A PRIVACY PRESERVING FEDERATED LEARNING PLATFORM SECURED WITH AUTHORITY CONSENSUS USING BLOCKCHAIN

A PROJECT REPORT

Submitted by

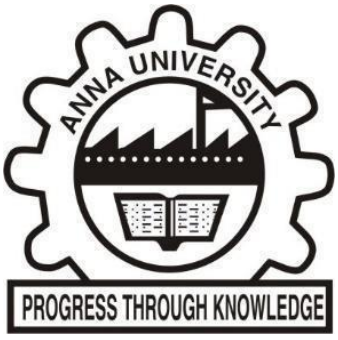
SRIVATSAV R	**2019103066**
SACHIN RAGHUL T	**2019103573**
SANJEEV K M	**2019103576**

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING



COLLEGE OF ENGINEERING, GUINDY

ANNA UNIVERSITY :: CHENNAI 600 025

MAY 2023

ANNA UNIVERSITY :: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**A PRIVACY PRESERVING FEDERATED LEARNING PLATFORM SECURED WITH AUTHORITY CONSENSUS USING BLOCKCHAIN**" is the *bonafide* work of "**Srivatsav R (2019103066)**, **Sachin Raghul T (2019103573)** and **Sanjeev K M (2019103576)**" who carried out the project work under my supervision, for the fulfilment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science and Engineering. Certified further that to the best of my knowledge, thework reported herein does not form part of any other thesis or dissertation on the basis of which a degree or an award was conferred on an earlier occasion on theseor any other candidates.

Place: Chennai

Date: 05/06/2023

Dr. S. Sudha

SUPERVISOR

Professor,
Computer Science and Engineering,
Anna University, Chennai – 600025

COUNTERSIGNED

Dr. S. Valli

HEAD OF THE DEPARTMENT
Computer Science and Engineering,
Anna University, Chennai – 600025

ACKNOWLEDGEMENT

We express our deep gratitude to our guide, **Dr. S. Sudha**, Professor,Department of Computer Science and Engineering for guiding us through every phase of the project. We appreciate her thoroughness, tolerance and ability to share her knowledge with us. We would also like to thank her for her kind supportand for providing necessary facilities to carry out the work.

We express our thanks to the panel of reviewers **Dr. S. Bose**, Professor, Department of Computer Science and Engineering and **Dr. P. Geetha**, Professor, Department of Computer Science and Engineering for their valuable suggestions and critical reviews throughout the course of our project

We are extremely grateful to **Dr. S. Valli**, Professor & Head of theDepartment, Department of Computer Science and Engineering, Anna University, Chennai – 25, for extending the facilities of the Department towards our project and for her unstinting support.

We express our thanks to all other teaching and non-teaching staff who helped us in one way or other for the successful completion of the project. We would also like to thank our parents, family and friends for their indirect contribution in the successful completion of this project.

SRIVATSAV R **SACHIN RAGHUL T** **SANJEEV K M**

ABSTRACT

The current explosion of data has made the development of distributed solutions necessary in order to use machine learning on a greater scale. These distributed learning systems can be centralized or decentralized to varying degrees. In this work, we describe a solution for a general, blockchain-based decentralized federated learning system that can support any machine learning model compatible with gradient descent optimization. Our system design comprises two decentralized actors: clients and checkers. Clients are responsible for training the machine learning models on local data and providing model updates to the checkers. Checkers ensure the accuracy of the model updates and reward or penalize clients accordingly. Our methodology ensures the reliable and efficient operation of the system. Decentralized federated learning offers an experimental sandbox for contrasting the influence of numerous variables on the performance of the entire system. The client-to-checker ratio, the authority consensus policy, and model synchronization techniques are some of these variables. With the help of our tests, we show that a decentralized federated learning system is a practical substitute for more centralized ones. We show that our system can support any machine learning model that is consistent with gradient descent optimization and is a viable alternative to more centralized designs. Our technique guarantees the system's dependable and effective functioning, and our studies shed light on the effects of different aspects on system performance.

திட்டப்பணி சுருக்கம்

சமீபத்திய ஆண்டுகளில் தரவுகளின் பெருக்கம் இயந்திரக் கற்றலை பெரிய அளவில் பயன்படுத்துவதற்கு விநியோகிக்கப்பட்ட தீர்வுகளின் வளர்ச்சியை அவசியமாக்கியுள்ளது. இந்த விநியோகிக்கப்பட்ட கற்றல் முறைமைகளை மையப்படுத்தலாம் அல்லது பல்வேறு அளவுகளில் பரவலாக்கலாம். இந்த வேலையில், பிளாக்செயின் தொழில்நுட்பத்தைப் பயன்படுத்தி ஒரு பொதுவான பரவலாக்கப்பட்ட கூட்டமைப்பு கற்றல் முறைக்கான எங்கள் தீர்வை நாங்கள் முன்வைக்கிறோம், இது சாய்வு வம்சாவளி மேம்படுத்தலுடன் இணக்கமான எந்த இயந்திர கற்றல் மாதிரிக்கும் இடமளிக்கும். எங்கள் கணினி வடிவமைப்பு இரண்டு பரவலாக்கப்பட்ட நடிகர்களைக் கொண்டுள்ளது: பயிற்சியாளர்கள் மற்றும் சரிபார்ப்பவர்கள். உள்ளூர் தரவுகளில் இயந்திர கற்றல் மாதிரிகளைப் பயிற்றுவிப்பதற்கும், வேலிடேட்டர்களுக்கு மாதிரி புதுப்பிப்புகளை வழங்குவதற்கும் பயிற்சியாளர்கள் பொறுப்பு. சரிபார்ப்பாளர்கள் மாதிரி புதுப்பிப்புகளின் துல்லியத்தை உறுதிசெய்து அதற்கேற்ப பயிற்சியாளர்களுக்கு வெகுமதி அல்லது அபராதம் விதிக்கின்றனர். எங்கள் முறையானது கணினியின் நம்பகமான மற்றும் திறமையான செயல்பாட்டை உறுதி செய்கிறது. பிளாக்செயின் மீதான கூட்டமைப்பு கற்றல், ஒட்டுமொத்த கணினி செயல்திறனில் பல்வேறு காரணிகளின் விளைவுகளை ஒப்பிட்டுப் பார்க்க ஒரு

சோதனை சாண்ட்பாக்ஸை வழங்குகிறது. இந்த காரணிகளில் பயிற்சியாளர்-க்கு-சரிபார்ப்பாளர் விகிதம், வெகுமதி-அபராதம் கொள்கை மற்றும் மாதிரி ஒத்திசைவு திட்டங்கள் ஆகியவை அடங்கும். எங்கள் சோதனைகள் மூலம், ஒரு பரவலாக்கப்பட்ட கூட்டமைப்பு கற்றல் முறை மிகவும் மையப்படுத்தப்பட்ட கட்டமைப்புகளுக்கு சாத்தியமான மாற்றாக உள்ளது என்பதை நாங்கள் நிரூபிக்கிறோம். முடிவில், பிளாக்செயின் தொழில்நுட்பத்தை மேம்படுத்தும் ஒரு பரவலாக்கப்பட்ட கூட்டமைப்பு கற்றல் முறைக்கான தீர்வை எங்கள் பணி வழங்குகிறது. இந்த அமைப்பு மிகவும் மையப்படுத்தப்பட்ட கட்டமைப்புகளுக்கு ஒரு சாத்தியமான மாற்றாகும் என்பதை நாங்கள் நிரூபிக்கிறோம் மற்றும் சாய்வு வம்சாவளி மேம்படுத்தலுடன் இணக்கமான எந்த இயந்திர கற்றல் மாதிரிக்கும் இடமளிக்க முடியும். எங்கள் முறையானது கணினியின் நம்பகமான மற்றும் திறமையான செயல்பாட்டை உறுதி செய்கிறது, மேலும் எங்கள் சோதனைகள் கணினி செயல்திறனில் பல்வேறு காரணிகளின் தாக்கத்தைப் பற்றிய நுண்ணறிவுகளை வழங்குகின்றன.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS AND NOMENCLATURE

D-FedL	Decentralized Federated Learning
Fed-Avg	Federated Averaging
CNN	Convolutional Neural Network
BSP	Bulk Synchronous Parallel
SSP	Stale Synchronous Parallel
BAP	Barrierless Asynchronous Parallel
ASP	Approximate Synchronous Parallel
Light client	Clients
FL	Federated Learning
SGD	Stochastic Gradient Descent
MNIST	Modified National Institute of Standards and Technology database

CHAPTER 1
INTRODUCTION

1.1 PROBLEM STATEMENT

A vast amount of research and development has been done in recent years with the goal of speeding up machine learning in every imaginable way. The days of inadequate computer hardware for handling machine learning algorithms are long gone, but it appears that the exploding market for machine learning applications has once again put us in a position where the capabilities of individual machines are being undermined. Using several machines in collaboration to train the current model is one such logical solution. Federated Learning (FL), which Google had conceptualised, was realised. A second significant issue is the lack of data on a single system, which is resolved by the concept of federated learning. This is resolved by working together among many training nodes, each of which uses its own local data and then shares model updates to realise a single model trained on their combined local data.

Federated learning allows for the remote sharing of data by a number of individuals in order to train a single deep learning model collaboratively and iteratively, much like a group presentation or report. Each participant downloads the model, which is often a foundation model that has already been trained, from a cloud datacentre. They train the model using their personal data, then condense and encrypt its updated settings. Returned to the cloud, the model updates are encrypted, averaged, and included into the centralised model. The cooperative training continues iteration after iteration until the model has received all necessary training.

There are three variations of this dispersed, decentralised training approach. The primary model in horizontal federated learning is trained on comparable datasets. The data are complimentary in vertical federated learning; movie and book

evaluations, for instance, are integrated to forecast a person's musical tastes. Finally, federated transfer learning involves training a foundation model that has already been trained to accomplish one job, such as detect automobiles, on a different dataset to do an other task, such as recognise cats. Foundation models are now being included into federated learning by Baracaldo and her coworkers. In one potential usage, banks may train an AI model to spot fraud before using it for different purposes.

Here, we introduce the D-FedL experimental framework for creating federated learning networks on blockchain. We also show how it provides workable answers to issues like privacy, efficiency, and Byzantine fault-tolerance that are typical in decentralised federated learning. D-FedL is a proof-of-concept that is used to create mini-systems that illustrate various features of our solution on a scale that is relatively small. On these systems, we conduct several experiments in later sections to determine the extent of the solution's effectiveness by contrasting its performance with that of a control group for which the relevant features are disabled.

1.2 OBJECTIVES

1. To achieve federated learning over decentralised network which eliminates following issues:
    a. Central node inferring important information
    b. Single point of failure on the server node

2. To prevent malicious clients, who tries to attack the global model with incorrect gradients (Byzantine Imposter)

3. To solve the issue of expensive communication where each training node shares its update with all the other nodes using authority consensus

1.3 CHALLENGES

• Generality: Aside from being compatible with gradient descent, the system design must be independent of the model's characteristics.

• Decentralisation: Decentralised consensus will be used to carry out system politics.

• No Data Sharing: Nodes will never be required to reveal their data, just insights. A distributed learning system has three basic components: synchronisation, degree of centralization, and parallelism.

A. Parallelism

The most time-consuming aspect of the entire machine learning process is, by far and away, training an ML model. In order to parallelize the process, distributed learning typically provides two paradigms: model parallelism and data parallelism, the latter of which was chosen for our system.

B. Degree of Centralization

The totally centralised and fully decentralised extremes of this spectrum are easily discernible. Therefore, a strictly completely centralised system would consist of only one device handling the whole training process from beginning to end. The traditional federated learning method, in which a central server or cluster of servers oversees the training carried out by other nodes, is a more decentralised and less centralised variation of that. We offer a system that is significantly more decentralised.

C. Synchronization

In order to prevent model deterioration owing to out of sync parameters, distributed model updates must take place simultaneously as part of the distributed learning process. There is a crucial trade-off between the velocity of convergence

and the expense of communication, though. In other words, each barrier costs money in terms of network and node bandwidth.

1.4 OVERVIEW OF THE THESIS

Chapter 2 focuses on the discussion of various related work to the project by different authors. The detailed description of the modules involved the proposed system and its implementation details are described in length in Chapter 3. The evaluation of the performance of the proposed system and the other results are discussed in Chapter 4. The proposed model undergoes a comparative analysis with other existing works which is also done in this chapter. Finally, in Chapter 5 the conclusion of the thesis and the discussion of future works are presented.

CHAPTER 2

RELATED WORKS

Blockchain was a well-known technology that tried to achieve decentralized consensus before federated learning was ever thought of. It is simple to understand how federated learning and blockchain may work together to increase decentralization and privacy. Decentralized federated learning has seen a variety of contributions; some of these use blockchain, while others use alternative decentralized protocols. The main goal is to do away with the requirement for a central server to acquire user data and carry out model training. This serves two purposes: it increases privacy while lowering the minimal amount of processing power required by dispersing computation over the network.

2.1 DECENTRALIZED NETWORKING AND SYNCHRONIZATION POLICIES

Besir Kurtulmus, Daniel Kenny et al. [7] discovered that it is possible to create contracts that offer a reward in exchange for a trained machine learning model for a particular data set. This would allow users to train machine learning models for a reward in a trustless manner. Contracts can be created easily by anyone with a dataset, even programmatically by software agents. This creates a market where parties who are good at solving machine learning problems can directly monetize their skillset, and where any organization or software agent that has a problem to solve with AI can solicit solutions from all over the world. This will incentivize the creation of better machine learning models, and make AI more accessible to companies and software agents.

Jakub Konečný, H. Brendan McMahan et al. [6] proposed two ways to reduce the uplink communication costs in implementing federated learning: structured

updates, where we directly learn an update from a restricted space parametrized using a smaller number of variables, e.g. either low-rank or a random mask; and sketched updates, where we the complete model gets update and then it is compressed using a combination of quantization, random rotations, and subsampling before sending it to the server. Both convolutional and recurrent networks show that the proposed methods can reduce the communication cost by two orders of magnitude.

2.2 BLOCKCHAIN AS A INFRASTRUCTIRE TO DECENTRALIZED FEDERATED LEARNING

Youyang Qu, Md Palash Uddin et al. [8] introduced a Blockchain-enabled Federated Learning that has been and will keep generating widespread attention in this big data era. However, the existing paradigms become increasingly impractical and difficult to follow from both perspectives of introductory and in-depth exploration of state-of-the-art models. They focussed on three crucial issues of blockchain-enabled FL, which are decentralisation, incentive mechanism and membership selection, and systematically introduce the evaluation matrix and analysis criteria. In addition, leading attacks have been categorised and the performance of all existing countermeasures is evaluated in a systematic manner.

Sepideh Avizheh, Mahmudun Nabi et al. [1] designed a decentralized resource sharing platform that uses a permissioned blockchain to allow users to share their digital items with their specified attributed-based access policies that are enforced through a set of smart contracts, and removes the need for a trusted intermediary. The system however allows user's accesses to be traced, and has limited availability as access to a resource requires its owner to be on-line. This decentralized attribute-based access control system achieves the same functionality

while preserving the privacy of user's access, and automating access which removes the need for the resource owner to be online.

Micah J. Sheller, Brandon Edwards et al. [9] investigated the effects of data distribution across collaborating institutions on model quality and learning patterns, indicating that increased access to data through data private multi-institutional collaborations can benefit model quality more than the errors introduced by the collaborative method. Finally, they compared with other collaborative-learning approaches demonstrating the superiority of federated learning, and discussed practical implementation considerations. Clinical adoption of federated learning is expected to lead to models trained on datasets of unprecedented size, hence having a catalytic impact towards precision/personalized medicine.

2.3 BYZANTINE FAULT TOLERANCE PROTOCOL

Marco Benedetti, Francesco De Sclavis et al. [2] explored one possible way to solve the problem where the Distributed Ledger Technologies, when managed by a few trusted checkers requires most but not all of the machinery available in public DLTs. Hence, a combination of a modified Practical Byzantine Fault Tolerant protocol and a revised Flexible Round-Optimized Schnorr Threshold Signature scheme is devised, and then the resulting proof-of-authority consensus algorithm is injected into Bitcoin, replacing its PoW machinery. The combined protocol may operate as a modern, safe foundation for digital payment systems and Central Bank Digital Currencies.

Anna Bogdanova, Akie Nakai et al. [3] explored an federated learning system that enables integration of dimensionality reduced representations of distributed data prior to a supervised learning task, thus avoiding model sharing among the parties. They compared the performance of this approach on image classification tasks to three alternative frameworks: centralized machine learning, individual machine

learning, and Federated Averaging, and analyze potential use cases for a federated learning system without model sharing. The Results show that this approach can achieve similar accuracy as Federated Averaging and performs better than Federated Averaging in a small-user setting.

Isiten Gorkey, Chakir El Moussaoui et al. [4] discovered the private blockchains use permissioned blockchain consensus algorithms as the participants need the permission of the authority to be able to join the system and investigated permissioned and permissionless blockchains and focus on permissioned blockchains to analyze it in terms of, e.g. trust models between the nodes, incentives, number of nodes & parties involved, and scalability regarding the number of transactions.

Jianping He, Lin Cai et al. [5] addresses to solve the challenges in privacy preserving data aggregation by exploiting the distributed consensus technique. They first proposed a secure consensus-based DA algorithm that guarantees an accurate sum aggregation while preserving the privacy of sensitive data, then proved that the proposed algorithm converges accurately. Extensive simulations have shown that the proposed algorithm has high accuracy and low complexity, and they are robust against network dynamics.

Stacey Truex, Nathalie Baracaldo et al. [10] proposed a Hybrid Approach to Privacy-Preserving Federated Learning by Combining differential privacy with secure multiparty computation which enables to reduce the growth of noise injection as the number of parties increases without sacrificing privacy while maintaining a pre-defined rate of trust. Hence, the system is therefore a scalable approach that protects against inference threats and produces models with high accuracy. Additionally, this system can be used to train a variety of machine learning models, which demonstrate that our approach out-performs state of the art solutions.

CHAPTER 3

SYSTEM DESIGN AND IMPLEMENTATION

3.1 SYSTEM ARCHITECTURE

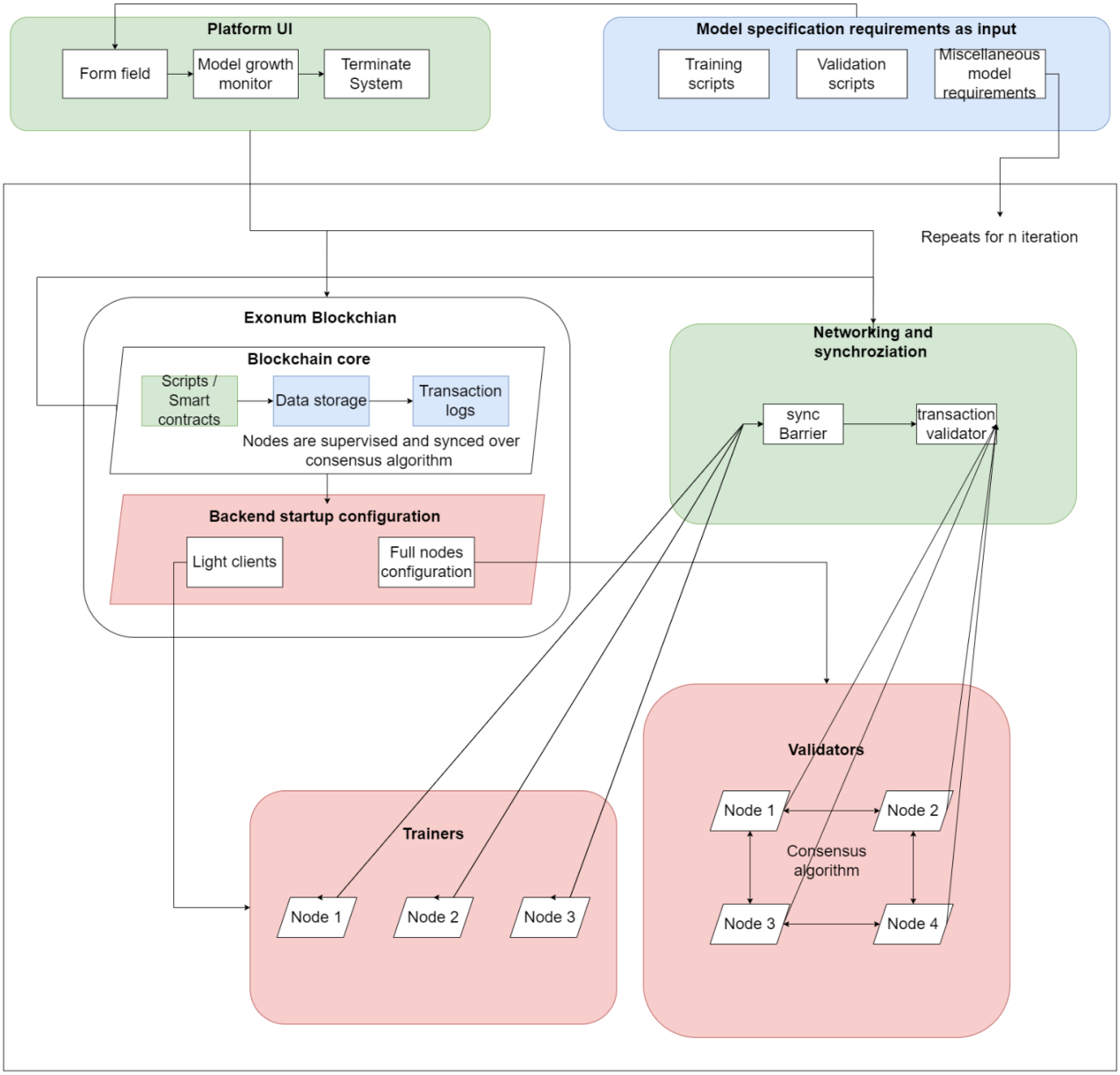


Figure 3.1. SYSTEM ARCHITECTURE

LEGEND:

Green - Proposed Enhancements

Red – Modifications

Blue – Existing Methodology

3.2 PROPOSED SYSTEM

- By employing blockchain technology, we propose Decentralised Federated Learning (D-FedL) framework (Figure 3.1) to create a generic decentralised federated learning system that can support any machine learning model that is appropriate for gradient descent optimisation.

