pragma solidity ^0.8.20;

contract SmartHomeFactory{

    address[] public deployedHouseholds;

    address public owner;

    event SmartHomeCreated(address smarthomeAddress);

    constructor() {

        owner = address(this);

    }

    function createHousehold(uint capacity) public {

        SmartHome h = new SmartHome(capacity, msg.sender, owner);

        address newHousehold = address(h);

        deployedHouseholds.push(newHousehold);

        emit SmartHomeCreated(newHousehold);

    }

    function getDeployedHouseholds() public view returns (address[] memory) {

        return deployedHouseholds;

    }

}

contract SmartHome{

    uint public currentDemand;

    uint public currentSupply;

    uint public batteryCapacity;

    uint public amountOfCharge;

    uint public excessEnergy;

    struct Bid{

        address origin;

        uint price;

        uint amount;

        uint date;

    }

    struct BidSuccessful{

        address recipient;

        uint price;

        uint amount;

        uint date;

    }

    Bid[] public Bids;

    Bid[] public Asks;

    BidSuccessful[] public SuccessfulBids;

    address public owner;

    address public contractAddress;

    address public parent;

    address public exchangeAddress;

    uint public balanceContract;

    MicrogridMarket ex;

    SmartHome hh;

    constructor(uint capacity, address creator, address watch\_address) payable{

        owner = creator;

        batteryCapacity = capacity;

        amountOfCharge = capacity;

        parent = watch\_address;

        contractAddress = address(this);

    }

    function deposit() public payable {

    }

    receive() external payable {

    }

    fallback() external payable {}

    function setSmartMeterDetails(uint \_demand, uint \_supply, uint \_excessEnergy) public {

        currentDemand = \_demand;

        currentSupply = \_supply;

        excessEnergy = \_excessEnergy;

    }

    function getSmartMeterDetails() public view returns(address, uint, uint, uint, uint, uint){

        return(

            owner,

            currentDemand,

            currentSupply,

            batteryCapacity,

            amountOfCharge,

            excessEnergy

        );

    }

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

        return (Bids[index].origin,

                Bids[index].price,

                Bids[index].amount,

                Bids[index].date

        );

    }

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

        return (Asks[index].origin,

                Asks[index].price,

                Asks[index].amount,

                Asks[index].date

        );

    }

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

        return (SuccessfulBids[index].recipient,

                SuccessfulBids[index].price,

                SuccessfulBids[index].amount,

                SuccessfulBids[index].date

        );

    }

    function getSuccessfulBidCount() public view returns(uint) {

        return SuccessfulBids.length;

    }

    function setExchange(address exchange) public {

        exchangeAddress = exchange;

    }

    function charge(uint amount) public restricted{

        if(amountOfCharge + amount >= batteryCapacity) {

            amountOfCharge = batteryCapacity;

        }

        else{

            amountOfCharge += amount;

        }

    }

    function discharge(uint amount) public {

        if(amountOfCharge - amount <= 0) {

            amountOfCharge = 0;

        }

        else{

            amountOfCharge -= amount;

        }

    }

    function submitBid(uint price, uint amount, uint timestamp) public restricted returns (bool){

        Bid memory newBid = Bid({

            origin: contractAddress,

            price: price,

            amount: amount,

            date: timestamp

        });

        Bids.push(newBid);

        ex = MicrogridMarket(payable(exchangeAddress));

        return ex.placeBid(price, amount, timestamp);

    }

    function submitAsk(uint price, uint amount, uint timestamp) public restricted returns(bool) {

        Bid memory newAsk = Bid({

            origin: contractAddress,

            price: price,

            amount: amount,

            date: timestamp

        });

        Asks.push(newAsk);

        ex = MicrogridMarket(payable(exchangeAddress));

        return ex.placeAsk(price, amount, timestamp);

    }

    function buyEnergy(uint \_amount, address payable \_recipient, uint \_price, uint \_date ) public payable returns(bool successful){

        BidSuccessful memory newBid = BidSuccessful({

            recipient: \_recipient,

            price: \_price,

            amount: \_amount,

            date: \_date

        });

        amountOfCharge += \_amount;

        hh = SmartHome(\_recipient);

        hh.discharge(\_amount);

