Introduction to Cryptocurrencies (67513)
Exercise 6

Alon Netser 31160253-6
Daniel Afrimi 20386583-7

June 29, 2020

1 First Scenario

Alice opens a channel with Bob. Bob does nothing (does not respond), and so Alice unilaterally closes the channel and withdraws her funds after the appeal period.

- In the beginning, before the channel's creation:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×409D42d5adEC6d5931357EF532A8DbE702C1a727	20.00 ETH	0	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×18f37C5e44287AbFcE00580Fdc56bd509918321d	20.00 ETH	0	1	

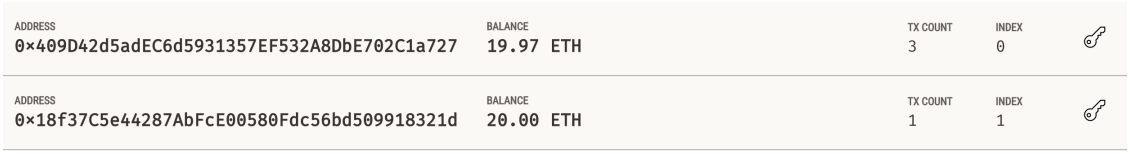
- Now Alice established the channel and deposited 10 ethers in it:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×409D42d5adEC6d5931357EF532A8DbE702C1a727	9.97 ETH	1	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×18f37C5e44287AbFcE00580Fdc56bd509918321d	20.00 ETH	0	1	

- Now Alice closes the channel, we wait 3 blocks (which is the appeal period) and then both Bob and Alice withdraw their money. After Bob'b withdrawal the balances are:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×409D42d5adEC6d5931357EF532A8DbE702C1a727	9.97 ETH	2	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×18f37C5e44287AbFcE00580Fdc56bd509918321d	20.00 ETH	1	1	

  and after Alice's withdrawal the balances are:



- The output of the python program:

```
Creating nodes
Creating channel
Notifying bob of channel
channel created 0xED337a2C9b6E3E847e05F1d396D2dAE2dD6d7611
------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------
ALICE CLOSING UNILATERALLY
------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = False
```

```
        block_number_at_closure = 3
------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = False
        block_number_at_closure = 3
------------------------------------------------------------
waiting
Bob Withdraws
Alice Withdraws
------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 0
        last_serial_num = 0
        channel_open = False
        block_number_at_closure = 3
------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 0
        last_serial_num = 0
        channel_open = False
        block_number_at_closure = 3
------------------------------------------------------------
```

2 Second Scenario

Alice opens a channel to Bob, they send some money back and forth. Bob then closes the channel. They wait for the appeal period to end and both withdraw funds.

- In the beginning, before the channel's creation:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×7EbDb082B22c338AC2B3c4869E9B954fE70172e7	9.97 ETH	1	0	
0×42B4c18488cf7B00ab60B36E16eF7B3fC6E7D2a2	20.00 ETH	0	1	

- Now Alice established the channel and deposited 10 ethers in it:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×7EbDb082B22c338AC2B3c4869E9B954fE70172e7	20.00 ETH	0	0	
0×42B4c18488cf7B00ab60B36E16eF7B3fC6E7D2a2	20.00 ETH	0	1	

- Now Alice sends 2 ethers to Bob, Bob sends her 1 ether back, and then she sends him 2 ethers twice (total of 4 ethers). The balances of each user can be seen in the python's output attached below. Afterwards, Bob closes the channel, we wait 3 blocks (which is the appeal period) and then both Bob and Alice withdraw their money. After Bob'b withdrawal the balances are:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×7EbDb082B22c338AC2B3c4869E9B954fE70172e7	9.97 ETH	1	0	
0×42B4c18488cf7B00ab60B36E16eF7B3fC6E7D2a2	25.00 ETH	2	1	

  and after Alice's withdrawal the balances are:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×7EbDb082B22c338AC2B3c4869E9B954fE70172e7	14.97 ETH	2	0	
0×42B4c18488cf7B00ab60B36E16eF7B3fC6E7D2a2	25.00 ETH	2	1	

