

**6CCS3EEP Electronic Engineering Individual Project**

QUIC Approach for Federated Learning

Final Project Report

**Author: Eugeniu Dimitriu**

**Student Number: 1843486**

**Supervisor: Dr Toktam Mahmoodi**

**Programme of Study: BEng Electronic Engineering**

**April 8, 2021**

**Originality Avowal**

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

I grant the right to King’s College London to make paper and electronic copies of the

submitted work for purposes of marking, plagiarism detection and archival, and to upload a

copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this

report does not exceed 25,000 words.

Eugeniu Dimitriu

April 8 ,2021

**Abstract**

Federated Learning is a relatively new domain of machine learning that enables the training of models avoiding the need for data collection. This is achieved by training a model provided by the server on a client device and sending the model back. Implementing federated learning raises new challenges, one of the major ones being communication costs. S**trategies targeting the reduction of the amount of data exchanged are actively developed. Another approach that can lower communication costs is to make data exchange more efficient. This thesis proposes the design for a federated learning system that uses QUIC for the transport layer protocol. The focus is to use the stream multiplexing property of QUIC and adapt the popular algorithms used in federated learning to this design. Furthermore, an evaluation of system performance is presented.**

Contents

[Chapter 1 Introduction 1](#_Toc68847713)

[1.1 Motivation and Objectives 1](#_Toc68847714)

[1.2 Paper Outline 2](#_Toc68847715)

[Chapter 2 Background 3](#_Toc68847716)

[2.1 Machine Learning 3](#_Toc68847717)

[2.1.1 Deep Learning 3](#_Toc68847718)

[2.1.2 Multi-layer perceptron 3](#_Toc68847719)

[2.1.3 Neuros 4](#_Toc68847720)

[2.1.4 Activation functions 5](#_Toc68847721)

[2.1.5 Training a multi-layer perceptron 6](#_Toc68847722)

[2.1.6 Convolutional Neural Networks 9](#_Toc68847723)

[2.2 Federated Learning 10](#_Toc68847724)

[2.2.1 Challenges and Properties of Federated Learning 11](#_Toc68847725)

[2.2.2 Federated Learning System Design 11](#_Toc68847726)

[2.2.3 Federated Averaging 12](#_Toc68847727)

[2.3 QUIC Transport layer protocol 14](#_Toc68847728)

[2.3.1 Transport layer 14](#_Toc68847729)

[2.3.2 QUIC Properties 15](#_Toc68847730)

[Chapter 3 System Design 17](#_Toc68847731)

[3.1 Protocol 17](#_Toc68847732)

[3.1.1 Waiting Phase 17](#_Toc68847733)

[3.1.2 Training Phase 17](#_Toc68847734)

[3.2 Server 18](#_Toc68847735)

[3.2.1 QUIC Server 19](#_Toc68847736)

[3.2.2 Federated Learning Process 19](#_Toc68847737)

[3.3 Client 24](#_Toc68847738)

[3.3.1 QUIC Client 25](#_Toc68847739)

[3.3.2 Federated Learning Task 25](#_Toc68847740)

[Chapter 4 Simulation 28](#_Toc68847741)

[4.1 Tools 28](#_Toc68847742)

[4.2 Setup 28](#_Toc68847743)

[4.3 Models 29](#_Toc68847744)

[4.3.1 Multi-layer Perceptron with hidden two layers 29](#_Toc68847745)

[4.3.2 Convolutional Neural Network 30](#_Toc68847746)

[4.4 Data 30](#_Toc68847747)

[Chapter 5 Results and Analysis 31](#_Toc68847748)

[5.1 Limited Upload 31](#_Toc68847749)

[5.2 Limited communication 32](#_Toc68847750)

[Chapter 6 Conclusion and Future Work 33](#_Toc68847751)

[6.1 Conclusion 33](#_Toc68847752)

[6.2 Future Research 33](#_Toc68847753)

[Bibliography 35](#_Toc68847754)

# Introduction

## Motivation and Objectives

In the last few years, significant advances in machine learning (ML) technologies revolutionised the fields of computer vision, self-driving cars, and speech recognition [1, 2, 3]. These improvements were possible due to the increased interest from the research community powered by the surge in computational power and data availability [4, 5].

ML is a tool that enables solving problems using data rather than domain knowledge. Nowadays, ML takes place in a centralised fashion, with the data held inside a datacentre where a machine learning model is trained on powerful computers [6]. Most of the time, this data is generated at edge devices(i.e. mobile phones, tablets, drones) and uploaded to the datacentre [7, pp. 1-1]. This process is time-consuming and requires a lot of bandwidth [4]. Storing and handling enough data required for training a good model demands a massive infrastructure. Moreover, data collection raises issues related to data ownership and privacy and is debated more and more in society [7].

With the advances in processing technology, edge devices have the power to train ML models [8]. Federated Learning (FL) emerged as a distributed learning approach that uses the power of edge devices to train a model [9]. Clients compute model updates using the local dataset, which are then aggregated by a server to update a global model. Since the data is not shared, FL provides increased privacy, less communication, and less power consumption [9].

However, to get the benefits mentioned above, a federated learning framework has to deal with new issues raised by the inability to control the edge devices. Device availability, unreliable connection, and interrupted execution are just a few of the issues [10]. Active research is carried out to find solutions to the problems emerging in FL. A lot of research focuses on developing methods to reduce the communication between the edge devices and the server [11]. The researchers focus on techniques that reduce the amount of information exchanged without affecting model performance, like federated averaging (FedAvg) and federated dropout [12, 13]. There appears to be no research that studies how to deliver the data more efficiently. The transport protocol is a set of rules that regulate the exchange of data in a system. Motivated to find an efficient way to exchange the data, this project focuses on the right choice of a transport protocol.

QUIC is a secure transport protocol with features that can prove to be handy in conditions provided by FL [14]. QUIC is resilient to network migration benefiting systems where the clients are expected to change networks relatively often [14]. As a prominent feature of QUIC, stream multiplexing provides a way to deliver data in parallel [14]. QUIC proves to be faster than other protocols in limited bandwidth scenarios [15].

