**Abstract**

This project aims to simulate decentralized peer-to-peer transfer of electricity in the microgrid. The houses which have installed solar panel systems can produce energy, sell their excess energy and can also purchase energy within the microgrid without involving any kind of middlemen or central authority. This is achieved using Blockchain technology. The Blockchain platform also provides additional reliability and security to the system. The local grids work in collaboration with the national grid to maintain the energy distribution system while providing a platform for local trading.

**I. Introduction**

With the continuously increasing demand and over-exploitation of the non-renewable energy resources, the renewable energy resources will play a major role in creating sustainable, inexpensive and eco-friendly energy generation.

The long distance between energy generation and energy consumption sites in the current centralized grid system causes a huge wastage of energy. Thus, we need to transit from a centralized system to a decentralized one.

But these renewable energy sourced distributed systems introduce new challenges to be solved. One of these is the introduction of the prosumer – the one who produces as well as consumes electrical power.

The erratic nature of wind and solar energy is another challenge. New market approaches are required to integrate these renewable energy generating prosumers into the energy system.

Microgrids can be constructed by forming a cluster of small-scale participants which locally form an energy trading market and trade energy within themselves.

The prosumers and consumers can trade energy on Microgrids. Microgrids provide a market access, marketplace as well as a mechanism for local energy trading for a specific group or society.

However, the execution of Microgrids would require the use of ICT(Innovative information and communication technology). Blockchain is such a fast growing technology for decentralized systems which provides user-friendly and transparent applications for prosumers to participate in the process of energy transaction.

**II. Motivation**

In the electricity grid, transmission of energy in a secure and stable manner is the principle task. Smart grids provide the solution of maintaining security of electric supply by integration of distributed energy sources. The losses caused due to transmission of energy is reduced by stimulating local consumption and production of energy. The management of transactions between prosumers and consumers which are participating within the grid in a centralized manner turns out to be costly and it would need a complex infrastructure of communication. Thus, we prefer a decentralized method for transaction of energy.

The Blockchain application provides a solution for establishing such decentralized infrastructure within the grid. It would enable prosumer and consumer in the smart grid to exchange electricity in a peer to peer fashion without the involvement of any other third party. Blockchain based application trading within the grid also provides plenty of other benefits including real-time market establishment, reduced cost of transaction, simplified trading and more privacy for users in smart grid.

We have witnessed an era of social media and e-commerce platforms which brought together producers and consumers of goods and services through peer-to-peer interactions. But there has always been some kind of involvement of untrustworthy middlemen .

The next generation Internet, the Decentralized Web would be possible with the help of exciting technology of Blockchain.

**III. Literary Survey**

**III.1 Blockchain Architecture**

Blockchain technology used nowadays is similar to the ledger system which was traditionally used for transacting records. The record of transactions is contained in a Blockchain. Every block in a Blockchain network points to the previous block with the help of a hash value which is a 256-bit hash. This creates a transactional record chain by which transactions can easily be tracked back. The contents of block include a block header which has the block version showing the guidelines to be strictly obeyed, the pointer to the previous block and a Merkle Tree Root hash which contains each transaction process in the block.

The blockchain is a distributed transactional database. The nodes of Blockchain are distributed globally and linked by a peer-to-peer communication network. These network nodes are machines of virtual kind in which communication takes place with the help of TCP and nodes can identify each other only by IP addresses. The transactions are cryptographically signed with the help of a corresponding private key. The transaction can be added to the chain only after being validated by each and every node which is present in the Blockchain network. This process only takes place if a transaction is included in a block and then mined according to the consensus algorithm rules. The longest chain is the most reliable and trusted chain. It is so because, if any invalid transaction is included by any node then, it would be ignored and discarded by remaining nodes of the network .

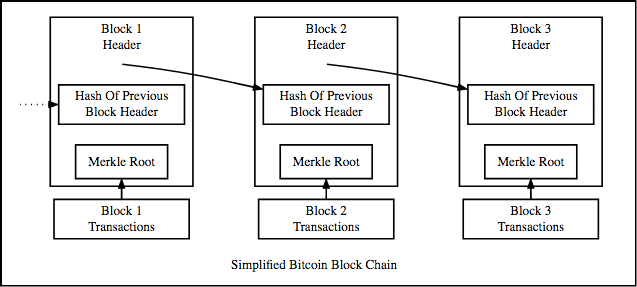


Figure 1. BlockChain Records

**III.2 Ethereum Smart Contracts**

Ethereum is a blockchain-based decentralized computing platform and operating system which is open sourced for use by developers. A non-centralized virtual machine is also made available by Ethereum. The Ethereum Virtual Machine is able to execute contracts. This all is possible with the help of a network of public nodes that is set up locally around the whole world. Gas is used for pricing of the Ethereum transactions.

The payment of transactions can only be done by Ether, which  is basically a  cryptocurrency made available by Ethereum itself. Ether is provided as a reward for performing some complex computations and this process of computation is sometimes referred to as mining.

1 Ether = (10 ^ 18) Wei.

Ethereum Smart Contracts can be written with help of different languages. We can write our own set of functions, data structures and rules to be used as contracts. Smart contracts are compiled into bytecode and then deployed to the Ethereum blockchain for execution.

They can be written in language libraries such as Solidity, Serpent, LLL, and Mutan. We have used Solidity in our project.

**III.3 Microgrid**

A group of decentralized energy resources and interconnected loads that can interact and trade energy locally within themselves is known as a microgrid. It is an entity wherein the resources which are produced as well as consumed into the microgrid can be controlled within its boundaries. It can also be connected as well as integrated within the main network grid. This makes one wonder about the advantages that can be brought by a microgrid to the currently used energy systems.

Microgrids are advantageous as they provide reliability, resilience, balanced loads and security of superior network grid. Moreover, microgrids can also be used in integration with the traditional energy distributing grids. This additional grid network can also help us to deduct and minimize the latency for congestion management. Furthermore, this distributed nature of a microgrid provides security against cyber-attack. Further, we can Interconnect several microgrids for a better and more stable balance of demand and supply. This also provides an advantage of a more active and engaging market.

The database used in our project is Brooklyn Microgrid Market Dataset.

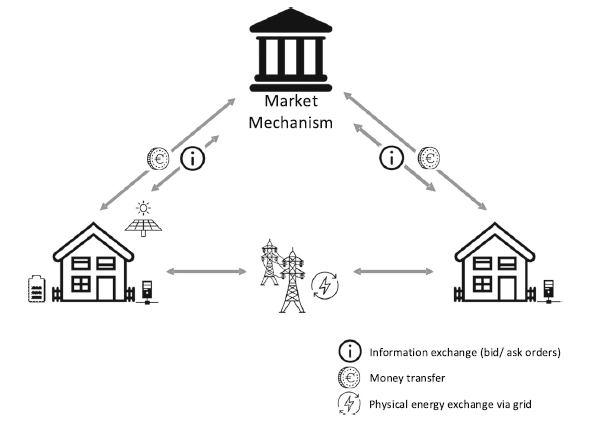


Figure 2. Microgrid

**IV. Contribution Technical**

**IV.1 Smart Contract using Solidity**

A smart contract is basically a set of defined rules written with help of different programming languages which are agreed upon by the participants to interact with each other. Whenever the predefined guidelines are met, the agreement guidelines are automatically set to be implemented. The machine on which it is run gives this contract various properties like immutability, also facilitating distributed & verifiable states.

We have written the contract using Solidity language. Each prosumer (entity in our project) can call the functions in the contract and send transactions which will automatically be published on the Ethereum Blockchain Network.

First of all the contract needs to be compiled to be uploaded on blockchain. The most important properties we’ll need to use on the compiled contract are the Application Binary Interface (ABI) and the bytecode. The bytecode is what will actually go onto the blockchain to make the smart contract work and the ABI will be the Javascript layer that acts like a human friendly map of the bytecode.

Then this Contract is deployed locally on test network i.e. Ganache in our case.

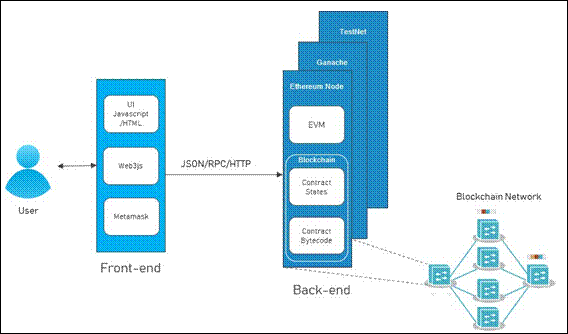


Figure 3. The structure of Decentralised Application

**IV.2 Web3.js API Setup for Smart Contracts**

Web3.js is an Application Programming Interface made available by javascript language that allows us to do interactions with an Ethereum node using JSON RPC endpoints enabled over the HTTP.

Web3.js provides the web3 object that helps us to utilize the Web3 API functions in JavaScript. Therefore, first of all a web3 object is instantiated as follows:

var web3 = new Web3();

This object needs to be connected to an RPC provider to communicate with the blockchain. We set a local or remote web3 provider using :

web3.setProvider(new web3.providers.HttpProvider(“http://RPC\_IP:RPC\_Port”));

where RPC\_IP is the RPC provider’s IP and RPC\_Port is its RPC port.

Web3 also provides a JavaScript object, web3.eth.Contract, which represents your deployed contract. To find and interact with your newly deployed contract on the blockchain, this object needs to know the contract’s address and its application binary interface (ABI):

var Contract = web3.eth.contract(“your contract’s ABI”);

The ABI is essentially a JSON object containing a detailed description (using a special data-encoding scheme) of the functions and their arguments, which describes how to call them in the bytecode.

**Invoking contract methods via a call**

A call is an invocation of a contract instance that cannot change the contract state, and includes calling view or pure functions or reading public states. The call only runs on your local node and saves you the expensive gas as there is no need for broadcasting the transaction.

**Invoking contract methods via a transaction**

To change the state of the contract instance, instead of making a call, you need to send a transaction that costs you real gas to validate your action.

For instance, to invoke the bid() method, you have to send a transaction to the blockchain with necessary arguments like address of the sender and fees.

**IV.3 Simulation using Ganache**

Ganache CLI is a command line interface which is a part of the Truffle framework of Ethereum development tools. It uses ethereum.js to simulate the behaviour of clients and make developing Ethereum applications a faster and easier task. It also includes all popular RPC functions and features which can be used to simplify the Blockchain development process.

* Using Ganache we have created 15 accounts, each account for a house acting as a prosumer.
* Then using the Database we calculate the amount of energy needed or amount of energy in excess with a prosumer. A Bid is placed by an account if it needs energy and Ask is placed if it has excess energy.
* Then a fixed price is calculated using an algorithm at which all transactions will occur. Finally Bids and Asks are matched satisfying the needs of customers. The matching is done if and only if the number of bids as well as the number of asks is greater than one. The total price for the energy is calculated in Wei and the amount is sent to the seller's account from the buyer's account.
* If any Bid is left unsatisfied it is fulfilled from National Grid at a price fixed by the government and money is sent to National Grid’s account.
* This simulation is carried out for 48 hours. The output csv file is prepared that shows the results.
* Further, we interpret the results by plotting the contents of the csv file on a graph.

