Cmpe483 BULOT Lottery Smart Contract HW1

Documentation

Serkan Özel

Bahadır Hocamoğlu

Contents:

1. Installation and test running instructions
2. Test description
3. Short project description
4. Function and implementation description.

Installation and Test running Instructions:

1. Install geth and parity. Note tests are intended to be done and tested on ubuntu.
2. Create a directory called PARITYDATA
3. In terminal go to folder that you extracted project files into.
4. Use the command in the parity\_command.txt file. Change –base-dir path to the full path of PARITYDATA folder.
5. Unlock the account 0 in order to be able to deploy contracts.
6. Deploy EIP20 contract.
7. After deploying eip20, we recommend you to copy its address to test1.js file and, if you want to use the watcher, to watcher.js to reduce your work after deploying bulot.
8. Deploy bulottest.sol as its stage duration is 60.
9. It is recommended to prepare loadScript("test1.js") in geth console so that you can quickly run the simulation after deploying the contract.
10. Copy their addresses to variables in “watcher.js” and “test1.js”.
11. Use the command geth attach to get into geth console.
12. First run the event watcher by running loadScript(“watcher.js”)
13. Then to run tests, open another geth console and run loadScript(“test1.js”).
14. You can see events happening from first geth console or test1.js will tell you the events.

Note: Remember you have 60 seconds until all the accounts can buy tickets, and because the loop is run sequentially (using forEach), it may take a little time to finish. We tested on up to 100 accounts 60 seconds is enough for the ticket buying stage whether accounts needs to be created or not. Creation of the accounts takes extra time.

Test and files’ description:

There is a file called “watcher.js” which has event listeners for events that are defined in the code for ticket buying, revealing and withdrawing. If you want you can use watcher but test gives same informations with console.log().

The watcher watches these events and informs by printing to the console. The “unlock.js” file unlocks first account if run with loadScript(“unlock.js”).

The number of accounts in the simulation is configurable by the variable called ACCOUNTNUMBER in the code. The test file created ACCOUNTNUMBER accounts if there are less than ACCOUNTNUMBER accounts and transfers 10 ether from main account to each of them(if needed for gas). Then, immediately all the accounts start buying ticket from the contract one by one. Each 30 seconds the test controls if the submission ended. If so, all the accounts reveal their number. Also, each 30 seconds the test controls if the lottery 1 ended and it is submission stage, meaning, winners are announced. Then, all the accounts check if they win any prize. If so, they withdraw the prize.

There are several intervals in the test file via setInterval(functions happen continuously with specific intervals):

**Block Creation Interval**: In simulation if no one sends any transaction no blocks are created, and the “now” variable do not change. To overcome this, we wrote this interval. This interval causes accounts 0 and 1 to send ethers to each other continuously.

**Reveal Trial Interval:** This interval is for trial of revealing every 30 seconds. Reveal is for the first lottery.

**Withdraw Trial Interval:**  This interval is for trial of withdrawing wined prizes if any. In this interval every user checks if he or she is the winner in any position, and if so withdraws his or her money.

Short Project Description:

This program written in solidity is a lottery smart contract. Users can buy a ticket with 10 EIP20 tokens in order to participate. Each token is equal to 1tl. Tokens-TL exchange are done using EIP20 Contact. There are two periods of the lottery: submission and reveal. In submission period users buy ticket, and in the reveal stage they need to reveal their random numbers with which they buy their ticket. According to the reveals, a random number is generated. By hashing this number multiple times, a few revealers which is proportional to logarithm of revealers win the lottery. Each winner takes half of the token prize of the previous winner. The first winner takes approximately half of the total tokens that is collected via ticket prices. The lottery lasts forever. More description is in the project description pdf.

Description of bulot.sol:

* This contract needs to be compiled with a compiler that’s version is greater than 0.4.22 but should not be 0.5 or newer.
* This contract interacts with eip20 contract, so we have an import statement of EIP20.sol

import "./EIP20.sol";

* Data Structures:

  struct Ticket {

         bytes32 hash;

         uint purchasedAt;

     }

* + Ticket: This data structure holds ticket data.
    - Bytes32 hash: Random hash that user sends
    - Uint purchasedAt: Lottery no that this ticket is purchased. If the ticket is revealed this number is deleted using “delete” so that all the other submission trials will not succeed.

struct Game {

        uint256 totalPrize;

        address[] revealedPlayers;

        mapping(uint=>bool) withdrawnPrizes;

        uint randomNumber;

        mapping(address=>Ticket) players;