The scope of this project is to design a federated learning system around the multi-stream capabilities of QUIC. The system follows some of the design techniques presented by Bonawitz et al. [10] and adapts popular algorithms, like FedAvg, to multi-stream.

## Paper Outline

**Chapter 2** presents the literature review of the topics this thesis builds on. It explores machine learning with a focus on neural networks, federated learning and the QUIC transport protocol.

**Chapter 3** dives into the design of a federated learning system that uses QUIC for the transport layer. It focuses on the multi-stream capabilities of QUIC and how popular techniques used in federated learning are adapted for this system.

**Chapter 4** goes over the setup used for simulation as well as provides an overview of the experimental settings simulated.

**Chapter 5** presents the results of the simulations together with an evaluation of the relative performance to each other.

**Chapter 6** concludes the work presented in this paper and suggests potential directions for future research.

# Background

## Machine Learning

Machine learning evolved as a concept in the second half of the 20th century [16]. ML is achieved through algorithms that extract knowledge from data without much human interaction. The learning process is referred to as training. The final product of this process is referred to as a model. Three categories are used to divide ML tasks: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. Supervised Learning uses labelled data (data that maps an input to output ) to train a model that can predict an output given an input . Unsupervised learning uses unlabelled data to find similarities in the data. In reinforcement learning, the model learns from interaction with the system it is built. The model receives rewards for its actions, intending to learn what actions will maximise total reward.

### Deep Learning

Deep Learning (DL) is a subfield in ML that employs a model structure called artificial neural networks (ANN). Although the concept of ANN is known for as long as ML, other machine learning concepts were preferred because of a more robust theoretical foundation and better performance results [17]. The popularity of ANN increased recently with the development of powerful GPUs that can support the training of large models that outperform classical ML approaches [18].

### Multi-layer perceptron

Multi-layer perceptron (MLP) is a class of ANN. As the name suggests, an MLP is composed of multiple layers: an input layer, one or more hidden layers, and one output layer. Each layer is formed of numerous neurons. Each neuron in layer has a connection to all the neurons in layer .

The input layer consists of neurons, where is the number of features in the data. A feature can be described as a column in the data table. Each hidden layer consists of neurons. The number of neurons is a parameter of model design and can differ on each layer. The output layer has one neuron. For classification tasks, the last layer is represented by a vector of dimension , where is the number of classes in the dataset. A typical MLP structure is presented in Figure 2.1. Each layer can be characterised by its respective output vector:

|  |  |  |
| --- | --- | --- |
| Input layer | Hidden layer | Output layer |

Moreover, each hidden layer has a weights matrix associated with it:

The weights matrix has the same number of rows as neurons in the layer and the same number of columns as neurons in the previous layer .

|  |
| --- |
|  |
| **Figure 2.1: Multi-layer perceptron with two hidden layers** |

### Neuros

The neuron is the building block of neural networks. Input layer neurons are the simplest type as they just output the input they receive. The neurons in the hidden layers apply transformations to the input to extract useful features from the data. Each neuron in the hidden layer applies two transformations to the input vector, a schematic of whose can be seen in Figure 2.2:

Matrix multiplication to associate weights with the input ( is the row j in the weight matrix )

|  |  |
| --- | --- |
|  | 2.1 |

A non-linear transformation using an activation function

|  |  |
| --- | --- |
|  | 2.2 |
|  | |
| **Figure 2.2:Computation blocks inside a neuron** | |

The neuron in the output layer applies an activation function that should predict the outputs. Commonly the softmax function is used for classification and the identity function for regression problems.

### Activation functions

Activation functions are used to add non-linearity to an ANN. Without them, the system’s output would be a combination of linear functions that is also linear. A linear function can learn only linear patterns in the data, which is not always the case.

Some examples of activation functions:

|  |  |  |
| --- | --- | --- |
| Identity |  | 2.3 |
| Sigmoid: |  | 2.4 |
| Softmax: |  | 2.5 |
| Tanh: |  | 2.6 |
| Rectified Linear Unit (ReLU): |  | 2.7 |
| Swish: |  | 2.8 |
|  | | |
| **Figure 2.3: Representation of 4 activation functions** | | |

The dominant functions used in DL architectures are ReLU for hidden layers and Softmax for the output layer [19].

### Training a multi-layer perceptron

The models are initialised with random weights. During the training, the weights are optimised to values that allow making correct predictions. To understand model performance, the output of the MLP is compared to the actual label from where the error is calculated. In the case of categorical data, the one-hot encoding of the label is used.

#### One hot encoding

The one-hot encoding of a categorical variable represents a vector of length k, where k is the number of categories. Each category is associated with an index in the vector. The value at the index is set to 1 while the others to 0. The one-hot encoding of digit 3 for a digit classification task is

#### Loss function

A Loss Function uses the labels and the predictions computed with input and weights to compute a quantitative measure of the error. The training aims to decrease the loss function of the training dataset . The training loss is the average loss across all training examples.

|  |  |
| --- | --- |
|  | 2.9 |

Generally, the loss functions fit in two types. *The cross-entropy loss* function is mainly used in classification tasks. It penalises with a logarithmic increase if the correct class’s probability is low. The other type of functions is mainly used for regression models having the form . It measures the distance between the correct answer and the predicted. The most used loss function of this type is *Mean Squared Error* with .