**V. Data Flow Diagrams (DFD)**

**V.1 Level 0 DFD**

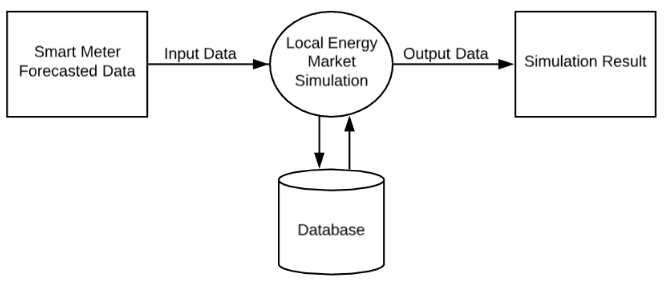
****

Figure 4. Level 0 DFD

**V.2 Level 1 DFD**

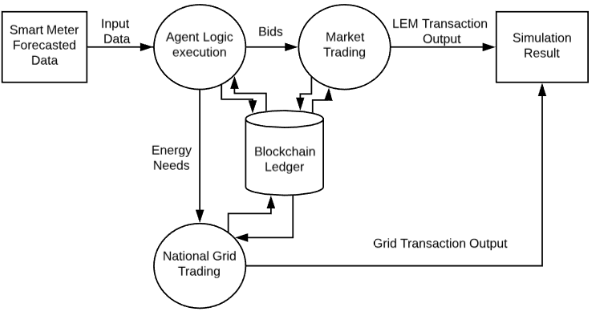
****

Figure 5. Level 1 DFD

**V.3 Level 2 DFD**

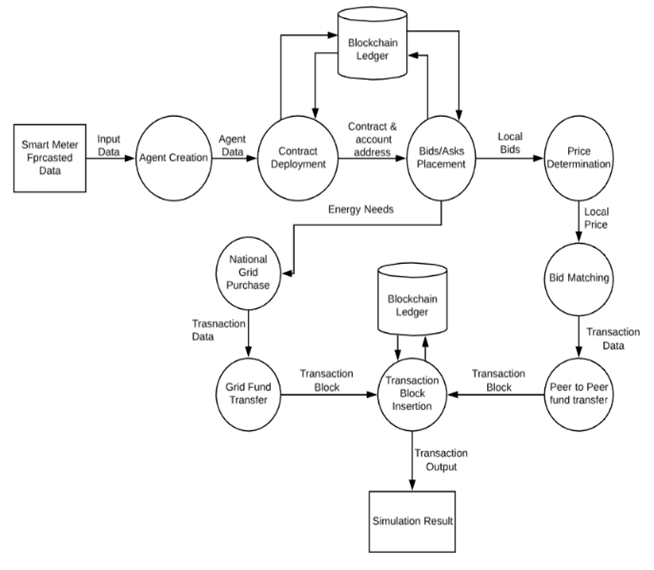
****

Figure 6. Level 2 DFD

**VI. Implementation**

**VI.1. Creating Prosumer**

The first step of our simulation is to create a real life user(a node in the network) that has all the functionalities. The user is a special type of consumer that can Consume as well as Produce the energy hence Prosumer. This Prosumer is created using Node class that has all the basic Data Members and Member Functions. The Prosumer can place Bid (requesting energy from market) or can place Ask (selling excess energy in the market) according to current supply, current demand and battery percentage.

**VI.2. Writing, Compiling and Deploying Ethereum Contract**

Ethereum allows users to create Contracts which are actually the applications that run on Ethereum Network. These Contracts are written using Solidity Language. The contracts, the data members and member methods that can be called by nodes. Whenever the node calls the method and manipulates any data or performs any transaction, it is stored in the transaction block and when the user only reads the data from the contract then it is not recorded in the transaction block. The contract includes all the important methods and parameters that must be public so that the market is transparent.

The contract is then compiled using the solc package of node. The contract after compilation returns Application Binary Interface (ABI) and Byte Code of the contract. The contract can be called using the functions in the ABI.

For deploying the contract we need an account on Ethereum Network. In this project we have Local Ganache Network and we have several test accounts with balance. We use one of these accounts to deploy the contract. The contract after deployment returns us the address through which it can be called by other nodes in the network.

**VI.3. Simulation using Node**

Now the Solidity Contract is deployed on Ethereum Network ready to be called by the Prosumers (nodes) on the network. Ganache provides us the test accounts with balance. We have the data of 15 houses on an hourly basis that has hourly Demand and Supply of each house in kWh.

To start the simulation we create Agents (Instances of Prosumer Class) that represent each house in the network. And every hour we place Bids and Asks in the market. The market price is determined using Regression Technique at which the market will be settled. If any demand is not met by prosumer then the electricity for that Prosumer is purchased from National Account i.e. from National Grid.

**VI.4. Placing Bids and Asks**

Bid = Requesting Energy from market

Ask = Selling Excess Energy in the market

The Bids and Asks are placed by prosumer using logic that include many parameters i.e. Battery Percentage, Current Demand, Current Supply, Previous day Market Price.

If Current Supply is greater than Current Demand then there is Excess Energy.

If Current Supply is less than Current Demand then there is Lack of Energy.

Here are the flow diagrams showing the Bidding strategy :

1. Bidding Strategy in case of Excess Energy

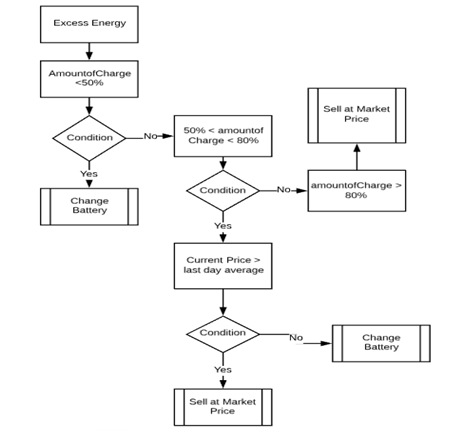


Figure 7. Flow chart for prosumer bidding strategy in case of excess energy

2. Bidding Strategy in case of Lack of Energy

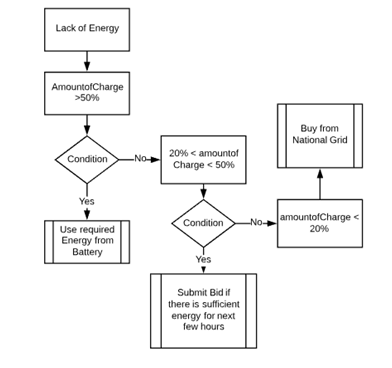


Figure 8. Flow chart for prosumer bidding strategy in case of lack of energy.

**VI.5. Determining Market Price using Regression**

Now we have a list of Bids and Asks each with a different Market Price set by its owner. For a fair settlement of the market we need to calculate the price to be fixed for the transactions to occur. Inside each Bid or Ask we have two parameters :

i. Amount (Wh of the energy the prosumer is asking or selling)

ii. Price (Amount of dollars per kWh of the energy)

Now the regression model is trained on these two parameters using the regression module of node. This module returns the equation in the form of y = mx + c. Since we have two equations, one for Bids and one for Asks, we solve these two equations for x (Amount). After getting x we predict y (Price) using the regression model of Bids. At this price the market will be settled.

**VII. Observations and Screenshots**

After the simulation is complete the results are compiled and returned in the form of JSON objects. The server is created in node using express and the results are displayed on the web page in a user-friendly manner. And for analysis of results the graphs are drawn using python script.

We ran the Energy trading simulation for about 2 weeks. We then stored the simulation results in an output csv file. It included parameters like average demand, average supply, price determined, number of market transactions, number of market trades and number of grid transactions for each hour. We plotted the following graphs using the parameters in the csv file to estimate the efficiency of simulated application.