- The output of the python program:

```
*** SCENARIO 2 ***
Creating nodes
Creating channel
Notifying bob of channel
----------------------------------------------------------------
Alice channel's information:
```

```
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
Alice sends money
-----------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 1
        last_owner2_balance = 20000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 1
        last_owner2_balance = 20000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
```

```
Bob sends some money
------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 2
        last_owner2_balance = 1000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 2
        last_owner2_balance = 1000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------
Alice sends money twice!
------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 4
        last_owner2_balance = 5000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 4
        last_owner2_balance = 5000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
```

```
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
BOB CLOSING UNILATERALLY
waiting
Bob Withdraws
Alice Withdraws
```

3 Third Scenario

Alice opens a channel to Bob, then sends him money several times. She then tries to close the channel using an old state message. Bob appeals successfully and they both withdraw funds.

- In the beginning, before the channel's creation:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×4bDB2F3D5f90c1c125364D8D23f0E6B1Df59F75d	20.00 ETH	0	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×f8C5057287Bb0Ea5E25948A40b973aAAa47B5173	20.00 ETH	0	1	

- Now Alice established the channel and deposited 10 ethers in it:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×4bDB2F3D5f90c1c125364D8D23f0E6B1Df59F75d	9.97 ETH	1	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×f8C5057287Bb0Ea5E25948A40b973aAAa47B5173	20.00 ETH	0	1	

- Now Alice sends 1 ether to Bob, saves the signed state (in order to "cheat" and publish this to the blockchain later) and then send twice more (1 ether each). The balances of each user can be seen in the python's output attached below. Afterwards, Alice closes the channel with the saved old state, indicating she sent Bob only 1 ether. But then Bob appeals with its latest state, indicating Alice sent him 3 ethers, and not just 1. This appeal is accepted because it has a higher serial-number (signed by Alice). Then both Bob and Alice withdraw their money. After Bob's withdrawal the balances are:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×4bDB2F3D5f90c1c125364D8D23f0E6B1Df59F75d	9.97 ETH	2	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×f8C5057287Bb0Ea5E25948A40b973aAAa47B5173	23.00 ETH	2	1	

  and after Alice's withdrawal the balances are:

ADDRESS	BALANCE	TX COUNT	INDEX	
0×4bDB2F3D5f90c1c125364D8D23f0E6B1Df59F75d	16.97 ETH	3	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0×f8C5057287Bb0Ea5E25948A40b973aAAa47B5173	23.00 ETH	2	1	

- The output of the python program:

```
*** SCENARIO 3 ***
Creating nodes
Creating channel
Notifying bob of channel
------------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 0
        last_owner2_balance = 0
        other_owner_last_signature = None
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
Alice sends money thrice
------------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 1
        last_owner2_balance = 10000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 1
        last_owner2_balance = 10000000000000000000
        other_owner_last_signature = ...
```

```
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
------------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 2
        last_owner2_balance = 2000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 2
        last_owner2_balance = 2000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
------------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
------------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
```

```
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 0
        channel_open = True
        block_number_at_closure = 0
-----------------------------------------------------------------
ALICE TRIES TO CHEAT
-----------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 1
        channel_open = False
        block_number_at_closure = 2
-----------------------------------------------------------------
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 1
        channel_open = False
        block_number_at_closure = 2
-----------------------------------------------------------------
Waiting one blocks
Bob checks if he needs to appeal, and appeals if he does
-----------------------------------------------------------------
Alice channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 3
        channel_open = False
        block_number_at_closure = 2
-----------------------------------------------------------------
```

```
Bob channel's information:
Local point of view of the channel:
        last_serial_number = 3
        last_owner2_balance = 3000000000000000000
        other_owner_last_signature = ...
Information on the channel from the blockchain:
        channel_balance = 10000000000000000000
        last_serial_num = 3
        channel_open = False
        block_number_at_closure = 2
-----------------------------------------------------------------
waiting
Bob Withdraws
Alice Withdraws
```

