BUFFALO FARMERS MARKET

Vishal Raman
vraman2@buffalo.edu
50376944

Sai Vishwanath Venkatesh
saivishw@buffalo.edu
50419728

1.0 Problem Description

A Farmer's market is a physical marketplace representative of local culture and promotes products developed specifically for an area by mostly the small business owners of the area. Some examples to understand - a vendor at a Farmer's Market in Buffalo[1] makes homemade cinnamon rolls and pasties while another vendor grows organic vegetable produce and sells it locally. Farmer's markets are usually seasonal and vary in size (can be as small as a couple of booths or can even cover a whole block)

The concept of Farmers' Market has been in our society for a long time and has a niche market of people who are willing to support their local community. With every locality having several disconnected farmers' markets there is a need for a singular system that can is not governed by middlemen to ensure trust, transparency, and assurance of the support local customers have for their local farmer community.

2.0 Proposal

The aforementioned system will benefit greatly from a blockchain application since it would ensure trust, security, and transparency within a local community. Our solution ensures a customer's payments to the local vendor will directly reach them. This benefits the vendor by cutting out the middleman. Furthermore, the customer will also have complete transparency about whether the vendor is actually part of the local community and with regards to if the vendor is complying with health and safety regulations, thus providing the customer with a level of protection. Although we name our application based on our local community ("Buffalo" Farmers Market) it can and is meant to be adopted for various local communities since they all have varying sizes but have the same needs that this app can fulfill.

3.0 Design Overview

3.1 Quad Chart

Problem Statement:
A blockchain-enabled farmers market for a group of small business owners(vendors) within a locality.

Issues with existing solutions:

- Third parties take undisclosed cuts to host a vendor at a farmer's market. This hinders local vendor profit by large margins and leaves the local customers wondering how much their compensation actually went to the vendors?
- There is no assurance that all the vendors have fair competition since some vendors can come from outside a local community to pass off as small business owners when they are actually large businesses from outside the local community trying to exploit the platform.
- Customers in this platform have no idea if the vendor's product is safe and according to health regulations since traditionally farmer's markets are local physical gatherings with no transparency of the inner workings.
- There exists no means for loyal customers to support their favorite local community vendor besides buying their product - such as rating them to show support.

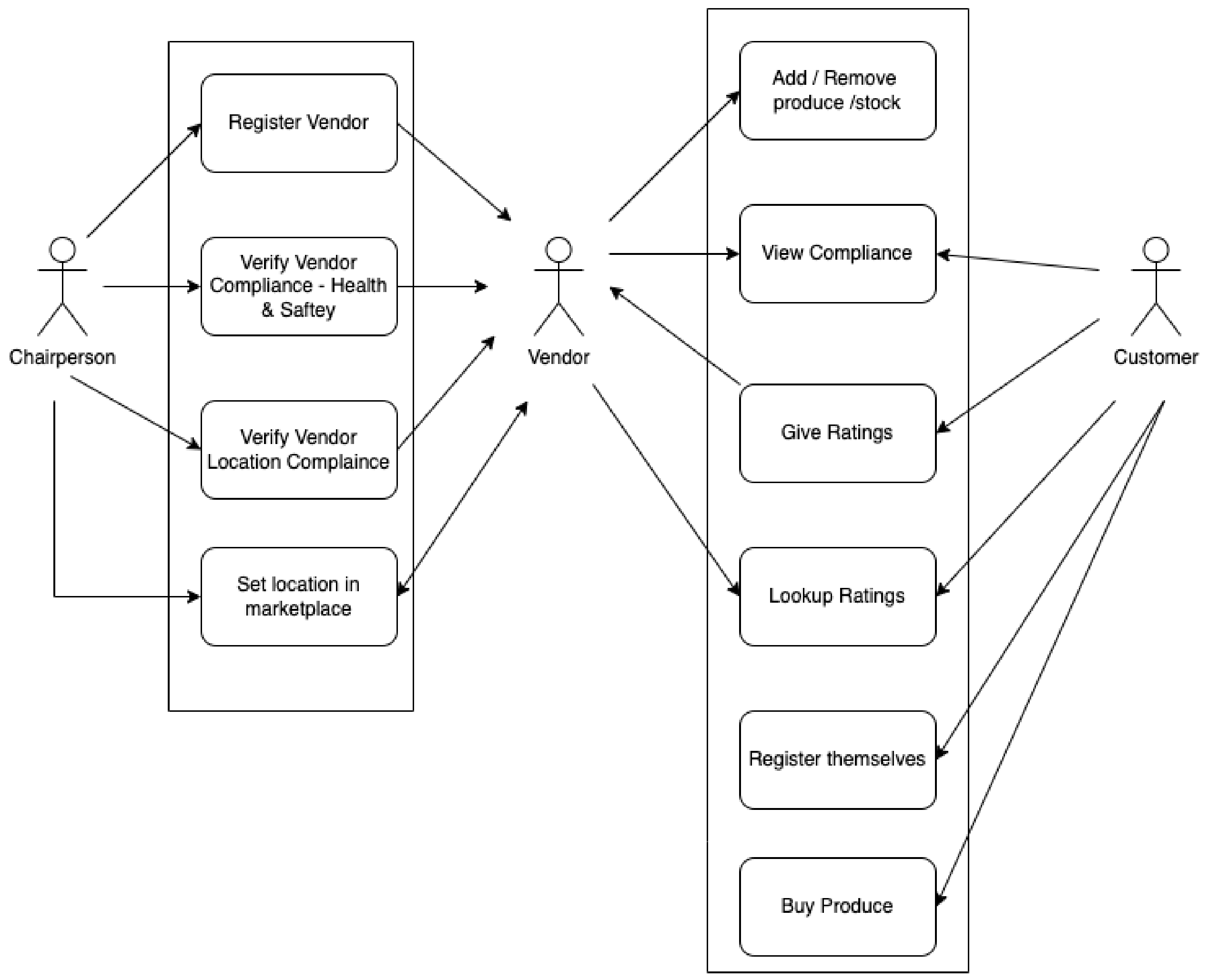
Proposed blockchain-based Solution:

- In a community of vendors any particular vendor can be chosen as chairperson democratically. This vendor(chair) can register other vendors based on location guidelines as well as health and safety guidelines.
- Customers on the BC can view these details before they buy anything - (registeredLocation compliance and health regulations)
- Customers pay the vendor directly via seamless payment on the blockchain
- Customers can rate their vendors to show support for their local community and discourage vendors that may supposedly misuse this system by ranking them worse.