1. Showing aggregate demand and aggregate supply Vs Time.

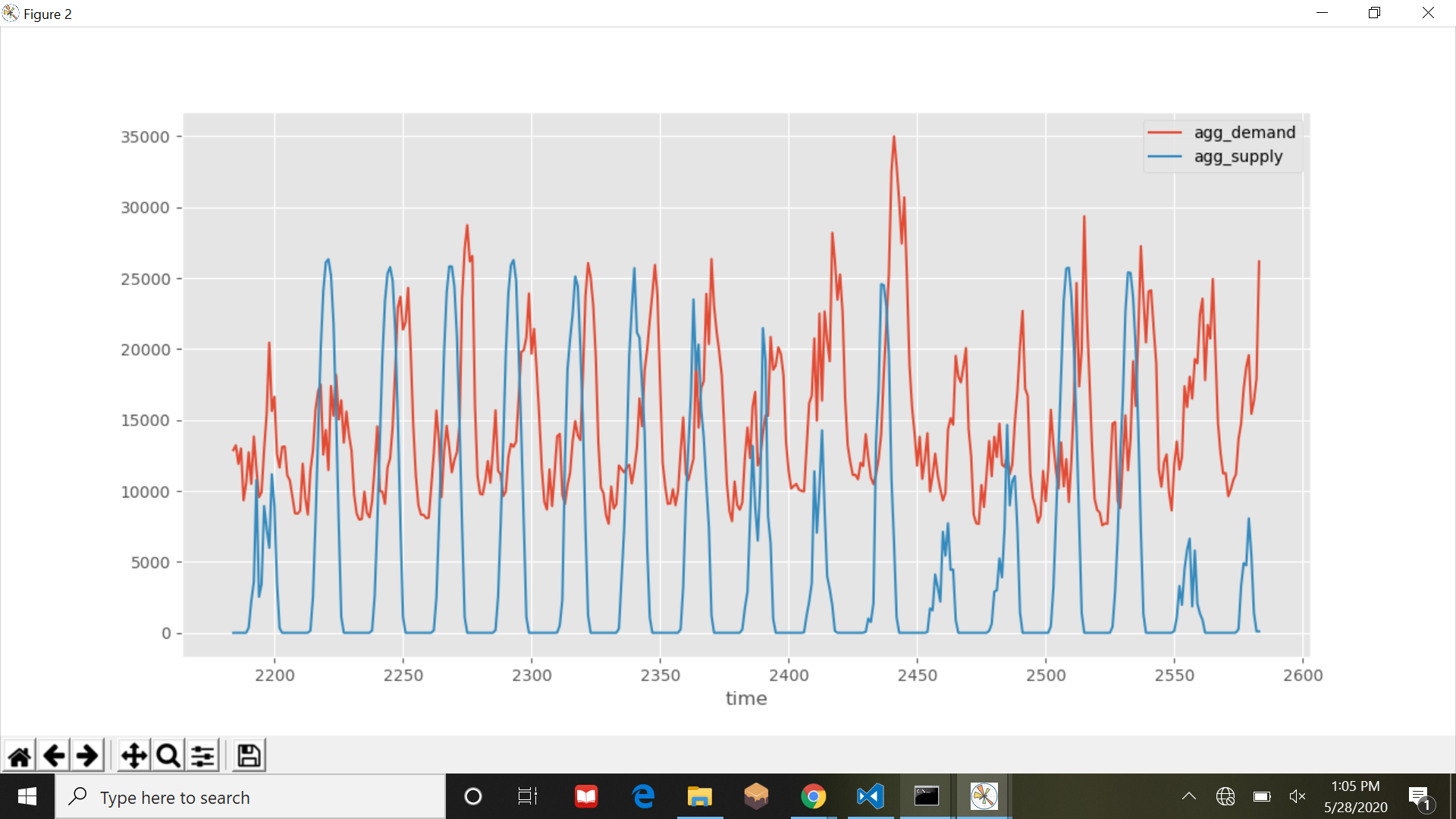


Figure 9. Graph between aggregate demand/supply Vs time

1. Showing market exchange price Vs time.

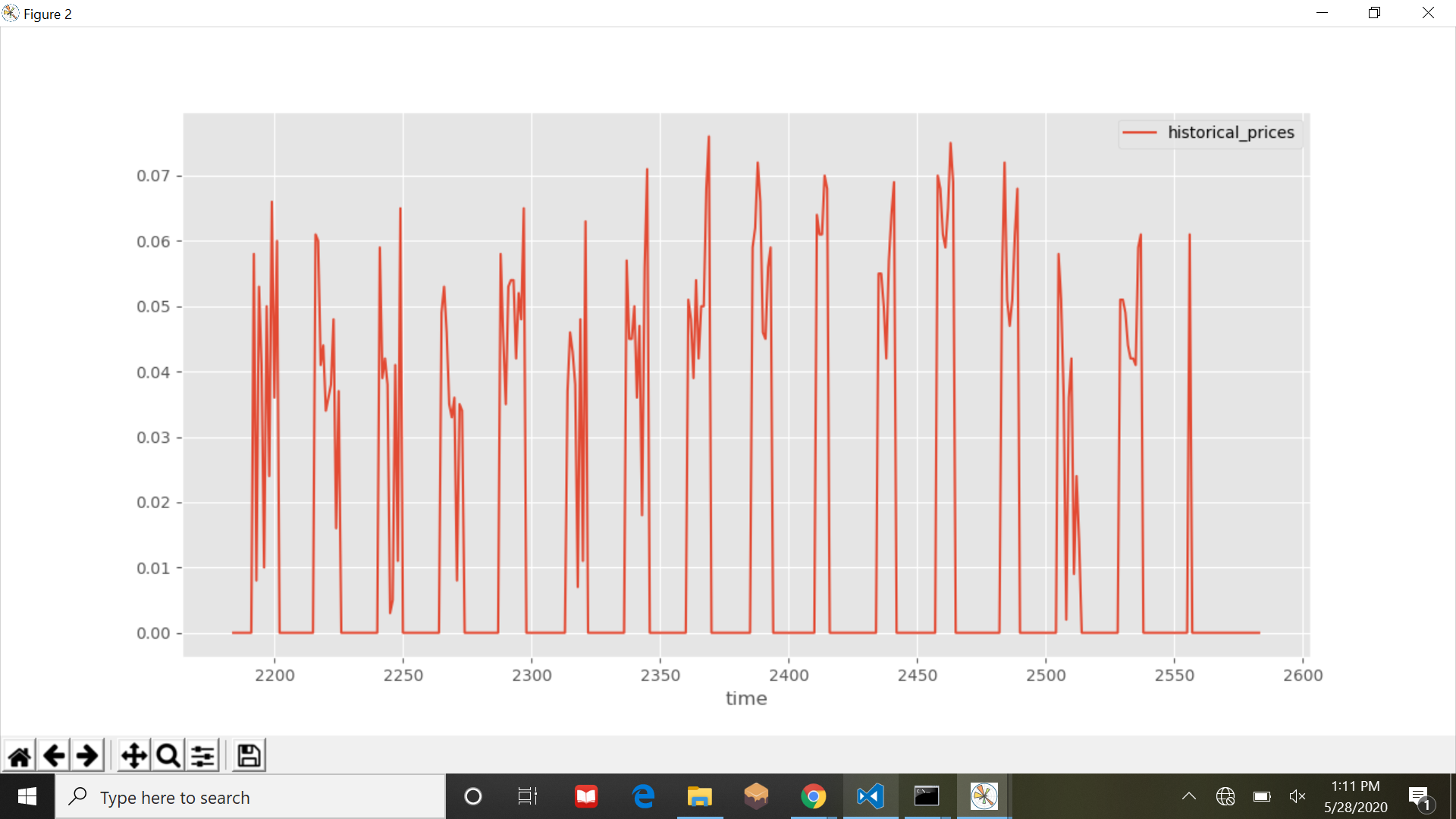


Figure. 10. Graph between Market Exchange Price Vs time

3.Showing market transactions and National Grid Transactions Vs Time.

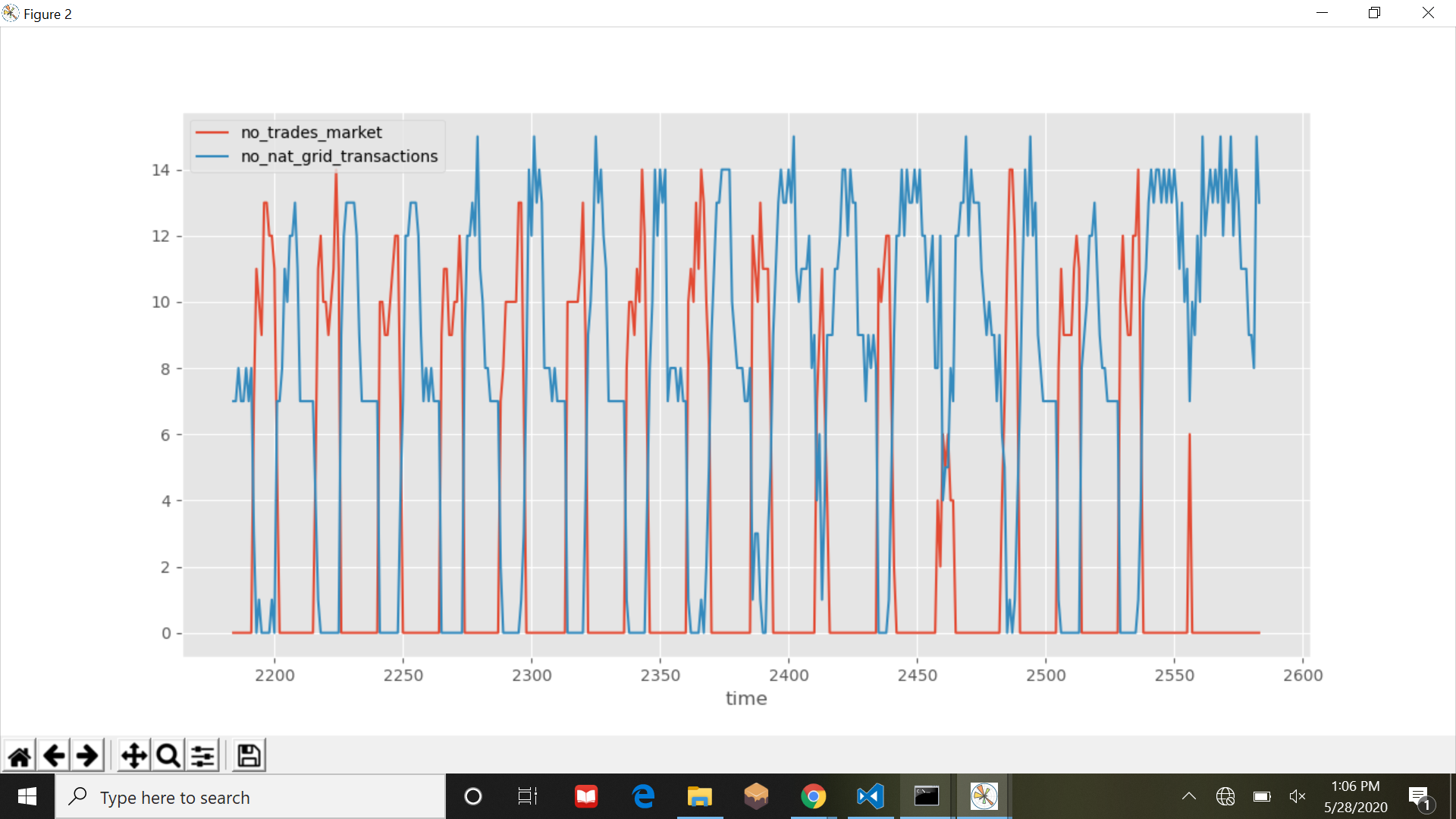


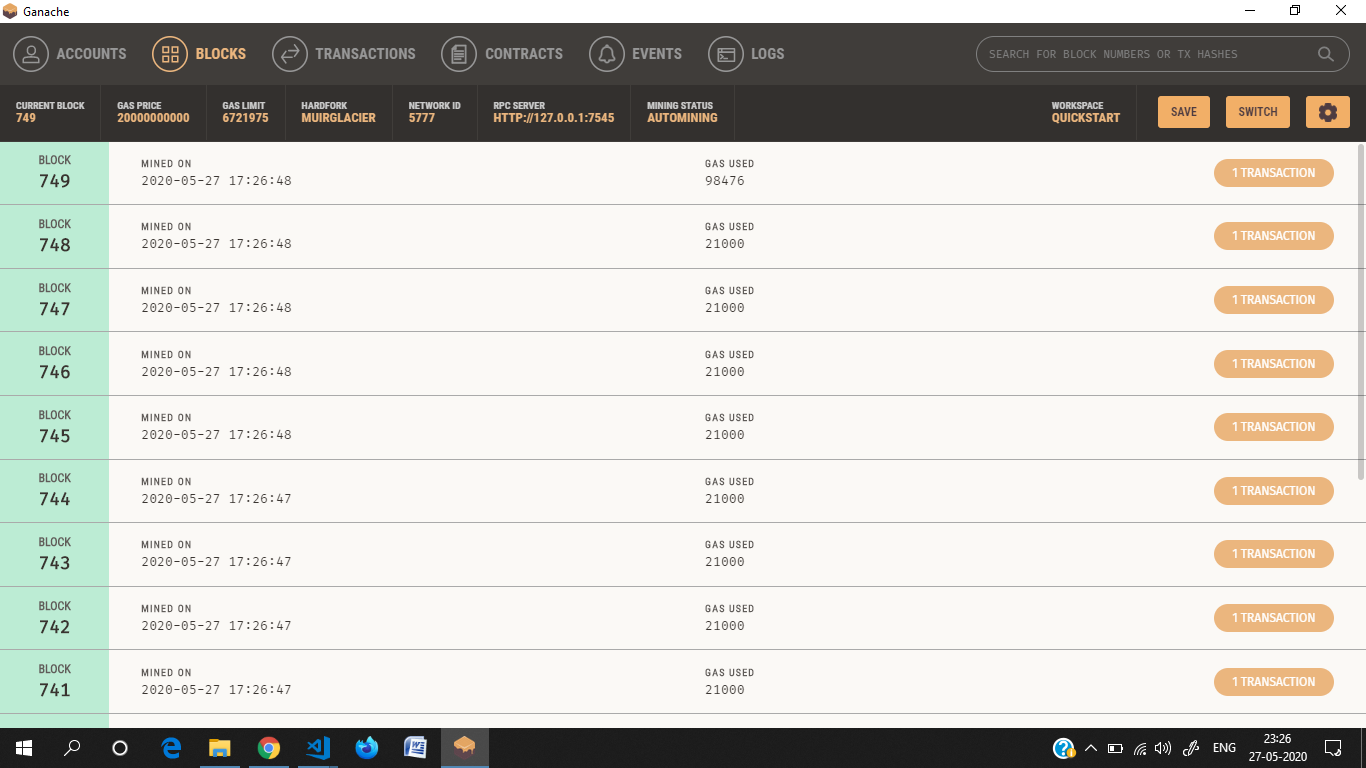
Figure. 11. Graph between number of market trades/ grid transaction Vs time

We can make the following inferences from the plots:

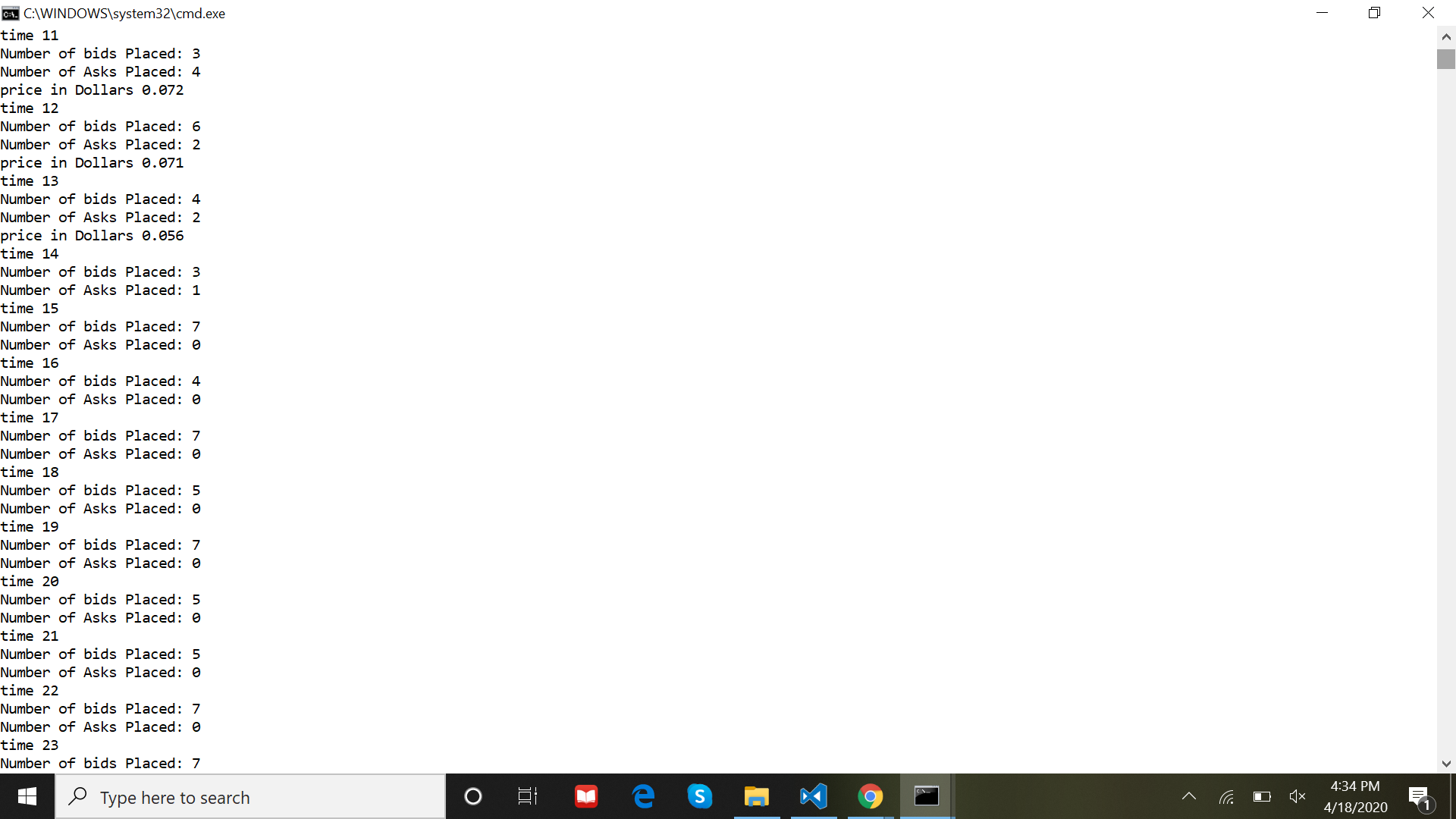
1. The price is proportional to the demand and varies with aggregate demand and supply. We can see that the price is greater when demand is higher.
2. Local market trading is significantly higher during day time as compared to grid transactions as local trading can only be done during day time when sunlight is available. At night, no local trading occurs due to absence of sunlight.
3. Local market price is always lower than the National grid price signifying that this decentralized approach is cheaper than the traditional way of purchasing energy.

