Blockchain in Federated Learning

Team Member 1:

First Name: Arjun Ramesh

Last Name: Kaushik

Email: kaushik3@buffalo.edu

Person Number: 50413327

Team Member 2:

First Name: Hari Preetham Reddy

Last Name: Nandyala

Email: haripree@buffalo.edu

Person Number: 50468240

Approved by: Prof. Bina Ramamurthy and Chen Xu

Reference Papers:

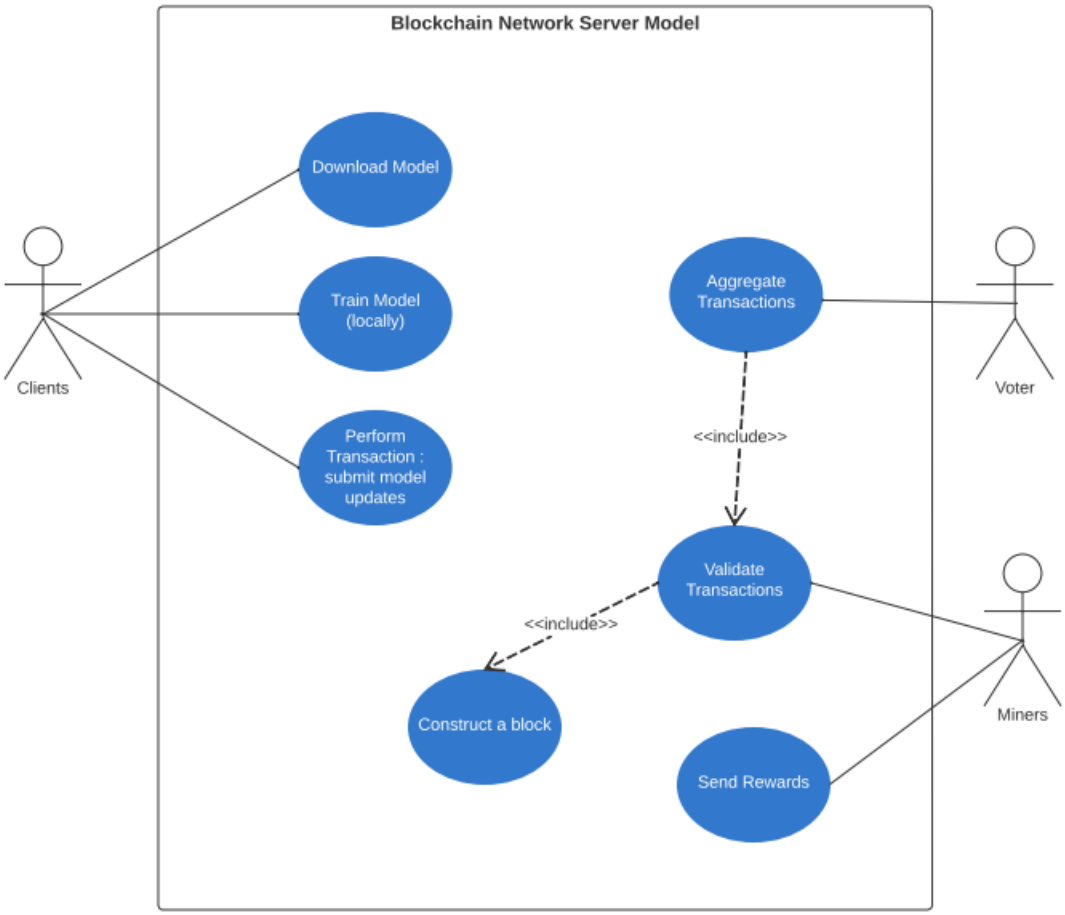
1. https://drive.google.com/file/d/1tbIfXJRVTmbeQYT6QmckV_Yq_u1rkow4/view?usp=sharing
2. https://drive.google.com/file/d/1BuzRDZioCtsEgFR9L0FLIYoCveJlaYY6/view?usp=sharing