#### Gradient Descent

In training, the model weights should be adjusted to minimise the loss function. Gradient descent(GD) is an optimisation algorithm used to minimise a function and find a local or global maximum [20]. The gradient of a function shows the direction of the fastest function increase at a particular point. To get to a minimum point the quickest, GD takes a step in the opposite direction of the gradient. This process is shown in Figure 2.4. The step size is determined by the learning rate . For a model with multiple parameters the gradient for each parameter is computed by the partial derivative. The model parameters update equation is 2.10:

|  |  |
| --- | --- |
|  | 2.10 |
|  | |
| **Figure 2.4: Example of gradient descent for a model with one parameter** | |

One of Gradient descent’s drawbacks is that the model will converge to a local minimum or an inflexion point where the gradient zero. Figure 2.5 has a representation of such a situation. With a zero gradient, the weights are updated to the same value.

|  |
| --- |
|  |
| **Figure 2.5: Gradient descent that converges to local minima** |

#### Stochastic Gradient Descent

Because the gradient should be computed for the whole dataset, another drawback is the number of computations required for the gradient. Stochastic Gradient Descent (SGD) uses a random choice of points from the training set to compute the gradient. SGD is much faster to compute, but there is no guarantee that the weights are updated in the most optimal direction. On average, SGD optimises the weights in the rights direction. In SGD, the weights could jump around the optimal point without reaching it.

|  |  |
| --- | --- |
|  | 2.11 |

#### Backpropagation

Due to a large number of weights in ANN computing, the derivatives numerically is not feasible. The training of the MLP is achieved through Backpropagation [21]. The training executes two procedures—f*orward pass,* which calculates the loss functions for the given round. *Backwards pass,* which uses the loss function value to calculate the gradient and update the weights.

In the forward pass, the loss function is calculated by applying a sequence of functions, as showed in **Figure 2.2**. This consecutive application of functions can be viewed as a graph— computation graph. Each node represents a function. For functions applied sequentially, the derivative is easy to calculate as the product of each function derivative. Therefore, each node in the computation graph has the derivative of the function. The gradient at the following nodes is required to compute the gradient of the loss at a node. Therefore the gradient calculation starts with the last layer, from where the name backwards pass.

#### Dropout

The final goal of ML is to train a model that will perform well when given inputs unseen in the training data. Overfitting and Underfitting are two main reasons why a model will showcase poor performance. Overfitting refers to the problem when the model learns the details in the training set rather than generalising for unseen examples. Underfitting refers to the model that can neither learn from training data nor generalise for unseen data. Overfitting is a common problem of large ANN models [22]. Dropout is a technique used to address overfitting in large models [23]. With dropout, each node has a probability of being active during training. Discarding neurons adds noise to the system that impacts the convergence rate but allows for better generalisation [23].

### Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of ANN used primarily on computer vision problems [18]. The main difference between MLP and CNN is a convolutional layer that allows to detect and output dominant image features across the training images. The features are extracted with the help of filters. The filters are matrices of any dimension, with 3x3 or 5x5 being the most popular [18]. An input to a convolutional layer is typically a 3D matrix, i.e. an image where the 3rd dimension of the matrix represents the colour channel information. Each colour channel, representing a 2D matrix, is convolved with a filter. Convolution is the operation of taking blocks from the input image B of the same size as the filter F and computing the scalar for all such blocks, and storing the results in a matrix.

|  |  |
| --- | --- |
|  | 2.12 |

If the input has multiple colour channels, the matrices resulting from convolution are simply added to create a 2-D layer in the feature map. The feature map is a 3D matrix with a width equal to the number of filters also referenced as channels. A schematic of these operations is provided in Figure 2.6. The training optimises the filters to detect mutual features in training images, i.e. wheels in images with cars. However, these features only make sense to the model rather than humans.

|  |
| --- |
|  |
| **Figure 2.6: A structural representation of a convolutional layer** |

#### Pooling

Training CNN requires far more computational power and memory than MLP because of the large number of parameters and operations. A pooling layer is introduced to downsample the output of a convolutional layer. Just like the convolution, for each block of size a fixed operation that outputs a scalar to the feature map is performed; see Figure 2.7. Usually, pooling is done by taking the maximum value in a 2x2 block called max pooling [17]. Max pooling allows halving both the width and the height of the future layer.

|  |
| --- |
|  |
| **Figure 2.7 Schematic of a pooling operation on 2x2 blocks** |

## Federated Learning

Recent interest in ML is fuelled inclusively by unprecedented amounts of data, namely big data [24]. To solve some of the challenges of Big Data, such as privacy concerns and storage space availability, Federated Learning (FL) was proposed by McMahan et al. in 2016. In a federated learning scenario, rather than bringing the data from users to the training machine, the training task is moved to the user devices where the data originates. Locally computed model updates are sent to a server that orchestrates the learning across multiple devices. FL brings the benefits of increased privacy and data security and saves communication costs associated with sending the data.

### Challenges and Properties of Federated Learning

With this approach, additional challenges unseen in a centralised system are introduced because of the distributed nature of the data that has the following properties:

* **Non-IID**: The data available on each device is not representative of the whole population. The data available on each device describes the behaviour of a single user or a subset of users.
* **Unbalanced**: Some devices may have a bigger number of training data points compared to others. Some users could use their devices more extensive than others.
* **Massively Distributed**: The number of clients can be much larger than the average number of data points available for training on each device.

With the training happening on edge devices such as mobile phones, FL could face the challenges of***limited processing resources*** and ***unreliable network connection*** when compared to the resources available in datacentres. However, the latest mobile devices have the power to train CNN relatively fast due to the increase in processing power and mobile GPU [25]. Moreover, the amount of computation performed on user devices is further decreased by the limited amount of data available on each device. Each round of learning is led and followed by client-server communication over a network. Considering that often edge-devices are connected to slow and unreliable networks, network connectivity is considered a bottleneck for large models [26].

### Federated Learning System Design

In their paper Bonawitz et al. [10] propose a design pattern for a federated learning platform. The design is used in Google’s Gboard application [10, 27]. The protocol involves several communication steps:

1. A number of devices that meet specific criteria connect to the server. For a system where the clients are mobile phones, typical criteria would be the phone to be charging and connected to Wi-Fi. These criteria are employed to avoid a negative impact on the device’s primary role.
2. The server selects a subset of devices and sends the current model weights to selected devices. The selection could be random or using custom protocols like FedCS [28].
3. Each client trains the model and computes the new model weights on the local dataset. The training typically uses SGD as an optimisation method [6].
4. After the training, the clients report the updated model back to the server. Due to uncontrolled factors such as *limited memory* or *limited computational resources,* some devices could abort the training. Moreover, the whole process of training and reporting could be delayed by a slow connection. Therefore, the server should define a timeout for the round and a minimum number of responses. These parameters dictate whenever to continue or abort the current round.
5. The server aggregates the updates in a global model. The most common used aggregation algorithm is federated averaging [6].
6. Steps 2 to 5 are repeated until the model converges.