    }

* + Game:This data structure holds game data.
    - Uint256 totalPrize: TotalPrize that will be given. Also equals total gain from ticket prices.
    - Address[] revealedPlayers: Array of players’ addresses that have revealed their random number.
    - Mapping(uint=>bool) withdrawnPrizes: Holds if winner I has withdrawn his/her money.
    - Uint randomNumber: The accumulating random number that is generated by xor ing revealed random numbers by users.
    - Mapping(address=> Ticket) players: Players that has bought ticket is added here to keep track of the ticket they bought.
* Global Variables

uint constant STAGEDURATION = 2 weeks

* + Uint constant STAGEDURATION: Equals to 2 weeks that is submission or reveal stage duration. Can be changed to debug or test the application. When testing it is recommended to make this something like 400 as ticket buying process takes time.

mapping(uint=>Game) public games;

* + Mapping(uint=>Game) public games: Holds games. Each submission and reveal stage form a game.

uint start;

* + Uint start: This is the creation timestamp of the contract. Used to determine stages and game number.

EIP20 network;

* + EIP20 network: This is the eip20 network instance. Its address should be given to constructor when deploying the contract.
* Events:

event PurchaseTicket(address sender, bytes32 randomHashed);

    event RevealNumber(address sender, uint randomNum);

   event PrizeWithdrawn(address \_player,uint \_lotteryNo,uint \_index);

* + PurchaseTicket(address sender, bytes32 randomHashed): The event that is fired when a ticket is bought
    - Sender: Address of the buyer.
    - randomHashed: sha256 hash of a random number and senders’ address. Submission is done like that, see description for further info.
  + RevealNumber(address sender, uint randomNum): The event that is fired when a reveal occurs.
    - Sender: Revealer’s address
    - randomNum: The random number that the revealer initially selected during submission.
  + PrizeWithdrawn(address \_player,uint \_lotteryNo, uint \_index): The event that is fired when somebody withdraws his or her prize from the contract. The prize is transferred to winners eip20 account.

function () public {

        revert();

    }

* Revert Function: Prevents random payments in fallback function

constructor(address \_network) public {

        network = EIP20(\_network);

        start = now;

    }

* Constructor: Called when the contract is created. Sets start to be now as start holds creation timestamp.
  + \_network: address of the EIP20 network. EIP20 network should be deployed before bulot contract.

function getCurrentLotteryNo() public view returns (uint) {

        return (now - start) / (2 \* STAGEDURATION) + 1;

     }

     function isSubmissionStage() public view returns (bool) {

        return (now - start) % (2 \* STAGEDURATION) < STAGEDURATION;

     }

* Stage functions:
  + getCurrentLotteryNo: This function returns lottery no(uint) by measuring how much STAGEDURATION passed starting from “start“ timestamp. If stageduration is 2 weeks
    - 1 week passed -> returns 1
    - 3 weeks passed -> returns 2
    - 80 weeks and 1 day passed -> return s41
  + isSubmissionStage: Returns bool representing if we are currently in submission stage or in reveal stage.

modifier submission() {

        require(isSubmissionStage(), 'Not in submission stage');

       \_;

    }

     modifier reveal() {

        require(!isSubmissionStage(), 'Not in reveal stage');

        \_;

     }

* Modifiers: These modifiers guard reveal and submission related functions and do not let them execute in the wrong stage.

function purchaseTicket(bytes32 randomHashed) public submission returns(bool success) {

        uint lotteryNo = getCurrentLotteryNo();

require(

            games[lotteryNo].players[msg.sender].hash == 0,

            'Already bought a ticket in the current lottery');

require(network.transferFrom(msg.sender, address(this), 10),

            'The payment failed. Please allow the transfer of 10 TL to the address of that contract');

games[lotteryNo].players[msg.sender].hash = randomHashed;

        games[lotteryNo].players[msg.sender].purchasedAt = lotteryNo;

        games[lotteryNo].totalPrize += 10;

        emit PurchaseTicket(msg.sender, randomHashed);

        return true;

    }

* purchaseTicket(bytes32 randomHashed): Returns true if ticket is bought successfully, false otherwise. This function should be called when a user wants to buy a ticket. This function should only be called in submission stage, so there is the submission modifier that is described above. This function does:
  + 1: Gets current lottery no that is going on.
  + 2: Prevents purchasing more than one ticket per lottery
  + 3: Tries to transfer 10 EIP20 tokens from EIP20 network. The user should allow the contract to do that beforehand.
  + 4: Add ticket to the current game, and record randomHashed that the user sent.
  + 5: Record purchase timestamp of the user.
  + 6: Increment totalPrize of this games by 10.
  + 7: Emit purchase event.

function revealNumber(uint randomNum) public reveal returns(bool success) {

        bytes32 hashed = keccak256(randomNum, msg.sender);

        uint lotteryNo = getCurrentLotteryNo();

        require(

            games[lotteryNo].players[msg.sender].purchasedAt == lotteryNo,

            "Haven't bought a ticket in the current lottery or already revealed your random number");

        require(

        games[lotteryNo].players[msg.sender].hash == hashed,

        'Failed submitted random number revelation attempt');

        games[lotteryNo].revealedPlayers.push(msg.sender);

        games[lotteryNo].randomNumber ^= randomNum;

        delete games[lotteryNo].players[msg.sender].purchasedAt;

        emit RevealNumber(msg.sender, randomNum);

        return true;