**Screenshots**

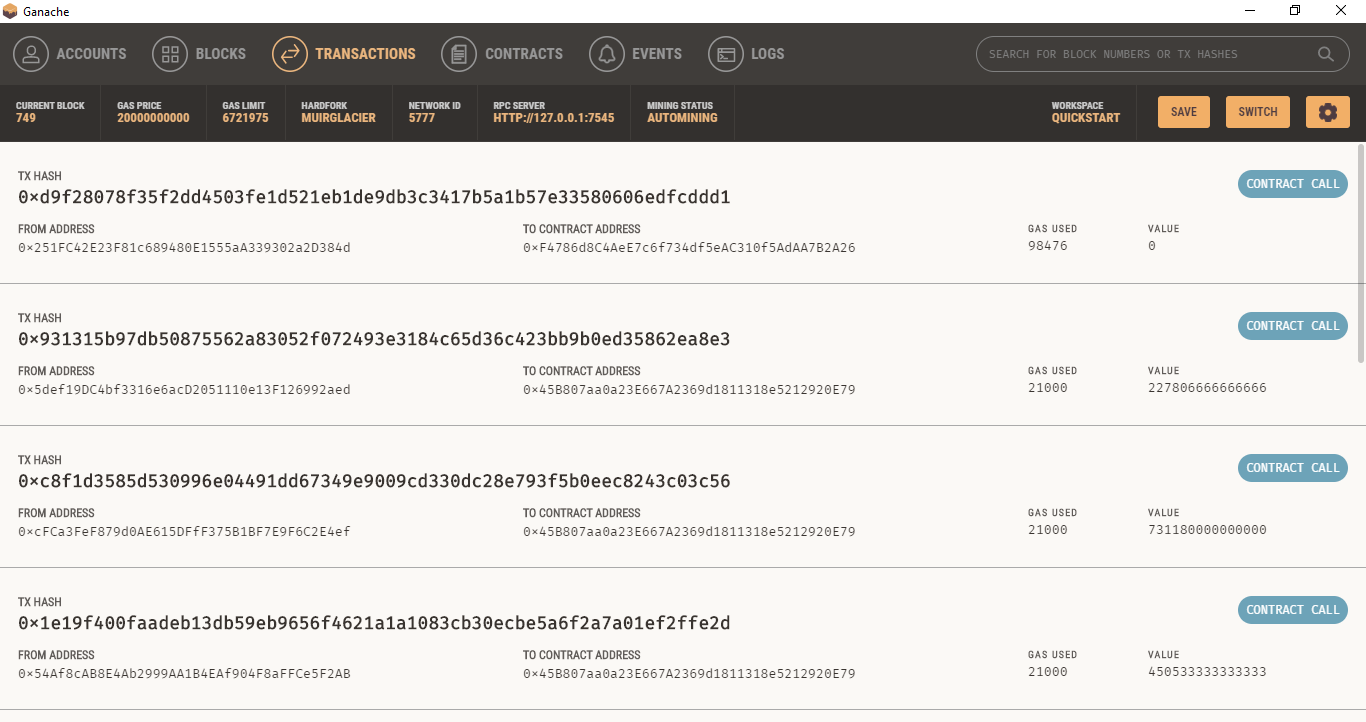
1. Account Addresses and their Private Keys created using Ganache



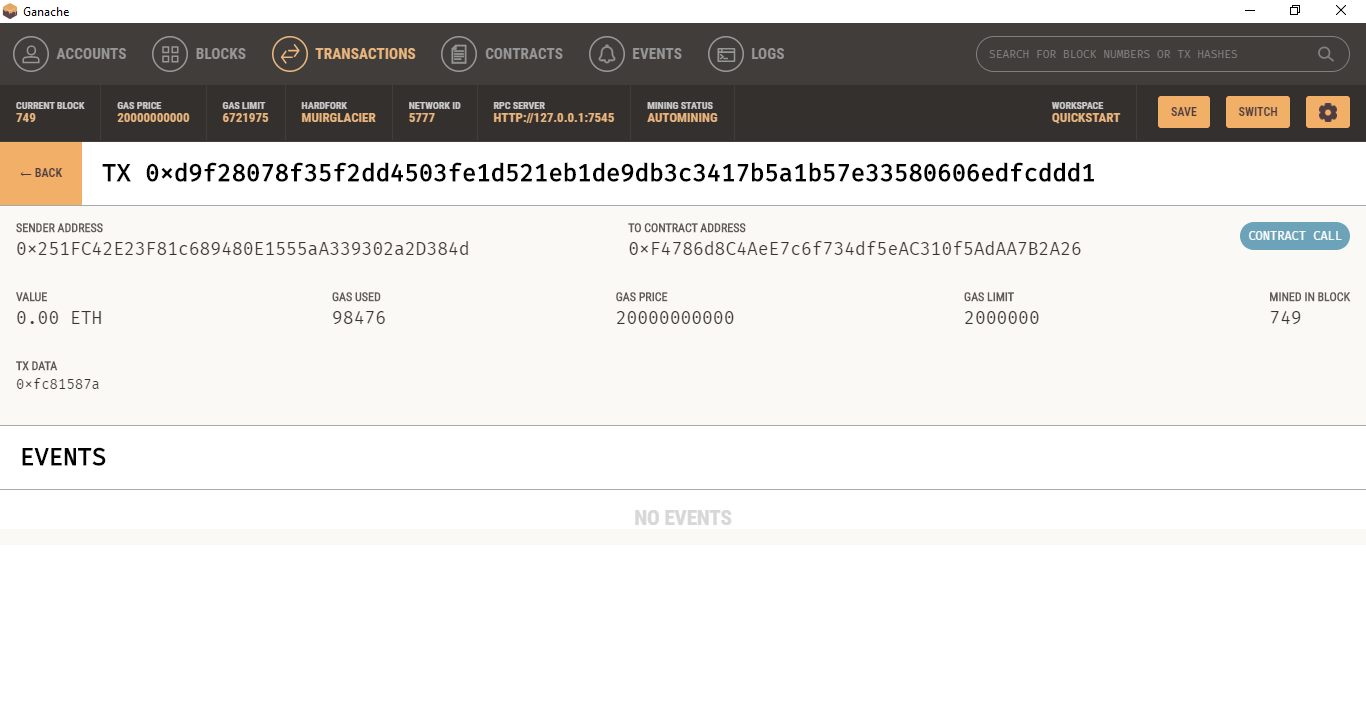
2. Simulation showing the number of Bids and Asks at a time



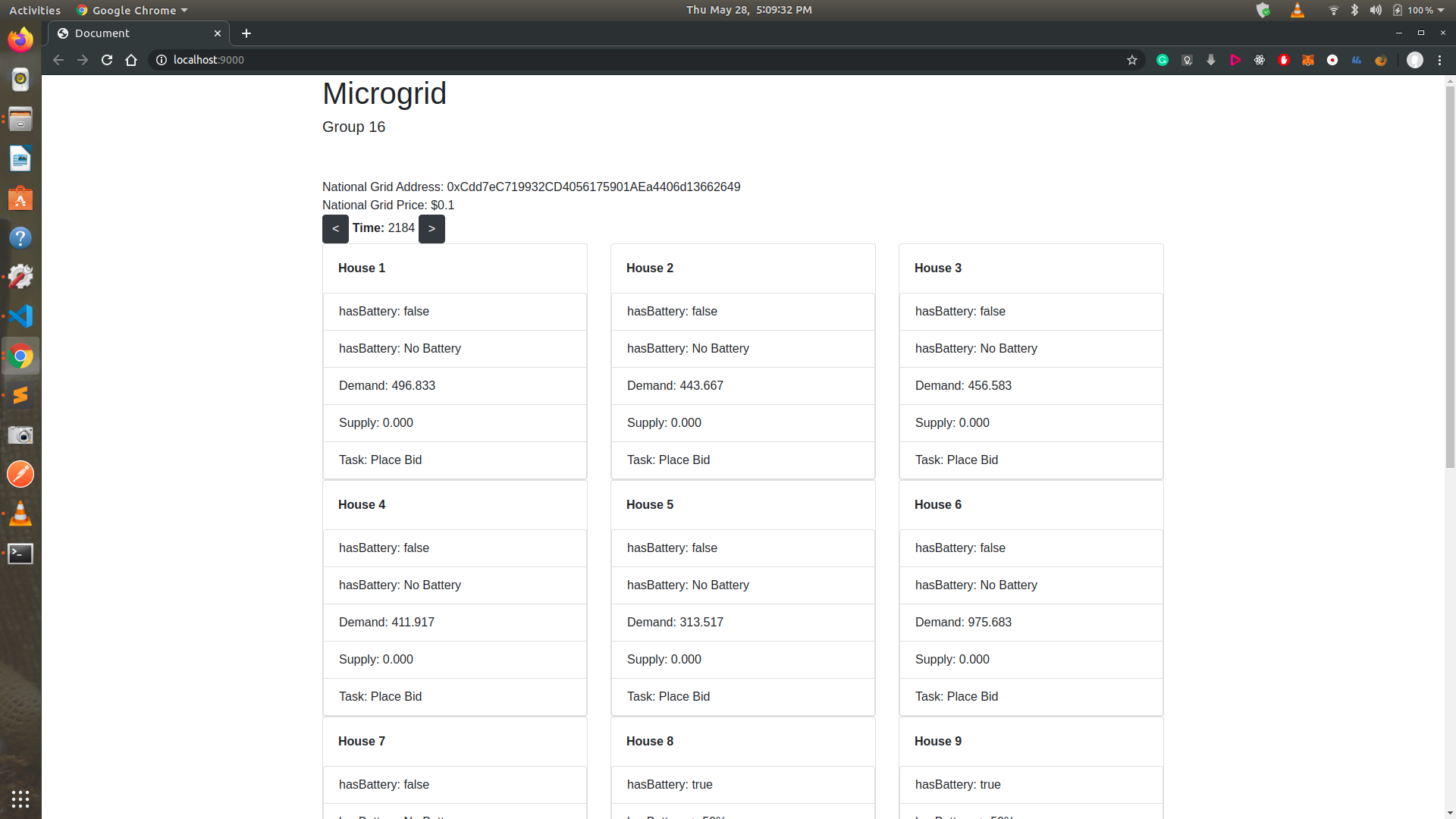
3. The transactions with their detail stored in Blockchain



4. Detail of each transaction in Ganache Local Network



5. Screenshot of frontend of application

****

**VIII. Conclusion and Future Scope**

Through this project we have provided the solution for the traditional centralized system of energy trading by simulating decentralized energy trading using the blockchain technology.

A local energy market was modelled to implement the bidding logic and we formulated the market price through the use of regression. After collection of asks and placing of bids, the bid matching algorithm was used.

We have run the simulation for about two weeks of time and shown inferences of the output recorded by plotting parameters on various graphs. Observing the graphs, we can conclude that the proposed model provides an inexpensive and transparent way to exchange energy. Also it promotes the use of renewable energy which is inexhaustible and free of pollution too.

A user friendly interface showing the simulation and results has also been developed. We can see various important parameters of the transactions at any desired point of time.

As an improvement in the future we can add other renewable energy resources like Biomass, Wind Energy into our microgrid. Doing this will promote the usage of Renewable resources and will reduce the pressure on the Conventional Methods of energy generation. The wastage of power during the transmission will reduce. The customers will also be benefitted as they will get energy at a cheaper rate.