4 Scenario 4 (our custom scenario)

The code is attached in the end of the PDF, here's the output (self-explanatory):

```
*** SCENARIO 4 ***
Creating nodes
Alice creates two channels and notify Bob about them.
Channel #1: Alice sends 5 ethers to Bob.
Channel #1: Bob tries to cheat - he saves this state for
future use in another channel.
Channel #1: Bob send back to Alice the 5 ethers.
Channel #1: Bob closes the channel, with the old state
(saying he got the money).
Waiting 1 block...
Channel #1: Alice appeals with her most updated state,
saying she got the money.
Waiting 3 blocks (appeal period)...
Channel #1: Bob Withdraws
Channel #1: Alice Withdraws
Channel #2: Bob tries to cheat and close the channel with
the state of Channel #1 saying he got 5 ethers.
Good :) Bob didn't succeed to do it, because the contract
itself reverted this action.
```

5 Solidity Code

```
pragma solidity ^0.5.9;

contract Channel {

        // The one that opened the channel (and deposited the initial funds).
    address payable public owner1;
    address payable public owner2;

    // After 'appeal_period_len' blocks the funds can be withdrawn
    // (and no appeal can be made).
    uint public appeal_period_len;

    // This mapping holds the balances of each of the owners, that
    // can be withdrawn (after the appeal period ends).
    mapping (address => uint256) private userBalances;

    // The serial number of the last split of the balance between
    // the two owners.
    // One of the owners can appeal with a different balances split
    // with a larger serial number.
    int8 public last_serial_num;

    bool public channel_open;     // Is the channel open or closed.
    uint public block_number_at_closure;  // The block number at closure.

    modifier onlyOwners{
        require(msg.sender == owner1 || msg.sender == owner2,
                "Only an owner can call this function.");
        _;
    }

    modifier openChannel{
        require(channel_open,
                "This function must be called when the channel is open.");
        _;
    }

    modifier closedChannel{
        require(!channel_open,
                "This function must be called when the channel is open.");
        _;
    }
```

```
modifier appealPeriod{
    require(block.number - block_number_at_closure < appeal_period_len,
        "This function must be called during the appeal period.");
    _;
}

modifier appealPeriodEnded{
    require(block.number - block_number_at_closure >= appeal_period_len,
        "This function must be called after the appeal period ends.");
    _;
}

constructor(address payable _other_owner,
        uint _appeal_period_len) payable public {
    require(msg.value > 0,
        "The channel must be initialized with some money.");
    require(msg.sender != _other_owner,
        "Cannot have a channel connecting to youself.");

    owner1 = msg.sender;
    owner2 = _other_owner;
    appeal_period_len = _appeal_period_len;
    channel_open = true;
}

function default_split() onlyOwners openChannel external {
    // Closes the channel according to a default_split,
    // gives the money to party 1.
    // Starts the appeal process.
    channel_open = false;
    block_number_at_closure = block.number;
    userBalances[owner1] = address(this).balance;
    userBalances[owner2] = 0;
    last_serial_num = 0;
}

function one_sided_close(uint256 balance, int8 serial_num ,
                             uint8 v, bytes32 r, bytes32 s)
         onlyOwners openChannel external {
    //closes the channel based on a message by one party.
    // starts the appeal period
    require(balance <= address(this).balance,
        "balance can not be greater than the balance in the channel.");

    // Needs to verfiy the signature of the other node
```

```
    address signerPubKey = (msg.sender == owner1) ? owner2 : owner1;
    require(verify_sig(balance, serial_num, v, r, s, signerPubKey),
        "Signatue verification failed.");

    channel_open = false;
    block_number_at_closure = block.number;
    userBalances[owner1] = address(this).balance - balance;
    userBalances[owner2] = balance;
    last_serial_num = serial_num;
}

function appeal_closure(uint256 balance, int8 serial_num ,
                            uint8 v, bytes32 r, bytes32 s)
         onlyOwners closedChannel appealPeriod external {
    // appeals a one_sided_close. should show a newer signature.
    // only useful within the appeal period
    require(balance <= address(this).balance,
        "balance can not be greater than the balance in the channel.");

    // Needs to verfiy the signature of the other node
    address signerPubKey = (msg.sender == owner1) ? owner2 : owner1;
    require(verify_sig(balance, serial_num, v, r, s, signerPubKey),
        "Signatue verification failed.");
    require(serial_num > last_serial_num,
        "Serial number is not larger than the serial number at closure.");

    // Checks if the channel is closed,  the serial number of the
    // transaction is newer and the signature is valid
    userBalances[owner1] = address(this).balance - balance;
    userBalances[owner2] = balance;
    last_serial_num = serial_num;
}

function withdraw_funds(address payable dest_address)
         onlyOwners closedChannel appealPeriodEnded external {
    //withdraws the money of msg.sender to the address he requested.
    // Only used after appeals are done.
    dest_address.transfer(userBalances[msg.sender]);
}

function () external payable{
    revert();  // we make this contract non-payable.
               //Money can only be added at creation.
}
```

```
    function verify_sig(uint256 balance, int8 serial_num,
                            uint8 v, bytes32 r, bytes32 s, address signerPubKey)
             view public returns (bool) {
        bytes32 hashMessage = keccak256(abi.encodePacked(address(this),
                                            balance, serial_num));
        bytes32 messageDigest = keccak256(
                abi.encodePacked("\x19Ethereum Signed Message:\n32", hashMessage));
        return ecrecover(messageDigest, v, r, s) == signerPubKey;
    }

    function get_balance() public view returns(uint256) {
        return address(this).balance;
    }
}
```