    }

* RevealNumber(uint randomNum): This function returns true if revealing succeed, false otherwise. This function is needed to be called in reveal stage and the caller should be submitted the same randomNum in the submission stage of the same game. Otherwise revealing will not succeed. The reveal modifier assures the first requirement. This function does:
  + 1: Takes sha256 of randomNum that is sent and the address of the sender.
  + 2: Get current lottery number to check if this user bought a ticket in the same lottery.
  + 3: Verifies the sender is revealing the number he committed in the submission stage
  + 4: If so, add the user to revealers array.
  + 5: Change randomNum accumulator by xoring the randomNum that is revealed.
  + 6: Delete purchased at from the ticket so that it is not revealed again.
  + 7: Emit revealNumber event

function hashRandomNumber(uint randomNum) public view returns (bytes32 hashed) {

        return keccak256(randomNum, msg.sender);

    }

* hashRandomNumber(uint randomNum): A helper view function that does not spend gas, for hashing randomNum in local. Players can use this function to get randomHashed that is necessary to be provided to purchaseTicket function.

function logarithm2(uint x) public pure returns (uint y)

* logarithm2(uint x): Returns ceil of logarithm2 of x. This function is written in assembly and taken from source: <https://ethereum.stackexchange.com/questions/8086/logarithm-math-operation-in-solidity/32900#32900>

function checkPrizeWon(uint \_lotteryNo, uint \_index, address \_player) public view returns (uint prize) {

        uint lotteryNo = getCurrentLotteryNo();

        require(\_lotteryNo < lotteryNo, "The lottery with given number hasn't finished yet");

        Game storage gameAsked = games[\_lotteryNo];

        uint M = gameAsked.totalPrize;  // total amount of money collected in the game with given number

        uint indexRange = logarithm2(M);

        require(\_index < indexRange, "no winner in that position and beyond");

        uint P;

        bytes32 hash = keccak256(gameAsked.randomNumber);

        for(uint i=0; i <= \_index; i++) {

            P = M % 2;

            M = M >> 1;

            P += M;

            hash = keccak256(hash);

        }

        address winner = gameAsked.revealedPlayers[uint(hash) % gameAsked.revealedPlayers.length];

        if(winner == \_player) {

            return P;

        }

        else {

            return 0;

        }

    }

* checkPrizeWon(uint \_lotteryNo, uint \_index, address \_player): Returns prize that \_player won during \_lotteryNo’th lottery as \_index’th winner. If the player did not win a prize durint that lottery as \_index’th winner, prize returned as 0. Typically users call this function many times to check they won. This function is view function so that no gas is spent. This function does:
  + 1: Check if the \_lotteryNo is less than or equal to current lottery.
  + 2: Copy asked game to storage.
  + 3: Get total prize.
  + 4: Calculate total number of winners that is logarithmic to totalPrize.(indexRange)
  + 5: Require that given \_index is more that that range.
  + 6: Calculate winners’ prizes as follows
    - i1 = hash(randomNum) mod number of revealers
    - i2 = hash(hash(randomNum)) mod number of revealers
    - …

where randomNum is the accumulated randomNum that is generated by xoring the randomNumbers during reveal stage. i1 is the index of first winner, i2 is the index of second winner and so on. Prize of ith winner is calculated as follows(as required in the description):

Pi = ⌊M /2i⌋ + (⌊M /2i− 1⌋ mod 2) i = 1,...,⌈log2(M)⌉

* + This way all the collected money is distributed to winners and each winner takes half of the prize that the previous winner takes.

function withdraw(uint \_lotteryNo, uint \_index) public {

        uint prize = checkPrizeWon(\_lotteryNo, \_index, msg.sender);

        require(prize > 0, 'Won no prize, sorry.');

        require(games[\_lotteryNo].withdrawnPrizes[\_index] == false, "Already withdrawn the prize");

require(

            network.transfer(msg.sender, prize),

            "Failed to transfer your prize to your address, somehow.");

        games[\_lotteryNo].withdrawnPrizes[\_index] = true;

       emit PrizeWithdrawn(msg.sender, \_lotteryNo, \_index);

    }

* Withdraw(uint \_lotteryNo, uint \_index): A winner uses this function to say to contract to transfer his or her prize to his or her EIP20 account. This function does:
  + 1. Revert if sender won nothing yet
  + 2. Verifies the player hasn't withdrawn his prize
  + 3. Transfers winner's tokens
  + 4. marks the prize as paid
  + 5. Emit withdraw event