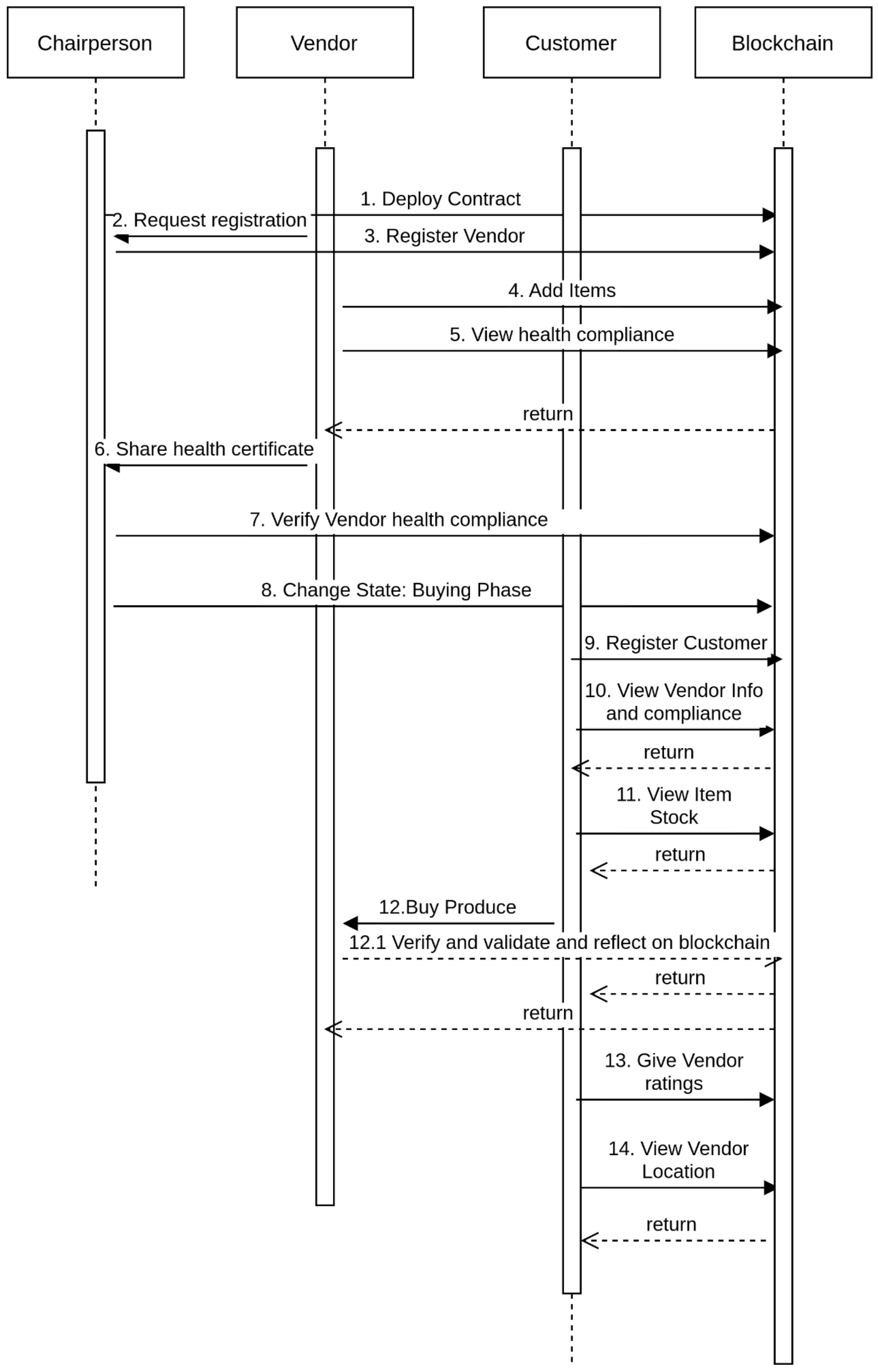
Benefits:

- Seamless direct transactions ensure that the customer's payment goes directly to the vendor. This eliminates profit losses due to middlemen and thus encourages vendors to reduce prices and inadvertently attract more customers to shift from commercial counterparts to local vendors.
- There is complete transparency with respect to location compliance and health regulation
    - This helps the local vendors community thwart unfair competition by large businesses that are not part of the local community from selling on this platform.
    - This helps the customer to know more about their product given they are from local vendors. This transparency in health and safety could encourage more customers to buy local rather than from big corporations.
- Rating encourages healthy development and support for local vendors and their products and thwarts bad/malicious commercial members that try to exploit the local market.