- D-FedL employs proof-of-concept to spawn checkers and clients, two small systems. The process for guaranteeing the system's dependable and effective operation, combined with the system design that includes two decentralised actors: the client and checker.

- Hence, D-FedL is used as an experimental sandbox to compare and contrast the effects of the client-to-checker ratio, authority consensus policy, and model synchronisation schemes on the performance of the overall system, ultimately demonstrating by way of example that a decentralised federated learning system is in fact a workable alternative to more centralised architectures.

3.3 LIST OF MODULES

1) Model Specification and platform UI
2) Blockchain core and startup configuration
3) Networking and synchronization
4) Model clients
5) Model checkers

3.3.1 Model Specification and platform UI

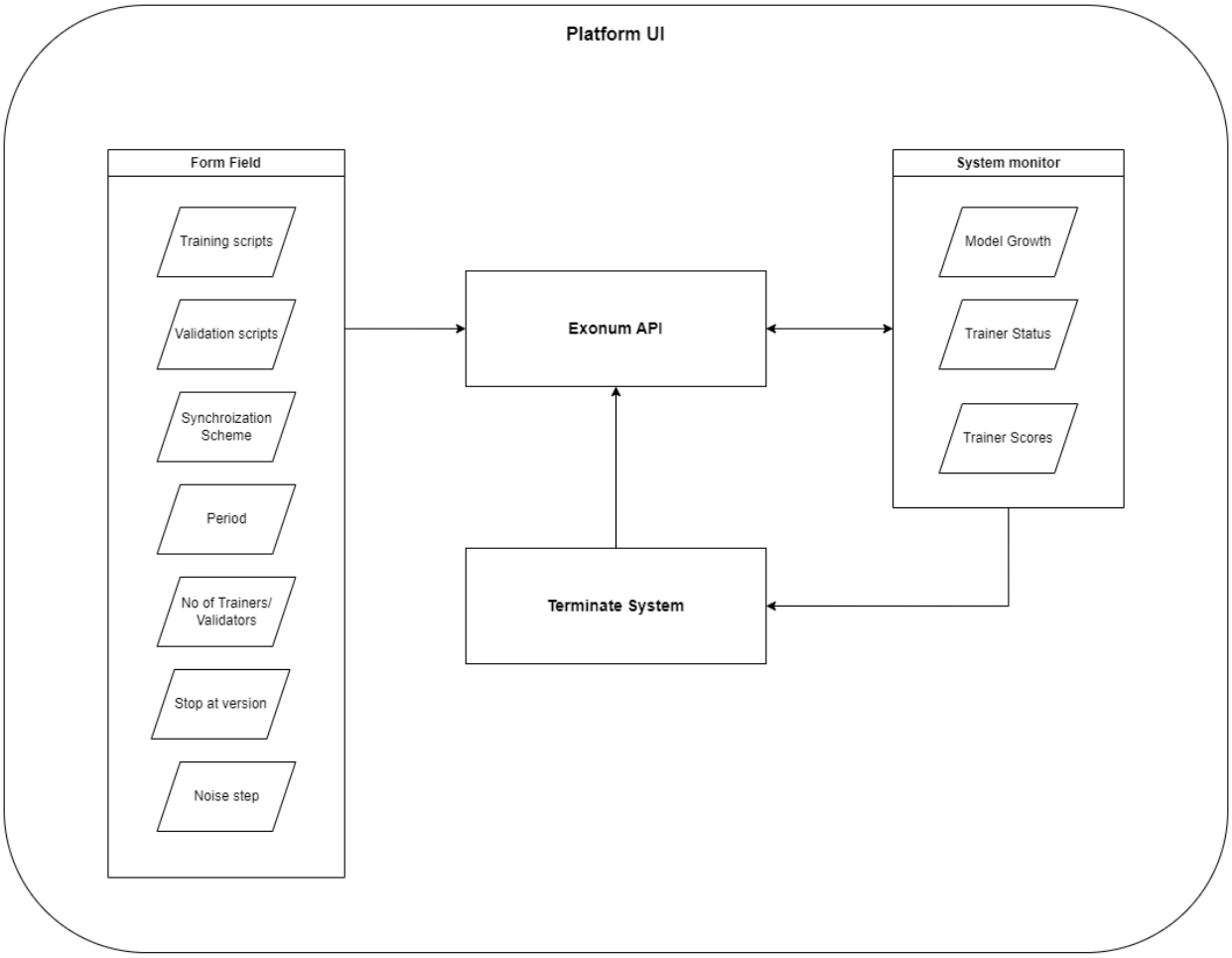


Figure 3.2. MODULAR VIEW OF PLATFORM UI

INPUT : Training and validation Scripts and Number, Synchronization scheme, Noise Step

OUTPUT : Forwards to startup configuration

Figure 3.2 shows the complete design diagram of Platform UI. The client and the checker are the two decentralised actors that make up the proposed system. The checker uses a consensus approach to guarantee that the trained model is accurate while the client is in charge of training the machine learning model on local data. The suggested approach is that uses methods like data sharing, peer selection, and reputation-based validation to guarantee the system operates dependably and effectively.

We also do experiments on D-FedL to examine the effects of different elements on system performance, including the client-to-checker ratio, authority consensus policy, and model synchronisation strategies.

Therefore, we must obtain the scripts so that we may distribute them across clients and checkers and train on the local dataset using them. Additionally, the system may be set up by the user to run the model with a set number of clients and checkers, choose the synchronisation scheme (BAP, BSP, SSP), stop at a certain version, and take a noise step.

Additionally, the frontend is set up with a system monitor that can keep tabs on client scores, model growth, and client status.

Pseudocode for Creation of User Interface

Import React, Component, and necessary components from reactstrap.
Define class InitializationPage.
In constructor, initialize state with default values (syncScheme, modelName, clients, checkers, period, version, noise).
Bind methods : handleChange and handleSubmit to the class.
Define handleChange to take an event parameter.
Retrieve name and value of input element that triggered the event.
If (input_name = "modelName") :
    extract model name from path of selected file,
    update the state.
Set state with updated name and value of input element.
Define handleSubmit to take an event parameter.
Prevent default form submission behavior.
Parse values(clients, checkers, period, version, noise).
If !(clients and checkers >= 1 and (typeof() = int))
    show alert and return.

If (syncScheme = BSP or SSP) ? validate period >= 1 and (typeof() = int) : show an alert and return.

If (syncScheme = BAP and period >= 0)

show an alert(period will not be used for BAP).

Construct URL for system spawning process, including the parsed values(clients, checkers, syncScheme, period, version, noise).

Create options for fetch request and send GET request to the constructed URL.

Show alert(system spawning is in progress).

Call startPolling() on props with appropriate parameters.

Define render() to return JSX that renders form for user input.

Bind form inputs to state properties using value and onChange attributes.

Render form inputs in card component with appropriate labels and styling.

Pseudocode for Graph Creation and Analysis

Import required modules and components from React and other libraries.

Define chart1_2_options object with various chart options.

Define GrowthGraph class component that extends Component class.

Define default props for component.

Implement render method of the component.

Define data function that returns data to be used for chart.

Use data function to create a Line chart using Line component from react-chartjs-2 library.

Define layout and content of Card component that wraps chart.

Export GrowthGraph component as default export.

3.3.2 Blockchain core and startup configuration

Role of blockchain in privacy preserving

INPUT : Parameters for Environment Creation
OUTPUT : Checkers and Clients Environment

Blockchain was a well-known technology before federated learning was thought of that aimed at achieving decentralised consensus. It is simple to understand the potential benefits of federated learning and blockchain integration for enhancing privacy and decentralisation. Decentralised federated learning has seen a number of contributions; some use blockchain, while others use other decentralised protocols.

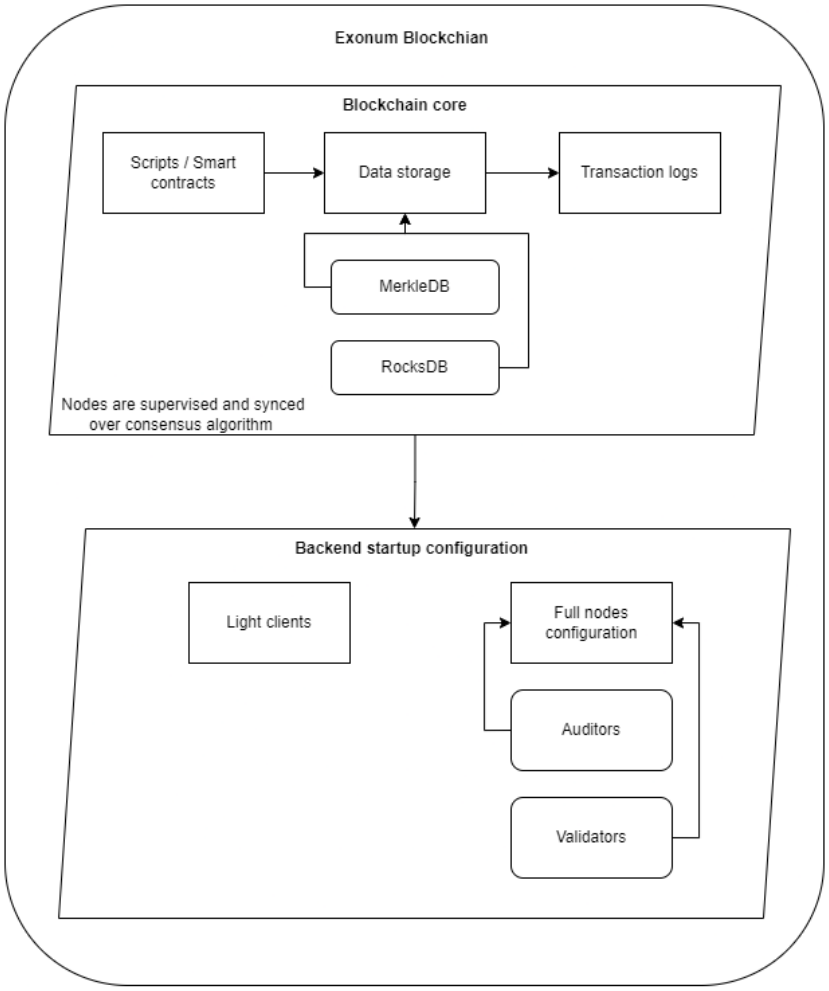


Figure 3.3. MODULAR VIEW OF EXONUM BLOCKCHAIN SETUP AND CONFIG

Figure 3.3 shows the complete design diagram of Blockchain core and startup configuration. The main goal is to do away with the requirement for a central server to collect user data and carry out model training. This serves two purposes: first, it increases privacy; second, by dispersing computing across the network, it relaxes the minimum amount of computational power necessary. Some of the current contributions use an All-Reduce strategy, where each training node communicates its updates with every other node in the system, resulting in a communication cost of O(n2) for n training nodes. Network topology is a crucial factor to take advantage of in order to lower communication costs. There are plans to share the node's model updates with its one-hop neighbours using the ring topology, tree topology, and graph topology.

All of these methods, however, necessitate many hops for the updates to reach every node in the system, which causes sluggish convergence.

Other contributions use the gossip protocol, which relies on the fact that each peer sends its updates to another peer in the network, disseminating the information throughout the entire network, to achieve the same goal of lowering communication overhead. It could be argued that this places an excessive amount of duty on the client to verify updates made by peers. In order to address this, our design introduces checker nodes (similar to bitcoin miners), which reduce the computational burden on clients and create a more transparent and traceable trust structure. Finally, and most importantly for our study, some previous work leverages blockchain as the communication backbone for decentralised federated learning.

Blockchain is used to speed up uploading and tracking updates, pay users for taking part in model training or validation, and make updates secure and immutable. Smart contracts function as the core server in centralised federated learning. The node conducts local training, after which the local model updates are forwarded to an elected consensus committee for verification and scoring. Each training node is allocated a checker that evaluates said client's modifications, computes proof of

work (PoW), and rewards the client appropriately. The modified global model is then published to the blockchain. The miner is compensated after the updates are recorded on the blockchain.

An analogous process, but the benefits are supplied by the model owner rather than the checker or the blockchain, and the client is free to communicate its modifications to any checker. To save the model checkpoints, we can use a distributed peer-to-peer file sharing system. In contrast, proof of stake (PoS) is used in our architecture as a considerably more portable substitute for cryptographic PoW. Additionally, because our methodology is general in nature, it is not restricted to a particular job or goal, provided the underlying network meets the necessary consensus assumptions.

Here, we employ the Exonum platform, which combines a service-oriented architecture with the Rust programming language to achieve the highest level of execution safety, as well as client-side verification based on cryptographic commitments (Merkle and Merkle Patricia trees) to guarantee system transparency and client security.

Pseudocode for Exonum Setup

Generate template for Exonum demo app using the common.toml config file and specify number of checkers to create.
Generate config files for each checker node using the common.toml config file.
Specify address of each node and directory to store the generated config files.
Finalize the config files generated in previous step by specifying
    public and private API addresses,
    public configuration files for each node.
Run checker node using config files generated in step 2.
Change current directory to frontend directory.
Install required dependencies for the frontend application.
Build the frontend application.
Start the frontend application, specifying the port to listen on and the API root URL.

Exonum

A blockchain framework called Exonum enables the creation of safe permissioned blockchain applications. Exonum has its unique collection of features and capabilities, just like every piece of software. This article lists the scenarios where Exonum might be helpful and highlights the key distinctions between Exonum and other distributed ledger technologies.

Exonum blockchain functions as a key-value storage and an online transaction processing service for an outside application. Processing transactions, preserving data, and responding to read requests from external clients are its three main operations.

The primary thing that Exonum deals with is transactions. An atomic patch for the key-value storage that should be applied is represented by a transaction. Digital signatures with a public key are used to authenticate transactions. Before a transaction is deemed approved or committed, it must be ordered and validated. The consensus method handles ordering and guarantees that only correctly validated transactions are committed.

The execution of transactions is influenced by a collection of variable parameters found in each transaction template. These parameters are also used to serialise transactions for network transmission and persistence. (Therefore, transactions and stored processes in RDBMSs might be compared.) Services define transaction templates and the processing guidelines for each template. Services, in particular, specify how transactions are applied to the key-value storage and the rules for transaction verification.

All data in the Exonum blockchain is divided into two parts:

Data storage, which contains data structured into objects. Objects represent high-level wrappers over the key-value store

Transaction log, i.e., the complete history of all transactions ever applied to the data storage.

The actual state of the data storage can be completely recovered from the transaction log since transactions often involve operations on the key-value storage, such as adding new values or changing previously saved values. As soon as a new node joins the Exonum network, it loads previously created blocks and applies each transaction one at a time to the data storage. Such a strategy makes auditing easier and allows for the full history of every data chunk to be seen.

State machine replication is carried out by Exonum by means of a transaction log. It ensures that network nodes agree on the states of data storage. Distributed DBs that are not blockchain-based, like PostgreSQL and MongoDB, frequently employ the same strategy.

Blocks

Exonum groups transactions into blocks, and each block is atomically accepted. A transaction is not regarded as accepted if it has not yet been written to any block. Each transaction in a block is carried out sequentially after it has been approved, and the data storage is updated.

Exonum blocks contain the following parts:

- The hash of the previous Exonum block
- The list of transactions that were approved. Each transaction is carried out by the nodes in the block's execution in the specified order, and their data storages are updated. The relevant Exonum service executes each sort of transaction.

- The new data storage state's hash. Although the state itself is excluded, transactions are applied deterministically and without hesitation. The Exonum consensus algorithm includes an agreement on the data storage hash, thus the hash will always be the same for all checkers.

It is impossible to alter one block without also making the necessary modifications to each of the following blocks since every block contains the hash of the block before it. This guarantees the transaction log's immutability; once a transaction is committed, it cannot be changed retrospectively or removed from the log. In the same way, inserting a transaction in the midst of the log is not possible.

Smart Contracting

Similar to how smart contracts function on other blockchain platforms, interfaces provided by services also serve this purpose. They specify the blockchain's business logic, enable data retrieval from the blockchain, and are reusable across several projects. Endpoints of REST web services and stored procedures for database management systems are two partial analogues for this execution model.

The key points differentiating Exonum smart contracts from other models used in blockchains are as follows:

Evolution is a top priority. Exonum supports the entire service lifecycle, including service updates, asynchronous data migrations (which adhere to the fundamental migration process for RDBMS), temporarily suspending a service, etc. The core guarantees invariants throughout lifecycle events, facilitating secure and simple service modifications.

Restricted execution environment. Exonum only executes preset request types by default; it does not permit the execution of erroneous client-supplied code. As a result, the setting is more controlled, which facilitates discussion about smart contract safety. At the same time, the supervisor service contains and is completely adjustable of the logic to update the blockchain contracts. Therefore, if the client code complies with the constraints imposed by the blockchain configuration, it can be deployed on an Exonum blockchain.

Flexible API support. Runtimes can introduce support for completely new APIs without the core having to do so (or even being aware of it).

Flexible runtime isolation. Although runtimes might be isolated within themselves, they are not separated from the system core. (For instance, by utilising a virtual computer like the Java runtime.) As a result, it is possible to execute contracts in the sandbox along with high-performance business logic. Exonum enables having both simultaneously because different runtimes may exist on the same blockchain.

Local state. Exonum services have the ability to specify a local state that is unique to the node that they are currently running on. It is possible to manage confidential data, such as private keys, through the local state. Private service endpoints may be in charge of managing the local state. Services can be more proactive than those in other blockchains by making use of the local state. For instance, the anchoring service fully automates anchoring transaction signing by using the local state.

Network Structure

Peer-to-peer connections between full nodes and light clients make up the Exonum network.

Full Nodes

Full nodes replicate the entire contents of the blockchain and correspond to replicas in distributed databases. All the full nodes are authenticated with public-key cryptography. Full nodes are further subdivided into 2 categories:

The blockchain's complete content is replicated by auditors. They can create new transactions, but they are unable to decide which of those transactions should be committed, or to create new blocks.

The network's liveness is provided by checkers. By employing a Byzantine fault tolerant consensus algorithm, only checkers are able to create new blocks. Transactions are received, verified, and added to a new block by checkers. Network administrators have placed restrictions on the list of checkers, which typically consists of 4–15 nodes.

Light Clients

In the client-server paradigm, light clients stand in for clients; they connect to full nodes to make and receive transactions as well as get data from the blockchain that interests them. A "proofs mechanism" is offered by Exonum and is based on cryptographic commitments made using Merkle and Merkle Patricia trees. With the use of this technique, it is possible to confirm that a response from the complete node has in fact been approved by a supermajority of checkers.

3.3.3 Networking and Synchronization

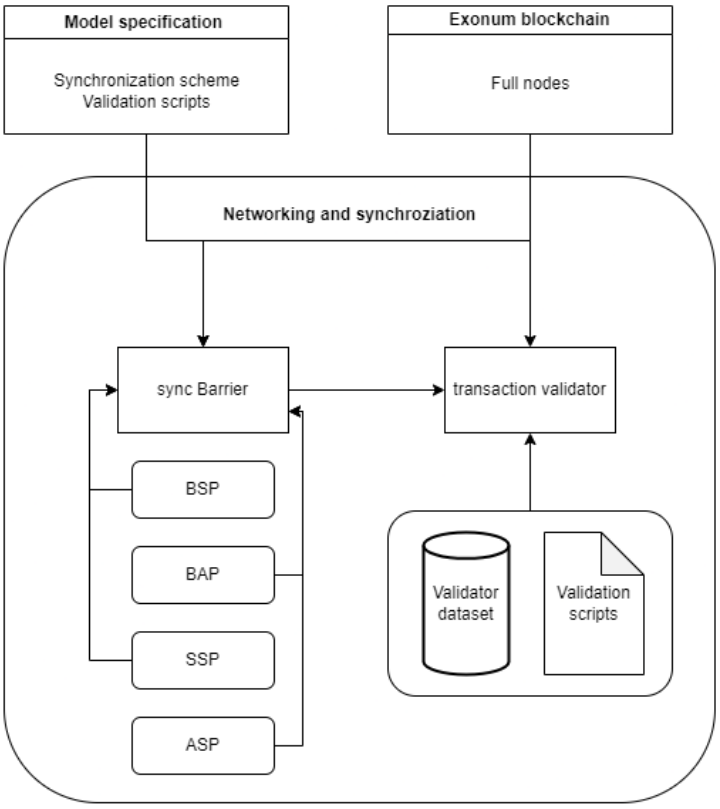


Figure 3.4. MODULAR VIEW OF NETWORKING AND SYNCHRONIZATION

INPUT : Sync updated parameters from clients and validates with checkers
OUTPUT : Commits the transaction logs in exonum blockchain

Figure 3.4 shows the complete design diagram of Networking and Synchronization Controller Plane.

