

Multisig Wallet for Ethereum

Smart Contracts for Cryptocurrencies

Presented by:

Shoval Loolian
David Strouk

Supervisor:

Oded Naor

March 2019

TABLE OF CONTENTS

Introduction	3
Blockchain	3
Bitcoin	4
Public and private keys	5
Ethereum	5
Problem description	7
Main goals	7
Secondary goals	7
Problem solution	7
Contract design	8
Multisig Wallet	8
K-of-N Multisig	8
Removal from group	9
Web-based application	9
Front/Back End Relation	12
Project tools	13
Programming languages	13
Solidity	13
Javascript/Vue	14
Work environment	15
Atom	15
Metamask / Web3	15
Remix IDE	16
Webstorm	17
Development process	17
Contract Writing	17
Contract Compiling	18
Test units	19
Documentation Generation	20
Contract Deployment	20
User Interface Building	21
Conclusion	21
Appendix	22
Bibliography	27

Introduction

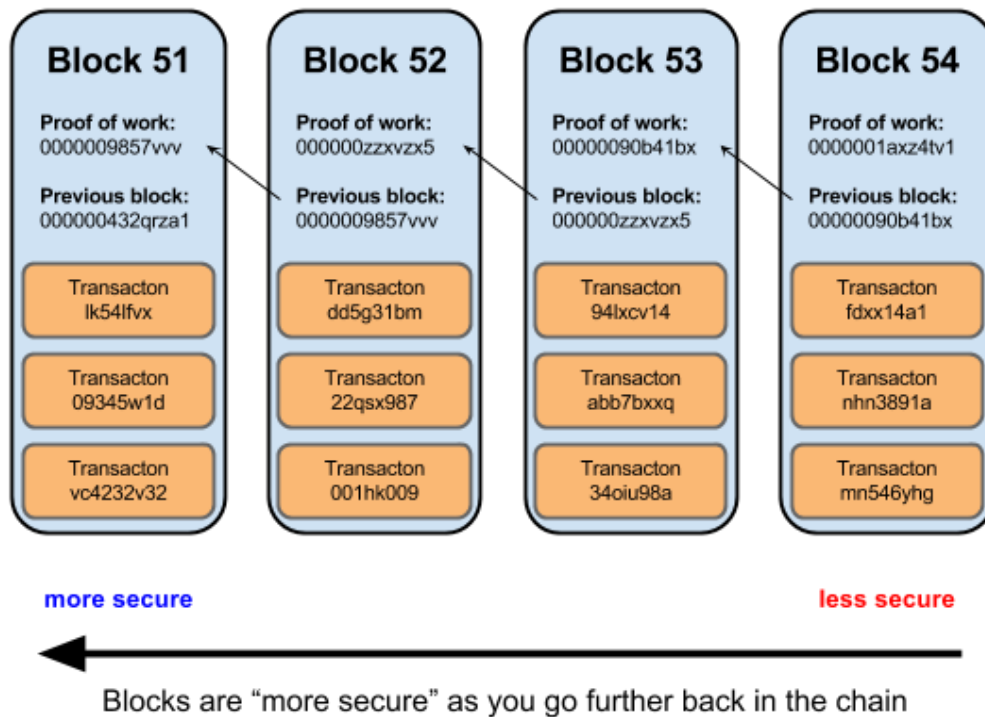
In this section, we will introduce relevant background and terms that are related to our project.

Blockchain

A blockchain is a growing general public ledger of cryptographically-signed transactions, where its main purpose is to handle money transfers between users. It consists of blocks, where each block is tied by a pointer to the previous block. Each block contains a set of transactions which were added to the network by the time of the block publication. Each transaction record consists of “*origin*” and “*destination*” addresses (one or more of each type) and an amount. An address is an identifier of alphanumeric characters, that represents a possible destination for a payment.

When someone wants to make a transfer, the transaction must be transmitted to the decentralized network and will be propagated on the network. It does not become part of the blockchain until it is verified and included in a block by a process called mining. Once added to the blockchain, the new block is increasingly trusted by the network as more blocks are added. Miners, as they verify transactions trustiness, receive a fee paid by the transaction emitter for adding it to the new block. The blockchain is secured by collective compute power of miners.

A blockchain is pseudo-anonymous: all transactions are public but are not tied to anyone's real identity. For each transaction, we can see that money was sent from address A to address B, but we can't know which entity is A or B unless he declares it as himself.



Bitcoin

Bitcoin is the first decentralized digital currency, which was introduced in January 2009. Bitcoins are digital coins you can send through the Internet. There is no institution controlling it.

Bitcoins are produced by people using software that solves mathematical problems (mining). As opposed to bank institutions where money transfers are centralized and controlled by a common authority, Bitcoin transactions are processed and validated independently by the Bitcoin network, and no centralized party is involved. Transfer of Bitcoins consists not of physically moving an object from A to B, but simply of adding a new, publicly accepted transaction to the blockchain.

Miners are awarded newly-minted Bitcoins or transaction fees for successfully finding blocks. The distributed algorithm ensures that the bounty of new Bitcoins will asymptotically approach 21 million, and the reward for mining will then become transaction fees only. The distributed algorithm dynamically adjusts how much computing power it takes to find a valid block, limiting block creation to around 1 block every 10 minutes for Bitcoin.