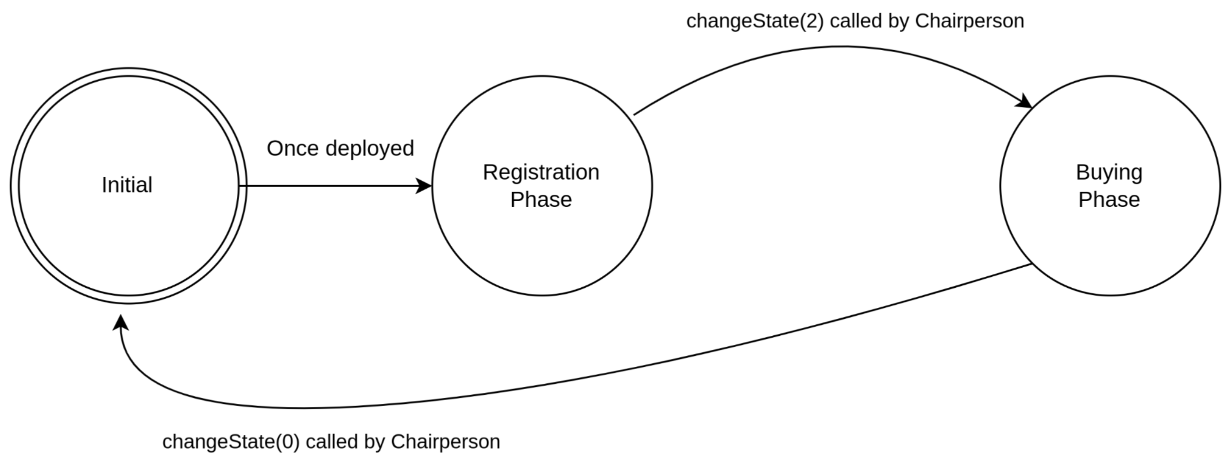
3.2 Use Case Diagram


Register Vendor
Verify Vendor Compliance - Health & Saftey
Verify Vendor Location Complaince
Set location in marketplace
Chairperson
Vendor
Add / Remove produce /stock
View Compliance
Give Ratings
Lookup Ratings
Register themselves
Buy Produce
Customer

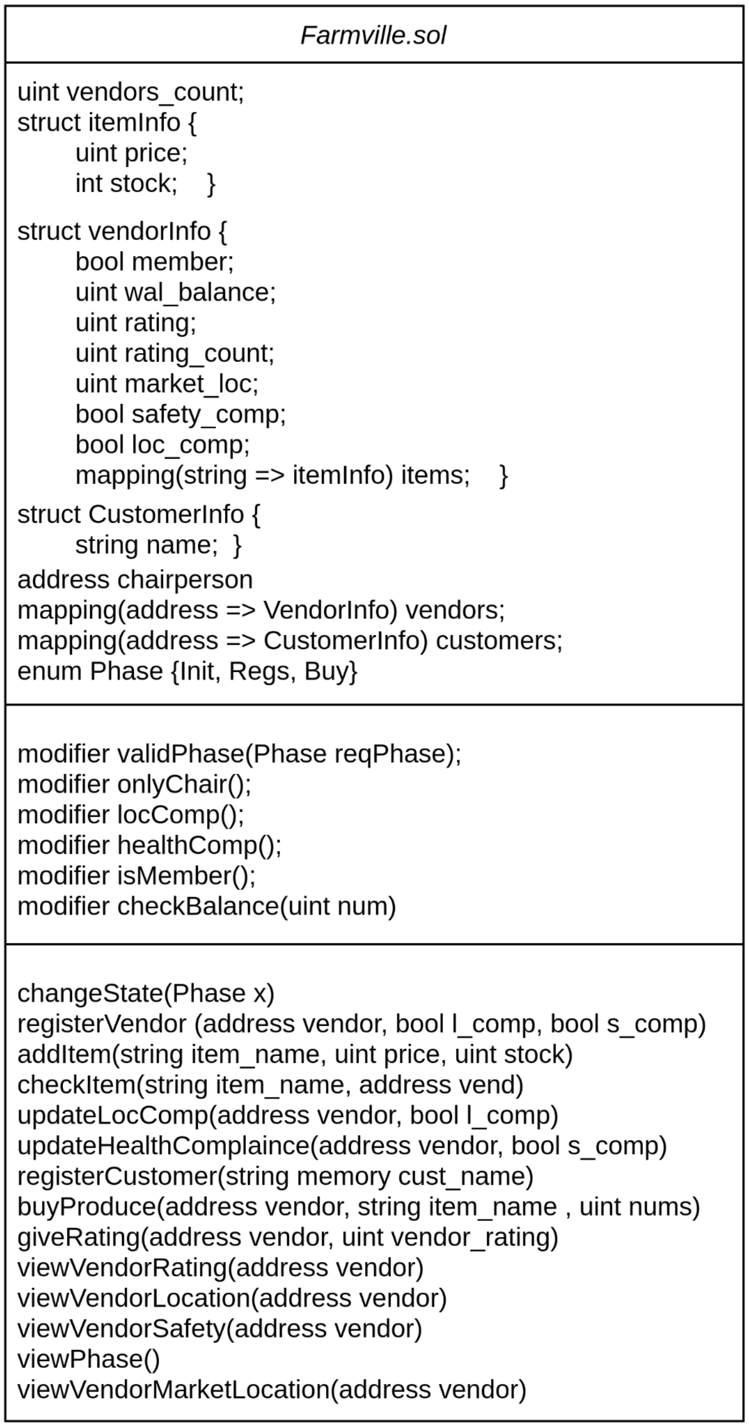
3.3 Sequence Diagram 1.0



3.4 State Machine Diagram


changeState(2) called by Chairperson
Initial
Once deployed
Registration Phase
Buying Phase
changeState(0) called by Chairperson

3.5 Contract Diagram 1.0



Farmville.sol

```
uint vendors_count;
struct itemInfo {
        uint price;
        int stock;    }
struct vendorInfo {
        bool member;
        uint wal_balance;
        uint rating;
        uint rating_count;
        uint market_loc;
        bool safety_comp;
        bool loc_comp;
        mapping(string => itemInfo) items;    }
struct CustomerInfo {
        string name;  }
address chairperson
mapping(address => VendorInfo) vendors;
mapping(address => CustomerInfo) customers;
enum Phase {Init, Regs, Buy}
```

```
modifier validPhase(Phase reqPhase);
modifier onlyChair();
modifier locComp();
modifier healthComp();
modifier isMember();
modifier checkBalance(uint num)
```

```
changeState(Phase x)
registerVendor (address vendor, bool l_comp, bool s_comp)
addItem(string item_name, uint price, uint stock)
checkItem(string item_name, address vend)
updateLocComp(address vendor, bool l_comp)
updateHealthComplaince(address vendor, bool s_comp)
registerCustomer(string memory cust_name)
buyProduce(address vendor, string item_name , uint nums)
giveRating(address vendor, uint vendor_rating)
viewVendorRating(address vendor)
viewVendorLocation(address vendor)
viewVendorSafety(address vendor)
viewPhase()
viewVendorMarketLocation(address vendor)
```