        \_recipient.transfer( (\_amount/1000)\*\_price);

        SuccessfulBids.push(newBid);

        return true;

    }

    function deleteBid(uint bid\_index) public {

        ex = MicrogridMarket(payable(exchangeAddress));

        ex.removeBid(bid\_index);

    }

    function deleteAsk(uint ask\_index) public {

        ex = MicrogridMarket(payable(exchangeAddress));

        ex.removeAsk(ask\_index);

    }

    function getBidsCount() public view returns(uint) {

        return Bids.length;

    }

    function getAsksCount() public view returns(uint) {

        return Asks.length;

    }

    modifier restricted() {

        require(msg.sender == owner);

        \_;

    }

}

contract MicrogridMarket {

    struct Bid {

        address payable owner;

        uint price;

        uint amount;

        uint date;

    }

    struct Ask {

        address payable owner;

        uint price;

        uint amount;

        uint date;

    }

    Bid[] public Bids;

    Ask[] public Asks;

    SmartHome hh;

    address public owner;

    constructor(address \_owner) payable{

        owner = \_owner;

    }

    function deposit() public payable {

    }

    receive() external payable {}

    fallback() external payable {}

    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 = payable(msg.sender);

        b.price = \_price;

        b.amount = \_amount;

        b.date = timestamp;

        for(uint i = 0; i < Bids.length; i++) {

            if(Bids[i].price > \_price) {

                Bid[] memory tempBids = new Bid[](Bids.length - i);

                for(uint j = i; j < Bids.length; j++) {

                    tempBids[j-i] = Bids[j];

                }

                Bids[i] = b;

                Bids.push();

                for(uint k = 0; k < tempBids.length; k++) {

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

                }

                if(Asks.length>0){

                    matchBid(Bids.length-1 ,Asks.length-1 );

                }

                return true;

            }

        }

        Bids.push(b);

        if(Asks.length>0){

            matchBid(Bids.length-1 ,Asks.length-1 );

        }

        return true;

    }

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

        Ask memory a;

        a.owner = payable(msg.sender);

        a.price = \_price;

        a.amount = \_amount;

        a.date = timestamp;

        for (uint i = 0; i < Asks.length; i ++) {

            if(Asks[i].price < \_price) {

                Ask[] memory tempAsks = new Ask[](Asks.length - i);

                for (uint j = i; j < Asks.length; j++) {

                    tempAsks[j-i] = Asks[j];

                }

                Asks[i] = a;

                Asks.push();

                for (uint k = 0; k < tempAsks.length; k++) {

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

                }

                if (Bids.length>0){

                    matchBid(Bids.length-1,Asks.length-1 );

                }

                return true;

            }

        }

        Asks.push(a);

        if(Bids.length > 0) {

            matchBid(Bids.length-1,Asks.length-1 );

        }

        return true;

    }

    function matchBid(uint bid\_index, uint ask\_index) public returns (bool) {

        if (Bids.length == 0 || Asks.length == 0 || Bids[bid\_index].price < Asks[ask\_index].price) {

            return true;

        }

        hh = SmartHome(Bids[bid\_index].owner);

        //uint price = (Asks[ask\_index].price + Bids[bid\_index].price) / 2;

        uint price = Bids[bid\_index].price;

        if(int(Bids[bid\_index].amount - Asks[ask\_index].amount) >= 0){

            uint remainder = Bids[bid\_index].amount - Asks[ask\_index].amount;

            uint calcAmount = Bids[bid\_index].amount - remainder;

            hh.buyEnergy(calcAmount, Asks[ask\_index].owner, price, Bids[bid\_index].date);

            Bids[bid\_index].amount = remainder;

            if(remainder==0){

                removeBid(bid\_index);

            }

            removeAsk(ask\_index);

            return (matchBid(Bids.length-1,Asks.length-1));

        }

        if(int(Bids[bid\_index].amount - Asks[ask\_index].amount) < 0){

            uint remainder = Asks[ask\_index].amount - Bids[bid\_index].amount;

            uint calcAmount = Asks[ask\_index].amount - remainder;

            hh.buyEnergy(calcAmount, Asks[ask\_index].owner, price, Bids[bid\_index].date);