**References**

[1] Mengelkamp, Esther, et al. "A blockchain-based smart grid: towards sustainable local energy markets." *Computer Science-Research and Development* 33.1-2 (2018): 207-214.

[2] Nehaï, Z., and G. Guerard. "Integration of the blockchain in a smart grid model." *Proceedings of the 14th International Conference Of Young Scientists On Energy Issues (CYSENI 2017), Kaunas, Lithuania*. 2017.

[3] Winter, Thomas. "The Advantages and Challenges of the Blockchain for Smart Grids." (2018).

[4] Zhang, Chenghua, et al. "Review of existing peer-to-peer energy trading projects." *Energy Procedia* 105 (2017): 2563-2568.

**APPENDIX**

**SOURCE CODE**

**contracts/Exchange.sol**

pragma solidity ^0.4.17;

contract Exchange {

struct Bid {

address owner;

uint price;

uint amount;

uint date;

}

Bid[] public Bids;

Bid[] public Asks;

function getBid(uint index) public view returns(address, uint, uint, uint){

return (Bids[index].owner, Bids[index].price, Bids[index].amount, Bids[index].date);

}

function getAsk(uint index) public view returns(address, uint, uint, uint){

return (Asks[index].owner, Asks[index].price, Asks[index].amount, Asks[index].date);

}

function placeBid(uint \_price, uint \_amount, uint timestamp) public returns (bool) {

Bid memory b;

b.owner = msg.sender;

b.price = \_price;

b.amount = \_amount;

b.date = timestamp;

Bids.push(b);

return true;

}

function placeAsk(uint \_price, uint \_amount, uint timestamp) public returns (bool) {

Bid memory a;

a.owner = msg.sender;

a.price = \_price;

a.amount = \_amount;

a.date = timestamp;

Asks.push(a);

return true;

}

function removeBid(uint index) public returns(uint){

if (index >= Bids.length)

return Bids.length;

for (uint i = index; i < Bids.length-1 ; i++){

Bids[i] = Bids[i+1];

}

delete Bids[Bids.length - 1];

Bids.length--;

return Bids.length;

}

function removeAsk(uint index) public returns(uint) {

if (index >= Asks.length)

return Asks.length;

for (uint i = index; i < Asks.length-1; i++){

Asks[i] = Asks[i+1];

}

delete Asks[Asks.length - 1];

Asks.length--;

return Asks.length;

}

function clearMarket() public {

delete Bids;

delete Asks;

}

function getBidsCount() public view returns(uint) {

return Bids.length;

}

function getAsksCount() public view returns(uint) {

return Asks.length;

}

}

**public/index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<!-- Jqeury -->

<script src='./jquery-3.5.1.js'></script>

<!-- bootstrap -->

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css" integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js" integrity="sha384-wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>

<!-- <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n" crossorigin="anonymous"></script> -->

<script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>

</head>

<body>

<div class="container">

<h1>Microgrid</h1>

<h5>Group 16</h5>

<br><br>

<div id="nationalGrid">

<span>National Grid Address: </span>

<span id="nationalGridAddress"></span>

<br>

<span>National Grid Price: </span>

<span id="nationalGridPrice"></span>

</div>

<div id="time">

<button id="previousTime" type="button" class="btn btn-dark" onclick="previousTime()"> < </button>

<strong>Time: </strong>

<span></span>

<button id="nextTime" type="button" class="btn btn-dark" onclick="nextTime()"> > </button>

</div>

<div id="houseData" class="row">

</div>

<br>

<div id="bidsAndAsks">

Bid Length: <span id="bidLength"></span>

<br>

Ask Length: <span id="askLength"></span>

</div>

</div>

<script src="./index.js"></script>

</body>

</html>

**public/index.js**

let time = null;

let data = null;

window.addEventListener('load', function() {

$.ajax({

url: 'http://localhost:9000/data',

type: 'GET',

success: function(res, status) {

if(status === 'success') {

// console.log(res);

data = res;

time = 0

printToScreen(0);

}

},

failure: function(err, status) {

console.log(err);

}

})

})

function printForTime(data) {

console.log('time', data);

$('#time').children('span').text(data.time);

let ap = data.purchaseLogic.map((item, idx) => {

return `

<div class='col col-lg-4 col-md-6 col-sm-12'>

<div class="card">

<div class="card-body">

<strong>House ${idx + 1}</strong>

</div>

<div class="card">

<ul class="list-group list-group-flush">

<li class="list-group-item">

hasBattery: <span>${item.hasBattery}</span>

</li>

<li class="list-group-item">

hasBattery: <span>${item.batteryPercentage}</span>

</li>

<li class="list-group-item">

Demand: <span>${item.demand.toFixed(3)}</span>

</li>

<li class="list-group-item">

Supply: <span>${item.supply.toFixed(3)}</span>

</li>

<li class="list-group-item">

Task: <span>${item.task}</span>

</li>

</ul>

</div>

</div>

</div>

`

});

$('#houseData').html(ap);

$('#bidLength').text(data.bidLength);

$('#askLength').text(data.askLength);

}

function previousTime() {

if(time == 0)

return;

time = time - 1;

printToScreen(time);

}

function nextTime() {

if(time == data.data.length)

return;

time = time + 1;

printToScreen(time);

}

function printToScreen(idx) {

$('#nationalGridAddress').text(data.nationalGridAddress);

$('#nationalGridPrice').text('$' + data.nationalGridPrice);

printForTime(data.data[idx]);

}

**compile.js**

const solc = require('solc');

const pth = require('path');

const fs = require('fs-extra');

// create build path

const build\_path = pth.resolve(\_\_dirname, 'build'); //currentdirectoy \_\_dirname;

// contract path

const contract\_path = pth.resolve(\_\_dirname, 'contracts', 'Exchange.sol');

let src = fs.readFileSync(contract\_path, 'UTF-8');

const compiledContracts = solc.compile(src, 1).contracts

fs.ensureDirSync(build\_path);

for(let contr in compiledContracts) {

fs.outputJsonSync(

pth.resolve(

build\_path,

contr.replace(':','') + '.json'

),

compiledContracts[contr]

);

}

config.js

const config = {

'contractAddress': '0x3Ab5d5b98BAAF81a066d65C3FEFA5a5c7E51e8AB',

provider: 'HTTP://127.0.0.1:7545',

debug: true

};

module.exports = config;

**conversion.js**

function convertArrayGasToDollars(array, GASPRICE, WEI\_IN\_ETHER, priceOfEther) {

let sumCost = array.reduce((a, b) => a + b, 0);

let calcPrice = sumCost \* GASPRICE;

let costEther = calcPrice / WEI\_IN\_ETHER;

let costDollars = costEther \* ( parseFloat(priceOfEther.toFixed(18)));

costDollars = parseFloat(costDollars.toFixed(3));

return costDollars;

}

function convertArrayWeiToDollars(arrayWei, WEI\_IN\_ETHER, priceOfEther) {

let sumCost = arrayWei.reduce((a, b) => a + b, 0);

let costEther = sumCost / WEI\_IN\_ETHER;

let costDollars = costEther \* ( parseFloat(priceOfEther.toFixed(18)));

costDollars = parseFloat(costDollars.toFixed(3));

return costDollars;

}

// to connvert Wei to dollars

function convertWeiToDollars(weiValue, WEI\_IN\_ETHER, priceOfEther) {

let costEther = weiValue / WEI\_IN\_ETHER;

let costDollars = costEther \* (parseFloat(priceOfEther.toFixed(18)));

costDollars = parseFloat(costDollars.toFixed(3));

return costDollars;

}

function convertGasToDollars(gasCost, GASPRICE, WEI\_IN\_ETHER, priceOfEther) {

let calcPrice = gasCost \* GASPRICE;

let costEther = calcPrice / WEI\_IN\_ETHER;

let costDollars = costEther \* ( parseFloat(priceOfEther.toFixed(18)));

costDollars = parseFloat(costDollars.toFixed(3));

return costDollars;

}

module.exports = {

convertArrayGasToDollars,

convertArrayWeiToDollars,

convertWeiToDollars,

convertGasToDollars

};

**deploy.js**

const Web3 = require('web3');

let config = require('./config');

const web3 = new Web3(

new Web3.providers.HttpProvider(`${config.provider}`)

);

// requires contract json

const compiledContract = require('./build/Exchange.json');

const deployExchange = async () => {

const account = await web3.eth.getAccounts();

console.log('Attempting to deploy from account 0: ', account[0]);

let Exchange = new web3.eth.Contract(

JSON.parse(compiledContract.interface)

)

let bytecode = compiledContract.bytecode;

let gasEstimate = await web3.eth.estimateGas(

{ data: bytecode }

);

let exchangeInstance = await Exchange.deploy({

data: bytecode

}).send({

from: account[0],

gas: gasEstimate

})

console.log('Contract Address: ', exchangeInstance.options.address);

};

deployExchange();

**exchange.js**

const Web3 = require('web3');

let config = require('./config');

const web3 = new Web3(

new Web3.providers.HttpProvider(`${config.provider}`)

);

const Exchange = require ('./build/Exchange.json');

const instance = new web3.eth.Contract(

JSON.parse(Exchange.interface),

config.contractAddress

);

module.exports = instance;

inter.js

const regression = require('regression');

const algebra = require('algebra.js');

var Fraction = algebra.Fraction;

var Expression = algebra.Expression;

var Equation = algebra.Equation;

const WEI\_IN\_ETHER = 1000000000000000000;

const PRICE\_OF\_ETHER = 250;

function slope(x1, y1, x2, y2) {

if (x1 == x2) return false;

return (y1 - y2) / (x1 - x2);

}

function yInt(x1, y1, x2, y2) {

if (x1 === x2) return y1 === 0 ? 0 : false;

if (y1 === y2) return y1;

return y1 - slope(x1, y1, x2, y2) \* x1 ;

}

function getIntersection(x11, y11, x12, y12, x21, y21, x22, y22) {

var slope1, slope2, yint1, yint2, intx, inty;

if (x11 == x21 && y11 == y21) return [x11, y11];

if (x12 == x22 && y12 == y22) return [x12, y22];

slope1 = slope(x11, y11, x12, y12);

slope2 = slope(x21, y21, x22, y22);

if (slope1 === slope2) return false;

yint1 = yInt(x11, y11, x12, y12);

yint2 = yInt(x21, y21, x22, y22);

if (yint1 === yint2) return yint1 === false ? false : [0, yint1];

if (slope1 === false) return [y21, slope2 \* y21 + yint2];

if (slope2 === false) return [y11, slope1 \* y11 + yint1];

intx = (slope1 \* x11 + yint1 - yint2)/ slope2;

return [intx, slope1 \* intx + yint1];

}

function setCharAt(str,index,chr) {

if(index > str.length-1) return str;

return str.substr(0,index) + chr + str.substr(index+1);

}

function nthIndex(str, pat, n){

var L= str.length, i= -1;

while(n-- && i++<L){

i= str.indexOf(pat, i);

if (i < 0) break;

}

return i;

}

function calculateIntersection(array1, array2){

let array1DescendingPrice = [];

let array2AscendingPrice = [];

// sort according to decreasing prices

array1DescendingPrice = array1.sort(sortDescending); // bids

array2AscendingPrice = array2.sort(sortAscending); //asks

let intersection = [null, null];

let array1x = new Array();

let array1y = new Array();

let array2x = new Array();

let array2y = new Array();

let array1Polynomial = new Array();

let array2Polynomial = new Array();

let array1xsub = Array(array1DescendingPrice.length).fill(0);

let array2xsub = new Array()

for(let i = 0; i< array1DescendingPrice.length; i++) {

for(let j = 0; j <= i; j++) {

array1xsub[i] += array1DescendingPrice[j].amount;

}

array1x.push(array1DescendingPrice[i].amount);

array1y.push(array1DescendingPrice[i].price);

array1Polynomial.push(new Array(array1xsub[i], array1y[i]));

}

array2xsub.push(0);

array2y.push(0);

array2Polynomial.push(new Array(array2xsub[0], array2y[0]));

for(let i = 0; i < array2AscendingPrice.length; i++) {

let value = 0;

for(let j = 0; j <= i; j++) {

value += array2AscendingPrice[j].amount;

}

array2xsub.push(value);

array2x.push(array2AscendingPrice[i].amount);

array2y.push(array2AscendingPrice[i].price);

array2Polynomial.push(new Array(array2xsub[i + 1], array2y[i + 1]));

}

const result1 = regression.linear(array1Polynomial)

const result2 = regression.linear(array2Polynomial);

let equation1 = result1.string;

let equation2 = result2.string;

equation1 = equation1.replace(/\+ -/g, "-");

equation1 = equation1.replace("y =", "");

equation2 = equation2.replace(/\+ -/g, "-");

equation2 = equation2.replace("y =", "");

let equationFinal = `${equation1} = ${equation2}`;

//put into equation and solve

var eq = new algebra.parse(equationFinal);

var ans = eq.solveFor("x");

let possibleIntersections = [];

ans = ans.numer / ans.denom;

let tempResult = result1.predict(ans);

intersection = tempResult;

let minimum = tempResult[1];

if(minimum == Infinity || minimum == undefined) {

minimum = 240000000000000;

}

intersection[1] = parseInt(minimum);

return intersection;

}

function sortDescending(a, b) {

if (a.price === b.price) {

return 0;

}

else {

return (a.price > b.price) ? -1 : 1;

}

}

function sortAscending(a, b) {

if (a.price === b.price) {

return 0;

}

else {

return (a.price < b.price) ? -1 : 1;

}

}

function sortFunctionByAmount(a, b) {

if (a[0] === b[0]) {

return 0;

}

else {

return (a[0] < b[0]) ? -1 : 1;

}

}

module.exports = calculateIntersection;

**prosumer.js**

const Web3 = require('web3');

const config = require('./config');

const { debug } = config;

const web3 = new Web3 (

new Web3.providers.HttpProvider(`${config.provider}`)

);

//compiled contract instance

const exchange = require('./exchange');

// Units of energy in database = kWh

// amount variable in code has energy = Wh unit.