4.0 Smart Contract Functionality

4.1 Smart Contract code

```
//SPDX-License-Identifier: UNLICENSED

pragma solidity >=0.4.22 <=0.6.0;

contract Farmville {
    uint vendors_count = 0;

   struct itemInfo {
       uint price;
       uint stock;

   }
   struct VendorInfo {
       bool member;
       uint wal_balance;
       uint rating;
       uint rating_count;
       uint market_loc;
       bool safety_comp;
       bool loc_comp;
       mapping(string => itemInfo) items;
   }

   struct CustomerInfo {
       string name;
   }

   address chairperson;
   mapping(address => VendorInfo) vendors;
   mapping(address => CustomerInfo) customers;

   enum Phase {Init, Regs, Buy}
   Phase public state = Phase.Init;

   modifier validPhase(Phase reqPhase)
```

```
{ require(state == reqPhase);
  _;
}

modifier onlyChair(){
    require(msg.sender == chairperson);
    _;
}

modifier locComp(){
    require(vendors[msg.sender].loc_comp);
    _;
}

modifier healthComp(){
    require(vendors[msg.sender].safety_comp);
    _;
}

modifier isMember(){
    require(vendors[msg.sender].member);
    _;
}

modifier checkBalance(uint num){
    require(msg.value>=num, 'Not enough money given to buy');
    _;
}
constructor() public {
    chairperson = msg.sender;
    state = Phase.Regs;
}

function changeState(Phase x) onlyChair public {

    require (x > state );

    state = x;
 }
```

```
    function registerVendor (address vendor, bool l_comp, bool s_comp)
onlyChair validPhase(Phase.Regs) public payable {
        if(vendors[vendor].member){revert();}
        vendors_count+=1;
        vendors[vendor].member = true;
        vendors[vendor].wal_balance = 0;
        vendors[vendor].loc_comp = l_comp;
        vendors[vendor].safety_comp = s_comp;
        vendors[vendor].rating = 0;
        vendors[vendor].market_loc = vendors_count;

    }

    function addItem(string memory item_name, uint price, uint stock)
locComp healthComp isMember public{
        itemInfo memory temp_item;
        temp_item.price = price;
        temp_item.stock = stock;
        vendors[msg.sender].items[item_name] = temp_item;
    }

    function checkItem(string memory item_name, address vend) view public
returns(uint) {
        return vendors[vend].items[item_name].stock;
    }

    function updateLocComp(address vendor, bool l_comp) onlyChair public{
        vendors[vendor].loc_comp = l_comp;
    }

    function updateHealthComplaince(address vendor, bool s_comp) onlyChair
public{
        vendors[vendor].safety_comp = s_comp;
    }

    function registerCustomer(string memory cust_name) public payable{
        customers[msg.sender].name = cust_name;
    }
    function buyProduce(address payable vendor, string memory item_name ,
uint nums) validPhase(Phase.Buy)
checkBalance(vendors[vendor].items[item_name].price * nums) public
payable{
        if((vendors[vendor].safety_comp == false) ||
(nums>vendors[vendor].items[item_name].stock)) {revert();}
        uint amt;
        vendors[vendor].items[item_name].stock =
vendors[vendor].items[item_name].stock - nums;
        amt = vendors[vendor].items[item_name].price * nums;

        vendors[vendor].wal_balance+=amt;
        address payable vendor_address = vendor;
        vendor_address.transfer(amt * (10 ** 18));
    }

    function giveRating(address vendor, uint vendor_rating) public payable{
        vendors[vendor].rating = vendors[vendor].rating + vendor_rating;
        vendors[vendor].rating_count = vendors[vendor].rating_count + 1;
    }

    function viewVendorRating(address vendor) view public returns(uint) {
        if(vendors[vendor].rating_count == 0) {revert('Not enough
ratings');}
        return uint(vendors[vendor].rating/vendors[vendor].rating_count);
    }

    function viewVendorLocation(address vendor) view public returns(bool) {
        return vendors[vendor].loc_comp;
    }

    function viewVendorSafety(address vendor) view public returns(bool) {
        return vendors[vendor].safety_comp;
    }

    function viewPhase() view public returns(Phase) {
        return state;
    }
    function viewVendorMarketLocation(address vendor) view public
returns(uint) {
        return vendors[vendor].market_loc;
    }
}
```

4.2 Data overview

The blockchain stores the following data for transactions:

1. **vendorInfo**: Struct containing vendor-specific information necessary for BC transactions.
    a. **member**: bool whether or not the vendor is a part of the marketplace
    b. **wal_balance**: vendors crypto balance in the marketplace
    c. **rating**: average customer rating for vendor
    d. **rating_count**: number of reviews for vendor
    e. **market_loc**: booth/stall number assigned to vendor in physical market
    f. **saftey_comp**: bool specifying whether the vendor complies with health & safety regulations
    g. **loc_comp**: bool specifying whether the vendor complies with location regulations
    h. **items**: mapping representing each item sold by vendor and its info
2. **vendors**: mapping representing all vendors on the blockchain (with vendor address) and their vendorInfo
3. **customerInfo**: Struct containing customer-specific information to store on blockchain
    a. **name**: customer name
4. **chairperson**: stores address of the chairperson(the person who deployed the contract)

4.3 Modifiers overview

1. **validPhase (Phase reqPhase)**: checks if the current state is the same as the state passed in the argument -reqPhase
2. **onlyChair()**: makes sure the sender is the chairperson
3. **locComp()**: checks if the sender is location compliant
4. **healthComp()**: checks if the sender is compliant with health & safety regulations
5. **isMember()**: checks if the sender is a vendor of the marketplace