6 Python Code

```
import web3
import time
import eth_account.messages
import web3.contract
from contract_info import ABI, compiled_sol


w3 = web3.Web3(web3.HTTPProvider("http://127.0.0.1:7545"))
APPEAL_PERIOD = 3  # The appeal period in blocks.


class ContractWrapper:
    """
    A Wrapper for the contract, being hold by the one of the owners.
    """

    def __init__(self, contract_obj, node_number):
        """
        Initialize a new wrapper for the contract object.

        :param contract_obj: The contract object to wrap.
        :param node_number: The index of the current owner in the channel.
                            0 means we are owner1, and 1 means we are owner2.
        """
        self.contract_obj = contract_obj
        self.node_number = node_number

        # Get the balance by querying the total balance in the channel by
        # calling the function 'get_balance'.
        self.channel_balance = contract_obj.functions.get_balance().call()

        if self.node_number == 0:
            self.other_party_address = self.contract_obj.functions.owner2().call()
        else:
            self.other_party_address = self.contract_obj.functions.owner1().call()

        self.last_serial_number = 0
        self.last_owner2_balance = 0
        self.other_owner_last_signature = None


def check_signature(message, sig, signer_public_key):
    """
```

```
    Checks a given signature on a given message by a given public key
    The message is given as a list of values, and the message types
    is a list of strings that describe their types in solidity.

    :param message: a list of values to sign (address of the contract,
                    balance and serial-number).
    :param sig: The signature of the signer key on this message.
    :param signer_public_key: The signer public key
                              (used to validate the signature).
    :return: True if the signature is valid, False otherwise.
    """
    message_types = ['address', 'uint256', 'int8']

    # Reconstruct the message hash.
    h1 = web3.Web3.soliditySha3(message_types, message)

    # This is the digest that ethereum actually signs.
    message_hash = eth_account.messages.defunct_hash_message(h1)

    return (w3.eth.account.recoverHash(message_hash, signature=sig) ==
            signer_public_key)


def get_v_r_s(signature):
    """
    Converts the signature to = 3 numbers that are accepted by ethereum.

    :param signature:
    :return: the signature in the format accepted by ethereum (3 numbers).
    """
    return (web3.Web3.toInt(signature[-1]) + 27,
            web3.Web3.toHex(signature[:32]),
            web3.Web3.toHex(signature[32:64]))


def wait_k_blocks(k: int, sleep_interval: int = 2):
    """
    Wait k blocks.
    :param k: the number of blocks to wait.
    :param sleep_interval: how much time to sleep between each check
                           with the blockchain.
    """
    start = w3.eth.blockNumber
    time.sleep(sleep_interval)
    while w3.eth.blockNumber < start + k:
```

```
        time.sleep(sleep_interval)


class LightningNode:
    def __init__(self, my_account):
        """
        Initializes a new node that uses the given local ethereum account
        to move money.

        :param my_account: the address of the account.
        """
        self.account_address = my_account

        # Dictionary mapping addresses of contracts to their
        # ContractWrapper objects.
        self.contracts = dict()

    def get_address(self):
        """
        :return: The address of this node on the blockchain
                 (its ethereum wallet).
        """
        return self.account_address

    def establish_channel(self, other_party_address, amount_in_wei):
        """
        Sets up a channel with another user at the given ethereum address.
        Returns the address of the contract on the blockchain.

        :param other_party_address: The other party address.
        :param amount_in_wei: The amount (in wei) to set the initial
                              value in the channel.
        :return: the contract address on the blockchain.
        """
        # Don't allow opening a channel with zero initial balance.
        if amount_in_wei <= 0:
            return

        txn_dict = {'from': self.account_address, 'value': amount_in_wei}

        # Submit the transaction that deploys the contract
        tx_hash = w3.eth.contract(
            abi=ABI, bytecode=compiled_sol["object"]
        ).constructor(other_party_address, APPEAL_PERIOD).transact(txn_dict)
```

```
        # Wait for the transaction to be mined, and get the transaction receipt.
        # In case of Timeout - an exception of type
        # web3.exceptions.TimeExhausted is raised,
        # and we don't want to catch it here (raise in upwards).
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)

        contract_address = tx_receipt.contractAddress

        # Create the contract instance with the newly-deployed address
        contract_obj = w3.eth.contract(address=contract_address, abi=ABI)

        # Add the contract's details to the contracts dictionary
        self.contracts[contract_address] = ContractWrapper(contract_obj,
                                                            node_number=0)

        return contract_address

    def notify_of_channel(self, contract_address):
        """
        A function that is called when someone created a channel with you
        and wants to let you know.
        The user that establishes the channel is the one that deposits the funds.

        :param contract_address: The address of the contract that was notified.
        """
        if contract_address not in self.contracts:
            contract_obj = w3.eth.contract(address=contract_address, abi=ABI)
            self.contracts[contract_address] = ContractWrapper(contract_obj,
                                                                node_number=1)

    def send(self, contract_address, amount_in_wei, other_node):
        """
        Sends money to the other address in the channel,
        and notifies the other node (calling its receive()).

        :param contract_address: The contract address.
        :param amount_in_wei: The amount (in wei) to send.
        :param other_node: The other node address to send the money to.
        """
        # Can not send money in a contract that is not in
        # our contracts dictionary.
        if contract_address not in self.contracts:
            return

        c = self.contracts[contract_address]
```

```
        # The given other_node address must equal the other_party_address
        # in the contract.
        if other_node.account_address != c.other_party_address:
            return

        # If we are node number 0 (i.e. owner1) then we are transferring
        # to owner2, so we need to add the amount to the balance of owner2.
        # If we are node number 1 (i.e. owner2) then we are transferring
        # to owner1, so we need to subtract the amount to the balance of owner2.
        sign = 1 if c.node_number == 0 else -1
        updated_owner2_balance = c.last_owner2_balance + sign * amount_in_wei

        # If the updated owner2 balance either is below zero,
        # it means that owner2 sent to owner1 more money than he has.
        # If it's above the channel's balance, it means that owner1
        # sent him too much money (more than the channel's balance).
        if not (0 <= updated_owner2_balance <= c.channel_balance):
            return

        # As stated in the forum:
        # https://moodle2.cs.huji.ac.il/nu19/mod/forum/discuss.php?d=98876
        # The only requirement for a valid serial is that it is strictly
        # larger than the last valid serial, so we simply keep track of the
        # largest seen serial (even if the message is invalid).
        updated_serial_number = c.last_serial_number + 1
        c.last_serial_number = updated_serial_number

        message = [contract_address,
                   updated_owner2_balance,
                   updated_serial_number]
        signature = self.sign(message)

        other_node_signature = other_node.receive(state_msg=(message, signature))

        # Check if the signature of the node is valid
        if other_node_signature is not None:
            other_node_signature_is_valid = check_signature(
                message, other_node_signature, other_node.account_address
            )