Synchronization

In order to prevent model degradation owing to out of sync parameters, distributed model updates must take place simultaneously as part of the distributed learning process. The idea of a synchronisation barrier is applicable to the scenario of a distributed learning system, just like it is to many other parallel and massively parallel paradigms. Each barrier in this scenario would denote a round of update-

sharing during which new information is combined to create an updated model. The model converges more quickly and suffers less degradation the more frequently synchronisation obstacles arise. There is a crucial trade-off between the velocity of convergence and the expense of communication, though. In other words, each barrier costs money in terms of network and node bandwidth. The primary concept is that the system is divided into two different phases: a computation phase and a communication phase for exchanging the results of the calculation. The next subsections introduce the most popular methods for handling node update synchronisation, which are many.

Bulk Synchronous Parallel (BSP): Abbreviated as BSP, it is widely acknowledged as the simplest method for maintaining consistency by alternating between rounds of computation and transmission. Although the best consistency of this model makes it the fastest converger, its fundamental drawback is that finished nodes must wait for all worker nodes to complete their computation at each synchronisation barrier. In our adaptation of BSP, every training round has a predetermined time restriction that concludes with a deadline after which no additional client updates are taken into account for that round.

Stale Synchronous Parallel (SSP): In contrast to BSP, this model has more lenient synchronisation barriers that permit an additional number of repetitions. All employees take a break after that buffer has been achieved. It has good convergence insurances in low to moderate staleness, below which convergence rates begin to degrade, given its nature as a compromise to BSP. In our design, SSP is modelled as BSP with the option of a deadline extension. This extension would be a portion of the initial round duration decided in proportion to the ratio of clients who have not yet concluded training for that round, known as the slack ratio. Clients who completed by the deadline are permitted to train for a maximum of N additional steps before the deadline extension expires.

Barrierless Asynchronous Parallel (BAP): In contrast to BSP, this model allows waitless, asynchronous communication between nodes, which achieves a very low overhead (hence, higher speedups), but it suffers from potentially slow and even incorrect convergence with increased delays. Clients can advance their local training in our version of BAP for as long as a round may last, sharing it occasionally with a checker, up till a new model is issued. When a minimum ratio of labour has been provided, a new model is issued, and clients should then request the new model.

Approximate Synchronous Parallel (ASP): The concept is still interesting even though our system does not support ASP. In contrast to the SSP approach, ASP is more focused on limiting parameter precision than staleness. To avoid the synchronisation costs associated with updates that are not important, it accomplishes this by ignoring them. The difficulty in determining which characteristics are unimportant as well as the complexity of implementation are drawbacks of the strategy.

Byzantine Fault Tolerance

Decentralised systems are perhaps the most vulnerable to Byzantine behaviour, and there are countless scenarios and circumstances that could occur. Due to the fact that malicious checker behaviour is largely handled by the decentralised consensus of the blockchain, in this study we concentrate on lazy and malicious client behaviour. We use an authoritative consensus policy to address this, whereby each client i is given a trust score (or reputation), such that:

$$0 \leq \phi_i \leq 1$$

$$\sum_{i=1}^{n} \phi_i = 1$$

The weight factor for a client update in the federated learning algorithm is the client's trust score. That has a double effect. First, it gives checkers the ability to manage the influence of the client's updates on the model according to the level of trust. Second, it fosters internal competition in the system because each increase in one client's score effectively causes a fall in the scores of other clients. Based on the outcomes of the validation, trust scores are modified. After receiving a client update, a checker computes its validation score by adding its gradients to the most recent model weights. Depending on whether it results in an increase or decrease, it then changes the corresponding client's trust score.

Pseudocode for Synchronization Barrier

Import the libraries: exonum-client, proto, and utils.
Set default period = 60 seconds.
If (command line argument), base period, else the default.
Define URL for explorer API.
Define service_id and transaction_id.
Create a new instance of class : Transaction with the schema, serviceId, methodId.
Define a sleep function that takes time (sec) and returns promise after that time.
Wait till base period.
while (true)

Create new_payload with a rand_seed.

Create a new transaction with SyncBarrier and payload, signed with generated key pair.

Serialize the transaction.

Send transaction to explorer API with timeout of 1sec for connection and 3sec for response.

Get slack_ratio using the get_slack_ratio function, and wait.

*If slack_ratio > 0, update current_period = base_period * slack_ratio, round it. else, use base_period.*

Wait for current_period.

3.3.4 Model Clients

INPUT	: Client scripts
OUTPUT	: Updated model parameters

Figure 3.5 shows the complete design diagram of Client module. Model training is carried out, together with any necessary intermediate transformations like model flattening and rebuilding at the client side. This layer, which is developed on top of Exonum's lightweight clients, transmits and receives flattened gradients and models.

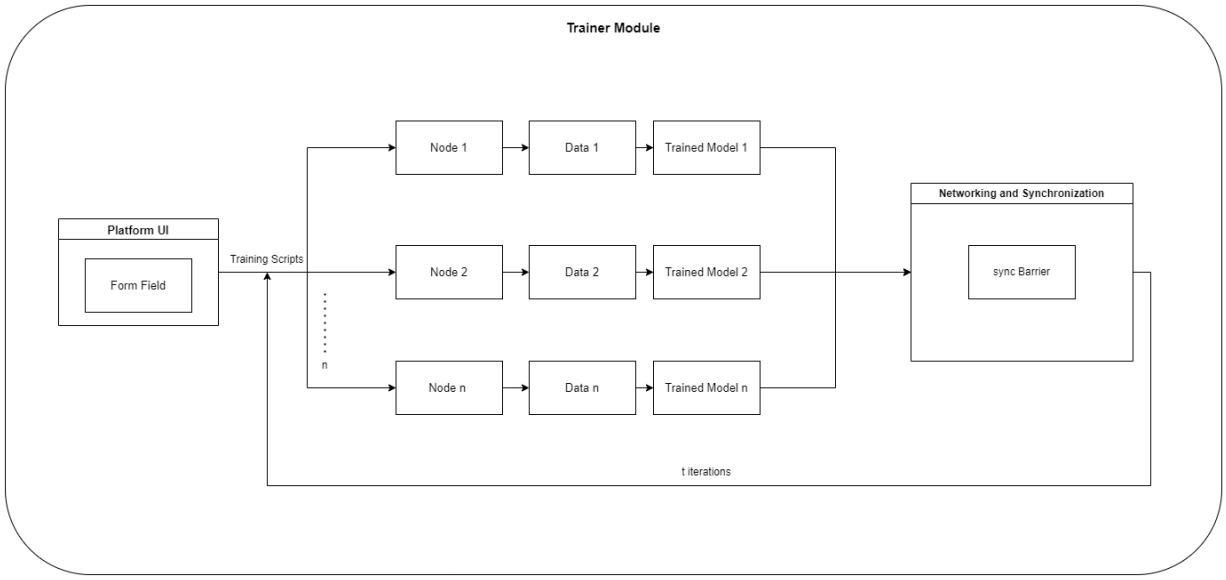


Figure 3.5. MODULAR VIEW OF CLIENT MODULE

Exonum comes with a light client to enhance system auditability. Front-end developers can make advantage of the assistance functions in the Light client JavaScript package. Using cryptographic proofs, these assistance functions are used to validate client-side blockchain answers.

The client functions are divided into the following submodules:

- **Data**. Functions for serialization and deserialization of data from the JSON format to Protobuf binary format
- **Cryptography**. Functions for calculating hashes, creating and validating digital signatures
- **Proofs**. Functions for verifying cryptographic proofs that the blockchain has returned, such as the Merkle and Merkle Patricia indices' proofs.
- **Blockchain integrity checks**. Function for checking the validity of a block, that is, its compliance with consensus algorithm

There are two typical use cases for the light client:

- Forming and sending transactions to the Exonum blockchain network
- Constructing requests to network complete nodes (often HTTP GET requests) and vetting the responses

Transaction Creation

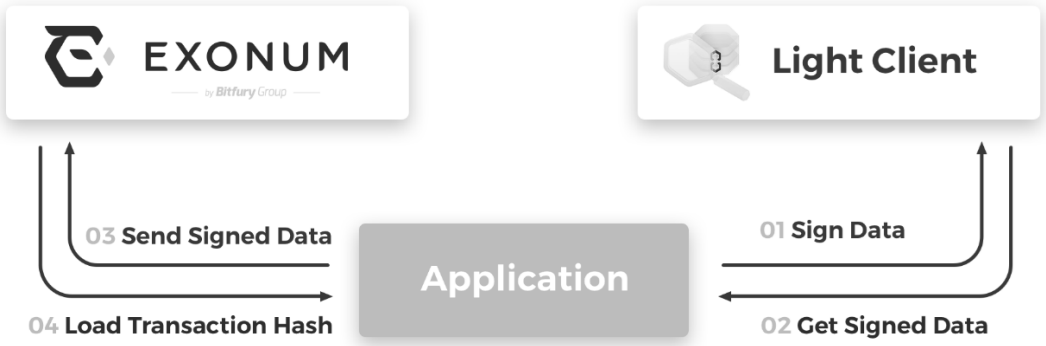


Figure 3.6. CREATING TRANSACTIONS IN EXONUM

- Figure 3.6 shows the state view of transaction creation in exonum. A new transaction is started when the front-end application is triggered, such as when a button is clicked. JSON format is used to store the transactional data. Finally, the

data is digitally signed using the lite client library and transformed to the protobuf binary format.

- Frontend application receives a digital signature from the light client library.
- A complete node receives the created transaction (JSON data plus the digital signature) via an HTTP POST request.
- In response to the HTTP POST request, the frontend application receives a notice (such as the transaction hash).

Sending Requests

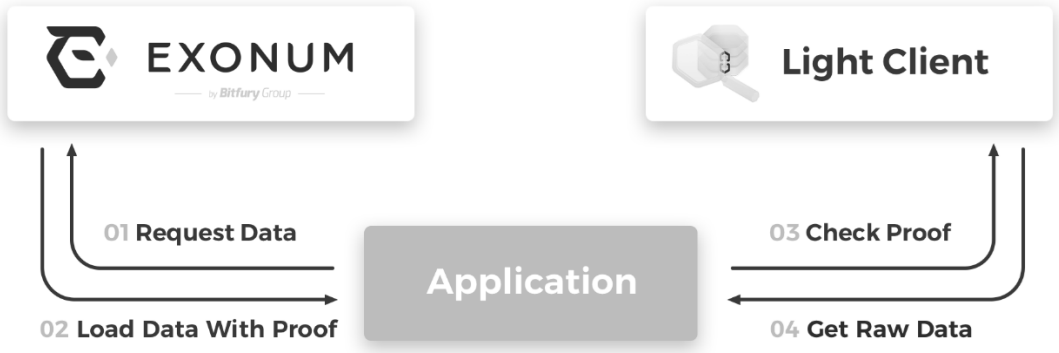


Figure 3.7. SENDING REQUESTS IN EXONUM

- Figure 3.7 shows the state view sending requests in exonum. The client forms an HTTP GET request and sends it to a full node in the Exonum blockchain network.

- The node delivers the client a response to the request along with the appropriate cryptographic evidence. The block header of the cryptographic proof is accompanied by Precommit messages certifying its authenticity, one or more Merkle routes connecting the response to the block header, and one or more Precommit messages.

- The client validates the response's cryptographic proofs and its structure after receiving the response from the blockchain. An aspect of the verification process is determining whether a provided response is outdated. This is done

by computing the median timestamp recorded in the Precommits and comparing it to the client's local time.

- If the median time in Precommits is too far in the past, the response is considered stale, and its verification fails.

- The result of checks and the data retrieved from the full node is shown in the user interface

An example of the cryptographic proof:

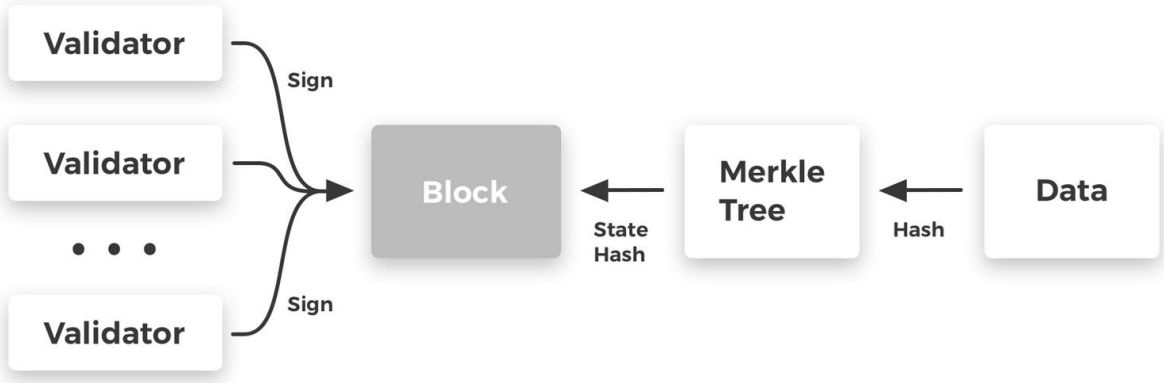


Figure 3.8. CRYPTOGRAPHIC PROOF

Blockchain suggests the potential for auditing or validating stored data as a technology that enhances the capabilities of distributed databases. There are various approaches to conduct this audit.

Figure 3.8 shows the demonstration for cryptographic proof. An audit can be carried out automatically when a full node is in operation in a system where full nodes are present (nodes that contain the complete copy of a blockchain). A full node checks the consistency of the consensus mechanism and the accuracy of transaction execution when a new block is received. If carried out by outside, impartial parties, such an audit by specialised auditor nodes may be successful. But it has a lot of shortcomings:

To perform audit, one needs a full copy of a blockchain and enough computational resources to stay in sync with the blockchain checkers.

In order to guarantee that the system state is accurate, this kind of audit must be ongoing. A full auditor node must climb to the current blockchain height in order to begin auditing. Especially if the blockchain has a high transaction flow, this could take a while.

Since he lacks the ability to independently confirm the accuracy of blockchain data, a system end user is obliged to rely on pair checkers and auditors. In actuality, from the user's perspective, such a blockchain is equivalent to a distributed database.

Introducing light clients, sometimes referred to as lightweight clients, thin clients, or merely clients, is one technique to alleviate the shortcomings of audit by specialised auditor nodes. These clients are also referred to as SPV (simple payment verification) clients for the Bitcoin blockchain. Light clients are computer programmes that can reproduce and validate a small subset of the data kept on the blockchain. Clients typically check details pertaining to a particular user (such as his transaction history). The usage of the Merkle and Merkle Patricia indices as particular data containers in a blockchain makes this verification possible.

Since their functions are distinct, the presence of light clients does not imply the lack of auditor nodes. While auditor nodes evaluate a blockchain as a whole, light clients only validate the data of a specific user.

The TLS security checks that web browsers incorporate could be compared to light client security. It doesn't completely replace the auditing done by auditor nodes, but it does offer a measureable level of protection from MitM attacks and maliciously acting nodes that the client might interact with. In addition, if light clients are used for all blockchain transactions, their combined security may match that of the auditor nodes.

A blockchain-based system's inclusion of light clients causes the following issues to arise during development:

Client developers and backend developers should agree on API requests and the structure of cryptographic proofs. (This is mostly handled by Protobuf being used as the serialisation format.)

Changes to the blockchain data architecture should be accompanied by pertinent adjustments to the logic used by light clients to verify proofs.

Despite the intricacy of the development, the only practical solution to largely eliminate the need for trust in third parties in a blockchain-based system is the presence of light clients.

Pseudocode for client module

Import necessary libraries : exonum-client, proto.
Import modules to fetch and manipulate data.
Define constants and variables :
    URL for the exonum blockchain,
    service_id and method_id of transaction for sharing updates,
    time interval between consecutive training sessions.
Define trainNewModel() with input as paths, and variables of model and its weights.
Retrieve new gradients from training dataset.
Apply trainNewModel() to the current model by adding the dataset to the existing weights.
Send updated model back to the server by creating and submitting a new transaction on the blockchain.
Define timeout() to suspend the script for specified amount of time.
Define randomizeDuration() to generates a random number between 0 and 6, and assign it to intervalDuration.

Define main() to fetch client keys.

while (true) :

Randomly select a duration for training,

Wait for that duration,

Train the model using the trainNewModel().

Check if the previous training session has finished, and whether there is any new data available for training.

3.3.5 Model Checkers

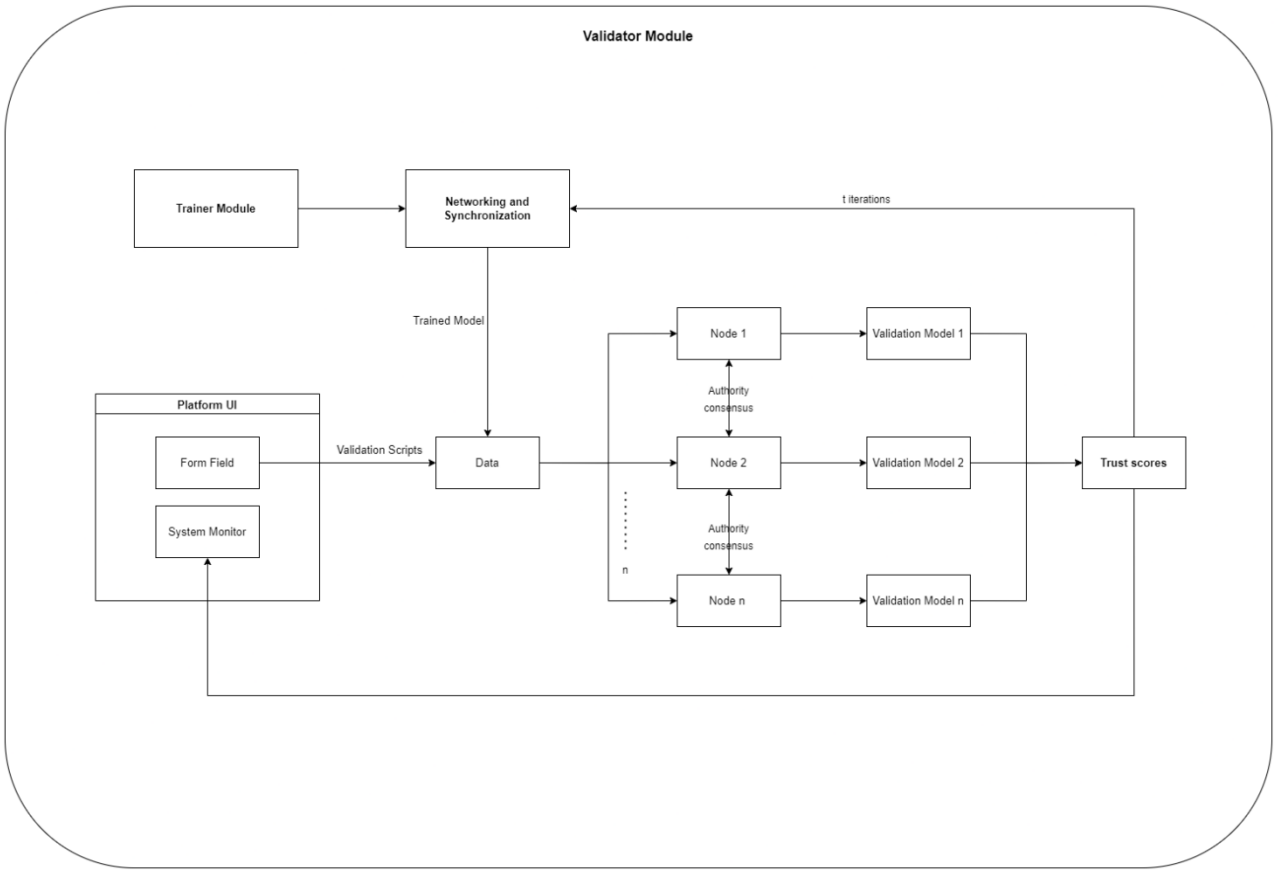


Figure 3.9. MODULAR VIEW OF CHECKER MODULE

INPUT	: Updated model parameters and client scripts
OUTPUT	: Trust scores

A strictly fully centralised system would consist of just one device handling the entire training process from beginning to end. Figure 3.9 shows the complete design diagram of Checker module. The traditional federated learning method, in which a central server or cluster of servers oversees the training carried out by other nodes, is a more decentralised and less centralised variation of that. Here, we describe a system that is significantly more decentralised. Our system has two types of nodes, client and checker, the latter of which is comparable to a crypto-miner, just as the majority of cryptocurrency networks. On the underlying blockchain network, checkers are in charge of maintaining models through decentralised consensus based on voting.