6. **checkBalance(uint num):** checks if the passed argument -nums is greater than the sender's account balance

4.4 Functions overview

The main functions of our app (using the blockchain) are

1. **registerVendor()** - This function is used to register vendors. This gets the vendor address, location and safety compliance as parameters. It has a modifier to check if only the chairperson is registering the vendor. Also, it has a modifier to check if the state currently is in the Registration state. It then assigns the compliancies to the vendor address's structure. It also makes the vendor as a member, and assigns his market location based on the vendor count.
2. **addItem()** - This function is used by the vendor to add produce. The parameters are item name, price and stock. The modifiers are there to check if the vendor is a member, and if he is location and safety compliant. Then the price, item, and stock are mapped to the item structure inside the vendor address's structure.
3. **buyProduce()** - This function is used by the Customer to buy produce from the Vendors. The parameters are Vendor address, item name, and number of items to buy. The modifiers are there to check if the Phase is in Buying Phase, and if the customer has given enough ether to buy the produce he wants. This function also takes in ether as a parameter option while calling from app.js. This function then decreases the stock of the corresponding item of the vendor, and calculates the amount to be transferred based on the rate and number of items, and then transfers the ethers to the vendor's account from the customer's account.

Furthermore, we use other functions to enhance the marketplace experience such as

- **changeState()** - To chair the state of the application to account for seasonality. Can be changed only by the chairperson.
- **viewVendorRating()** - To look at a particular vendor's rating.
- **viewVendorLocation()** - To check whether a particular vendor is location compliant.
- **viewVendorSafety()** - To check whether a particular vendor is safety compliant.
- **updateLocComp()** - To update the vendor's location compliance status. Can be changed only by the chairperson.

- **updateHealthComplaince()** - To update the vendor’s safety compliance status. Can be changed only by the chairperson.
- **checkItem()** - To check the stock of a particular item sold by a particular vendor.

5.0 Unique functionality / Novelties

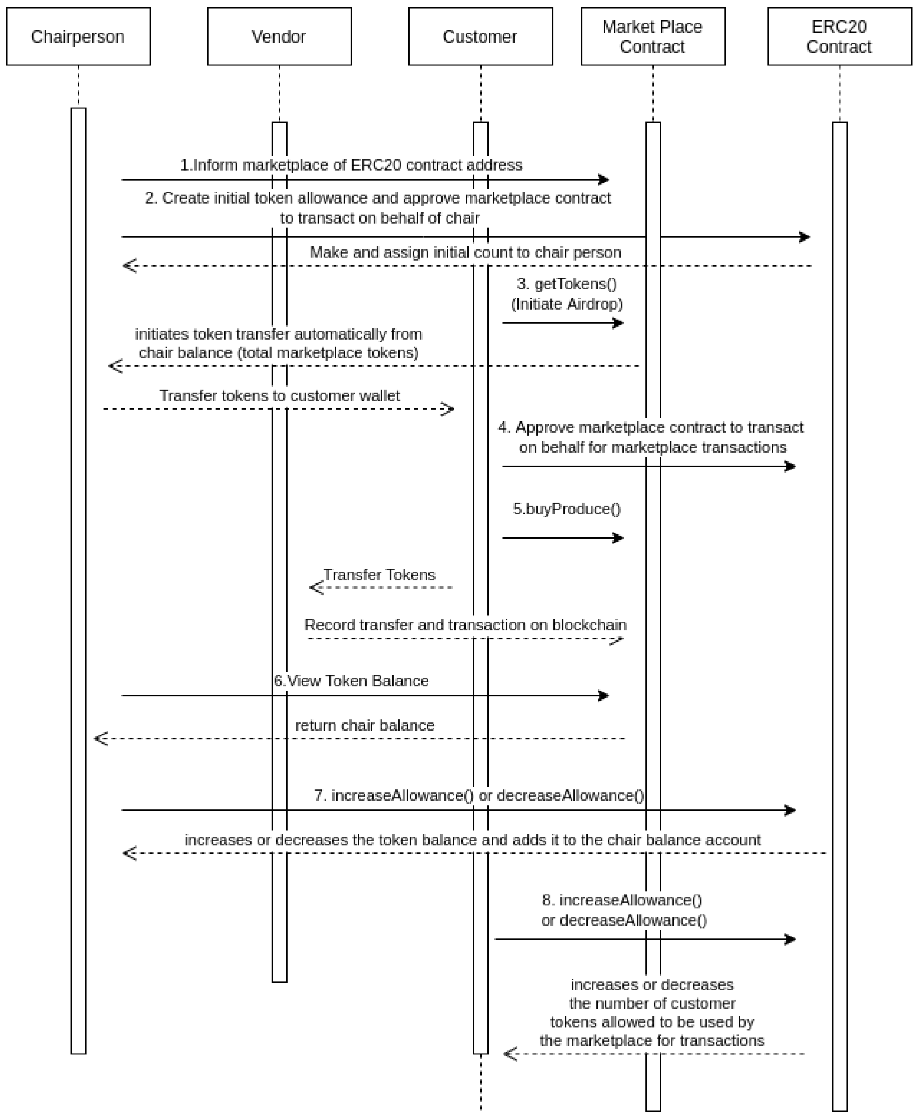
- Usage of states to account for the seasonality of the farmer's market. Keeps changing between registering vendors for the next season and buying season. This is accessible only by the chair via the onlyChair() modifier.
- We have incorporated a field for Location Compliance which can be granted by the chair. This indicates that the vendor is part of the local community. Most farmer’s markets have a similar regulation for registration, however, there is no means to ensure transparency of this to all actors. Putting this on a blockchain and making this viewable by all actors makes our implementation uniquely transparent.
- In a similar spirit of transparency to benefit and empower the customer's right to information about a product we incorporate a field indicating whether a vendor is complying with the health and safety regulations of the farmer’s market. This is also viewable by all actors and can be verified by the chair.
- We incorporate rating functionality for the customers. This encourages healthy development and customer support for local vendors and their products and thwarts bad/malicious commercial members that may try to exploit the local market.
- **Off-chain data** - Use of Dynamo DB for data storage outside the blockchain and viewing of vendors and items. The DynamoDB is hosted in AWS, and the vendor details and the item details of the vendors are stored as JSON in the DB. This Json is queried and displayed as the vendor information in the customer page. This helps us in not calling the smart contract to view the details, and helps in saving gas.