            if other_node_signature_is_valid:
                c.other_owner_last_signature = other_node_signature
                c.last_owner2_balance = updated_owner2_balance

    def receive(self, state_msg):
```

```
    """
    A function that is called when you've received funds.
    You are sent the message about the new channel state that is
    signed by the other user.

    :param state_msg:
    :return: a state message with the signature of this node acknowledging
             the transfer.
    """
    message, sender_signature = state_msg
    contract_address, updated_owner2_balance, updated_serial_number = message

    # Can not receive money in a contract that is not in our
    # contracts dictionary.
    if contract_address not in self.contracts:
        return

    c = self.contracts[contract_address]

    self_is_owner1 = (c.node_number == 0)
    self_is_owner2 = (c.node_number == 1)

    updated_owner2_balance_is_sane = (
            0 <= updated_owner2_balance <= c.channel_balance
    )
    self_balance_increased = (
        (self_is_owner1 and updated_owner2_balance <= c.last_owner2_balance)
        or
        (self_is_owner2 and updated_owner2_balance >= c.last_owner2_balance)
    )
    signature_is_valid = check_signature(message,
                                         sender_signature,
                                         c.other_party_address)

    if updated_serial_number <= c.last_serial_number:
        return

    # As stated in the forum:
    # https://moodle2.cs.huji.ac.il/nu19/mod/forum/discuss.php?d=98876
    # The only requirement for a valid serial is that it is strictly
    # larger than the last valid serial, so we simply keep track of the
    # largest seen serial (even if the message is invalid).
    c.last_serial_number = updated_serial_number

    if not (signature_is_valid and
```

```
            self_balance_increased and
            updated_owner2_balance_is_sane):
        return

    c.other_owner_last_signature = sender_signature
    c.last_owner2_balance = updated_owner2_balance
    signature = self.sign(message)

    return signature

def unilateral_close_channel(self, contract_address, channel_state=None):
    """
    Closes the channel at the given contract address.

    :param contract_address:
    :param channel_state: This is the latest state which is signed by the
                          other node, or None, if the channel is to be
                          closed using the current balance allocation.
    """
    # Can not close a channel not in our contracts dictionary.
    if contract_address not in self.contracts:
        return

    c = self.contracts[contract_address]

    tx_dict = {"from": self.account_address}

    # If there was not given a channel_state, and the other owner last
    # signature is None, it means that the channel needs to be closed
    # using the default_split function (which gives everything to owner1).
    if (channel_state is None) and (c.other_owner_last_signature is None):
        tx_hash = c.contract_obj.functions.default_split().transact(tx_dict)
    else:
        # If there was not given a channel_state, take the current state as c
        # hannel_state.
        if channel_state is None:
            channel_state = self.get_current_signed_channel_state(
                contract_address)

        # Now send the transaction containing a call to the 'one_sided_close'
        # function, using the 'channel_state'.
        balance, serial_num, v, r, s = channel_state
        tx_hash = c.contract_obj.functions.one_sided_close(
            balance, serial_num, v, r, s).transact(tx_dict)
```

```
        w3.eth.waitForTransactionReceipt(tx_hash)

    def get_current_signed_channel_state(self, chan_contract_address):
        """
        Gets the state of the channel (i.e., the last signed message from the
        other party).

        :param chan_contract_address:
        :return: The last signed state of the channel (balance, serial-number
                 and signature in 3 numbers).
        """
        # Can not get the current signed state of a channel not in
        # our contracts dictionary.
        if chan_contract_address not in self.contracts:
            return