A checker's vote on the acceptability of a model update (made by a client node) should ideally be determined by validation on information that is presumed hidden from the client. This necessitates the following conditions and presumptions in order for the system to function properly, given that consensus is based on voting.

More than two thirds of checkers are not Byzantine.

- To obtain a meaningful consensus, the datasets of the several checkers should be sufficiently positively connected.

  At least a portion of the underlying network is synchronous.
- The quantity of collectively usable data utilised during training and validation is a limiting factor on the quality of models generated by the system.

The choice of the gradient descent (or any other optimisation approach) variation that fits the way the system is structured is one important consequence of the centralization level of the system. We will only take into consideration the distributed centralised version and its fully decentralised adversary because the subject of this study is distributed learning. Since using traditional gradient descent for such a distributed system is difficult, we will instead utilise a stochastic variant of it to estimate the genuine gradient while keeping in mind the pace at which this approximation converges to the true value.

Our approach use Federated Averaging to carry out parallel stochastic gradient descent (P-SGD), much like the majority of federated learning systems. It is based on the premise that several nodes training on various datasets and aggregating or averaging their updates onto a single model would eventually converge to a reliable global model.

The data flow from each high-level module to the others can be seen by taking a closer look within a checker. In order to decide whether to accept or reject the update onto the blockchain, clients first contact with checkers through HTTP, whose endpoint then transfers the transactions to validation module. The validation module uses the validation dataset to validate the model-update transactions. In addition, Checkers provide a Wire API that may be used for system diagnostics and monitoring to respond to questions about the blockchain state, such as "What is the most recent model version?" The service-level view, on the other hand, places emphasis on the division of issues and duties throughout the system into several services.

As the primary interface for implementing persistent modifications, the storage schema service serves as the hub for all other checker-side services. The validation service is in charge of evaluating client work and presenting the results to the reputation service, which can then alter trust scores accordingly. The ML service is in charge of integrating client updates and managing the client flow (e.g., synchronisation). In addition to the primary blockchain interfaces used for transmitting transactions, the Wire APIs on the sides provide auxiliary querying interfaces for various uses (such as accessing the most recent model version).

One significant characteristic is that while checkers are fully networked, clients are not. It is important to note that while though the roles (client or checker) are not fixed or devoted, a node cannot play both parts concurrently for the same subnetwork. However, a node can play the checker role for one subnetwork while playing the client role for a different one. Additionally, at least one checker is necessary for each model subnetwork to operate.

Pseudocode for Checker Module

Define a function called validation() that does the following:

Decode transaction from command line argument as TxShareUpdates message.

Extract the min_score field from transaction.

store (encoded_gradients_vector) from transaction and obtain path of vector.

Call validate_vector() function with path to encoded vector and a callback function.

In callback function:

clear(encoded vector).

If (validation is successful) ? print "VALID" : print "INVALID".

Fetch the latest model using fetchLatestModel() function.

In the promise resolution callback function:

Fetch min_score required for validation by fetchMinScore().

Read the file specified by the command line argument.

Remove "[","]" and use split(","), then convert resulting string array to int array.

Decode resulting array as TxShareUpdates message.

Obtain validation_id from command line argument.

store (encoded_gradients_vector) with store_encoded_vector() from transaction and obtain path of vector.

If (latest_model = 0):

Call validate_vector() function with 1 as the first argument, an empty string as the second argument, the encoded_vector_path, the command line argument, the min_score, and callback function.

In callback function:

clear(encoded vector) and clear(encoded_model).

If (validation is successful) ? print "VALID" : print "INVALID".

print(score).

else :

store (encoded_vector) with store_encoded_vector() and obtain path of vector.

Call validate_vector() function with 0 as the first argument, the encoded_model_path, the encoded_vector_path, the second command line argument, the min_score, and callback function.

In callback function:

clear(encoded vector) and clear(encoded_model).

If (validation is successful) ? print "VALID" : print "INVALID".

print(score).

3.4 IMPLEMENTATION

1) Blockchain Setup (Exonum):

Exonum is written in Rust and depends on the following third-party system libraries:

- RocksDB (persistent storage)
- libsodium (cryptography engine)
- Protocol Buffers (mechanism for serializing structured data)

Prerequisites:

- git
- Node.js with npm
- Rust compiler
- Exonum launcher

Dependencies installation:

For distributives with deb-based package managers (such as Debian or Ubuntu), use

```
add-apt-repository ppa:exonum/rocksdb
apt-get update
apt-get install build-essential jq libsodium-dev libsnappy-dev libssl-dev \
    librocksdb6.2 pkg-config clang-7 lldb-7 lld-7
```

For `protobuf` installation add the following dependencies:

```
add-apt-repository ppa:maarten-fonville/protobuf
apt install libprotobuf-dev protobuf-compiler
```

Package names and installation methods may differ in other Linux distributives; use package manager tools to locate and install dependencies.

Depending on the version of your distributive, libsodium, RocksDB and Protobuf may not be present in the default package lists. In this case you may need to install these packages from third-party PPAs, or build them from sources.

Adding Environment Variables

If your OS contains pre-compiled rocksdb or snappy libraries, you may setup ROCKSDB_LIB_DIR and/or SNAPPY_LIB_DIR environment variable to point to a directory with these libraries. This will significantly reduce compile time.

```
export ROCKSDB_LIB_DIR=/usr/lib
export SNAPPY_LIB_DIR=/usr/lib/x86_64-linux-gnu
```

Rust Toolchain

Exonum repositories use the stable Rust toolchain that can be installed by using the **rustup** program:

```
curl https://sh.rustup.rs -sSf | sh -s -- --default-toolchain stable
```

Compiling Exonum

We can verify that you installed dependencies and the Rust toolchain correctly by cloning the `exonum` repository and running its built-in unit test suite:

```
git clone https://github.com/exonum/exonum.git
cd exonum
cargo test -p exonum
```

Necessary dependencies(Cargo.toml):

Required dependencies (Figure 3.10.1 and 3.10.2) and configurations are written on Cargo.toml file in rust framework similar to yarn.json in node.

```
[package]
name = "cryptocurrency"
version = "0.1.0"
edition = "2018"
authors = ["Your Name <your@email.com>"]

[dependencies]
exonum = "1.0.0"
exonum-crypto = "1.0.0"
exonum-derive = "1.0.0"
exonum-proto = "1.0.0"
exonum-rust-runtime = "1.0.0"
```

Figure 3.10.1. NECESSARY DEPENDENCIES

```
failure = "0.1.5"
protobuf = "2.8.0"
serde = "1.0"
serde_derive = "1.0"
serde_json = "1.0"

[build-dependencies]
exonum-build = "1.0.0"
```

Figure 3.10.2. NECESSARY DEPENDENCIES

Generate node configuration template:

```
mkdir example
exonum-<project-name> generate-template \
  example/common.toml \
```

```
  --checkers-count 4
```

Generate public and secret keys for each node:

```
exonum-<project-name> -advanced generate-config \
  example/common.toml example/1 \
  --peer-address 127.0.0.1:6331 -n
exonum-<project-name> generate-config \
  example/common.toml example/2 \
  --peer-address 127.0.0.1:6332 -n
exonum-<project-name> generate-config \
  example/common.toml example/3 \
  --peer-address 127.0.0.1:6333 -n
exonum-<project-name> generate-config \
  example/common.toml example/4 \
  --peer-address 127.0.0.1:6334 -n
```

Finalize configs:

```
exonum-<project-name> finalize \
  --public-api-address 0.0.0.0:8200 \
  --private-api-address 0.0.0.0:8091 \
  example/1/sec.toml example/1/node.toml \
  --public-configs example/{1,2,3,4}/pub.toml
```

Similar commands for other 3 nodes, with adjusted paths and socket addresses

Run nodes:

```
exonum--<project-name> run \
  --node-config example/1/node.toml \
  --db-path example/1/db \
  --public-api-address 0.0.0.0:8200 \
  --master-key-pass pass
```

Similar commands for other 3 nodes, with adjusted paths and socket addresses

2) Checker (Exonum Blockchain)

Filling Requirements in Spawn Page :

Frontend is connected to node server via REST api. The necessary system configuration is collected from end user via form (Figure 3.11.1). All the requirements to spawn the system is passed to node server which will run script to initiate client and checker nodes.

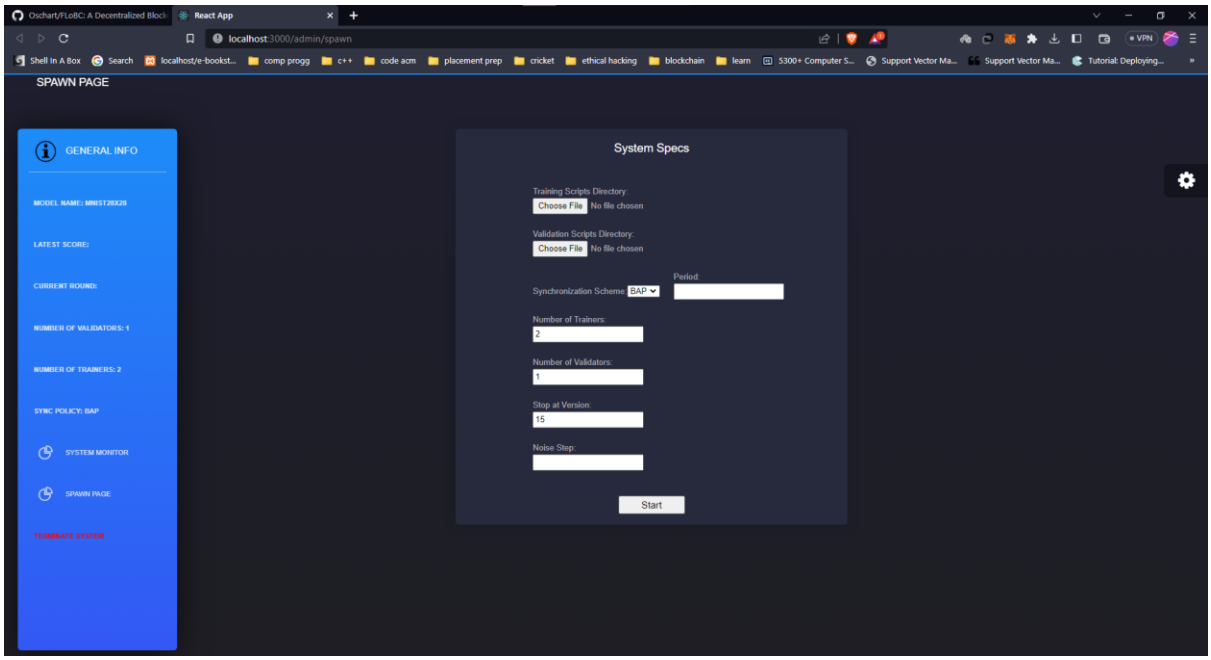


Figure 3.11.1. Filling the form fields

Starting Checker Node :

```
exonum-ML generate-template common.toml --checkers-count ${node_count}
     for i in $(seq 0 $((node_count - 1))); do
        peer_port=$((start_peer_port + i))
        exonum-ML generate-config common.toml $((i + 1)) --peer-address
127.0.0.1:${peer_port} -n
    done
```

Backend node spawn script starts the mentioned number of checkers (Figure 3.11.2) and the checker nodes starts receiving for model updates by the clients (Figure 3.11.3).

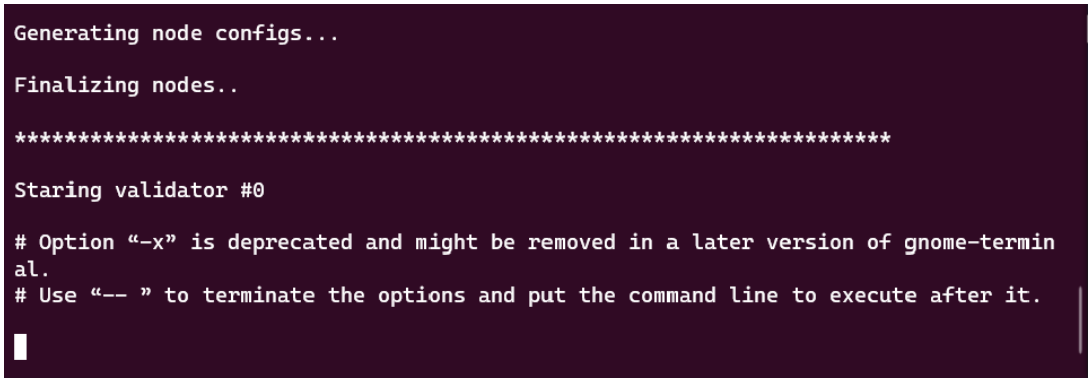


Figure 3.11.2. STARTING CHECKER #0

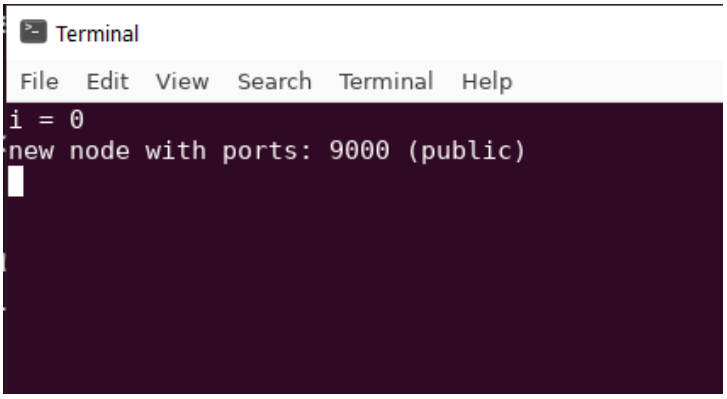


Figure 3.11.3. CHECKER CONSOLE

3) Client (Exonum Blockchain) :

Starting Client Node :

```
mode=$(($1))
rm ModelMetadata
npm start -- 9000 "models/test_model/data.csv" $mode
```

Backend node spawn script starts the mentioned number of trainers (Figure 3.12.1) that trains the model with local data and sends it to the validator node (Figure 3.12.2).

```
1

'./lightclient' -> './/lightclient1'
'./lightclient/.babelrc' -> './/lightclient1/.babelrc'
'./lightclient/models' -> './/lightclient1/models'
'./lightclient/models/MNIST20X20' -> './/lightclient1/models/MNIST20X20'
'./lightclient/models/MNIST20X20/apply.py' -> './/lightclient1/models/MNIST20X20/ap
ply.py'
'./lightclient/models/MNIST20X20/metadata' -> './/lightclient1/models/MNIST20X20/me
tadata'
'./lightclient/models/MNIST20X20/training_script.py' -> './/lightclient1/models/MNI
ST20X20/training_script.py'
'./
lightclient/models/MNIST20X20/utils.py' -> './/lightclient1/models/MNIST20X20/utils
.py'
'./lightclient/models/MNIST28X28' -> './/lightclient1/models/MNIST28X28'
'./lightclient/models/MNIST28X28/metadata' -> './/lightclient1/models/MNIST28X28/me
tadata'
'./lightclient/models/MNIST28X28/training_script.py' -> './/lightclient1/models/MNI
ST28X28/training_script.py'
'./lightclient/models/MNIST28X28/utils.py' -> './/lightclient1/models/MNIST28X28/ut
ils.py'
'./lightclient/models/MNIST28X28/__pycache__' -> './/light
```

Figure 3.12.1. STARTING LIGHT CLIENT #1

```
Terminal
File  Edit  View  Search  Terminal  Help

> lightclient@1.0.0 start /root/FLoBC/lightclient1
> babel src -d dist && node dist/app.js "9000" "models/MNIST28X28/data.csv" "0"
"MNIST28X28"

src/app.js -> dist/app.js
src/proto.js -> dist/proto.js
src/utils/fetchClientKeys.js -> dist/utils/fetchClientKeys.js
src/utils/fetchDatasetDirectory.js -> dist/utils/fetchDatasetDirectory.js
src/utils/fetchLatestModel.js -> dist/utils/fetchLatestModel.js
src/utils/fetchPythonWeights.js -> dist/utils/fetchPythonWeights.js
src/utils/generateNormalNoise.js -> dist/utils/generateNormalNoise.js
src/utils/parsePythonList.js -> dist/utils/parsePythonList.js
src/utils/store_encoded_vector.js -> dist/utils/store_encoded_vector.js
Will pause for 1 secs
http://127.0.0.1:9000/api/services/ml_service/v1/models/latestmodel
New model released by the validator!, #-1
```

Figure 3.12.2. LIGHT CLIENT #1

Version manager (Figure 3.12.3) is used to manage the current version of the released model with the target version

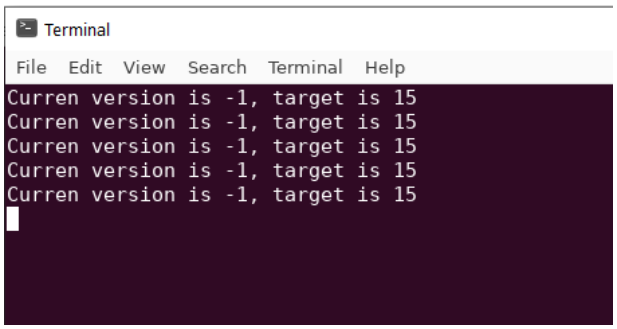


Figure 3.12.3. VERSION MANAGER

Backend Environment:

Total consolidated backend environment (Figure 3.12.4) at any given part of time

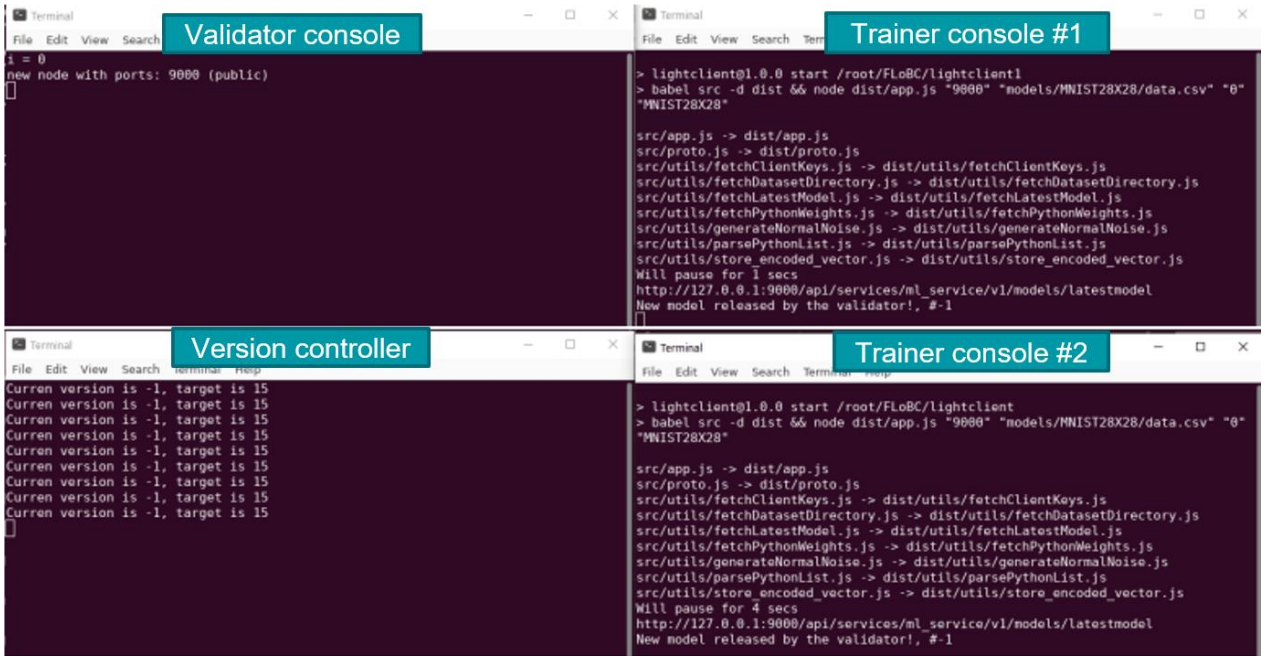


Figure 3.12.4. CONSOLIDATED BACKEND ENVIRONMENT

Model Released by Client :

```
async function main(){
    await fetchClientKeys()
    .then((client_keys) => {
        CLIENT_KEY = client_keys
    });

    while(1){
        randomizeDuration();
        console.log("Will pause for " + intervalDuration + " secs")
        await timeout(intervalDuration);
```

```
        if(!can_train){
            console.log("training is in progress")
        }
        else{
            await fetchLatestModelClient(CLIENT_KEY.publicKey, MODEL_LENGTH)
            .then(async fetcherResult => {
                let newModel = fetcherResult[0];
                let isLocallyCached = fetcherResult[1];
                let firstIteration = fetcherResult[2];
                if (newModel !== -1){
                    if (can_train){
                        can_train = false;
                        let newModel_path = store_encoded_vector(newModel);
                        await trainNewModel(firstIteration, newModel_path, newModel,
isLocallyCached)
                    }
                }
                else{
                    console.log("No retrain quota at the moment, will retry in a bit")
                }
 })}}}
```

Client trains the model with local data and shares the model parameters with the checkers, asynchronously. Clients are allowed to train for as long as a round may extend; that is, until a new model is released, clients can keep advancing local training, periodically sharing it with a checker. A new model is released when a minimum ratio of labor has been submitted (Figure 3.13), and that's when clients should pull in the new model.