PHASE 3

6.0 ERC20 Interaction and functionality

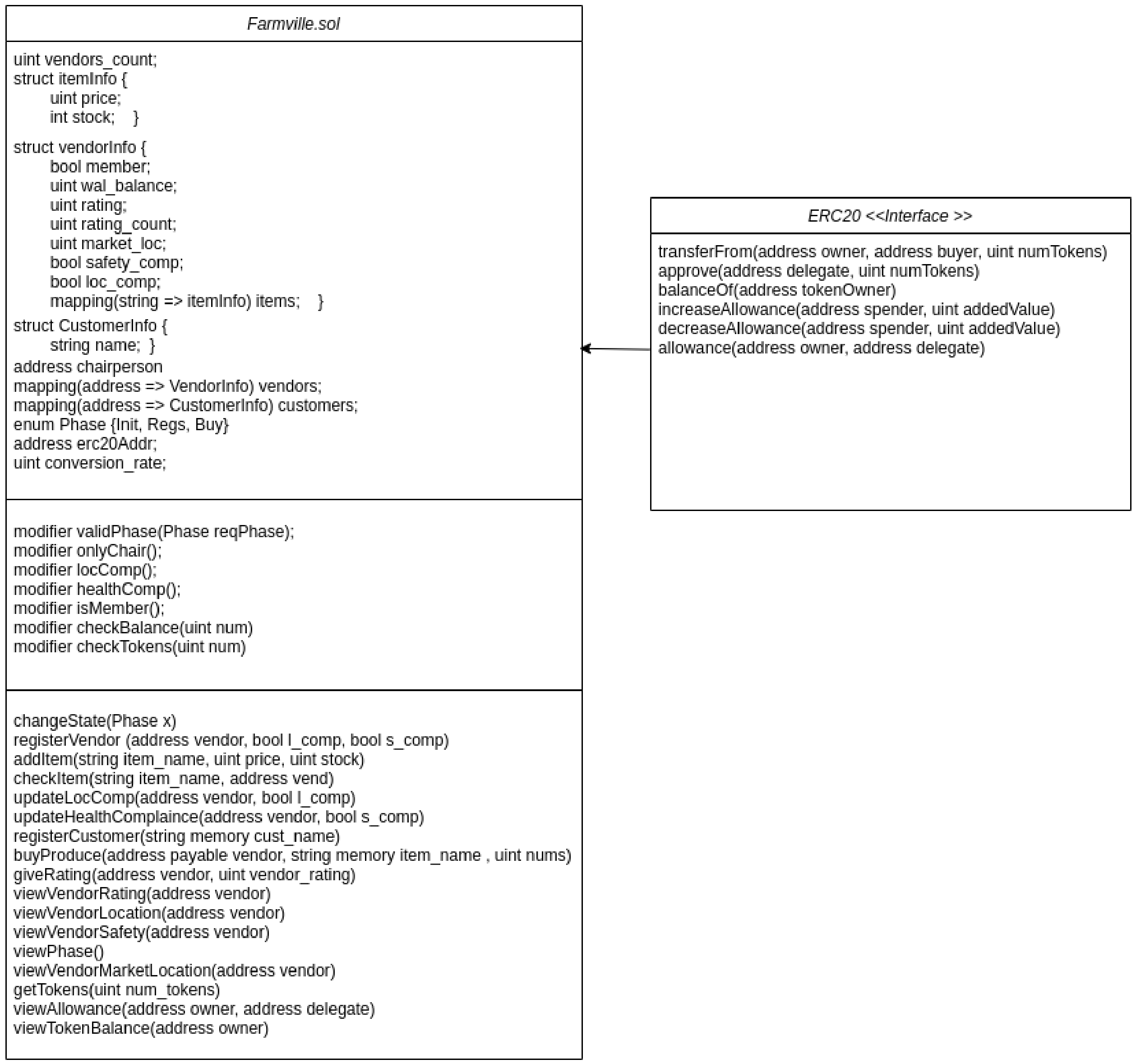
ERC20 functions and smart contract are used to make the initial airdrop of our own new token (BuFA) and allow the marketplace (transactions and actions) to function via BuFA instead of Ether. It facilitates the following functions illustrated in Section 6.1. Open Zeppelin ERC20 contract is used as an interface in our marketplace smart contract as illustrated in Section 6.2. The smart contract code used for this phase is specified in Section 6.5 and the deployment instructions are on Section 7.0.

6.1 ERC20-specific Sequence Diagram 2.0



Chairperson
Vendor
Customer
Market Place Contract
ERC20 Contract
1.Inform marketplace of ERC20 contract address
2. Create initial token allowance and approve marketplace contract to transact on behalf of chair
Make and assign initial count to chair person
3. getTokens() (Initiate Airdrop)
initiates token transfer automatically from chair balance (total marketplace tokens)
Transfer tokens to customer wallet
4. Approve marketplace contract to transact on behalf for marketplace transactions
5.buyProduce()
Transfer Tokens
Record transfer and transaction on blockchain
6.View Token Balance
return chair balance
7. increaseAllowance() or decreaseAllowance()
increases or decreases the token balance and adds it to the chair balance account
8. increaseAllowance() or decreaseAllowance()
increases or decreases the number of customer tokens allowed to be used by the marketplace for transactions