        c = self.contracts[chan_contract_address]

        # Decompose the signature to 3 numbers that are accepted by ethereum
        v, r, s = get_v_r_s(c.other_owner_last_signature)

        return c.last_owner2_balance, c.last_serial_number, v, r, s

    def appeal_closed_chan(self, contract_address):
        """
        Checks if the channel at the given address needs to be appealed.
        If so, an appeal is sent to the blockchain.
        :param contract_address: The contract address to appeal in.
        """
        # Can not appeal a channel not in our contracts dictionary.
        if contract_address not in self.contracts:
            return

        c = self.contracts[contract_address]

        # Check if the channel is closed, and the serial number of the
        # last message is newer. If it does, an appeal can occur.
        channel_is_closed = not c.contract_obj.functions.channel_open().call()
        last_serial_number_in_contract = \
            c.contract_obj.functions.last_serial_num().call()

        if channel_is_closed and (last_serial_number_in_contract <
                                  c.last_serial_number):
            txn_dict = {"from": self.account_address}
            balance, serial_number, v, r, s = self.get_current_signed_channel_state(
```

```
            contract_address)
        tx_hash = c.contract_obj.functions.appeal_closure(
            balance, serial_number, v, r, s).transact(txn_dict)

        w3.eth.waitForTransactionReceipt(tx_hash)

def withdraw_funds(self, contract_address):
    """
    Allows the user to withdraw funds from the contract into his address.
    :param contract_address: The contract address to withdraw from.
    """
    # Can not withdraw from a channel not in our contracts dictionary.
    if contract_address not in self.contracts:
        return

    c = self.contracts[contract_address]

    # contract = self.contracts[contract_address]["contract"]
    tx_dict = {"from": self.account_address}

    # Call the contract function to withdraw the money to account address
    tx_hash = c.contract_obj.functions.withdraw_funds(
        self.account_address).transact(tx_dict)

    # Wait for the transaction to be mined, and get the transaction receipt
    w3.eth.waitForTransactionReceipt(tx_hash)

def debug(self, contract_address):
    """
    A useful debugging method. prints the values of all variables
    in the contract (public variables have auto-generated getters).
    :param contract_address:
    :return:
    """
    pass

def sign(self, message):
    """
    This function signs a message by the given signer account.
    The account is assumed to be unlocked (i.e., its private keys
    are managed by the connected ethereum node).
    The message is given as a list of values, and the message types
    is a list of strings that describe their types in solidity.

    :param message: The message to sign (address of the contract,
```

```
                    balance and serial-number).
        :return: The signature.
        """
        message_types = ['address', 'uint256', 'int8']
        message_hash = web3.Web3.soliditySha3(message_types, message)
        signature = w3.eth.sign(self.account_address, message_hash)

        return signature


# Opening and closing channel without sending any money.
def scenario1():
    print("\n\n*** SCENARIO 1 ***")
    print("Creating nodes")
    alice = LightningNode(w3.eth.accounts[0])
    bob = LightningNode(w3.eth.accounts[1])
    print("Creating channel")

    # creates a channel between Alice and Bob.
    chan_address = alice.establish_channel(bob.get_address(), 10 * 10 ** 18)
    print("Notifying bob of channel")
    bob.notify_of_channel(chan_address)

    print("channel created", chan_address)

    print("ALICE CLOSING UNILATERALLY")
    alice.unilateral_close_channel(chan_address)

    print("waiting")
    wait_k_blocks(APPEAL_PERIOD)

    print("Bob Withdraws")
    bob.withdraw_funds(chan_address)
    print("Alice Withdraws")
    alice.withdraw_funds(chan_address)


# sending money back and forth and then closing with latest state.
def scenario2():
    print("\n\n*** SCENARIO 2 ***")
    print("Creating nodes")
    alice = LightningNode(w3.eth.accounts[0])