Issue(s) addressed: Improving privacy in a world driven by increasing needs for privacy

Abstract:

Federated learning is a type of Machine Learning which works like a decentralised system. Here, learning happens locally in the devices of clients; the gradients from client devices are aggregated in a server and then each client is updated with a new ML model from the server called “global model”. The idea of Federated learning is to preserve privacy of client data. But it doesn’t prevent the clients from using malicious data to train, which might degrade the performance of the global model. This is where we bring in Blockchain Technology. Before updating the server model, we validate, using smart contracts, that the global model is performing better with the aggregation of locally trained gradients. And then, add the global model (aggregated gradients) to the blockchain network.

Use Case Diagram:



Contract Diagram:

FedLearning

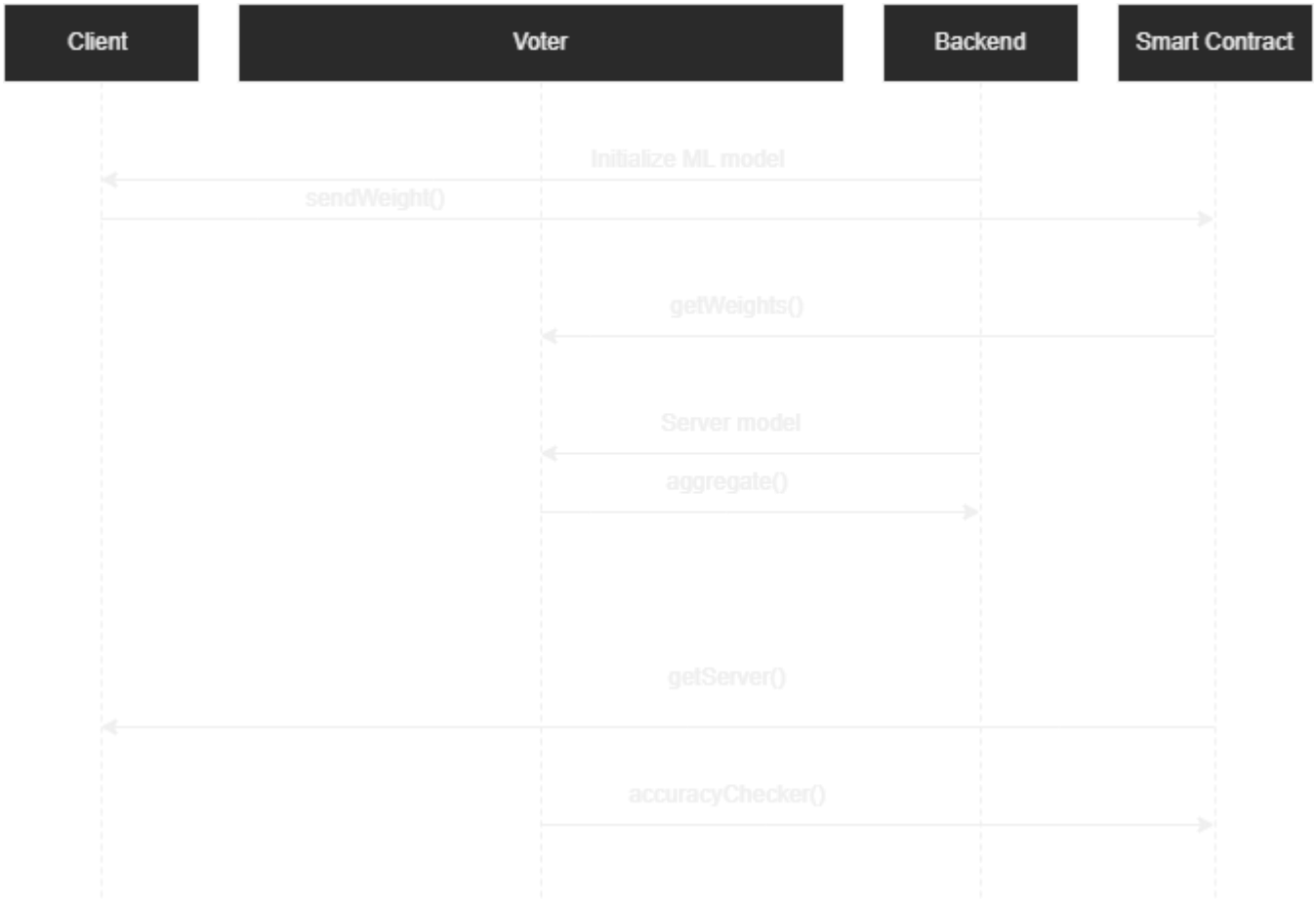
mapping(address => string) public client_weights;
address[] client_address;
string server;