            Asks[ask\_index].amount = remainder;

            if(remainder == 0){

                removeAsk(ask\_index);

            }

            removeBid(bid\_index);

            return (matchBid(Bids.length-1,Asks.length-1));

        }

        return false;

    }

    function removeBid(uint index) public {

        if (index >= Bids.length) return;

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

            Bids[i] = Bids[i+1];

        }

        Bids.pop();

    }

    function removeAsk(uint index) public {

        if (index >= Asks.length) return;

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

            Asks[i] = Asks[i+1];

        }

        Asks.pop();

    }

    function getBidsCount() public view returns(uint) {

        return Bids.length;

    }

    function getAsksCount() public view returns(uint) {

        return Asks.length;

    }

}

give me the proposed method chapter based on this contract which is divided into 4 parts:

1. initialization phase
2. registration phase
3. power order phase
4. trust rating phase (which is not included in code, but is proposed for future works)

**Proposed Method: Enhancing Trust in Smart Grids using Blockchain**

This thesis proposes a blockchain-based system to enhance trust and security within smart grids, focusing on energy trading between households equipped with smart meters and energy storage capabilities. The system utilizes smart contracts to automate energy trading and ensure transparency, enabling a decentralized and trustless environment for peer-to-peer energy exchange. The proposed method can be divided into four key phases:

**1. Initialization Phase:**

* **Smart Contract Deployment:** The SmartHomeFactory contract is deployed on the blockchain. This factory contract serves as a central point for creating and managing individual SmartHome contracts, each representing a participating household within the smart grid.
* **Microgrid Market Establishment:** A MicrogridMarket contract is deployed, acting as a decentralized marketplace where households can submit energy bids and asks. This contract facilitates the matching of energy buyers and sellers, ensuring a transparent and efficient trading process.

**2. Registration Phase:**

* **Household Onboarding:** Each participating household, through its designated owner, interacts with the SmartHomeFactory contract to create a SmartHome contract. This process registers the household within the system and establishes its initial energy capacity and ownership details.
* **Smart Meter Integration:** The smart meter at each household is linked to its respective SmartHome contract. The smart meter continuously monitors energy consumption and generation data, which is then used to update the SmartHome contract's state, reflecting the household's current energy demand, supply, and excess energy available for trading.

**3. Power Order Phase:**

* **Energy Trading:** Households with excess energy can submit sell orders ("asks") to the MicrogridMarket contract, specifying the price and amount of energy they are willing to sell. Conversely, households requiring additional energy can submit buy orders ("bids") with their desired price and amount.
* **Order Matching and Execution:** The MicrogridMarket contract automatically matches bids and asks based on price and time priority. When a match is found, the contract facilitates the energy transfer between the buyer and seller, updating the respective SmartHome contract states and recording the transaction on the blockchain.
* **Secure Payment:** The system utilizes the blockchain's native cryptocurrency for secure and transparent payments. Upon successful energy transfer, the buyer's SmartHome contract automatically transfers the agreed-upon amount to the seller's SmartHome contract, ensuring immediate and tamper-proof settlement.

**4. Trust Rating Phase (Future Work):**

While not implemented in the current system, a trust rating mechanism is proposed as a future enhancement. This mechanism would assign trust scores to participating households based on their past behavior in the energy market, such as:

* **Successful transactions:** Completing energy trades without issues would contribute positively to a household's trust score.
* **Failed transactions:** Cancelling orders or failing to deliver on agreed-upon energy transfers would negatively impact the trust score.
* **Responsiveness:** The timeliness of responses to bids and asks could also factor into the trust rating.

This trust rating system could offer several benefits:

* **Enhanced Security:** Households with higher trust scores could be prioritized in the order matching process, reducing the risk of interacting with unreliable or malicious actors.
* **Reputation-based Incentives:** Trust scores could be used to incentivize good behavior and discourage fraudulent activities within the energy market.
* **Dynamic Pricing:** Trust scores could influence the pricing of energy bids and asks, with higher trust scores potentially commanding premium prices or receiving preferential treatment.

By incorporating this trust rating mechanism, the proposed system can further enhance trust and security within the smart grid, fostering a more reliable and efficient energy trading environment for all participants.