// National Grid Price = $0.1 / kWh

class AgentNationalGrid {

constructor() {

// $0.1 per kWh

this.national\_GridPrice = 0.1;

}

async getAccount(idx) {

let accts = await web3.eth.getAccounts();

this.ethereum\_Address = accts[idx];

return this.ethereum\_Address;

}

}

class Agent{

constructor(batteryCapacity, batteryBool) {

this.timeRow = 0;

this.balance =0;

this.householdAddress = 0;

this.household = 0;

this.nationalGridAddress = 0;

this.hasBattery = batteryBool; // $250 = 1 ether

this.priceOfEther = 250;

this.WEI\_IN\_ETHER = 1000000000000000000;

this.balanceHistory = new Array();

this.batteryCapacity = batteryCapacity;

this.amountOfCharge = batteryCapacity;

this.excessEnergy = 0;

this.shortageEnergy = 0;

this.currentDemand = 0;

this.currentSupply = 0;

// last days data

this.historicalDemand = new Array();

this.historicalSupply = new Array();

this.historicalPrices = new Array();

// history

this.successfulBidHistory = new Array();

this.successfulAskHistory = new Array();

// maintain data

this.nationalGridPurchases = new Array();

// jo b data hra h

this.bidHistory = new Array();

this.askHistory = new Array();

this.householdID = 0;

this.nationalGridPrice = 0.1;

}

async loadSmartMeterData(historicData, householdID) {

this.householdID = householdID;

for (i = 1; i < historicData.length - 1; i++){

let currentDemand = {

time: historicData[i][0],

demand: parseFloat(historicData[i][1]) \* 1000 // kWh to Wh

}

let currentSupply = {

time: historicData[i][0],

supply: parseFloat(historicData[i][2]) \* 1000

}

this.historicalDemand.push(currentDemand);

this.historicalSupply.push(currentSupply);

}

return true;

}

async getAccount(index) {

let accounts = await web3.eth.getAccounts();

this.ethereumAddress = accounts[index];

return this.ethereumAddress;

}

async getAgentBalance() {

let balance = await web3.eth.getBalance(this.ethereumAddress);

this.balance = balance;

return balance;

}

async setAgentBalance() {

let balance = 0;

balance = await web3.eth.getBalance(this.ethereumAddress);

this.balanceHistory.push(balance);

}

async setNationalGrid(nationalGridPrice, nationalGridAddress) {

// $250 = 1 ether

let nationalGridPriceEther = nationalGridPrice / 250;

let nationalGridPriceWei = await web3.utils.toWei(`${nationalGridPriceEther}`, 'ether');

this.nationalGridPrice = nationalGridPriceWei;

this.nationalGridAddress = nationalGridAddress;

}

addSuccessfulAsk(amount) {

let date = (new Date).getTime();

let newReceivedTransaction = {

amount: amount,

date: date,

timeRow: this.timeRow

}

this.successfulAskHistory.push(newReceivedTransaction);

}

async buyFromNationalGrid(amount) {

// buys from national grid, amount units of enery

console.log('national\_GridPrice:', this.nationalGridPrice);

let amountTransaction = (this.nationalGridPrice) \* (amount / 1000);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING FOR API

let returnData = {};

returnData.task = 'Buying From National Grid';

returnData.nationalGridPrice = this.nationalGridPrice;

// returnData.amountTransaction = amountTransaction;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

amountTransaction = parseInt( + amountTransaction.toFixed(18));

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING FOR API

returnData.amountTransaction = amountTransaction;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

let transactionReceipt = null;

try{

if(debug) {

console.log();

console.log('Buying from National grid');

console.log('addresss: ', this.ethereumAddress);

console.log('to: ', this.nationalGridAddress);

console.log('amount: ', amount);

console.log('amountTransaction: ', amountTransaction);

console.log();

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING FOR API

returnData.amount = amount;

returnData.nationalGridAddress = this.nationalGridAddress;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// send funds to national grid

transactionReceipt = await web3.eth.sendTransaction({

to: this.nationalGridAddress,

from: this.ethereumAddress,

value: amountTransaction,

gas: '2000000'

});

} catch (err) {

console.log('Cannot buy from national grid, prosumer:buyFromNationalGrid() ', err);

return null;

};

// current timestamp

let date = (new Date).getTime();

// create transaction object

let newTransactionReceipt = {

transactionReceipt: transactionReceipt,

transactionCost: transactionReceipt.gasUsed,

transactionAmount: amountTransaction,

date: date,

quantity: amount,

timeRow: this.timeRow

}

this.nationalGridPurchases.push(newTransactionReceipt);

this.charge(amount);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING FOR API

returnData.receipt = transactionReceipt;

return returnData;

}

async sendFunds(price, amount, receiver) {

let amountTransaction = price \* (amount / 1000);

amountTransaction = parseInt(amountTransaction); // assuming asmountTransaction is in Wei's

let transactionReceipt = null;

try {

transactionReceipt = await web3.eth.sendTransaction({

to: receiver,

from: this.ethereumAddress,

value: amountTransaction

});

} catch (err) {

console.log('error in sending funds, prosume:L230', err);

return null;

}

let date = (new Date).getTime();

let newTransactionReceipt = {

transactionReceipt: transactionReceipt,

transactionCost: transactionReceipt.gasUsed,

transactionAmount: amountTransaction,

timeRow: this.timeRow,

quantity: amount,

receiver: receiver,

date: date

}

this.successfulBidHistory.push(newTransactionReceipt);

this.charge(amount);

return transactionReceipt;

}

convertWeiToDollars(weiValue) {

let costEther = weiValue / this.WEI\_IN\_ETHER;

let costDollars = costEther \* ( + this.priceOfEther.toFixed(18));

costDollars = + costDollars.toFixed(3);

return costDollars;

}

async placeBuy (price, amount, date) {

let returnData = {};

returnData.type = 'Placing Bid';

returnData.address = this.ethereumAddress;

returnData.amount = amount;

returnData.price = price;

let transactionReceipt = null;

try {

if(debug) {

console.log('Placing Bid');

console.log("address: ", this.ethereumAddress);

console.log('amount: ', amount);

console.log('price: ', price);

}

transactionReceipt = await exchange.methods.placeBid(

Math.floor(price),

Math.floor(amount),

date)

.send({

// from current agents address

from: this.ethereumAddress,

gas: '3000000'

});

} catch(err) {

console.log('Cannot place bid - prosumer:placeBuy() ', err);

return false;

}

let newBid = {

address: this.ethereumAddress,

price: price,

amount: amount,

date: date,

timeRow: this.timeRow,

transactionCost: transactionReceipt.gasUsed

}

this.bidHistory.push(newBid);

returnData.transactionReceipt = transactionReceipt;

return returnData;

}

async placeAsk(price, amount, date) {

let returnData = {};

returnData.price = price;

returnData.amount = amount;

returnData.date = date;

let transactionReceipt = null;

try {

transactionReceipt = await exchange.methods.placeAsk(

Math.floor(price),

Math.floor(amount),

date

).send({

from: this.ethereumAddress,

gas: '3000000'

});

} catch(err) {

console.log('Error in placeAsk, prosumer:L281', err);

return false;

}

let newAsk = {

address: this.ethereumAddress,

price: price,

amount: amount,

date: date,

timeRow: this.timeRow,

transactionCost: transactionReceipt.gasUsed

}

// main ask history record

this.askHistory.push(newAsk);

returnData.action = 'Place Ask';

returnData.receipt = transactionReceipt;

return returnData;

}

// charges your battery

charge(amount) {

this.amountOfCharge += amount;

if(this.amountOfCharge > this.batteryCapacity) {

// 100% charged

this.amountOfCharge = this.batteryCapacity;

}

}

discharge(amount){

this.amountOfCharge -= amount;

if(this.amountOfCharge <= 0) {

this.amountOfCharge = 0;

}

}

setCurrentTime(row) {

this.timeRow = row;

}

async unfilledOrdersProcess() {

let ret = {};

let demand = this.historicalDemand[this.timeRow].demand;

let supply = this.historicalSupply[this.timeRow].supply;

ret.demand = demand;

ret.supply = supply;

let shortageOfEnergy = demand;

if(this.hasBattery)

shortageOfEnergy = demand - supply;

ret.hasBattery = this.hasBattery;

ret.shortageOfEnergy = shortageOfEnergy;

let s = await this.buyFromNationalGrid(shortageOfEnergy);

ret.buyFromNationalGrid = s;

return ret;

}

calculateYesterdayAverage() {

if (this.timeRow - 24 <= 0) {

return this.timeRow - 24;

}

// get the day

let scaledTime = (this.timeRow - 24) / 24;

let startOfDay = Math.floor(scaledTime) \* 24;

let endOfDay = startOfDay + 24;

let sumPrices = 0;

for (let i = startOfDay; i <= endOfDay; i++) {

sumPrices += this.historicalPrices[i]

}

// returns average of yesterday’s historical prices

return sumPrices / 24;

}

async purchaseLogic() {

if(debug) {

console.log('Purchase Logic: ', this.ethereumAddress, ": ", this.timeRow);

}

let demand = this.historicalDemand[this.timeRow].demand;

let supply = this.historicalSupply[this.timeRow].supply;

let excessEnergy = null;

let shortageOfEnergy = null;

let time = (new Date()).getTime();

let price = 0;

let bidsCount = 0;

let bid = 0;

let asksCount = 0;

let ask = 0;

if(supply >= demand) {

excessEnergy = supply - demand;

}

else if(supply < demand) {

shortageOfEnergy = demand - supply;

}

if(debug) {

console.log('excessEnergy: ', excessEnergy);

console.log('shortageOfEnergy: ', shortageOfEnergy);

console.log('Battery: ', this.hasBattery);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING TO API

let returnData = {};

// returnData.address = this.ethereumAddress;

returnData.supply = supply;

returnData.demand = demand;

if(excessEnergy)

returnData.excessEnergy = excessEnergy;

if(shortageOfEnergy)

returnData.shortageEnergy = shortageOfEnergy;

returnData.hasBattery = this.hasBattery;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if(this.hasBattery === true) {

if(excessEnergy !== null) {

// if less than 50%, to use yourself

if (this.amountOfCharge <= 0.5 \* this.batteryCapacity) {

console.log('charging < 50');

this.charge(excessEnergy);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING TO API

returnData.batteryPercentage = '< 50%';

returnData.action = 'Battery Charged';

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}

// if battery is 50% to 80%

else if ((this.amountOfCharge > 0.5 \* this.batteryCapacity) && (this.amountOfCharge < 0.8 \* this.batteryCapacity)) {

bidsCount = await exchange.methods.getBidsCount().call();

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// RETURNING TO API

returnData.batteryPercentage = '50-80%';

returnData.bidsCount = bidsCount;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if(debug) {

console.log('Battery: 50-80%');

console.log('Bids Count: ', bidsCount);