function sendWeights(address x, string memory y) public
function getServer() public view returns(string memory)
function getWeights(address a) public view returns(string memory)
function accuracyChecker(string memory z, uint acc) public returns (bool)

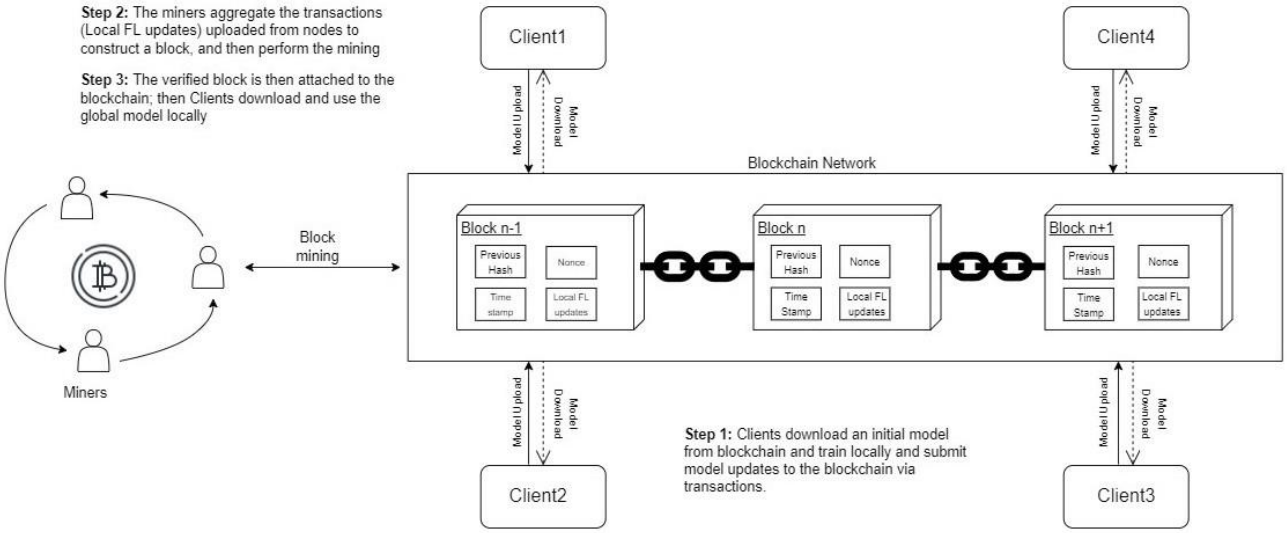
Quad Chart Diagram

Use case: Federated Learning **Problem statement:** A decentralized blockchain based federated learning	**Issues with existing centralized model:** 1.Single point of failure: In centralized federated learning integration of local training results to update the global model server is required which is also called as aggregator if the aggregator is compromised the whole system will fail and intentionally dishonest aggregation, accidental network connection failure, unexpected external attacks, 2.Malicious clients and false data: Here clients can train the model with invalid data or false data and update it to the global model which affects the whole model. 3. The lack of incentives: It is challenging to persuade clients to follow the protocol honestly and supply accurate data in the classic federated learning model because they are seen as providing their processing resources without being compensated.
Proposed blockchain based solution: 1. Clients download an initial model from blockchain and train locally and submit model updates to the blockchain via transactions. 2. The Voter will aggregate the transactions (Local FL updates) and then the miners will validate the transactions, perform the mining 3. The verified block is then attached to the blockchain; then Clients download and use the global model locally	**Benefits:** 1. Model will no longer collapse due to a single server failure/attack. 2. The proposed model is more robust and secure, ensuring data security/confidentiality. 3. Decentralization aids in significantly reducing malicious clients and false data. 4. Providing incentives encourages clients to follow protocol in a distributed system.