### Federated Averaging

Federated averaging is introduced by McMahan et al. [12] in 2016 as an efficient method, in terms of communication, to aggregate models that were trained using SGD on non-IID and unbalanced data. The system has the following properties:

* The data is available across clients. The entire dataset is represented as a set of data points .
* is the number of all data points.
* Each client is represented as a set of data points that is a subset of or as a set that stores the indices of elements in that create .
* is the number of data points associated with client .

From Equation 2.9 can be shown that the loss of the whole dataset can be computed as a sum of weighted loses computed at each client .

|  |  |
| --- | --- |
|  | 2.13 |
|  | 2.14 |

This property allows computing the new weights of the global model from the new model weights computed at each client relatively easily.

All clients will update their weights with one step of SGD. From equations 2.11, 2.14, and 2.15, it can be shown that the global weights can be computed as a weighted average of the weights received from clients on the server-side.

|  |  |
| --- | --- |
|  | 2.15 |

Usually, global weights are not computed at each step of SGD taken by the clients; instead, the client performs full training for multiple epoch on baches of data of size before the server aggregates the weights. The algorithm has the following steps presented in formalised by McMahan, Moore, Ramage, Hampson, & Arcas in their work:

|  |
| --- |
| FedAvg |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

## QUIC Transport layer protocol

### Transport layer

The transport layer is one of the layers that enable applications on two hosts to communicate [29]. The transport layer is built on top of other layers that connect two hosts, such as physical connection and network. The lower layers provide a way to exchange data between the hosts but do not guarantee delivery or the quality of data. The transport layer introduces the logic to achieve reliable delivery. Transport layer protocol is a set of rules known by both hosts that ensure that all the required data is delivered. The most used protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [30]. The main difference between the two is that TCP applies logic to deliver all the data, i.e. if some data is lost, stop sending new data and send the missing pieces. UDP does not resend the lost packets.

Depending on the application, the transport layer can be followed by a security layer. The security layer is a cryptographic protocol that provides security to network communication [31]. The newest security protocol is Transport Layer Security (TLS) version 1.3.

QUIC is a relatively new transport protocol designed at Google to reduce the latency of applications [32]. QUIC is built to be used for HTTPS traffic and aims to replace the traditional Transmission Control Protocol (TCP) + Transport Layer Security (TLS) combination used in most applications [14]. QUIC encrypts all the data by default using TLS 1.3 [33]. QUIC takes advantage of User Datagram Protocol (UDP) as a substrate adding additional features that aim to improve performance compared to other transport protocols.

|  |
| --- |
|  |
| **Figure 2.8: QUIC vs TCP+TLS stack** [14] |
|  |

### QUIC Properties

#### Reduced Latency Connection Establishment

QUIC reduces the number of round trips required to set up a secure connection by combining the cryptographic and transport handshakes. On the initial handshake with the server, the QUIC client caches a received token that can be used to achieve Zero Round Trip Time (0-RTT) on further connections establishment with the server [14]. However, the first connection is still 1-RTT. 0-RTT allows exchanging messages straight away before the handshake is completed. Figure 2.8 presents the steps taken by QUIC to establish a connection in comparison to the classic TCP+TLS.

|  |
| --- |
|  |
| **Figure 2.9: Initialisation steps for a) TCP+TLS 1.3 2-RTT, b) First QUIC connection 1-RTT, c) Further QUIC connections 0-RTT** [14] |

#### Stream Multiplexing

QUIC streams are an abstraction that allows avoiding the problem of head-of-line blocking. Head-of-line blocking occurs when a packet is lost, and the transmission of the subsequent packets is blocked until the lost packet is retrieved. If a UDP packet is lost, this will affect only the streams whose data was in that packet. The retransmission of lost UDP is programmed at the QUIC level as UDP does not have the means of handling lost packets. A QUIC packet can carry data from multiple streams and is discussed further. A Stream ID identifies each stream.

#### Connection Migration

A 64-bit Connection ID is used to define a QUIC connection. This approach allows the connection to continue even if the client’s IP and port change, which happens when changes in network connection occur. The client is automatically verified since it uses the same session key. Moreover, it allows the migration of the server for the same reasons.

# System Design

The system follows the classical client-server model. The server has two main tasks: to *coordinate* the learning and to *aggregate* the results. The client has the task to *train* the machine learning model received from the server on the data available on the device. In the presented design, the model is a part of the application running on the client device. The server will send only the model weights to the client.

In the following sections, a detailed description of the processes executed by each instance is provided. Firstly, the protocol is presented to provide an overview of the meaningful actions executed by each instance. After, the detailed implementations of these actions are described from both server and client perspective.

## Protocol

The client-server interaction can be divided into two phases. Waiting phase — when the client waits for indications from the server. During the connection phase, there is a small amount of communication. The training phase — the client and server start the federated learning. The later phase is communication heavy.

### Waiting Phase

The communication begins by the client connecting to the server and opening a bidirectional stream referenced as the *main-stream*. The connection signals to the server that the client is ready to work. The process by which a client decides when it is ready to work and when to connect to the server is out of this project’s scope and will not be discussed. All the connected clients are considered fit to work. After the connection, the waiting phase starts.

During the waiting phase, the client and the server communicate periodically to keep the connection alive. The waiting phase is also reached after a training round.

### Training Phase

The training phase is divided into three main processes. Each process covers at least one research area in federated learning.

#### Selection process

A minimum number of clients should be connected to start the selection process. For each training round, the server selects a subset of clients. Depending on the algorithm used for selection, this round may or may not require communication with the client. A random selection of clients is used in this project.