6.2 ERC20-specific Contract Diagram 2.0



```
Farmville.sol

uint vendors_count;
struct itemInfo {
        uint price;
        int stock;    }
struct vendorInfo {
        bool member;
        uint wal_balance;
        uint rating;
        uint rating_count;
        uint market_loc;
        bool safety_comp;
        bool loc_comp;
        mapping(string => itemInfo) items;    }
struct CustomerInfo {
        string name;  }
address chairperson
mapping(address => VendorInfo) vendors;
mapping(address => CustomerInfo) customers;
enum Phase {Init, Regs, Buy}
address erc20Addr;
uint conversion_rate;

modifier validPhase(Phase reqPhase);
modifier onlyChair();
modifier locComp();
modifier healthComp();
modifier isMember();
modifier checkBalance(uint num)
modifier checkTokens(uint num)

changeState(Phase x)
registerVendor (address vendor, bool l_comp, bool s_comp)
addItem(string item_name, uint price, uint stock)
checkItem(string item_name, address vend)
updateLocComp(address vendor, bool l_comp)
updateHealthComplaince(address vendor, bool s_comp)
registerCustomer(string memory cust_name)
buyProduce(address payable vendor, string memory item_name , uint nums)
giveRating(address vendor, uint vendor_rating)
viewVendorRating(address vendor)
viewVendorLocation(address vendor)
viewVendorSafety(address vendor)
viewPhase()
viewVendorMarketLocation(address vendor)
getTokens(uint num_tokens)
viewAllowance(address owner, address delegate)
viewTokenBalance(address owner)
```

```
ERC20 <<Interface >>

transferFrom(address owner, address buyer, uint numTokens)
approve(address delegate, uint numTokens)
balanceOf(address tokenOwner)
increaseAllowance(address spender, uint addedValue)
decreaseAllowance(address spender, uint addedValue)
allowance(address owner, address delegate)
```

6.3 ERC-20 functionality and methods:

1. **Transfer()**: This is the ERC20 contract function called by an actor (chairperson, customer, marketplace) to transfer BuFAs between to another actor. Here the token is transferred from msg.sender to the address mentioned in the parameter.
2. **Approve()**: This is the ERC20 contract function called by an actor (in this case - chairperson or customer) to approve the marketplace to transfer “numTokens” on their behalf.
3. **TransferFrom():** This is the ERC20 contract function called by an actor (chairperson) to transfer BuFA to another actor of the

marketplace. Here both the from and to address is specified in the parameter, and token is transferred between them.

4. **Allowance()**: This is the ERC20 contract function called by an actor to tell us how many tokens are approved for use by the given address.
5. **BalanceOf():** This returns the total token balance of the given actor(given via address)
6. **increaseAllowance():** This is the ERC20 contract function called by an actor (in this case - chairperson or customer) to increase the number of tokens approved for the marketplace to use for any transactions.
7. **decreaseAllowance():** This is the ERC20 contract function called by an actor (in this case - chairperson or customer) to decrease the number of tokens approved for the marketplace to use for any transactions.

<u>6.4 Methods in Marketplace for connecting with ERC20:</u>

1. **setAddress():** This is the marketplace contract function to set the ERC20 contract address to be an accessible variable for use in other functions.
2. **getTokens():** This is a marketplace contract function called by an actor (in this case - customer) in order to receive tokens to interact within the marketPlace (Done by calling **TransferFrom())**. The actor spends Ether to get tokens (1Ether = 20 Tokens), and the token is “airdropped” from the chairperson to the customer.
3. **buyProduce():** This is a marketplace contract function called by an actor (in this case - customer) in order to buy produce from a vendor using the marketplace token. This initiates a **TransferFrom()** transaction and token is “airdropped” from the customer to the vendor.
4. **viewAllowance(address owner, address delegate) :** This is a marketplace contract function to view the actors (in this case can be chairperson, or customer) number of tokens allowed to the marketplace for transaction purposes
5. **viewTokenBalance()**: This is a marketplace contract function to view the actor’s remaining token balance. (This is the remaining balance that can be dispersed to users entering the marketplace)