Sequence Diagram:



Working Model:



Instructions to start the DApp

1. Start the server by running the python file in *api/main.py*
2. Run *npm install* in the project folder to install requisite dependencies
3. Deploy smart contract to the test network on Ganache and replace the credentials in *src/components/sc_config.js*
4. Run *npm start*

Smart Contract

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;
pragma experimental ABIEncoderV2;

contract FedLearning{
    mapping(address => string) public client_weights;
    address[] client_address;
    string server;

    function sendWeights(address x, string memory y) public {
        client_weights[x] = y;
        client_address.push(x);
    }

    function getServer() public view returns(string memory){
        return server;
    }

    function getWeights(address a) public view returns(string memory){
        return client_weights[a];
    }

    function accuracyChecker(string memory z, uint acc) public returns (bool){
        if(acc > 8000000){
            server = z;
            return true;
        }
        return false;
    }


}
```

sendWeights(address x, string memory y **)**

This function stores the hash value obtained from IPFS against the address of the participating client.

getWeights(address a **)**

This function returns the hash value stored in the HashMap against the address of the client who has already contributed. This function is only called during aggregation.

accuracyChecker(string memory z, uint acc **)**

This function checks the accuracy of the server model and stores it in the server string if the accuracy is above 80%. Here, we have $80x10^5$ since solidity can't handle floating-point numbers, yet.

Working of the DApp

Currently, the working of the DApp can be explained in the following steps, assuming Number of Clients = 3 –

1. On clicking the Train Clients and Upload Gradients button, the control is passed to the backend through the *@app.post("/train/")* API call.
2. In the backend, the 3 clients are trained independently on a preset dataset.
3. The gradients of the 3 clients are stored into 3 different text files.
4. The text files are stored on IPFS through 3 API calls. The hash values returned from the *https://ipfs.infura.io:5001/api/v0/add* API call is sent to the frontend.
5. The returned hash values are stored on blockchain
6. Each *Voter* retrieves the stored hash values from the blockchain. Then, the hash values are used in the *https://ipfs.infura.io:5001/api/v0/cat* API call to get the gradients.
7. The gradients are now aggregated using *FedAvg* algorithm and fed into a new Machine Learning model called *Server* model. The *Server* model calculates the accuracy of its newly aggregated model on the test dataset. This is done in backend through *@app.post("/aggregate/") API* call.
8. *@app.post("/aggregate/")* API call returns the accuracy multiplied by a factor of 10^5 and the hash of the *Server* gradients stored on IPFS. The accuracy is multiplied by 10^5 because *Solidity* can't handle floating-point numbers yet.
9. From the front-end, we call the smart contract function *accuracyChecker()* with accuracy and hash as its input (obtained from *@app.post("/aggregate/")* API). This function stores the *Server* hash on the blockchain only when the accuracy is more than 80% or $80x10^5$.

Token Design

The principal idea of the project is to improve privacy and avoid a single point of failure. To counter this, we will be issuing Governance Tokens to a particular number of people (called Voters) apart from the participating clients. Each voter has a separate dataset, they aggregate the client weights and test it on their dataset. Based on the obtained accuracy, each voter votes on the aggregated model. With a majority, the aggregated model is stored in IPFS. The hash obtained from IPFS is stored on the blockchain.

Smart contract functions will be updated to register the voters and only these voters can call the getWeights() and accuracyChecker() functions.