a chance of packet loss, specifically starting from a packet loss rate of 20%, UDP demonstrates significantly faster performance.
Overall, this project has provided us with valuable insights into the capabilities and trade-offs between UDP and TCP when applied to decentralized Machine Learning environments.

5.1 Future Work

For future work, we can think of the following:

- Running similar experiments on datasets other than CIFAR-10 to see if the results generalizes or not.
- Running similar experiments on different graph topologies.
- Communication Compression techniques have demonstrated good performance when the underlying communication networks has both high latency and low bandwidth. It can be interesting to see how these approaches (DCD-PSGD / ECD-PSGD) [3] behave when using UDP.

5.2 Acknowledgments

I would like to thank the project supervisors Rishi Sharma and Dr. Rafael Pires for their help during the semester as well as Professor Anne-Marie Kermarrec for making this project possible.

6 References

[1] DecentralizePy. SaCS. URL: https://gitlab.epfl.ch/sacs/decentralizepy

[2] Pieter Hintjens. ZeroMQ: messaging for many applications. " O'Reilly Media, Inc.", 2013.

[3] Tang, Hanlin and Gan, Shaoduo and Zhang, Ce and Zhang, Tong and Liu, Ji. Advances in Neural Information Processing Systems. Communication Compression for Decentralized Training. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/44feb0096faa83-Paper.pdf.

[4] Xiangru Lian et al. "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent". In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017.

[5] Master Thesis: "Improving communication-efficiency in Decentralized and Federated Learning Systems" by Jeffrey Wigger, 2022.

[6] Semester project: "Data sharing for tackling non-iidness" by Joseph Abboud, 2023.