6.5 Smart Contract Code :

Farmville.sol :

```
//SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.6.0;
interface ERC20 {
  function transferFrom(address owner, address buyer, uint numTokens)
external returns (bool);
  function approve(address delegate, uint numTokens) external returns
(bool);
  function balanceOf(address tokenOwner) external view returns (uint);
  function increaseAllowance(address spender, uint addedValue) external
returns (bool);
  function decreaseAllowance(address spender, uint addedValue) external
returns (bool);
  function allowance(address owner, address delegate) external view
returns (uint);
}

contract Farmville {
    uint vendors_count = 0;

   struct itemInfo {
       uint price;
       uint stock;

   }
   struct VendorInfo {
       bool member;
       uint rating;
       uint rating_count;
       uint market_loc;
       bool safety_comp;
       bool loc_comp;
       mapping(string => itemInfo) items;
   }
```

```
struct CustomerInfo {
    string name;
}

address chairperson;
mapping(address => VendorInfo) vendors;
mapping(address => CustomerInfo) customers;

enum Phase {Init, Regs, Buy}
Phase public state = Phase.Init;

address erc20Addr;

uint conversion_rate = 20;

modifier validPhase(Phase reqPhase)
{ require(state == reqPhase);
  _;
}

modifier onlyChair(){
    require(msg.sender == chairperson);
    _;
}

modifier locComp(){
    require(vendors[msg.sender].loc_comp);
    _;
}

modifier healthComp(){
    require(vendors[msg.sender].safety_comp);
    _;
}

modifier isMember(){
    require(vendors[msg.sender].member);
    _;
}
```

```
    modifier checkBalance(uint num){
        require(msg.value*(10**18)>=(num/(conversion_rate)), 'Not enough
money given to buy');
        _;
    }

    modifier checkTokens(uint num){
        require(viewTokenBalance(msg.sender)>=num*(10**18), 'Not enough
tokens given to buy');
        _;
    }

    constructor() public {
        chairperson = msg.sender;
        state = Phase.Regs;

    }

    function setAddress(address erc20) public {
        erc20Addr = erc20;
    }

    function changeState(Phase x) onlyChair public {

        require (x > state );

        state = x;
     }

    function registerVendor (address vendor, bool l_comp, bool s_comp)
onlyChair validPhase(Phase.Regs) public {
        if(vendors[vendor].member){revert();}
        vendors_count+=1;
        vendors[vendor].member = true;
        vendors[vendor].loc_comp = l_comp;
        vendors[vendor].safety_comp = s_comp;
        vendors[vendor].rating = 0;
        vendors[vendor].market_loc = vendors_count;

    }
    function addItem(string memory item_name, uint price, uint stock)
locComp healthComp isMember public{
         itemInfo memory temp_item;
         temp_item.price = price;
         temp_item.stock = stock;
         vendors[msg.sender].items[item_name] = temp_item;
    }

    function checkItem(string memory item_name, address vend) view public
returns(uint) {
         return vendors[vend].items[item_name].stock;
    }

    function updateLocComp(address vendor, bool l_comp) onlyChair public{
         vendors[vendor].loc_comp = l_comp;
    }

    function updateHealthComplaince(address vendor, bool s_comp) onlyChair
public{
         vendors[vendor].safety_comp = s_comp;
    }

    function registerCustomer(string memory cust_name) public{
         customers[msg.sender].name = cust_name;
    }

    function getTokens(uint num_tokens) checkBalance(num_tokens) public
payable {
         ERC20(erc20Addr).transferFrom(chairperson, msg.sender, num_tokens);
    }

    function buyProduce(address payable vendor, string memory item_name ,
uint nums) validPhase(Phase.Buy)
checkTokens(vendors[vendor].items[item_name].price * nums) public payable{
         if((vendors[vendor].safety_comp == false) ||
(nums>vendors[vendor].items[item_name].stock)) {revert();}
         uint amt;
         vendors[vendor].items[item_name].stock =
vendors[vendor].items[item_name].stock - nums;
```

```
        amt = vendors[vendor].items[item_name].price * nums;

        ERC20(erc20Addr).transferFrom(msg.sender, vendor, amt * (10 **
18));
    }

    function viewAllowance(address owner, address delegate) public view
returns (uint) {
        return ERC20(erc20Addr).allowance(owner, delegate);
    }

    function viewTokenBalance(address owner) public view returns (uint) {
        return ERC20(erc20Addr).balanceOf(owner);
    }

    function giveRating(address vendor, uint vendor_rating) public{
        vendors[vendor].rating = vendors[vendor].rating + vendor_rating;
        vendors[vendor].rating_count = vendors[vendor].rating_count + 1;
    }

    function viewVendorRating(address vendor) view public returns(uint) {
        if(vendors[vendor].rating_count == 0) {revert('Not enough
ratings');}
        return uint(vendors[vendor].rating/vendors[vendor].rating_count);
    }

    function viewVendorLocation(address vendor) view public returns(bool) {
        return vendors[vendor].loc_comp;
    }

    function viewVendorSafety(address vendor) view public returns(bool) {
        return vendors[vendor].safety_comp;
    }

    function viewPhase() view public returns(Phase) {
        return state;
    }

    function viewVendorMarketLocation(address vendor) view public
returns(uint) {
        return vendors[vendor].market_loc;
    }
}
```

Zep_ERC20.sol:

```
pragma solidity ^0.8.13;
//SPDX-License-Identifier: UNLICENSED

import "@openzeppelin/contracts/token/ERC20/ERC20.sol" ;

contract FarmToken is ERC20{
    constructor() ERC20("FarmToken", "BuFA"){
        _mint(msg.sender,100000*10**18);
    }
}
```

7.0 Deployment, Test and Interact

1. Go inside the farmville-app folder, and open a terminal there and give **Npm install.** This will install all the dependencies.
2. Now open the **Farmville.Sol** code in Remix, and connect to injected Web3 and deploy it.
3. Copy the Abi and the address of the smart contract and paste it in app.js.
4. Now open the **Zep_ERC20.Sol** code in Remix, and connect to injected Web3 and deploy it. Copy the Abi and the address of the smart contract and paste it in app.js.
5. Now come back to the terminal, and give npm start. This will start the app.
6. Go to http://localhost:3010/ and now you can start interacting with the app as a chairperson.
7. Go to the chairperson page, and click on the “Set ERC20 Address” to set the ERC20 Address to the marketplace and disperse the initial amount of tokens to the chairperson.
8. Enter the amount of BuFA within the bounds of the total marketplace balance that you want to approve for the usage inside the marketplace by selecting the Approve option. You can also increase or decrease this allowance using this form.
9. Give vendor details and register a vendor.

10. To do this, give the address of the vendor, and true, true as the compliance details.
11. Next switch to the vendor account in Metamask, and go to the vendor page.
12. Now give the item name, price, and stock, add the item to the smart contract and the DynamoDB (Hosted in AWS).
13. Now again go to the chairperson account, and set the Phase to Buying Season
14. Now switch account to Customer, and go to the Customer page.
15. Scroll down to the end of the page to exchange Exchange Ether for BuFA coins. Specify an appropriate amount to purchase the BuFA coins. Complete the transaction on the MetaMask pop-up
16. Specify how many of the bought BuFA you would like to allow the marketplace to use for transactions you are interested in. Click the Approve option. Likewise you can increment or decrement the amount of BuFA you have allowed with the options.
17. Check the Vendor Information, and give the vendor address, num of items to buy and click the "Buy" button to buy items from the vendor. Complete the same on metamask.
18. Now BuFA will get transferred from the customer to the vendor.
19. Give ratings to the vendor.

REFERENCES

1. http://www.buffalofarmersmarket.com/
2. Counter Dapp by Professor Bina
3. https://gourabp17.github.io/codespace/#/aws/su4dz-qgr1y
4. Open Zeppelin ERC20 contract from @openzeppelin/contracts/token/ERC20/ERC20.sol
5. https://medium.com/coinmonks/handling-events-of-a-smart-contract-part-1-2-b086eb6696cf
6. https://ethereum.stackexchange.com/questions/46457/send-tokens-using-approve-and-transferfrom-vs-only-transfer
7. https://www.youtube.com/watch?v=tbjyc-VQaQo