}

if(bidsCount > 0) {

// get last bid

bid = await exchange.methods.getBid(bidsCount - 1).call();

if(this.historicalPrices[this.timeRow - 24] !== null || this.historicalPrices[this.timeRow - 24] !== undefined) {

let averagePrice = this.calculateYesterdayAverage();

if(debug) {

console.log('Ask or charge');

console.log('yesterdayAverage: ', averagePrice);

console.log('lastBidPrice: ', bid[1]);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

returnData.yesterdayAverage = averagePrice;

returnData.lastBidPrice = bid[1];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if(bid[1] >= averagePrice) {

console.log('Placing ask from: ', this.ethereumAddress);

let success = await this.placeAsk(bid[1], Math.floor(excessEnergy), time);

if(typeof(success) === 'boolean' && success === false) {

// error occurred

console.log('Could not place Ask, prosumer:purchaseLogic,bidsCount>0');

console.log('Ask Error');

return returnData;

}

returnData.task = 'Ask Placed';

}

// if last bids price is less than the yesterday's average price

else if(bid[1] < averagePrice) {

returnData.task = 'Charge Battery';

this.charge(excessEnergy);

}

}

else

{

let success = await this.placeAsk(

bid[1],

Math.floor(excessEnergy),

time

);

if(typeof(success) === 'boolean' && success === false) {

// error occurred

console.log('Could not place Ask, prosumer:purchaseLogic()bid > 0 else');

console.log('Place Ask Error');

return returnData;

}

returnData.task = 'Ask Placed';

}

}

else {

if(debug) {

console.log('Charged Own Battery')

}

returnData.task = 'Charged Battery';

this.charge(excessEnergy);

}

}

// if battery is more than 80%

else if (this.amountOfCharge >= this.batteryCapacity \* 0.8 ) {

if(debug) {

console.log('Battery > 80%');

}

// TODO - why energy multiplied by 100

excessEnergy \*= 100;

// random price generated

price = generateRandomPriceInDollar();

price = await this.convertToWei(price);

// ask place for this pricr

let success = await this.placeAsk(

price,

Math.floor(excessEnergy),

time

);

if(typeof(success) === 'boolean' && success === false) {

console.log('Could not place Ask, prosumer:L493');

return;

}

returnData.batteryPercentage = '> 80 %';

returnData.task = 'Ask Placed';

}

}

else if (shortageOfEnergy !== null) {

// if battery is more than 50%

if (this.amountOfCharge >= 0.5 \* this.batteryCapacity) {

if(debug) {

console.log();

console.log('Battery > 50%');

console.log('Discharging: ', shortageOfEnergy);

console.log();

}

returnData.batteryPercentage = '> 50%';

returnData.task = 'Battery Discharged';

this.discharge(shortageOfEnergy);

}

// if battery is between 20% to 50%

else if(this.amountOfCharge < 0.5 \* this.batteryCapacity && this.amountOfCharge > 0.2 \* this.batteryCapacity){

let price = generateRandomPriceInDollar();

let amount = this.formulateAmount();

if(debug) {

console.log('Battery 20-50%');

console.log('formulateAmount: ', amount);

}

if(typeof(amount) === 'boolean' && amount === false) {

return;

}

// convert random price to dollar

price = await this.convertToWei(price);

// place the bid

let success = await this.placeBuy(

Math.floor(price),

Math.floor(amount),

time

);

if(typeof(success) === 'boolean' && success === false) {

// error occured

console.log('Could not place buy, pronsumer:purchaseLogic() Battery@20-50');

console.log('Place Buy Error');

returnData.task = 'Place Buy Error';

return returnData;

}

returnData.batteryPercentage = '20-50%';

returnData.task = 'Place Bid';

}

else if (this.amountOfCharge <= 0.2 \* this.batteryCapacity) {

if(debug) {

console.log('Battery < 20%, buying shortage from national grid');

}

await this.buyFromNationalGrid(0.5 \* this.batteryCapacity);

returnData.batteryPercentage = '< 20%';

returnData.task = 'Buy From National Grid';

}

}

}

else {

if(shortageOfEnergy === null || shortageOfEnergy === undefined) {

returnData.task = 'Nothing';

return returnData;

}

shortageOfEnergy = demand;

price = generateRandomPriceInDollar();

price = await this.convertToWei(price);

let success = await this.placeBuy(price, shortageOfEnergy, time);

if(typeof(success) === 'boolean' && success === false) {

// error occured

console.log('Could not place Ask, purchaseLogic:noBattery');

console.log('No Battery, Place Buy Error');

return returnData;

}

returnData.batteryPercentage = 'No Battery';

returnData.task = 'Place Bid';

}

if(debug) {

console.log();

console.log();

}

console.log(returnData);

return returnData;

}

formulateAmount() {

let timeInterval = 10;

let supplySum = 0;

let demandSum = 0;

let energyNeeded = 0;

for(let i = this.timeRow ; i < this.timeRow + timeInterval; i++) {

supplySum += this.historicalSupply[i].supply;

demandSum += this.historicalDemand[i].demand;

}

if(supplySum - demandSum >= 0) {

return false;

}

if(supplySum - demandSum < 0) {

energyNeeded = Math.abs(supplySum - demandSum);

}

if(this.amountOfCharge + energyNeeded >= this.batteryCapacity) {

energyNeeded = this.batteryCapacity - this.amountOfCharge;

}

return energyNeeded;

}

// converts price in dollar to Wei

async convertToWei(price) {

let ethers = (price / this.priceOfEther);

ethers = ethers.toFixed(18);

try {

price = await web3.utils.toWei(ethers, 'ether');

} catch(err) {

console.log('Error - prosumer.js:convertToWei() ', err);

};

price = parseInt(price);

// returns price in Wei

return price;

}

}

function generateRandomPriceInDollar() {

// return Math.random() \* 0.03 + 0.04;

return 0.1;

}

module.exports = {

Agent,

AgentNationalGrid

};

**server.js**

const express = require('express');

const fs = require('fs');

const dd = require('./API.json');

const simulation = require('./simulation');

const app = express();

app.use(express.static('public'));

let ret = null;

async function run() {

if(ret === null)

ret = await simulation();

// let data = {

// success: true,

// data: ret

// };

fs.writeFileSync('./result.json', JSON.stringify(ret), err => {

if(err) {

console.log('Cant write to file: ', err);

} else {

console.log('File written');

}

})

}

// run();

app.use('/data', async (req, res) => {

res.status(200).json(dd);

});

const PORT = 9000;

app.listen(PORT, (err) => {

if(err) {

console.log("Could not start server");

console.log('Error: ', err);

return;

}

console.log(`Server started at http://localhost:${PORT}`);

})

simulation.js

// const ganache = require('ganache-cli');

const Web3 = require('web3');

// requires

let fs = require('fs');

var csv = require('fast-csv');

let csvWriteStream = require('csv-write-stream');

let parse = require('csv-parse');

let async = require('async');

let calculateIntersection = require('./inter');

const config = require('./config');

const { debug } = config;

// Agent and NationalGridAgent

const { Agent, AgentNationalGrid } = require('./prosumer');

//compiled contracts

const exchange = require('./exchange');

// functions imports

const { readCSV }= require('./utils');

const web3 = new Web3(

new Web3.providers.HttpProvider(`${config.provider}`)

);

const {

convertArrayGasToDollars,

convertArrayWeiToDollars,

convertWeiToDollars,

convertGasToDollars

} = require('./conversion');

let id = new Array();

let agentsNoBattery = new Array();

let agentsBattery = new Array();

let numberOfBids = new Array();

// constants

const outputFile = 'output.csv';

const defaultBatteryCapacity = 12000;

const GASPRICE = 20000000000; // 20 GWEI

const PRICE\_OF\_ETHER = 250; // $250 = 1 ether

const WEI\_IN\_ETHER = 1000000000000000000; // 10^18

const NATIONAL\_GRID\_PRICE = 0.1;

// RUN THIS FUNCTION FOR SIMULATION

// init();

async function init() {

let unFilledBids = new Array();

let unFilledAsks = new Array();

let aggregatedDemand = new Array();

let aggregatedSupply = new Array();

let historicalPricesPlot = new Array();

var accounts = await web3.eth.getAccounts();

let { householdHistoricData } = await getFiles();

let {

agents,

agentNationalGrid

} = await createAgents(householdHistoricData, defaultBatteryCapacity, false);

let agentsBattery = agents;

// last account as national grid address

let nationalGridAddress = await agentNationalGrid.getAccount(accounts.length - 1);

console.log(`${agentsBattery.length} agents created.`);

console.log('Starting Simulation');

let dataToReturnApi = [];

dataToReturnApi.nationalGridAddress = nationalGridAddress;

dataToReturnApi.NATIONAL\_GRID\_PRICE = NATIONAL\_GRID\_PRICE;

if(debug) {

console.log();

console.log('NATIONAL\_GRID\_PRICE: ', NATIONAL\_GRID\_PRICE);

console.log('nationalGridAddress: ', nationalGridAddress);

console.log();

}

let timeArray = new Array();

for (let i = 2184; i < 2232; i++) {

timeArray.push(i);

let retData = {};

retData.time = i;

if(debug)

console.log('Time: ', i, '\n');

let agentPurchaseLogic = [];

for (let j = 0; j < agentsBattery.length; j++) {

agentsBattery[j].agent.setCurrentTime(i);

if(i == 2184) {

await agentsBattery[j].agent.setNationalGrid(

NATIONAL\_GRID\_PRICE,

nationalGridAddress

);

}

// run purchase logic for the each agent at that time

try{

let some = await agentsBattery[j].agent.purchaseLogic();

if(some === null || some === undefined) {

console.log('Some= ', some);

console.log('TIme: ', i);

console.log('Agent: ', j);

}

agentPurchaseLogic.push(some);

} catch(err){

console.log('error from purchase logic', err);

}

}

retData.purchaseLogic = agentPurchaseLogic;

// all bids and asks in array

let { bids, asks } = await getExchangeBids();

console.log('Bid Length: ', bids.length);

console.log('Ask Length: ', asks.length);

retData.bidLength = bids.length;

retData.askLength = asks.length;

if (bids.length >= 2 && asks.length >= 2) {

let intersection = calculateIntersection(bids, asks);

let priceDollars = convertWeiToDollars(intersection[1], WEI\_IN\_ETHER, PRICE\_OF\_ETHER);

console.log('price in Dollars', priceDollars);

let paidBids = new Array();

// bid/ask sorted in desc by the amount

bids = bids.sort(sortByAmount);

asks = asks.sort(sortByAmount);

numberOfBids.push(bids.length);

for (let j = 0; j < agentsBattery.length; j++) {

agentsBattery[j].agent.historicalPrices[i] = intersection[1];

}

let {

bids: unfilledBids,

asks: unfilledAsks,

agentsBattery: agentsBattery2

} = await matchBids(bids.length - 1, asks.length - 1, bids, asks, agentsBattery, intersection);

bids = unfilledBids;

asks = unfilledAsks;

agentsBattery = agentsBattery2;

// unfulfilled bids

if(bids.length > 0) {

for (let i = 0; i < bids.length; i++) {

let obj = agentsBattery.find(function (obj) {

return obj.agentAccount === bids[i].address;

});

obj.agent.unfilledOrdersProcess();

unFilledBids.push(bids[i]);

}

}

// ask unfullfilled

if(asks.length > 0) {

for (let i = 0; i < asks.length; i++) {

let obj = agentsBattery.find(function (obj) {

return obj.agentAccount === asks[i].address;

});

obj.agent.charge(asks[i].amount);

unFilledAsks.push(asks[i]);

}

}

try{

console.log('clearing Market');

await clearMarket();

} catch(err) {

console.log('Error while trying to clear market, simulation:L213', err);

}

}

// one of them can be large

else if (bids.length < 2 || asks.length < 2) {

numberOfBids.push(bids.length);

for (let j = 0; j < bids.length; j++) {

unFilledBids.push(bids[j]);

let obj = agentsBattery.find(function (obj) {

return obj.agentAccount === bids[j].address;

});

if(!obj) {

if(debug) {

console.log('Bidder not found, simulation:bids<2asks<2', j, i);

console.log('For bid', bids[j]);

}

continue;

}

let s = obj.agent.unfilledOrdersProcess();

}

for (let x = 0; x < asks.length; x++) {

unFilledAsks.push(asks[x]);

let obj = agentsBattery.find(function (obj) {

return obj.agentAccount === asks[x].address;

});

if(!obj) {

if(debug) {

console.log('Asker not found simulation:bids<2asks<2', x, i);

console.log('For ask', asks[x]);