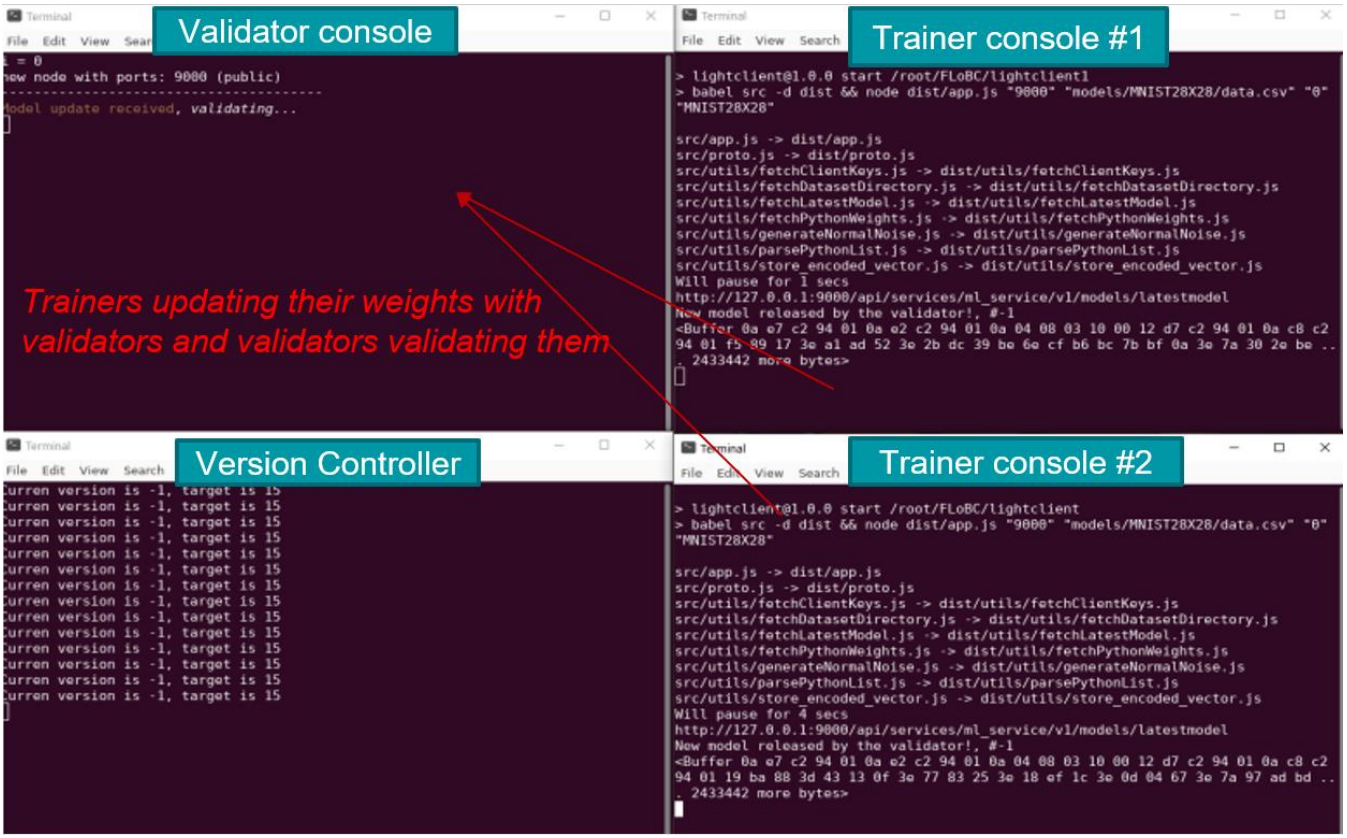


Figure 3.13. CLIENT RELEASING THE MODEL

Validating Model :

```
MODELS_DIR = '../src/models/'
model_id = sys.argv[6]
data_dir = MODELS_DIR + model_id + '/data.csv'

def parse_gradients(gradients_path):
    gradients = open(gradients_path, "r").readline()
    split = gradients.split("|")
    split = [float(element) for element in split]
    return np.array(split)

def send_valid(is_valid):
    verdict = 'valid' if is_valid else 'invalid'
    print("VERDICT" + verdict + "ENDVERDICT")

def send_score(score):
    print("SCORE" + str(score) + "ENDSCORE")

gradients = parse_gradients(sys.argv[4])
newModel_flag = str(sys.argv[1])
if (newModel_flag == "true"):
    newModel_flag = 1
elif (newModel_flag == "false"):
    newModel_flag = 0
else:
    newModel_flag = int(newModel_flag)

if newModel_flag:
    evaluate_model = gradients
else:
    base_model = parse_gradients(sys.argv[3])
    evaluate_model = base_model + gradients

min_score = float(sys.argv[5])
score = model_mod.compute_validation_score(evaluate_model, data_dir)
is_valid = score >= min_score

send_valid(is_valid)
send_score(score)

if max(gradients >= 1.0):
    send_valid(False)
else:
    send_valid(True)
```

The model released by the client is validated by the checker using validating sample to check for any Byzantine attack (Figure 3.14). If the model accuracy is greater than the threshold accuracy then new model version is released by aggregating the previous version with the current version using aggregator function. Then the new aggregated model is synced across other checkers and clients. If the current version reaches the target version then the model is available for download.

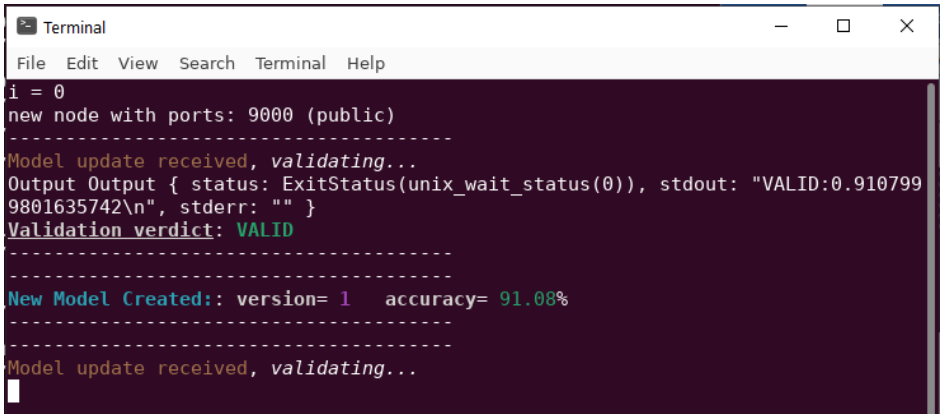


Figure 3.14. CHECKER VALIDATING THE MODEL

Environment Endpoints :

```
pub fn wire(builder: &mut ServiceApiBuilder) {
        builder
            .public_scope()
            .endpoint("v1/models/info", Self::model_info)
            .endpoint("v1/models/getmodel", Self::get_model)
            .endpoint("v1/models/clientsscores", Self::get_clients_scores)
            .endpoint("v1/models/latestmodel", Self::latest_model)
            .endpoint("v1/models/getmodelaccuracy", Self::get_model_accuracy)
            .endpoint("v1/sync/slack_ratio", Self::get_slack_ratio)
            .endpoint("v1/client/retrain_quota", Self::get_retrain_quota)
            .endpoint("v1/client/client_status", Self::get_client_status)
            .endpoint("v1/client/all_clients_status", Self::get_all_clients_status);
    }
```

Releasing Model :

New model resulting from aggregating the models from the trainers is released into the blockchain with the respective model version number (Figure 3.15.1).

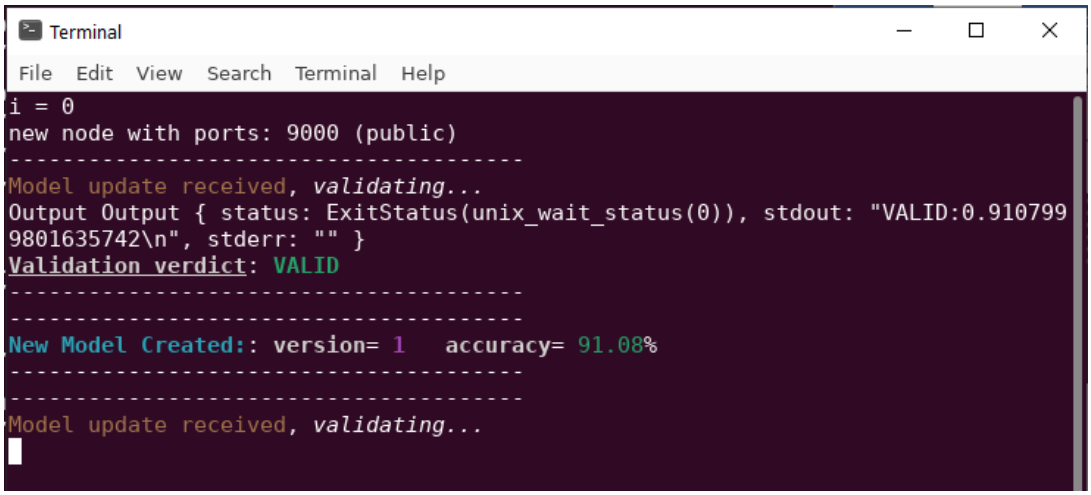


Figure 3.15.1. CHECKER CREATING NEW MODEL

Releasing Model :

New model release metadata (Figure 3.15.2) is simultaneously picked up by the frontend REST API endpoints and corresponding metadata information is displayed in the same like model growth graph, trainers status, trainers scores (Figure 3.15.3).

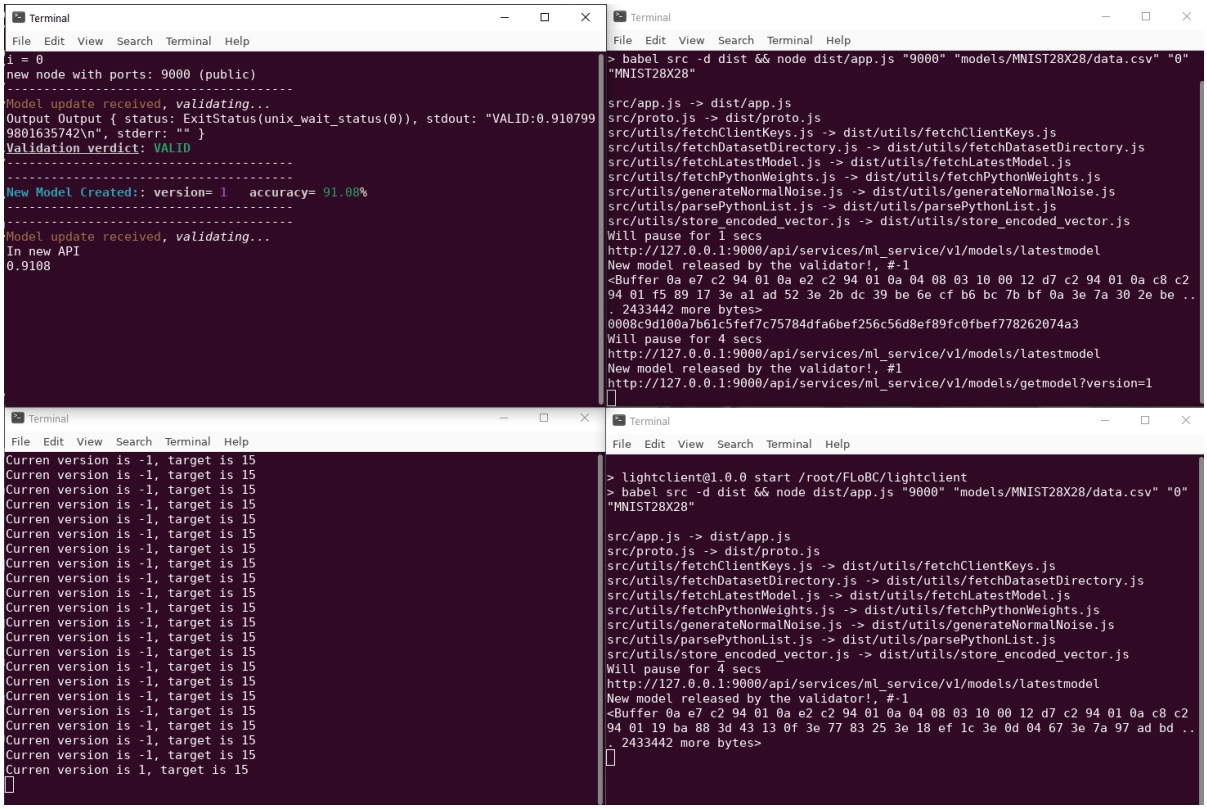


Figure 3.15.2. NEW MODEL SYNCHRONISED ACROSS CLIENTS

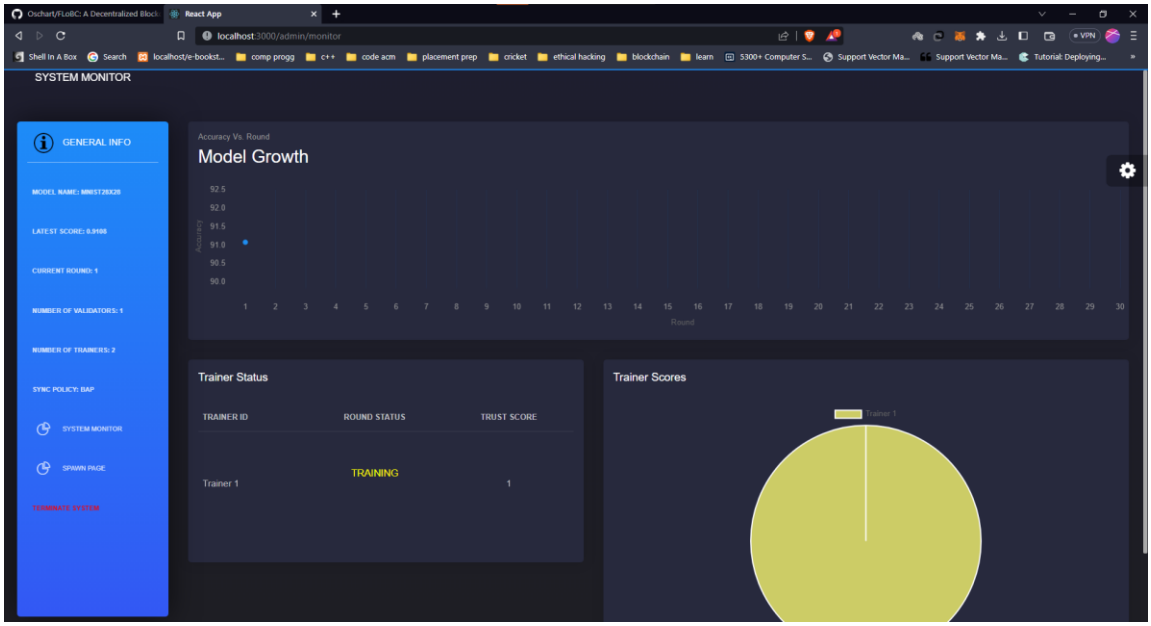


Figure 3.15.3. NEW MODEL REFLECTED IN DASHBOARD

Frontend visualization of the model growth for each iteration:

The framework trains the model for mentioned number of threshold iterations and for reach iteration new model is created and shared across trainers for training on it and resultant meta data is updated on frontend for each iteration (Figure 3.15.4).

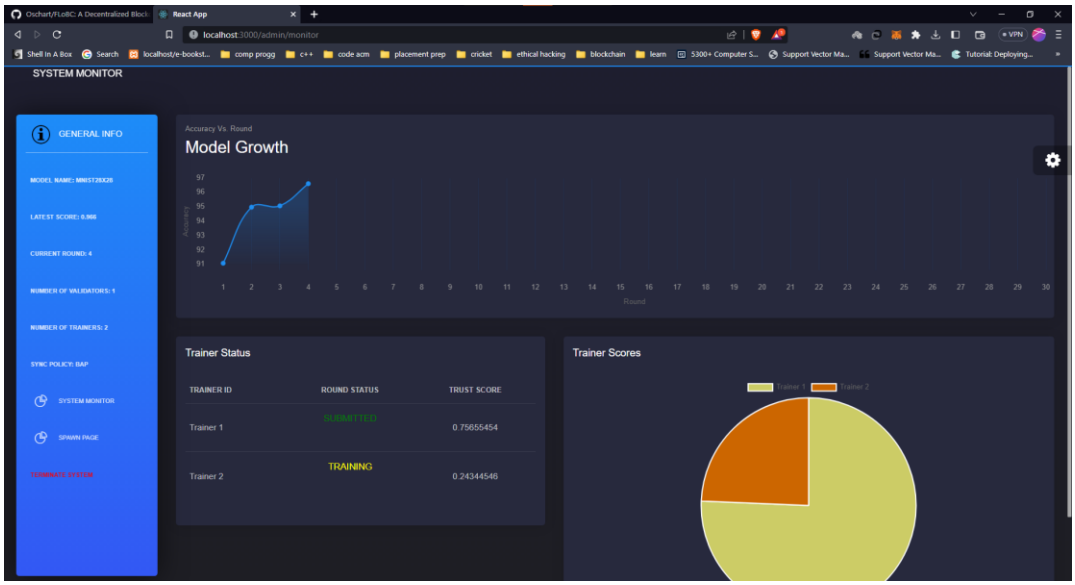


Figure 3.15.4. MODEL VERSION #4

Final model growth graph:

Final state of the frame work after threshold no of iterations and resultant growth of the model accuracy (Figure 3.15.5).