#### Weights exchange process

The training happens synchronously on all the selected clients. Global model weights are sent to all selected clients.

* For each selected client, the server opens a number of additional bidirectional streams — *weight streams*.
* The model weights are divided into parts and are sent to the clients using the created streams.

|  |
| --- |
|  |
| **Figure 3.1: Multi-stream communication paths for QUIC federated learning** |

A time window limits the time available to receive the weights. A client should receive the weights on at least streams to be fit for training. If not enough clients are fit, the training round is aborted. If now all streams reported the weights, the weights from the local model are used. The local model is trained on local data.

#### Aggregation process

The new weights are divided and reported back through the same streams. The weights should be received in a time window. The server firstly aggregates the weights for each client. Finally, all received weights are aggregated in a global model using an adapted version of FedAvg that considers partially received weights. After, the interaction returns to the waiting phase.

## Server

The *federated learning server* (*FLS*) comprises a *QUIC server* and a federated learning process (FLP).

The *QUIC server* handles the connection and communication with clients. The FLP is the coordinator of federated learning. FLP is build of strategies that manage the learning process — selection strategy, communication strategy, and aggregation strategy. The optimal choice of strategy has a significant impact on system performance.

|  |
| --- |
|  |
| **Figure 3.2 Building blocks of a federated learning server** |

### QUIC Server

QUIC server is the instance that handles the QUIC traffic between the federated learning server and the clients. On a new client connection, the QUIC server notifies the FLP and provides a reference for the client connection — . This reference has application programming interfaces (API) to exchange messages on specific streams and create new streams. The API available are:

* — send a message on
* — wait for receipt of a message on
* — create a new stream and return the

QUIC server holds the received messages in a buffer. The API that waits for a messages looks in the buffer for the message. QUIC server is characterised by a timeout — . If no messages are exchanged during the window, the connection will automatically close. When the connection closes, the QUIC server also notifies the FLP.

### Federated Learning Process

The FLP is the instance that executes a series of tasks to enable federated learning. The algorithms implemented in the tasks depend on the choice of strategies. In this project, the following strategies are used:

* Selection strategy: a random choice of clients.
* Communication strategies: The client-server connection is kept alive. The weights are exchanged over multiple streams. The algorithm for this is described further.
* Aggregation strategy: FedAvg adapted to multiple streams.

The FLP starts with the FLS. FLP runs a loop of federated training rounds. In parallel, it runs a service to coordinate client connections. On each notification of a client connection from the QUIC server, a for the client starts. The reference received with the notification is added to the set that stores all the references — .

The sends periodic messages to the client. This is done to keep the connection alive — a requirement of the communication strategy. The period should be less than the . The pseudo-code is presented in .

|  |
| --- |
| periodic communication |
| W |
| : |
|  |
|  |

The federated training loop waits for enough clients to connect. As soon as the number of elements in reaches the minimum number of clients required to do a round of training, the FLP calls the . The will execute the algorithm provided by the selection strategy. The pseudo-code for is given in

|  |
| --- |
| steps required to select clients for training |
| : |
|  |
|  |
|  |

When a client is selected for training by the FLP, the is called. The executes the server-side algorithms provided by the communication strategy to exchange the weights. The executes the following steps to send the weights over multiple streams.

First, it stops the . A message is sent to the client signalling that it was chosen for training — . The model weights are divided into parts. The FLP uses the QUIC Server API to look if streams are open. If not enough streams are available, more streams are opened using the API. The client does not know what the weights received on a stream mean. A message with the information required to aggregate the weights is sent on the *main-stream —* . The weights are sent to the client.

These steps are easily implemented for an ANN. The weights of an ANN are an array of matrices. The element at index is the weights matrix corresponding to layer . The weights can be divided by layer. Each is sent on a separate stream. In this case, the could be an array of stream IDs. The client places the weights received on a stream with at the index corresponding to the index of in the .

|  |
| --- |
|  |
| **Figure 3.3 Structure of for ANN** |

A time window is allocated to send the weights. The server waits for acknowledgement of weights receipt on each stream — . The server decided to proceed further if the is received on enough streams in the allocated time window. The server sends a message indicating if the client should proceed with the training— .

The server waits for a message that indicates that the client finished training— . The server waits for the message carrying the amount of training data the client possesses — . The server waits for the that will allow the server to reconstruct the full weights. The weights are received across the streams specified in the . The weights should be received in a time window. A is sent on each stream that reported in the time window. The weights are aggregated in a. If a layer is not received, it remains empty in the. The  *and the are reported back to the FLP.* The pseudo-code for is given in .

|  |
| --- |
| server task that sends and receives the weights |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| : |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

After all the clients return the model weights, the FLP proceeds to the . The executes the algorithm specified in the aggregation strategy. For each layer , get the set of clients that reported the weights for this layer . For the clients in calculate the total number of data points . Calculate the weighted average of the weights in layer for clients in and set the result as the layer weights in the global model. gives the pseudo-code for this task.

|  |
| --- |
| : Federated averaging for multi-stream; k is the client index |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

After aggregation, the FLP changes the old model weights with the new ones. Finally, the FLP training loop starts over. The complete pseudo-code for FLP is provided in .

|  |
| --- |
| : m is the minimum number of clients to start a round; timeout is the time window to receive the weights |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| : |
|  |
|  |
|  |
|  |
|  |

## Client

The federated learning client (FLC) comprises a QUIC client and a Federated Learning Task (FLT), a local model, and the data. The QUIC client handles the connection with the server and the communication. The FLT coordinates the federated learning process on the client device. The FLT comprises a communication strategy and a training strategy. The communication strategy is the same as the strategy on the server-side, with the difference that it provides the algorithm executed by the client. The training strategy represents the algorithm used to train the local model. The training strategy dictates how the FLC uses the model and the data.

|  |
| --- |
|  |
| **Figure 3.4 Building blocks of federated learning client** |

### QUIC Client

Similar to the QUIC server, the QUIC client is the instance that handles the QUIC traffic on the client-side. When the QUIC client connects to the server, it provides a connection reference to the FLT that is used to send and receive messages. It provides the same APIs as the server. On startup, a timeout window can be set, similar to the server.