}

continue;

}

obj.agent.charge(asks[x].amount);

}

for (let j = 0; j < agentsBattery.length; j++) {

agentsBattery[j].agent.historicalPrices[i] = 0;

}

try {

await clearMarket();

} catch(err) {

console.log('Error clearMarket, simulation:bids<2asks<2 ', err);

}

}

dataToReturnApi.push(retData);

}

let agentBalanceAverage = new Array();

let history = agentsBattery[0].agent.historicalPrices;

let aggActualDemand = new Array();

let transactionCostBid = new Array();

let transactionCostAsk = new Array();

let transactionCostAvg = new Array();

let transactionCost = new Array();

let nationalGridBidsAggAmount= new Array();

let nationalGridBidsAggGas = new Array();

let nationalGridPurchases = new Array();

let nationalGridTotalCost = new Array();

let totalNumberTransactions = new Array();

let successfulBidsAggAmount = new Array();

let successfulBidsAggGas = new Array();

let successfulBidsTotalCost = new Array();

let totalExpenditureHourly = new Array();

let totalExpenditure = new Array();

//averages parameters (for each agent)

let averageNumberTransactions = new Array();

let averageNationalGridPurchases = new Array();

let agent;

let simulationCSV = new Array();

let csvData = new Array();

const sumPrices= history.reduce((a, b) => a + b, 0);

for (let i = 2184; i < 2564 ; i++) {

let demand = new Array();

let supply = new Array();

let gasCostBids = new Array();

let gasCostAsks = new Array();

let nationalGridBidsGas = new Array();

let successfulBidsGas = new Array();

historicalPricesPlot[i] = convertWeiToDollars(

agentsBattery[0].agent.historicalPrices[i],

WEI\_IN\_ETHER,

PRICE\_OF\_ETHER

);

for (let j = 0; j < agentsBattery.length; j++) {

demand.push(agentsBattery[j].agent.historicalDemand[i].demand);

if(j>=8)

supply.push(agentsBattery[j].agent.historicalSupply[i].supply);

for(let k = 0; k < agentsBattery[j].agent.bidHistory.length; k++) {

if(agentsBattery[j].agent.bidHistory[k].timeRow == i){

gasCostBids.push(agentsBattery[j].agent.bidHistory[k].transactionCost);

}

}

for(let z=0; z < agentsBattery[j].agent.askHistory.length; z++) {

if(agentsBattery[j].agent.askHistory[z].timeRow == i){

gasCostAsks.push(agentsBattery[j].agent.askHistory[z].transactionCost);

}

}

for(let k = 0; k < agentsBattery[j].agent.successfulBidHistory.length; k++) {

if (agentsBattery[j].agent.successfulBidHistory[k].timeRow == i) {

successfulBidsGas.push(agentsBattery[j].agent.successfulBidHistory[k].transactionCost);

}

}

for(let k=0; k < agentsBattery[j].agent.nationalGridPurchases.length; k++) {

if (agentsBattery[j].agent.nationalGridPurchases[k].timeRow == i) {

nationalGridBidsGas.push(agentsBattery[j].agent.nationalGridPurchases[k].transactionCost);

}

}

}

if(gasCostBids.length > 0) {

let bidCostDollars = convertArrayGasToDollars(gasCostBids, GASPRICE, WEI\_IN\_ETHER, PRICE\_OF\_ETHER);

transactionCostBid[i] = bidCostDollars;

}

else if(gasCostBids.length == 0) {

transactionCostBid[i] = 0;

}

if(gasCostAsks.length > 0) {

let askCostDollars = await convertArrayGasToDollars(gasCostAsks, GASPRICE, WEI\_IN\_ETHER, PRICE\_OF\_ETHER);

transactionCostAsk[i] = askCostDollars;

}

else if(gasCostAsks.length == 0) {

transactionCostAsk[i] = 0;

}

let sumTransactions = nationalGridBidsGas.length + gasCostAsks.length + gasCostBids.length + successfulBidsGas.length;

totalNumberTransactions.push(sumTransactions);

// numbermarket = total - national grid

let numberMarketTransactions = gasCostAsks.length + gasCostBids.length + successfulBidsGas.length;

// accumulate demand and supply

const sumDemand = demand.reduce((a, b) => a + b, 0);

const sumSupply = supply.reduce((a, b) => a + b, 0);

// aggregates demand and supply

aggregatedDemand[i] = sumDemand;

aggregatedSupply[i] = sumSupply;

// create new csv entry

let newCsvEntry = {

time: i,

agg\_demand: aggregatedDemand[i],

agg\_supply: aggregatedSupply[i],

historical\_prices: historicalPricesPlot[i],

no\_total\_transactions: totalNumberTransactions[i-2184],

no\_trades\_market: successfulBidsGas.length,

no\_market\_transactions: numberMarketTransactions,

no\_nat\_grid\_transactions: nationalGridBidsGas.length,

}

// push to csv data

csvData.push(newCsvEntry);

}

// write to output.csv file

console.log(`writing results of simulation to csv file : ${outputFile}`);

// var csvStream = csv.createWriteStream({ headers: true }),

let csvStream = csvWriteStream();

writableStream = fs.createWriteStream(outputFile);

writableStream.on('finish', function () {

console.log('DONE!');

});

csvStream.pipe(writableStream);

for(let i = 0; i < csvData.length; i++) {

csvStream.write(csvData[i]);

}

csvStream.end();

console.log('Returnning Data for API');

return {

data: dataToReturnApi,

nationalGridAddress: nationalGridAddress,

nationalGridPrice: NATIONAL\_GRID\_PRICE

};

};

async function getFiles() {

console.log('readings files...');

let householdHistoricData = new Array();

for (i = 1; i <= 15; i++){

householdHistoricData.push(

await readCSV(`./dataset/house${i}.csv`)

);

}

return {

householdHistoricData

};

}

async function createAgents(householdHistoricData, batteryCapacity, batteryBool) {

console.log('Creating Agents...');

// national grid agent

let agentNationalGrid = new AgentNationalGrid();

// battery household agents

let agents = new Array();

for (let i = 1; i <= 15; i++) {

if(i >= 8) {

batteryBool = true;

}

// create agent for each house

agent = new Agent(batteryCapacity, batteryBool);

agentAccount = await agent.getAccount(i);

await agent.loadSmartMeterData(

householdHistoricData[i - 1],

i

);

let newAgent = {

id: i,

agent,

agentAccount

}

agents.push(newAgent);

}

return {

agents,

agentNationalGrid

};

}

async function getExchangeBids() {

// variable declaration

let bids = new Array();

let asks = new Array();

let bid = 0;

let ask = 0;

// get bids and ask count

let bidsCount = await exchange.methods.getBidsCount().call();

let asksCount = await exchange.methods.getAsksCount().call();

console.log("Number of bids Placed: " + bidsCount);

console.log("Number of Asks Placed: " + asksCount);

for (let i = 0; i <= bidsCount - 1 ; i++) {

try {

bid = await exchange.methods.getBid(i).call();

} catch (err) {

console.log('Error in getting bid, simulation: L507')

}

// get bid date

let date = new Date(parseInt(bid[3]));

date = date.toLocaleString();

if(bid[0] === '0x0000000000000000000000000000000000000000')

{

if(debug) {

console.log('Zero Bid', i, ' with ', bidsCount);

}

continue;

}

newBid = {

price: parseInt(bid[1]),

amount: parseInt(bid[2]),

address: bid[0],

date: date

}

// add to bids array

bids.push(newBid);

}

for (let j = 0; j <= asksCount - 1; j++) {

try {

ask = await exchange.methods.getAsk(j).call();

} catch(err){

console.log('Error in getting ask, simlation:L468', err);

}

let date = new Date(parseInt(ask[3]));

date = date.toLocaleString();

if(ask[0] === '0x0000000000000000000000000000000000000000')

{

console.log('Zero Ask', i, ' with ', asksCount);

continue;

}

newAsk = {

price: parseInt(ask[1]),

amount: parseInt(ask[2]),

address: ask[0],

date: date

}

asks.push(newAsk);

}

// return all bids and asks

return {

bids,

asks

};

}

//decreasing amount

function sortByAmount(a, b) {

if (a.amount === b.amount) {

return 0;

}

else {

return (a.amount > b.amount) ? -1 : 1;

}

}

async function clearMarket() {

let bidsCount = await exchange.methods.getBidsCount().call();

let asksCount = await exchange.methods.getAsksCount().call();

let accounts = await web3.eth.getAccounts();

if(debug) {

console.log('clearMarket()')

console.log('bidsCount: ', bidsCount);

console.log('asksCount: ', asksCount);

}

await exchange.methods.clearMarket().send({

from: accounts[accounts.length - 2],

gas: '2000000'

})

bidsCount = await exchange.methods.getBidsCount().call();

asksCount = await exchange.methods.getAsksCount().call();

if(debug) {

console.log('Market Cleared');

console.log('bidsCount: ', bidsCount);

console.log('asksCount: ', asksCount);

}

}

async function matchBids(bid\_index, ask\_index, bids, asks, agentsBattery, intersection) {

if (bids.length == 0 || asks.length == 0) {

return { bids, asks, agentsBattery};

}

let obj = agentsBattery.find(function (obj) {

return obj.agentAccount === bids[bid\_index].address;

});

// if buyer wants more than seller

if(bids[bid\_index].amount - asks[ask\_index].amount >= 0) {

let remainder = bids[bid\_index].amount - asks[ask\_index].amount;

let calcAmount = asks[ask\_index].amount;

await obj.agent.sendFunds(intersection[1], calcAmount, asks[ask\_index].address);

let objSeller = agentsBattery.find(function (obj) {

return obj.agentAccount === asks[ask\_index].address;

});

// objSeller.agent.discharge(calcAmount);

objSeller.agent.addSuccessfulAsk(calcAmount);

bids[bid\_index].amount = remainder;

if(remainder === 0) {

bids.splice(bid\_index, 1);

}

asks.splice(ask\_index, 1);

// recursive call

return (matchBids(bids.length-1, asks.length-1, bids, asks, agentsBattery, intersection));

}

// buyer needs less than than the giver

else if(bids[bid\_index].amount - asks[ask\_index].amount < 0) {

let remainder = asks[ask\_index].amount - bids[bid\_index].amount;

let calcAmount = bids[bid\_index].amount;

await obj.agent.sendFunds(intersection[1], calcAmount, asks[ask\_index].address);

let objSeller = agentsBattery.find(function (obj) { return obj.agentAccount === asks[ask\_index].address; });

objSeller.agent.discharge(calcAmount);

objSeller.agent.addSuccessfulAsk(calcAmount);

asks[ask\_index].amount = remainder;

if(remainder == 0) {

asks.splice(ask\_index, 1);

}

bids.splice(bid\_index, 1);

// recursive call for remaining indexes

return (matchBids(bids.length-1, asks.length-1, bids, asks, agentsBattery, intersection));

}

}

module.exports = init;

**utils.js**

var fs = require('fs');

let csv = require('fast-csv');

let csvParser = require('csv-parser');

// fast csv method

async function readCSV (inputFile) {

return new Promise((resolve, reject) =>{

let csvData = [];

csv.parseFile(inputFile)

.on("data", function(data) {

csvData.push(data);

})

.on("end", function() {

resolve(csvData);

});

});

}

async function getFiles() {

console.log('reading files...');

let householdHistoricData = new Array();

for (i = 1; i <= 15; i++){

householdHistoricData.push(await readCSV(`./dataset/house${i}.csv`));

}

return { householdHistoricData };

}

module.exports = {

readCSV,

};

**plot.py**

import pandas as pd

import matplotlib

import matplotlib.pyplot as plt

matplotlib.style.use('ggplot')

data = pd.read\_csv('output.csv')

plt.legend()

data.set\_index('time')[['agg\_demand','agg\_supply']].plot()

plt.show()

**plot2.py**

import pandas as pd

import matplotlib

import matplotlib.pyplot as plt

matplotlib.style.use('ggplot')

data = pd.read\_csv('output.csv')

plt.legend()

data.set\_index('time')[['no\_trades\_market','no\_nat\_grid\_transactions']].plot()

plt.show()

**plot3.py**

import pandas as pd

import matplotlib

import matplotlib.pyplot as plt

matplotlib.style.use('ggplot')

data = pd.read\_csv('output.csv')

plt.legend()

data.set\_index('time')[['historical\_prices']].plot()

plt.show()