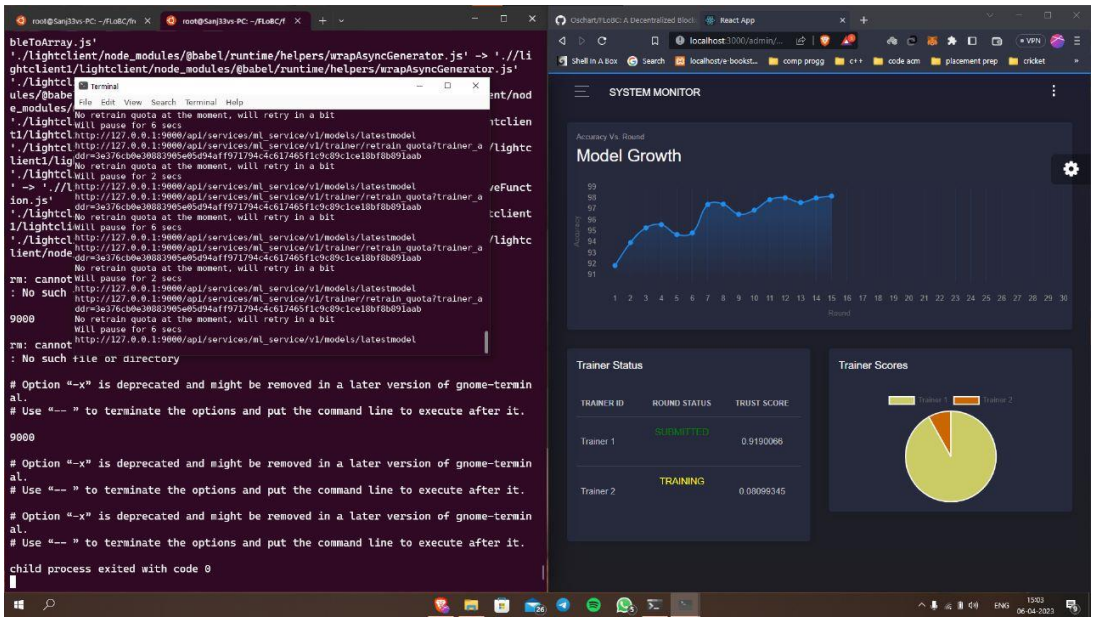


Figure 3.15.5. FINAL MODEL GROWTH GRAPH VERSION #15

Final model weights at version no 15:

The final model weights after threshold iterations (Figure 3.15.6) and the final updated model can be downloaded from backend REST API

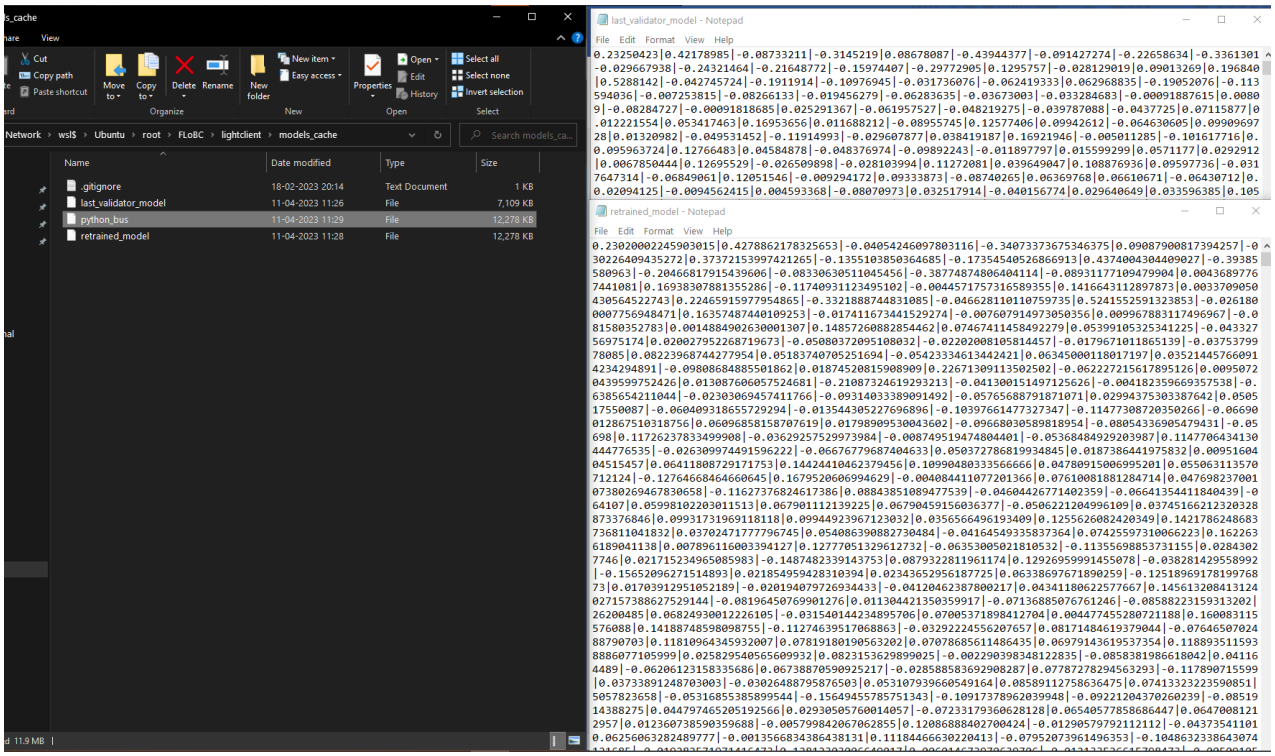


Figure 3.15.6. FINAL MODEL WEIGHTS VERSION #15

CHAPTER 4

RESULTS AND DISCUSSION

4.1 EXPERIMENTAL SETUP

All of the experiments in this section included training and verifying a convolutional neural network (CNN) to forecast handwritten MNIST digits from pictures with a 20 by 20 pixel resolution from the standard MNIST dataset. An independent 30% sample of the MNIST training dataset is used by each client in the system. In these studies, accuracy is the measure used to assess training, however the system will function equally well with any other machine learning model metric. The Python machine learning validation script allows for simple integration of the measure.

4.2 TEST CASES

1)	When Clients and Checkers are less than or equal to 0
2)	When period is 0 and the scheme is synchronous
3)	When stop at version is left null or is filled with float value
4)	Force Terminating System
5)	Synchronization policy
6)	Sync Signal Propagation
7)	Sending Model Parameters
8)	Validating Model Parameters
9)	Releasing New Model
10)	Reflecting model update in frontend

Table 4.2 TEST CASES

Case 1 (When Clients and Checkers are less than or equal to 0)

Input: Clients and checkers are <= 0

Output: Alert prompt appears if clients and checkers are less than or equal to 0 (Figure 4.2.1)

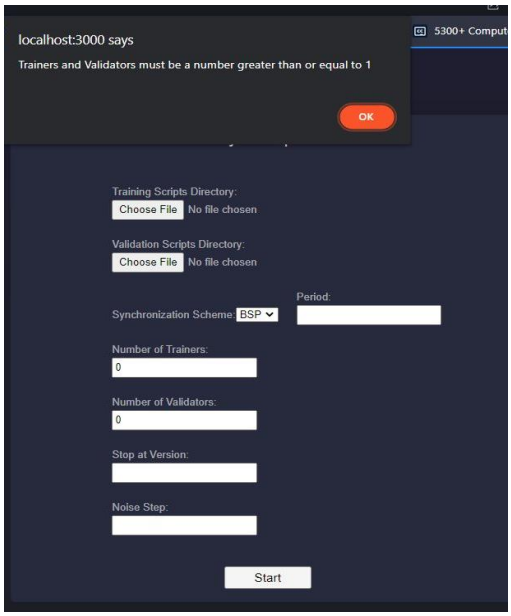


Figure 4.2.1. PROMPT ARAISES IF CONSTRAINTS ARE VIOLATED [CLIENTS AND CHECKERS ARE NULL]

Case 2 (When period is 0 and the scheme is synchronous)

Input: Synchronous scheme and period is 0

Output: Alert prompt appears if period is 0 and scheme is synchronous (Figure 4.2.2 and 4.2.3)

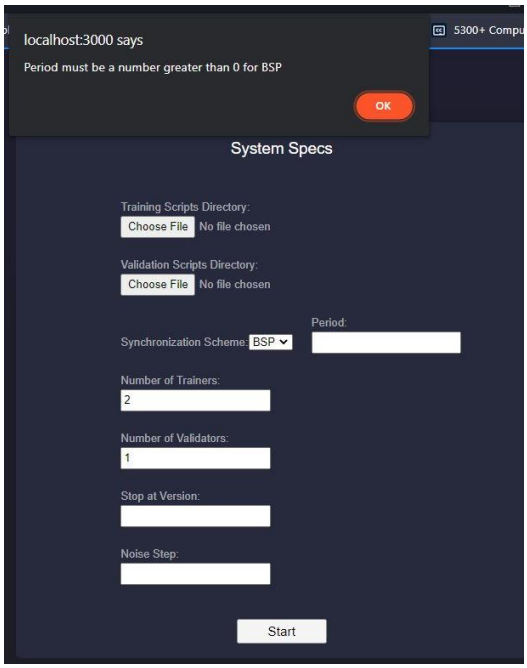


Figure 4.2.2. PROMPT ARAISES IF CONSTRAINTS ARE VIOLATED [BSP]

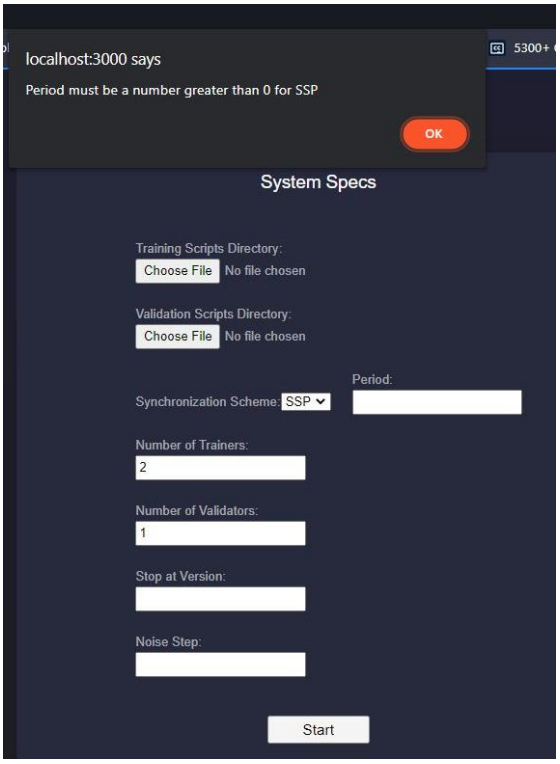


Figure 4.2.3 PROMPT ARAISES IF CONSTRAINTS ARE VIOLATED [SSP]

Case 3 (When stop at version is left null or is filled with float value)

Input: Stop at version field is left null or filled with float value

Output: Alert prompt appears to rectify the stop at version field (Figure 4.2.4 and 4.2.5)

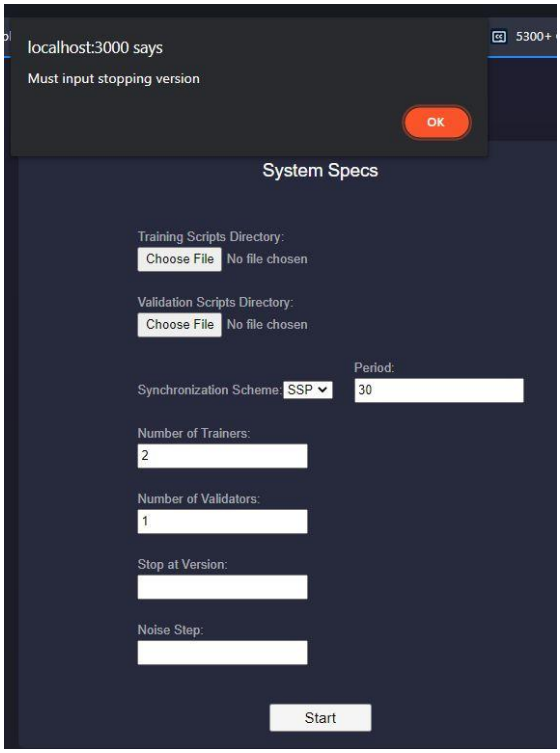


Figure 4.2.4. PROMPT ARAISES IF CONSTRAINTS ARE VIOLATED [VERSION CONTROL]

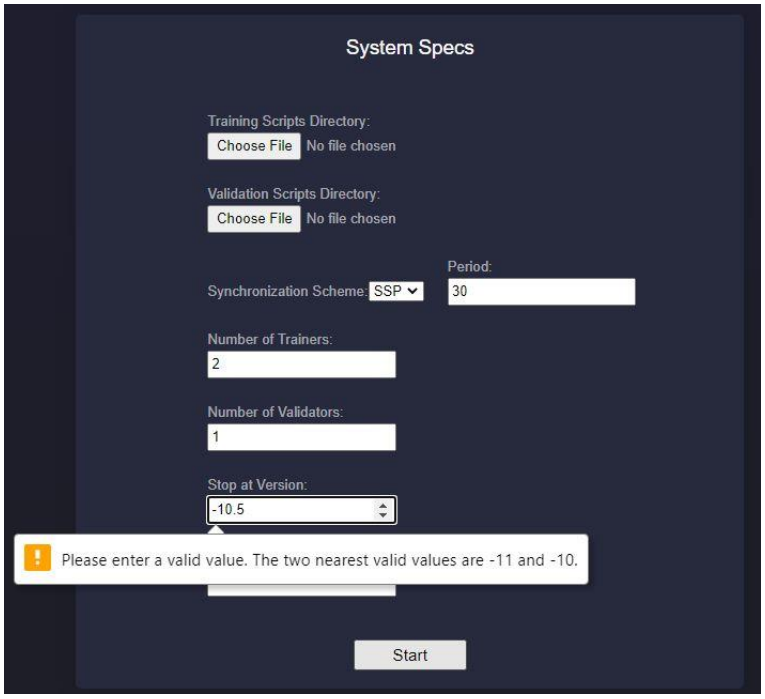


Figure 4.2.5. PROMPT ARAISES IF CONSTRAINTS ARE VIOLATED [VERSION CONTROL]

Case 4 (Force Terminating System)

Input: Fore kill framework instance signal sent to REST API

Output: The framework instance should kill and clear all the residues from blockchain (Figure 4.2.6)

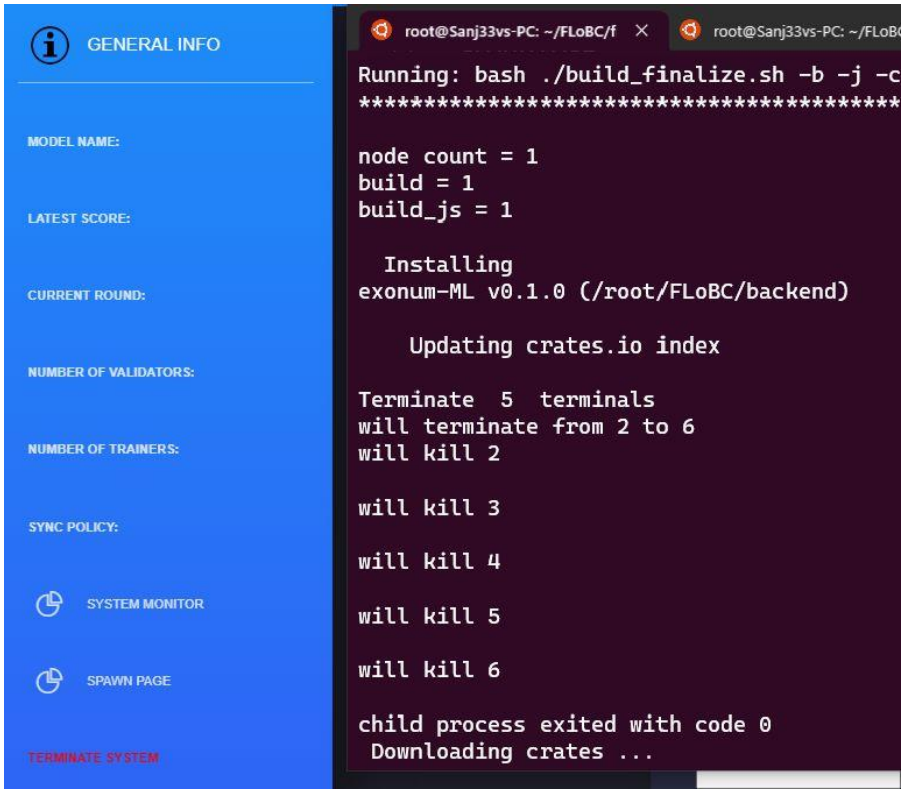


Figure 4.2.6 TERMINATING SYSTEM

Case 5 (Synchronization policy)

Imput: Synchronization scheme chosen from the frontend dropdown

Output: API correctly interprets the policy (Figure 4.2.7) and spawns clients and checkers corresponding to the policy (Figure 4.2.8)

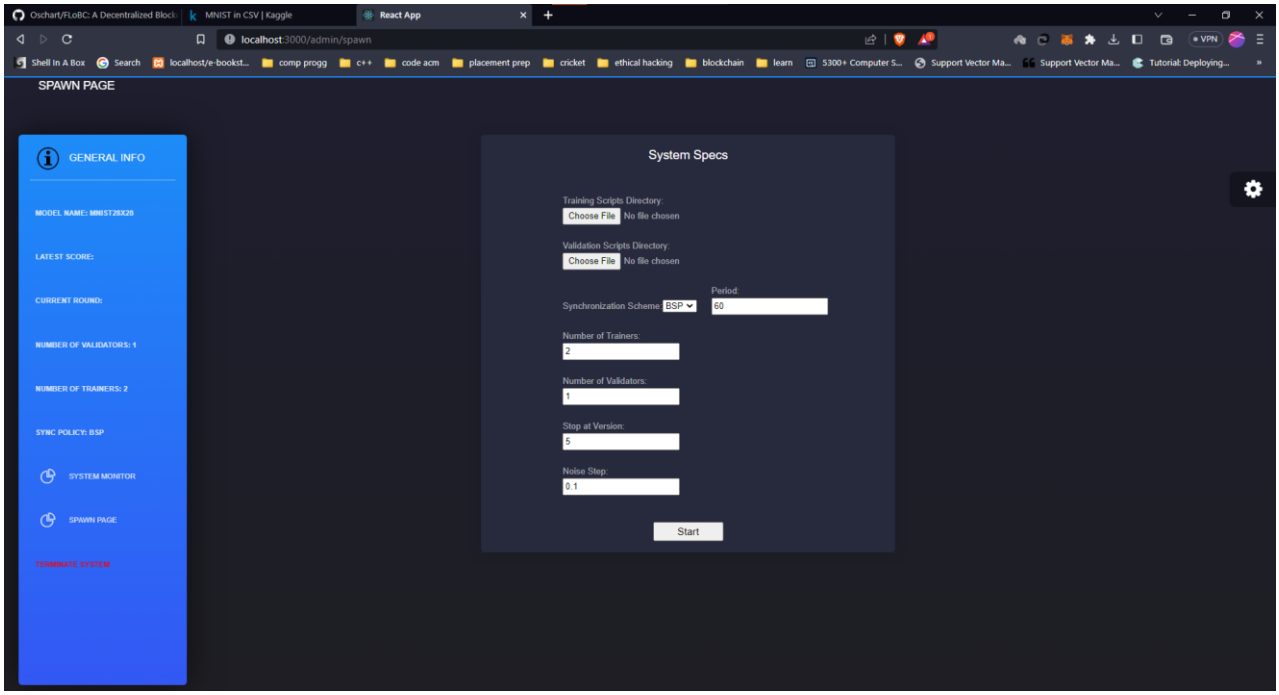


Figure 4.2.7. SPAWN PAGE

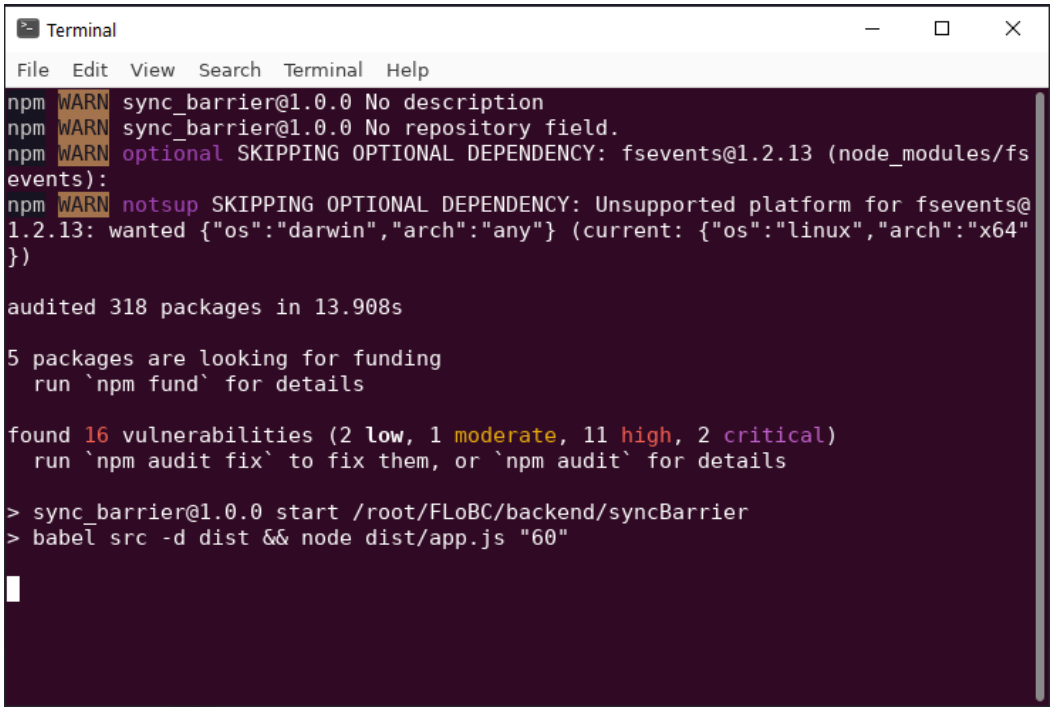


Figure 4.2.8 SYNCHRONIZER

Case 6 (Sync Signal Propagation)

Input: Clients and checkers are created

Output: Clients and checkers communicate with each other and shares model updates (Figure 4.2.9).