### Federated Learning Task

The FLT executes a series of steps required to exchange the weights and train the local model. The FLT is started after the QUIC clients connect to the server. It receives a reference to the server connection and executes the client part of the communication algorithm.

When the client connects to the server, it enters a state where it waits for the from the server. Upon receipt of *,* the client expects the *.*

The weights are received on the enumerated in the . For each stream, a time window is defined. If the weights on a stream are received in the time window, the client sends back a confirmation of receipt on that stream — *.* When weights are received on a stream, the local model is updated with the new weights.

The client waits for confirmation from the server to proceed to the training —*.* If the server signals to abort the round, the client goes to the initial state waiting for *.* Before proceeding to the training, a task is created to send messages periodically to the server in order to keep the connection alive. The pseudo-code is provided in .

|  |
| --- |
| Task to keep the connection alive |
| S |
| : |
|  |
|  |

Next, a is called. The training tasks executes the algorithm provided by the training strategy. The weights are loaded in the local model using the API provided by the model. For convenience is used as the API. The data is loaded from the data storage using the API provided by the data storage — . The model is trained on the data —. After training, the are extracted from the model — .

|  |
| --- |
| : Execution of the machine learning task |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

After the the returned the required data the client signals to the server with a *.* It sends the and the *.* The client sends the new weights across the and returns to the initial state*.* The pseudo-code for this protocol is provided in

|  |
| --- |
| : FLT algorithm |
|  |
| () |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| S |
| n |
| : |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

# Simulation

## Tools

**Python** programming language version 3.6.8 is used to build the QUIC federated learning system. Python is chosen because it provides convenient ways of building machine learning models. **TensorFlow** version 2.3.2 is used to create the machine learning model [34]. TensorFlow is a python framework that allows to build and train machine learning models. The **aioquic** version-0.9.7 library is used to implement the QUIC protocol [35]. Aioquic is a python library that allows the creation of both the server and the client that communicate through QUIC protocol. Both QUIC Server and QUIC Client are created using aioquic. The code is available at <https://github.com/dimitsqx/bachelor_thesis/tree/main/model/quic>.

## Setup

This simulation aims to test how the proposed multi-stream systems design and algorithms impact the global model convergence and performance. The simulations run on two systems:

* One stream system — representing the baseline
* Multistream system

The multi-stream system is the implementation of the design presented in section 3. The one stream system is built on the same proposed template with two differences:

* Communication strategy: The weights are sent as a whole entity on the
* Aggregation strategy: FedAvg is used to aggregate the weights

The client applications run concurrently on a PC. The asyncio python library is used to run the clients concurrently. The server runs on another PC. Running client tasks concurrently on one PC have two benefits over parallel processes:

* Concurrent execution does not overload the CPU as much as parallel execution and allows a bigger number of clients to be connected simultaneously. Parallel training of a large number of models is an expensive task.
* Parallel processes that use the same network for communication can create a network bottleneck if all of them start to send data at the same time.

The number of client instances running at the same time is set to 100. The data for each client is available in a separate file on the PC. The number of clients required for training is set to 10. For local training of the models, a batch size and number of epochs is specified.

The server and the clients are connected to a local area network (LAN). The LAN provides near perfect network conditions in which all the streams report in time. The FedAvg and the multi-stream FedAvg are mathematically equivalent when all the streams deliver the weights. Additional steps are incorporated in the system to simulate the situation when the weights are not received. This is done on both clients and server.

After the weights on a stream are received, the weights are disregarded with probability . It mimics the situation when the weights on the stream are not reported in the specified time window, and a timeout occurs. The probability is the parameter that will vary across the simulations. For convenience, the probability that the data on a client stream is kept is indicated by ; on the server by .

The first group of simulations aims to study the effect of **limited upload** speed, which resembles a realistic scenario where the model can be downloaded but not uploaded due to low upload speed [26]. The values of are varied while keeping the constant at 1.

The **limited communication** case aims to simulate the system with different combinations of and .

After each round of training, the server loads the aggregated weights in a global model. The server evaluates the global model on data that was not available for training — *test data*. Two performance metrics — *loss* and *accuracy* are calculated. The loss is the numerical value of the loss function. The accuracy is the numerical interpretation of how many labels were classified correctly. Both measures provide an overview of how the learning is evaluating with each round of communication.

## Models

### Multi-layer Perceptron with two hidden layers

The model has the following layers:

* A layer with 128 neurons with ReLU activation
* A layer with 256 neurons with ReLU activation and 0.5 dropout probability
* An output layer with softmax activation

The model is trained using SGD with a learning rate of 0.01.

The loss function used is the categorical cross-entropy.

### Convolutional Neural Network

The model is built of the following layers:

* A convolution layer with 32 output channels and 3x3 windows
* A max-pooling layer with a 2x2 window
* A convolution layer with 64 output channels and 3x3 windows
* A max-pooling layer with a 2x2 window
* A layer with 128 nodes with ReLU activation and 0.5 dropout probability
* An output layer with softmax activation

The model is trained using SGD with a learning rate of 0.01.

The loss function used is the categorical cross-entropy.

## Data

The data used for training is a federated version of the MNIST dataset —EMNIST [36, 37]. Each data point has a 28x28x1 image representing a digit and the label associated with it. The data is grouped by the original writer of the digits. Data from 3382 writers is available. The EMNIST dataset is split into the training part and testing. Both parts have at least one image example for every writer. Not all groups have examples for all ten labels. Moreover, the number of images varies by the writer. These properties mimic the **non-IID** and **unbalanced** properties of a federated learning dataset.

The training part is used to train local client models. The test part is used to evaluate the performance of the aggregated model at the server. The simulation station cannot host 3382 clients. Therefore, the data is grouped into 100 groups. All the images belonging to a writer will be in the same entity.