    bob = LightningNode(w3.eth.accounts[1])
    print("Creating channel")
    # creates a channel between Alice and Bob.
```

```
    chan_address = alice.establish_channel(bob.get_address(), 10 * 10**18)
    print("Notifying bob of channel")
    bob.notify_of_channel(chan_address)

    print("Alice sends money")
    alice.send(chan_address, 2 * 10**18, bob)
    print("Bob sends some money")
    bob.send(chan_address, 1 * 10**18, alice)
    print("Alice sends money twice!")
    alice.send(chan_address, 2 * 10**18, bob)
    alice.send(chan_address, 2 * 10**18, bob)

    print("BOB CLOSING UNILATERALLY")
    bob.unilateral_close_channel(chan_address)

    print("waiting")
    wait_k_blocks(APPEAL_PERIOD)

    print("Bob Withdraws")
    bob.withdraw_funds(chan_address)
    print("Alice Withdraws")
    alice.withdraw_funds(chan_address)


# sending money, alice tries to cheat, bob appeals.
def scenario3():
    print("\n\n*** SCENARIO 3 ***")
    print("Creating nodes")
    alice = LightningNode(w3.eth.accounts[0])
    bob = LightningNode(w3.eth.accounts[1])
    print("Creating channel")
    # creates a channel between Alice and Bob.
    chan_address = alice.establish_channel(bob.get_address(), 10 * 10**18)
    print("Notifying bob of channel")
    bob.notify_of_channel(chan_address)

    print("Alice sends money thrice")

    alice.send(chan_address, 1 * 10**18, bob)
    old_state = alice.get_current_signed_channel_state(chan_address)
    alice.send(chan_address, 1 * 10**18, bob)
    alice.send(chan_address, 1 * 10**18, bob)

    print("ALICE TRIES TO CHEAT")
    alice.unilateral_close_channel(chan_address, old_state)
```

```
    print("Waiting one blocks")
    wait_k_blocks(1)

    print("Bob checks if he needs to appeal, and appeals if he does")
    bob.appeal_closed_chan(chan_address)

    print("waiting")
    wait_k_blocks(APPEAL_PERIOD)

    print("Bob Withdraws")
    bob.withdraw_funds(chan_address)
    print("Alice Withdraws")
    alice.withdraw_funds(chan_address)


# Alice is participating in a channel with Bob, and had previously closed
# a channel with him. Now Bob closes the channel using a message that belongs
# to the old channel. This should fail because the signature is also being
# done on the channel address
def scenario4():
    print("\n\n*** SCENARIO 4 ***")
    print("Creating nodes")
    alice = LightningNode(w3.eth.accounts[0])
    bob = LightningNode(w3.eth.accounts[1])

    print("Alice creates two channels and notify Bob about them.")
    chan1_address = alice.establish_channel(bob.get_address(), 5 * 10**18)
    bob.notify_of_channel(chan1_address)
    chan2_address = alice.establish_channel(bob.get_address(), 5 * 10 ** 18)
    bob.notify_of_channel(chan2_address)

    print("Channel #1: Alice sends 5 ethers to Bob.")
    alice.send(chan1_address, 5 * 10**18, bob)

    print("Channel #1: Bob tries to cheat - he saves this state for future "
          "use in another channel.")
    old_chan1_state = bob.get_current_signed_channel_state(chan1_address)

    print("Channel #1: Bob send back to Alice the 5 ethers.")
    bob.send(chan1_address, 5 * 10**18, alice)

    print("Channel #1: Bob closes the channel, with the old state "
          "(saying he got the money).")
    bob.unilateral_close_channel(chan1_address, old_chan1_state)
```

```
    print("Waiting 1 block...")
    wait_k_blocks(1)

    print("Channel #1: Alice appeals with her most updated state, "
          "saying she got the money.")
    alice.appeal_closed_chan(chan1_address)

    print("Waiting 3 blocks (appeal period)...")
    wait_k_blocks(APPEAL_PERIOD)

    print("Channel #1: Bob Withdraws")
    bob.withdraw_funds(chan1_address)
    print("Channel #1: Alice Withdraws")
    alice.withdraw_funds(chan1_address)

    print("Channel #2: Bob tries to cheat and close the channel "
          "with the state of Channel #1 saying he got 5 ethers.")
    bob.unilateral_close_channel(chan2_address, old_chan1_state)

    print("Waiting 3 blocks (appeal period)...")
    wait_k_blocks(APPEAL_PERIOD)

    print("Channel #2: Bob Withdraws")
    bob.withdraw_funds(chan2_address)
    print("Channel #2: Alice Withdraws")
    alice.withdraw_funds(chan2_address)


scenario1()
scenario2()
scenario3()
scenario4()
```