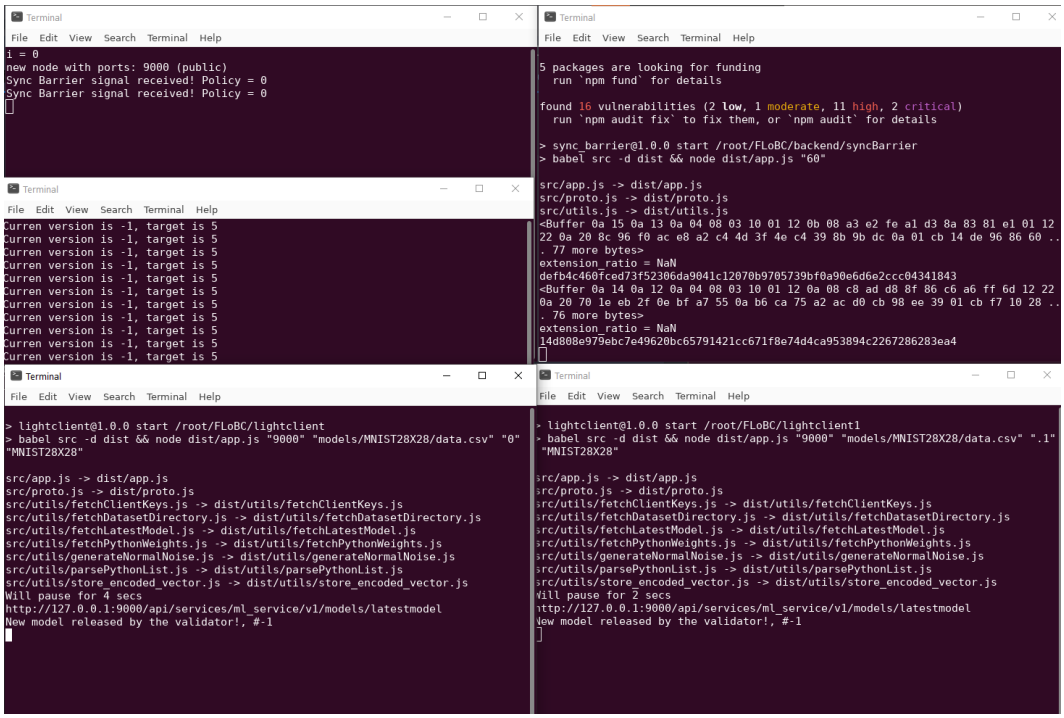


Figure 4.2.9. CLIENT SYNC SIGNALS RECEIVED BY CHECKER

Case 7 (Sending Model Parameters)

Input: Clients train the model with local dataset

Output: Trained model updates are shared to checkers (Figure 4.2.10).

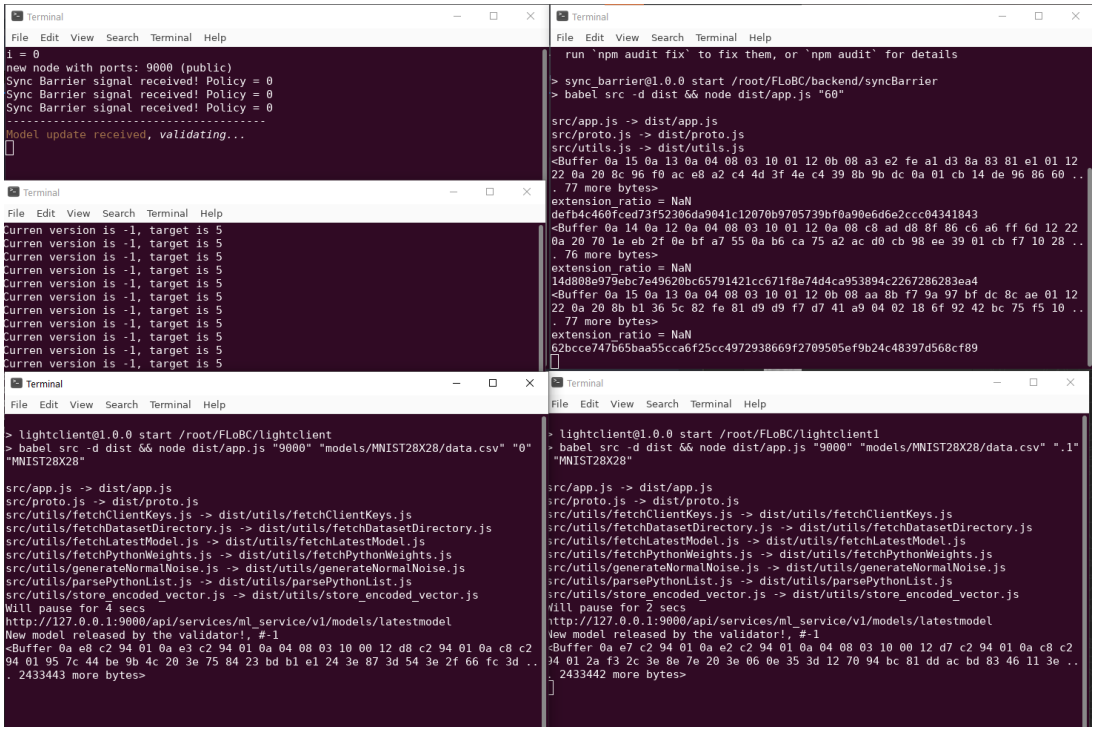


Figure 4.2.10. CLIENTS SHARING MODEL PARAMETERS WITH CHECKERS

Case 8 (Validating Model Parameters)

Input: All trainers shared their model updates

Output: A new model is released which aggregates the updates from all clients (Figure 4.2.11)

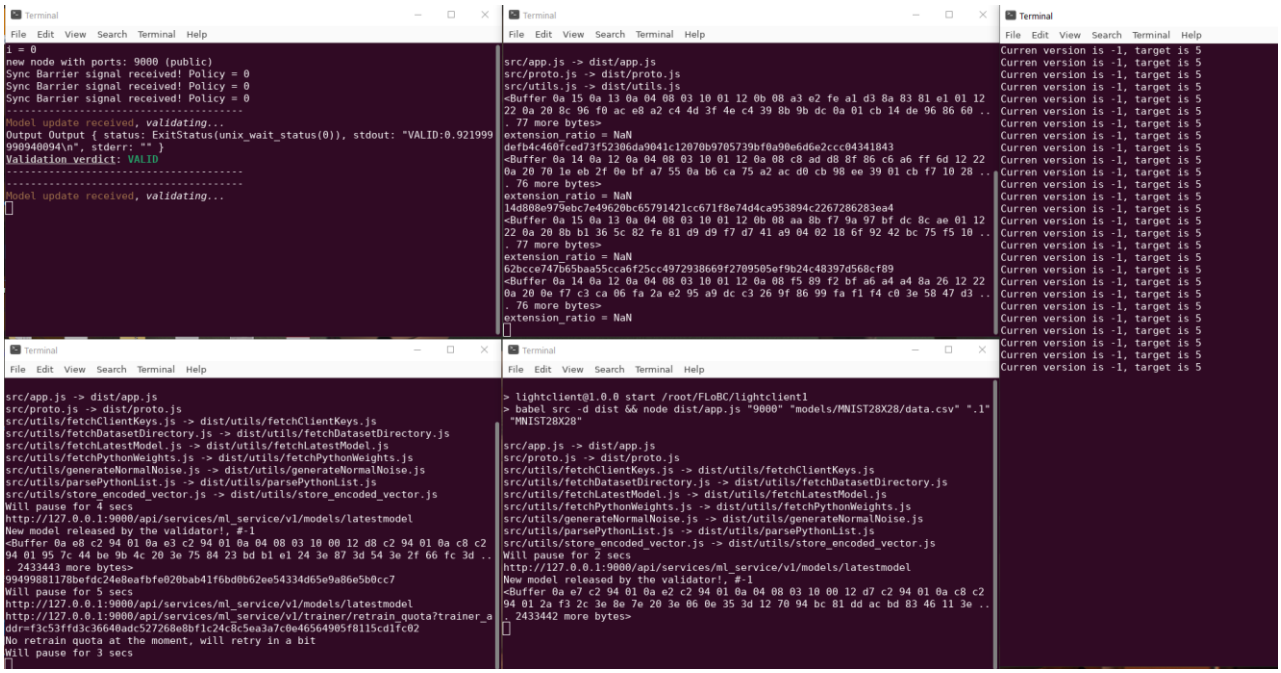


Figure 4.2.11. CHECKERS VALIDATING CLIENT MODELS

Case 9 (Releasing New Model)

Input: New model is released by the checker

Output: All clients will receive the new model update from the checker (Figure 4.2.12)

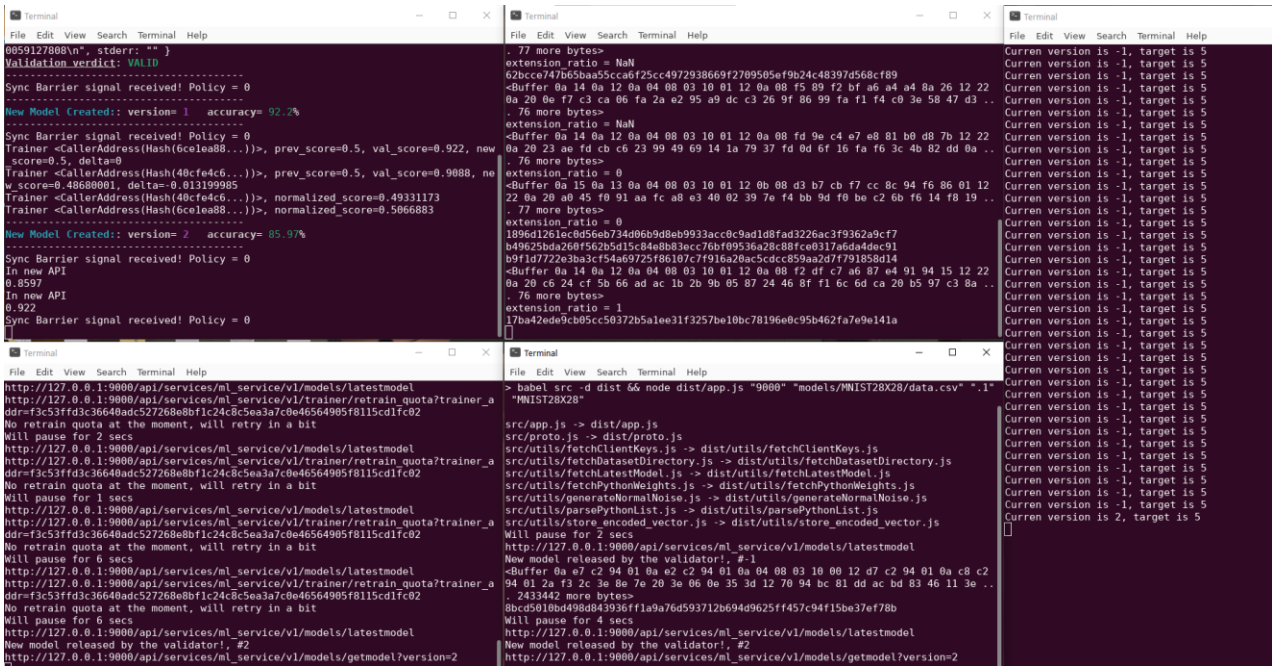


Figure 4.2.12. CHECKERS RELESING THE NEW MODEL

Case 10 (Reflecting model update in frontend)

Input: New model is released

Output: New model metadata is updated in the frontend (Figure 4.2.13) and in its buffer (Figure 4.2.14)

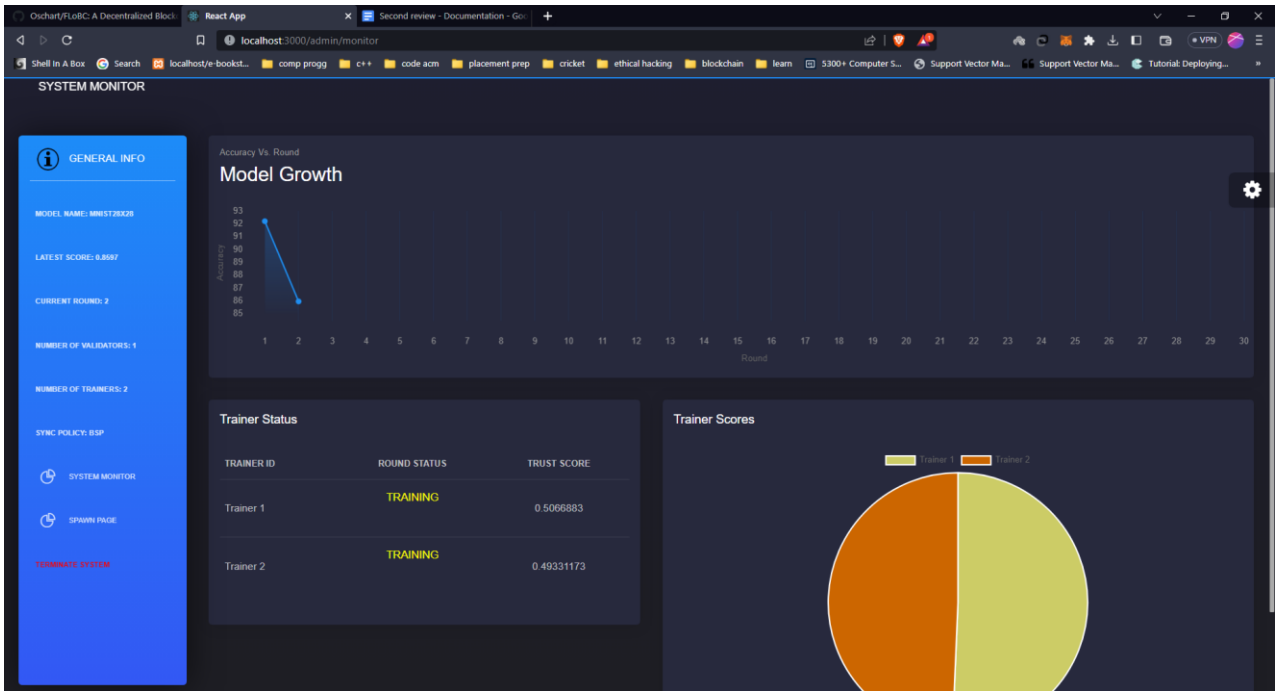


Figure 4.2.13. MODEL UPDATED IN FRONTEND

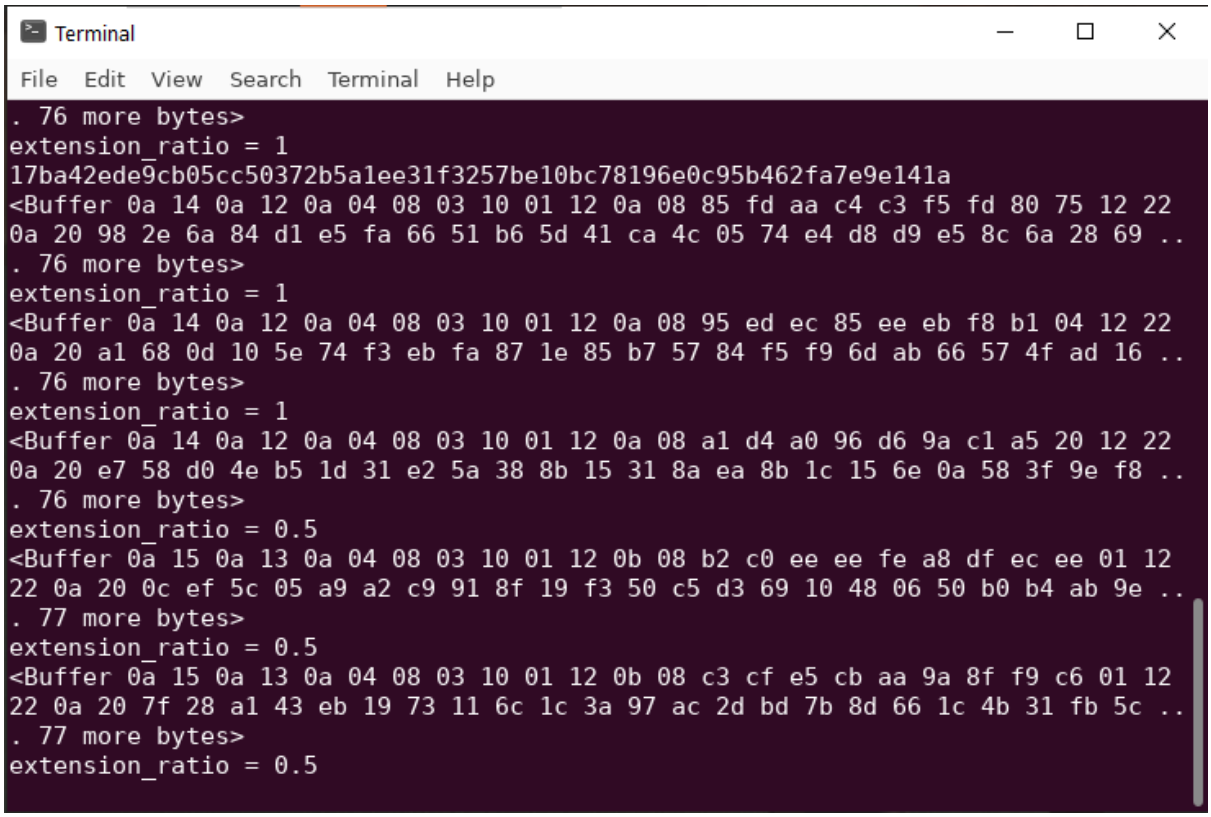


Figure 4.2.14. CORRESPONDING TESTCASE BUFFER

4.3PERFORMANCE METRICS

4.3.1 Benchmark: Decentralized vs. Centralized Performance

The purpose of this comparison is to check whether the security and privacy achieved from decentralized computing comes with a trade-off from loss of model quality and accuracy or whether model accuracy is unaffected by the advantages imposed by the decentralized model.

Setup: We ran both centralized and decentralized framework for 15 iterations and below is the resultant graph. Decentralized model was ran using 10 nodes (8 clients and 2 checkers)

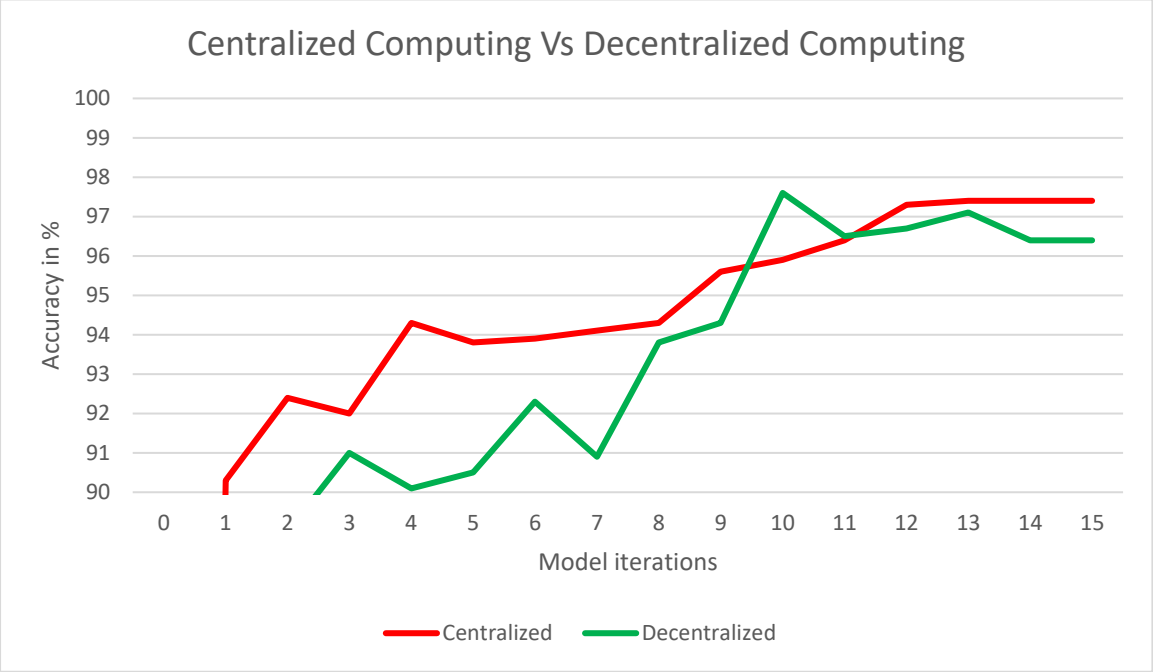


Figure 4.3.1 Accuracy Against iterations for the centralized and decentralized runs

Results and Discussion: From the graph (Figure 4.3.1) we can infer that with decentralized approach the model accuracy stays low as compared to centralized model for initial few numbers of iterations (6 – 7), it takes time to build up on the model accuracy as many decentralized actors would be contributing to build a model in each iteration with different datasets for them to train on. After initial few iterations we can infer

that accuracy of centralized and decentralized model are in comparable with each other, and ends up with decentralized model having just 0.8% accuracy less than the centralized model. As for the benefits of security and privacy the decentralized model provides a very small percentage loss in accuracy is acceptable, also the trend can be subjected to change on different dataset or with more iterations where accuracy of both models would be similar.

4.3.2 Experiment 1: Comparing different consensus mechanism

The purpose of this experiment is to compare the different available consensus algorithms, which maintains the integrity of the blockchain and check whether different consensus mechanism affect the model accuracy, if so then we have to find the consensus mechanism that yields higher accuracy as compared to others

Setup: We ran the decentralized framework using 10 nodes (8 clients and 2 checkers) for 15 iterations with different consensus mechanisms.