## Results and Analysis

In this section, the simulation results are displayed and interpreted where possible. This section comprises a series of plots. The orange lines in the plots correspond to the results obtained from the simulation of a multi-stream system; the blue lines — the results from the one-stream system.

### Limited Upload

**Figure 4.1** and **Figure 4.2** show the performance of the CNN model across three values of for both system designs. The models were succesfully trained and reached an accuracy of 98 % in 150 rounds. The models show similar performance at each training round, with the plots overlapping in some areas. However, the data for the smaller values of has more noise with sharp jumps between adjacent point. This is the expected behaviour due to the fact that a smaller value of means fewer weights are received. A smaller number of reported weights introduces bias to the model. The bias appears when the weights from a small number of devices are aggregated. The new model is tailored more towards the devices that reported the weights impacting generalisation capabilities of the model. The impact on the generalisation can be seen on the loss and accuracy plot as the spikes in **Figure 4.3** and **Figure 4.6**.

|  |
| --- |
|  |
| **Figure 4.1: Loss vs communication rounds for the CNN trai****ned with B=10 and E=10. Orange lines represent the multi-stream system. Blue represent the one stream system** |

|  |
| --- |
|  |
| **Figure 4.2: Accuracy vs communication rounds for the CNN trained with B=10 and E=10. Orange lines represent the multi-stream system. Blue represent the one stream system** |

The described behaviour is the most prominent in the one-stream system, with represented by the blue dotted line. This system has the highest probability of not receiving any client data. The model still manages to achieve a low loss and a high accuracy but slower when compared to other models. It is interesting to point out that in the case of multi-stream with the spikes are not as prominent. This could happen because of the aggregation mechanism. With the aggregation by layers, the bias is introduced only in a layer. Other layers could have been received from a slightly different pool of clients introducing bias in other direction. Therefore the newly aggregated model is flatter.

|  |
| --- |
|  |
| **Figure 4.3 Loss vs communication rounds for the MLP trained with B=10 and E=20** |
|  |
| **Figure 4.5:Accuracy vs communication rounds for the MLP trained with B=10 and E=15.** |
|  |

From the above simulations, it is clear that the efficient delivery of information is vital for the systems where the complex models are trained for a large number of rounds.

### Limited communication

**Figure 4.6** and **Figure 4.7** show the model performance when the same probability of dropout is observed at both client and server. The one-stream system outperforms the multi-stream system in the situation when perfect weights delivery is not possible. The problem cannot be at the upload as previous results showed that even when the weights are reported back to the server with a probability of 30%, the model still learns relatively fast.

|  |
| --- |
|  |
| **Figure 4.6: Loss vs communication rounds for the CNN trained with B=10 and E=15. Orange lines represent the multi-stream system. Blue represent the one stream system** |
|  |
| **Figure 4.7 Accuracy vs communication rounds for the CNN trained with B=10 and E=15. Orange lines represent the multi-stream system. Blue represent the one stream system** |

The problem is in the decision taken when the weights on a stream do not arrive at the client. The proposed solution was to use the old weights available to the client. However, this proves to be an unfeasible approach. The main reason could be that the weights available at the client are outdated. In the simulation context, the global model could have a performance of 80% accuracy and the local model 11%. Combining the weights from both models would not benefit the federated learning system as the client would train a model worse than the global one, and the chances that the trained model would be better than the existing one are low. Therefore, the proposed algorithm in the system design should be revised to account for these findings. A straight forward solution is to drop the clients that did not receive the full weights.

# Conclusion and Future Work

## Conclusion

Federated Learning is a distributed method of training machine learning models. This paper provided an overview of federated learning, focusing on federated learning system design and popular algorithms. Federated Learning has unique properties different from classic machine learning that bring benefits but also new challenges. Communication cost is one of the issues that stands out and is actively researched. Most of the techniques developed to decrease communication cost involve reducing the amount of transmitted information. There is a gap in research regarding efficient methods of message delivery that can influence communication costs.

This project focused on designing a federated learning system that uses QUIC as a transport protocol. QUIC has several properties that make it an excellent choice for systems where network quality is a problem. The project concentrates on the multiplexing property of QUIC and proposes tailored algorithms, inspired from existing ones, that can make use of partially received weights.

The proposed system was build using python to carry out simulations in order to evaluate the performance compared to a similar system that uses one stream. The simulation results showed a design flaw that could badly impact the performance. A solution to tackle the flaw was proposed. The simulation did not provide enough proof to make a conclusion about the multi-stream performance relative to single-stream. However, it showed that getting more weights back from the client is vital. To understand the multi-stream approach enables the receipt of a increased number of weights, the systems should be tested in an environment with the possibility to tweak network parameters.

## Future Research

This being one of the first works targeting the choice of a transport protocol for a federated learning system, there are plenty of topics that can be addressed in research. Firstly, it is critical to research other transport layer protocols in the context of federated learning.

Further research on other QUIC properties is required. This project focused mainly just on one property of QUIC. The system design should further be tailored to use other properties of QUIC, such as 0-RTT, that could allow getting rid of unnecessary communication to keep the connection alive because the communication can be restored without any delays. Moreover, scenarios such as the internet of vehicles (IoV) could profit from fast connection migration [38]. Getting a quantitative evaluation of how implementing these properties would be helpful. The connection migration mechanism could also provide a convenient way to migrate a connection from an overloaded node to an empty one.