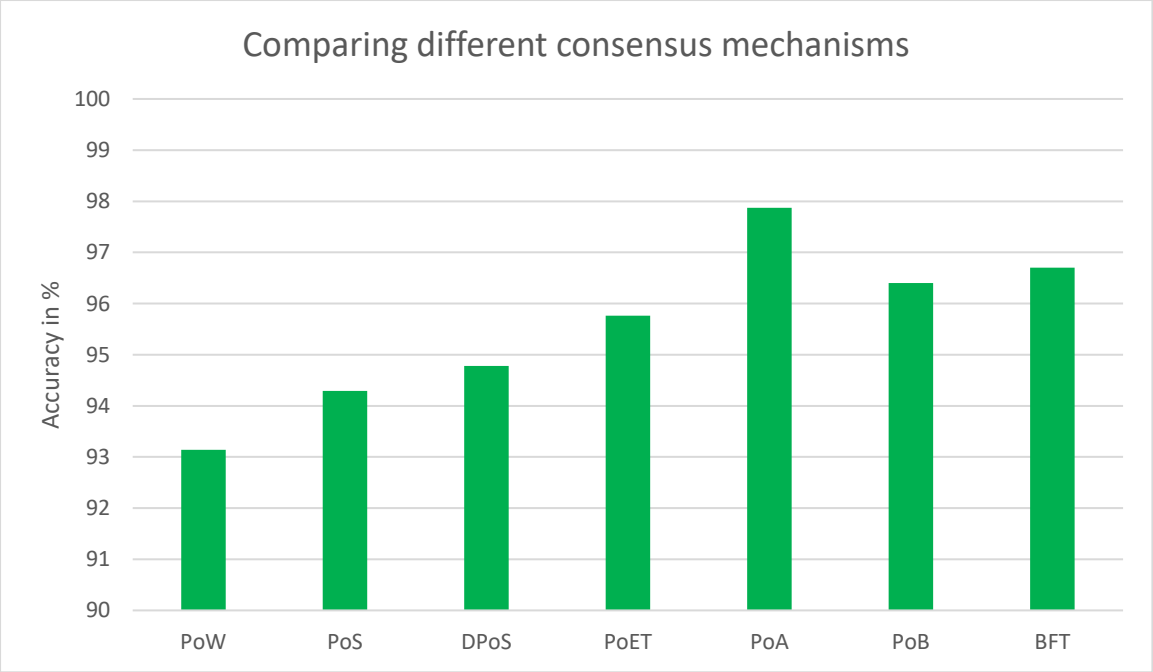


Figure 4.3.2 Accuracy Against different Consensus mechanisms

Results and Discussion: From the graph (Figure 4.3.2) we can infer that PoA (Proof of Authority) consensus mechanism gave highest accuracy 97.9% and BFT (Byzantine Fault Tolerance) has the second-best accuracy 96.7% after 15 iterations and PoW (Proof of Work) has the least accuracy among the list with 93.2%. So as PoA has higher accuracy and seems compatible with running machine learning algorithms with consortium blockchain, we chose PoA (Proof of Authority) as consensus mechanism for this project.

4.3.3 Experiment 2: Clients-to-Checkers Ratio

The purpose of this comparison is to check whether the difference in number of clients and checkers affects model accuracy, if so then we have to estimate the ratio of clients and checkers which would lead us to higher accuracy model as compared to others.

Setup: We chose 10 nodes with variable number of Clients and Checkers in different setup starting from 1 client and 9 checkers to 9 clients and 1 checkers andwe ran each setup for 15 iterations and below is the resultant graph.

Figure 4.3.3 Maximum Accuracy and its iteration index for different splits of clients and checkers

Results and Discussion: As we can infer from the graph (Figure 4.3.3) lower number of clients has very ow accuracy, because they are trained on very low number of dataset rows, thus leading to lowest accuracy. And as the clients increases the model is trained on diverse number of dataset instances thus accuracy increases. And during 9 clients and 1 checker there is a bottleneck in checkers side as it cannot validate all 9 model instances which decreases the accuracy eventually. Thus Clients: Checkers ratio 8:2 should be maintained in order to achieve higher accuracy model with maximum utilization of available resources.

4.3.4 Experiment 3: Authority consensus Policy

Another main component of D-FedL framework is Authority consensus which is responsible to make the overall system stable by avoiding model updates from the clients which constantly provides low quality model by penalising them and rewarding the clients with good quality model. The goal of the experiment is to compare the model growth with consensus and without it, and to infer whether usage of authority consensus increases model accuracy and stability.

Setup: We ran the framework both with consensus and without consensus setting, with best performance setting 8 clients and 2 checkers, for 15 iterations.

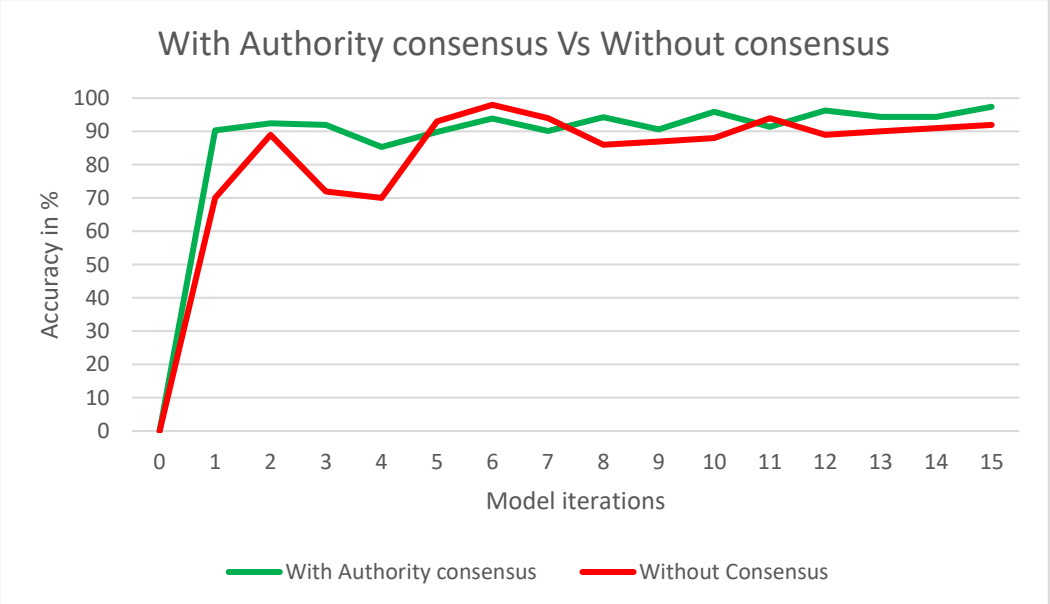


Figure 4.3.4 Accuracy Performance of scoring group vs control (uniform) group over 15 training rounds

Results and Discussion: As we could infer from the graph (Figure 4.3.4), we could see large deviation in accuracy of the model trained without consensus, because nodes that provide low quality model would eventually bring down the accuracy of the model thus more fluctuation in model accuracy. Whereas on other hand model trained with authority consensus had very low deviation in the accuracy in successive rounds because it will stop receiving model updates from low performing nodes which increases model stability and model accuracy. After 15 rounds accuracy of the model trained with consensus came out to be 97.4% and the model without consensus was 91.9%, as we could notice the huge 5.5% difference in model accuracy between models trained with and without consensus. Thus using authority consensus seems to have improved model accuracy and stability.

4.4 COMPARATIVE ANALYSIS

4.4.1 Comparing accuracy of different synchronization schemes

The critical component of the D-FedL framework is synchronization schemes which are responsible for receiving updates from the clients and synchronizing model updates across other clients. They can be implemented in two ways synchronous and in asynchronous manner.

In synchronous scheme, the clients will wait till the threshold period for all the other clients to complete their training and the clients will resume their training once a new model version is released by the checkers. BSP is synchronous algorithm, whereas SSP is similar to BSP but with more relaxed conditions, which will extend the deadline if threshold % of clients haven’t submitted their model.

In asynchronous scheme, the clients will continue training on the same model version irrespective of other clients training completion status once all the clients have submitted their model new model version is released by checkers. BAP is asynchronous algorithm. The framework in the base paper was built on asynchronous algorithm. Thus, the goal of this comparison is to check the model

accuracy and efficient utilization of resources under different synchronization scheme.

Setup: We ran the framework with best performance setting 8 clients 2 checkers, and we came up with two results one with model accuracy over 15 iterations and another one with model accuracy and no of iteration over 15 minutes.

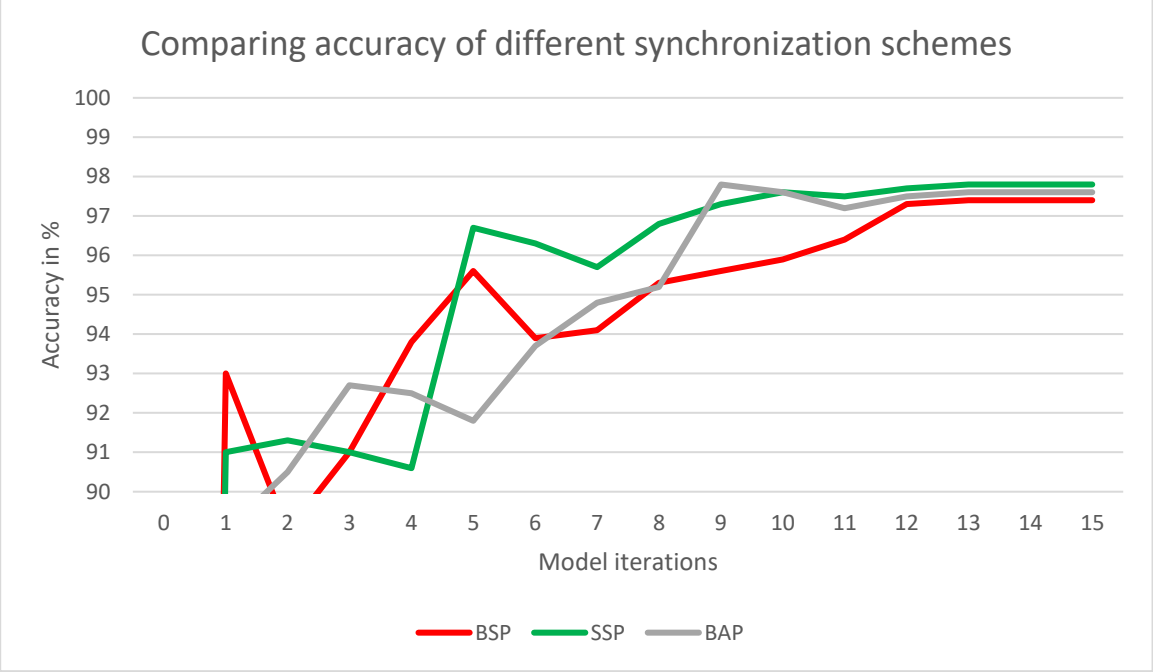


Figure 4.4.1 Accuracy performance for 4 different schemes across 15 training iterations

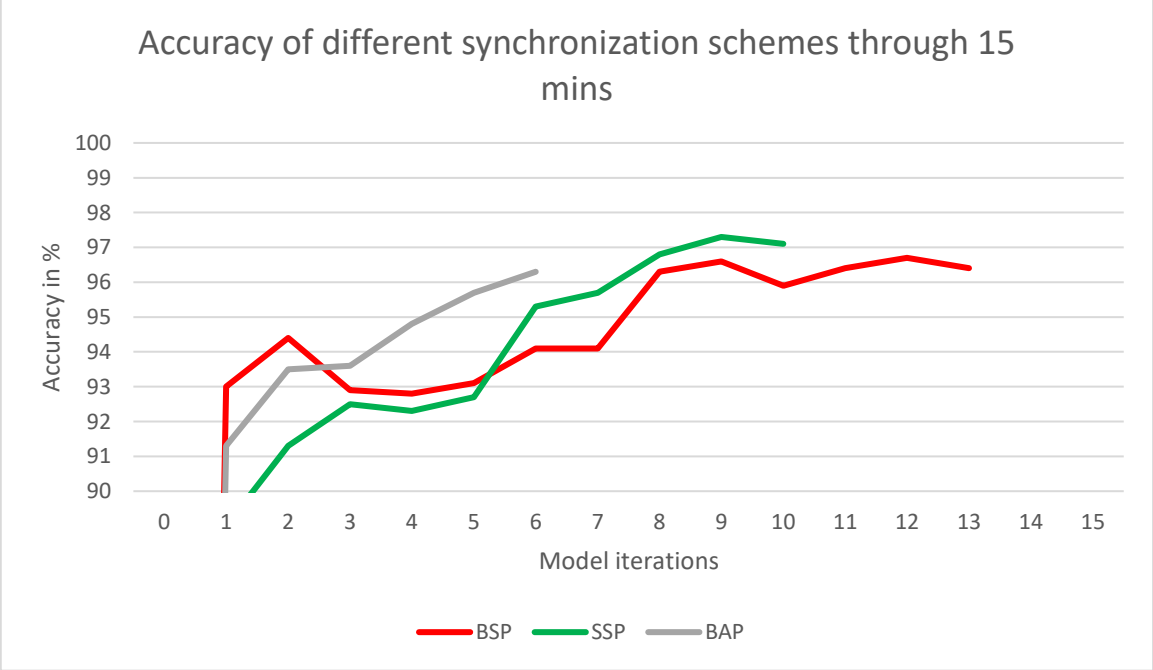


Figure 4.4.2 Accuracy performance for 4 different schemes across 15 minutes of training time

Results and Discussion: As we could infer from the graphs (Figure 4.4.1 and Figure 4.4.2) at the end of 15 iterations the model accuracy of both synchronous and asynchronous schemes is almost same, but when we ran the system through particular time frame i.e) 15 minutes we could see BSP has higher number of iterations in 15 minutes but it has lower accuracy, followed by SSP which had completed 10 iterations in 15 minutes with higher accuracy and asynchronous scheme which has completed just 6 iterations.

Though model accuracy is same at similar iteration, when it comes into the efficiency of using their resources synchronization schemes outperform asynchronous scheme by completing more iterations in fixed time frame.

4.4.2 Model Accuracy vs No of Iterations

From the previous experiment we find that all the synchronization scheme has similar accuracy but synchronous algorithm has better efficiency compared to asynchronous algorithm. But as the synchronous algorithm is concerned the nodes will pause training until it receives updates from other nodes, but as private/consortium blockchain in concerned we can guarantee that all the other trainer nodes are live within the private network. But when it has to be implemented through the public platform asynchronous algorithm might have an edge over synchronous algorithm.

From this experiment, the goal is to find whether the highest number of iterations till which the model is not affected by overfitting which decreases the model accuracy and find which synchronization scheme is more susceptible to overfitting and to find the one which is least affected by it.

Setup: We ran the framework with best performance setting 8 clients 2 checkers for 15 iterations over various synchronization scheme.

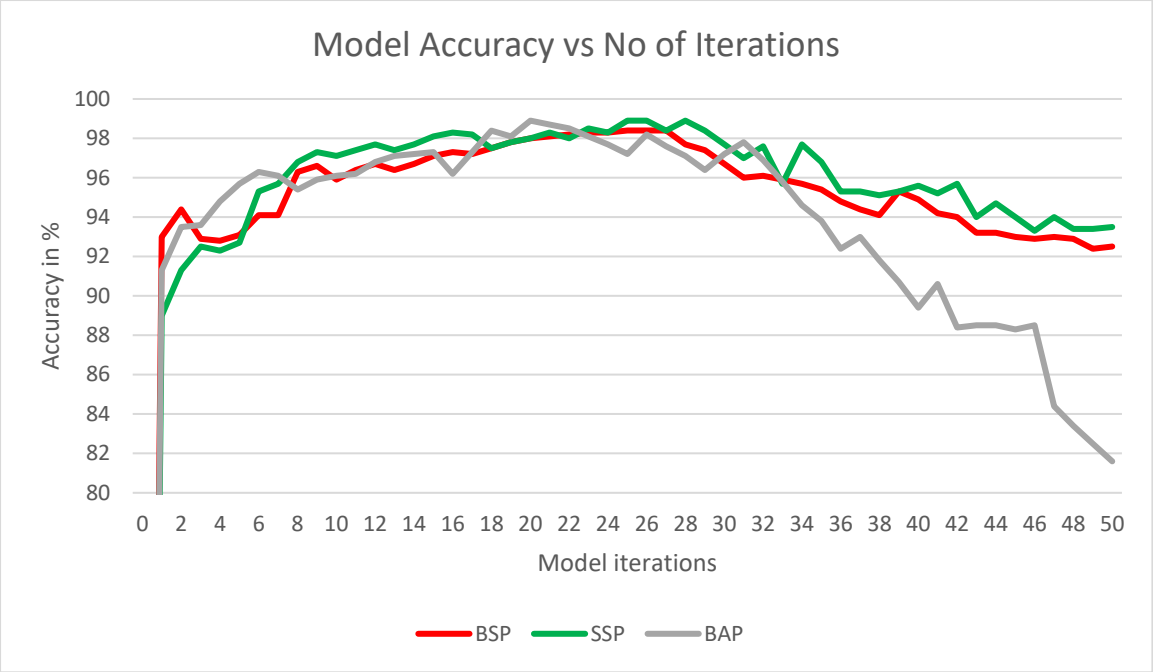


Figure 4.4.3 Accuracy performance for 4 different schemes across model Iterations

Synchronization scheme	Iteration number
BSP	25
SSP	26
BAP	22

Table 4.4.2 Threshold Iteration number for different Synchronization scheme

Results and Discussion: From the table and graph (Figure 4.4.3) we can infer that the maximum iterations we can train without model overfitting is 30, after which the model accuracy gradually decreases. Also, from the graph we could find that BAP model accuracy is drastically affected by overfitting, while BSP and SSP can withstand the model overfitting to a certain limit. The conclusion from the experiment is in order to avoid overfitting the threshold no of iterations is 30 and synchronous scheme seems to withstand overfitting to a certain margin.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

In conclusion, D-FedL is a proof-of-concept that demonstrates the ability of its constituent parts to work together to create a coherent system with the requisite levels of generality, decentralization, efficiency, privacy, and Byzantine fault-tolerance. We can tell from our trials that the decentralized model has just 0.8% less accuracy than the centralized model. Regarding the advantages of security and privacy the decentralized model offers, a very little percentage loss in accuracy is acceptable. Additionally, the trend may be changed on various datasets or with additional iterations so that the accuracy of both models is comparable. We initially demonstrated that there is a quasi-optimal balance to be struck on client-to-checker partitioning, empirically establishing that an 8:2 client-to-checker ratio works well. Second, we showed that even a straightforward authority consensus approach may significantly improve the caliber of models that are created. Finally, we contrasted and highlighted the various trade-offs of the three main synchronization systems (BSP, SSP, and BAP). SSP has been shown to be more economical than the other two approaches. Also, from the trails we found that the maximum no of iterations that can run without overfitting is 30 iterations (threshold number), and BSP, SSP (synchronous schemes) can withstand overfitting to certain extent, but BAP (asynchronous scheme) found very vulnerable to overfitting.

5.2 FUTURE WORK

D-FedL may easily be modified to support additional computational models even though the SGD-compatibility of a model is an assumption built into the system. By giving the sync periods greater flexibility based on how long it is anticipated that most clients will take to submit their updates in a single round, synchronisation schemes can be further improved. Additionally, a layer of differential privacy might be added to transaction communication to restrict how far gradients can be inverted in order to extract client data, which would provide a better level of data privacy.

REFERENCES

[1] Avizheh, S., Nabi, M., Rahman, S., Sharifian, S., & Safavi-Naini, R. (2021). Privacy-Preserving Resource Sharing Using Permissioned Blockchains: (The Case of Smart Neighbourhood). In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25* (pp. 482-504). Springer Berlin Heidelberg.

[2] Benedetti, M., De Sclavis, F., Favorito, M., Galano, G., Giammusso, S., Muci, A., & Nardelli, M. (2022). A PoW-less Bitcoin with Certified Byzantine Consensus. *arXiv preprint arXiv:2207.06870*.

[3] Bogdanova, A., Nakai, A., Okada, Y., Imakura, A., & Sakurai, T. (2020). Federated learning system without model sharing through integration of dimensional reduced data representations. *arXiv preprint arXiv:2011.06803*.

[4] Görkey, I., El Moussaoui, C., Wijdeveld, V., & Sennema, E. (2020). Comparative Study of Byzantine Fault Tolerant Consensus Algorithms on Permissioned Blockchains.

[5] He, J., Cai, L., Cheng, P., Pan, J., & Shi, L. (2019). Consensus-based data-privacy preserving data aggregation. *IEEE Transactions on Automatic Control*, *64*(12), 5222-5229.

[6] Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., & Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

[7] Kurtulmus, A. B., & Daniel, K. (2018). Trustless machine learning contracts; evaluating and exchanging machine learning models on the ethereum blockchain. *arXiv preprint arXiv:1802.10185*.

[8] Qu, Y., Uddin, M. P., Gan, C., Xiang, Y., Gao, L., & Yearwood, J. (2022). Blockchain-enabled federated learning: A survey. *ACM Computing Surveys*, 55(4), 1-35.

[9] Sheller, M. J., Edwards, B., Reina, G. A., Martin, J., Pati, S., Kotrotsou, A., ... & Bakas, S. (2020). Federated learning in medicine: facilitating multi-institutional collaborations without sharing patient data. *Scientific reports*, *10*(1), 1-12.

[10] Truex, S., Baracaldo, N., Anwar, A., Steinke, T., Ludwig, H., Zhang, R., & Zhou, Y. (2019, November). A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM workshop on artificial intelligence and security* (pp. 1-11).