# Bibliography

|  |  |
| --- | --- |
| [1] | S. Hong, S. Kwak and B. Han, “Weakly Supervised Learning with Deep Convolutional Neural Networks for Semantic Segmentation: Understanding Semantic Layout of Images with Minimum Human Supervision,” *IEEE Signal Processing Magazine,* pp. 39-49, November 2017. |
| [2] | R. Kulkarni, S. Dhavalikar and S. Bangar, “Traffic Light Detection and Recognition for Self Driving Cars Using Deep Learning,” in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, Pune, India, 2018. |
| [3] | A. Conneau, D. Kiela, H. Schwenk, L. Barrault and A. Bordes, “Supervised Learning of Universal Sentence Representations from Natural Language Inference Data,” in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark, 2017. |
| [4] | S. H. Kaisler, F. Armour and J. A. Espinosa, “Big Data: Issues and Challenges Moving Forward,” in *46th Hawaii International Conference on System Sciences (HICSS)*, 2013. |
| [5] | C. A. Mack, “Fifty Years of Moore's Law,” *IEEE Transactions on Semiconductor Manufacturing,* pp. 202 - 207, 2011 . |
| [6] | S. A. Rahman, H. Tout, H. Ould-Slimane, A. Mourad, C. Talhi and M. Guizani, “A Survey on Federated Learning: The Journey from Centralized to Distributed On-Site Learning and Beyond,” *IEEE Internet of Things Journal,* pp. 1-1, 2020. |
| [7] | Q. Yang, Y. Liu, Y. Cheng, Y. Kang, T. Chen and H. Yu, Federated Learning, Morgan & Claypool, 2020. |
| [8] | M. T. Beck, M. Werner and S. Feld, “Mobile Edge Computing: A Taxonomy,” in *The Sixth International Conference on Advances in Future Internet*, 2014. |
| [9] | B. McMahan and D. Ramage, 6 April 2017. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html. [Accessed 24 March 2021]. |
| [10] | K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. McMahan, T. Overveldt, D. Petrou, D. Ramage and J. Roselander, “Towards Federated Learning at Scale: System Design,” 2019. |
| [11] | W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato and C. Miao, “Federated Learning in Mobile Edge Networks: A Comprehensive Survey,” *IEEE Communications Surveys & Tutorials,* pp. 2031-2063, 2020. |
| [12] | H. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. A. y. Arcas, “Communication-Efficient Learning of Deep Networks,” in *20 th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017. |
| [13] | S. Caldas, J. Konečný, H. B. McMahan and A. Talwalkar, “Expanding the Reach of Federated Learning by Reducing Client Resource Requirements,” 2018. |
| [14] | A. Langley, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, A. Riddoch, W.-T. Chang, Z. Shi, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov and I. Swett, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *the Conference of the ACM Special Interest Group*, 2017. |
| [15] | P. Megyesi, Z. Krämer and S. Molnár, “How quick is QUIC?,” in *2016 IEEE International Conference on Communications (ICC)*, Kuala Lumpur, Malaysia, 2016. |
| [16] | A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal of Research and Development,* pp. 210 - 229, 1959. |
| [17] | A. Géron, Hands-On Machine Learning with Scikit-Learn and TensorFlow, O’Reilly Media, 2017. |
| [18] | F. CHOLLET, Deep Learning with Python, Manning Publications Co., 2018. |
| [19] | C. Nwankpa, W. Ijomah, A. Gachagan and A. Gachagan, “Activation Functions: Comparison of trends in Practice and Research for Deep Learning,” 2020. |
| [20] | C. Lemarechal, “Cauchy and the Gradient Method,” *Documenta Mathematica,* p. 251–254, 2012. |
| [21] | D. E. Rumelhart, G. E. Hinton and R. J. Williams, “Learning representations by back-propagating errors,” *Nature,* no. 323, p. 533–536, 1986. |
| [22] | S. Lawrence and C. Giles, “Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, Como, Italy, 2000. |
| [23] | N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from,” *Journal of Machine Learning Research 15,* pp. 1929-1958, 2014. |
| [24] | F. Liang, W. Yu, D. An, Q. Yang, X. Fu and W. Zhao, “A Survey on Big Data Market: Pricing, Trading and Protection,” *IEEE Access ( Volume: 6),* pp. 15132 - 15154, 2018. |
| [25] | S. S. L. Oskouei, H. Golestani, M. Hashemi and S. Ghiasi, “CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android,” *Proceedings of the 2016 ACM Multimedia Conference, Open Source Software Track,* pp. 1201-1205, 2016. |
| [26] | J. Konecny, H. B. McMahan, F. X. Yu, A. T. Suresh and D. Bacon, “FEDERATED LEARNING: STRATEGIES FOR IMPROVING COMMUNICATION EFFICIENCY,” 2017. |
| [27] | T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage and F. Beaufays, “Applied Federated Learning: Improving Google Keyboard Query Suggestions,” 2018. |
| [28] | T. Nishio and R. Yonetani, “Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge,” in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China, 2019. |
| [29] | V. Cerf and R. Kahn, “A Protocol for Packet Network Intercommunication,” *IEEE Transactions on Communications,* pp. 637 - 648, 1974. |
| [30] | M. Polese, F. Chiariotti, E. Bonetto, F. Rigotto, A. Zanella and M. Zorzi, “A Survey on Recent Advances in Transport Layer Protocols,” *IEEE Communications Surveys & Tutorials,* pp. 1-1, 2019. |
| [31] | H. Krawczyk and H. Wee, “The OPTLS Protocol and TLS 1.3,” in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbruecken, Germany, 2016. |
| [32] | J. Roskind, “QUIC: Design Document and Specification Rationale,” 2012. |
| [33] | J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-34,” 2021. |
| [34] | 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. Mane, 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. Viegas, . O. Vinyals, . P. Warden, . M. Wattenberg, . M. Wicke, . Y. Yu and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” 2015. |
| [35] | J. Lainé, “aioquic,” 2019. [Online]. Available: https://github.com/aiortc/aioquic. |
| [36] | Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE,* pp. 2278 - 2324, 1998. |
| [37] | “EMNIST,” [Online]. Available: https://www.tensorflow.org/federated/api\_docs/python/tff/simulation/datasets/emnist. |
| [38] | M. Gerla, E. Lee, G. Pau and U. Lee, “Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds,” in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, Korea, 2014. |
| [39] | J. Konecny, H. B. McMahan, D. Ramage and P. Richtarik, “Federated Optimization:Distributed Machine Learning for On-Device Intelligence,” 2016. |