Public and private keys

Each person who wants to get an identity on the blockchain (and initiate a transfer or receive one) must own a pair of private and public keys.

The private key is a string of numbers and letters randomly generated, and it is required to control funds that belong to the owner. In other words, a private key represents the “password” of the account: transfer to another entity is possible only after signing a message using it. This message can later be verified by the public key associated to this private key.

The public key represents a possible destination for payment. It is derived from the private key by a hash function (given a private key, it is simple to deduce what is the public key, but the inverse operation is impossible).

Let's see how it works by describing a simple use case:

Alice's public key: 1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2

Alice's private key: E9873D79C6D8B0FB6A577863339F4453213303DA61F262

Bob's public key: 3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy

Alice wants to send 1 Bitcoin (BTC) to Bob. Of course, we suppose that she is the owner of an address which contains at least 1 BTC. To do that, she must prove that she is the owner of her address (1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2) by signing a message using its private key (E9873D79C6D8B0FB6A577863339F4453213303DA61F262). When the signed message has been verified, Alice can control her funds and choose to make a transfer of 1 BTC to Bob's public address (3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy). The transaction initiated by Alice is then published to the network, and will be confirmed once the miners approve it.

Ethereum

Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference.

Ethereum allows developers to program their own **smart contracts**. The language is Turing-complete, meaning it supports a broader set of computational instructions.

Smart contracts can, for example:

- Function as ‘multi-signature’ accounts, so that funds are spent only when a required percentage of people agree;
- Manage agreements between users, say, if one buys insurance from the other;
- Provide utility to other contracts (similar to how a software library works);

- Store information about an application, such as domain registration information or membership records.

Ethereum started as a way to make a general-purpose blockchain that could be programmed for a variety of uses. But very quickly, Ethereum's vision expanded to become a platform for programming apps.

DApps (decentralized applications) represent a broader perspective than smart contracts. A DApp is consisted at least of a smart contract and a web user interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services.

Ethereum has been introduced in 2015, and is based on same principles as Bitcoin, but with some main differences.

- The most significant difference is that Bitcoin serves only as a digital currency, when Ethereum introduced smart contracts, which interacts with the blockchain and developers can build DApps using those smart contracts (as described before);
- Bitcoin's supply cap is limited to 21 million in total, whereas Ethereum's supply cap has no limitation, and 18 million Ether are added every year;
- In Bitcoin, a new block is published on the blockchain approximately every 10 minutes, whereas in Ethereum it is about 12 to 14 seconds.

Problem description

The problem we tried to solve in this project is an everyday use case.

As in standard banking system, there is a need of shared accounts, in which funds are shared between two or more people.

For example, a joint account between a husband and a wife, or a shared business account between coworkers.

Main goals

Our goal is to implement a smart contract for creating shared wallets on the Ethereum blockchain.

Every shared wallet of this kind will be composed of N public keys (addresses) where each address represents a person, and in which every transaction within the wallet will require approval of K people in the group. Such a wallet will be called K -of- N Multisig, where K approvals out of N will be requested, as explained.

But what happens if a member of the group loses his access to the shared wallet? In order to prevent the whole group for losing all the money in the shared wallet, we want to give users the ability to remove an inactive user from the group. This means that users who have been removed from the group won't be able to approve transactions from this wallet anymore. An attempt of removal of a user from the group will be done by sending him a challenge. The targeted user will be removed only if he didn't answer the challenge in a pre-defined time.

Secondary goals

- Get familiar with concepts of blockchain, Bitcoin, Ethereum and smart contracts.
- Learn how to write in Solidity - programming language for writing smart contracts.
- Designing and programming the code of our smart contract.
- Build and run smart contracts on Ethereum blockchain.
- Search and solve misuses and issues in our code.
- Deploy our contract on a Test Network.
- Build a web dApp for user interface.

Problem solution

Our solution will consist of two main parts: the smart contract (back-end), and the web-based application (front-end). We will describe each one of these components, and then show the relation between both.

Contract design

We created two different contracts: **Multisig Wallet** and **K-of-N Multisig**.

Multisig Wallet

This is the factory contract, which is used to create and deploy **K-of-N Multisig** contract instances. We created this contract for enabling to create different shared wallets, each one with different users, N and K , such that every contract represents other shared wallet.

This contract contains two functions:

- `constructor` - initializes the factory contract
- `addGroup` - receives a list of users, and a number of approvals K , and creates an instance of **K-of-N-Multisig** (when N is the group size)

K-of-N Multisig

Shared wallet of N people which requests the approval of K members for making a payment.

When a user wants to make a transfer from the shared funds, he can publish a request for payment with a destination address and an amount he wants to transfer. When publishing this request, a new transaction is created and inserted to the shared wallet ledger. The ledger is keeping all the transactions, including their amount, destination address, number of approvals, which user approved it, and if the transaction was sent or is still waiting for approval.

The user who published the transfer will automatically give his approval on the published transaction. Then, the other group members will see this request as a transaction from the ledger on the blockchain, and every member who wants to approve the transfer, has to send an approval. When approving the transaction, the relevant transaction in the ledger will be updated by indicating the new user's approval, and increment the approvals counter for this transaction. Once the user approved this transaction, he can't cancel his approve.

When the transaction reaches K approvals, an appropriate amount will be transferred from the shared wallet to the transaction's destination address.

Beside all the functionalities that are related to payment transfers, there is also a possibility to remove an inactive member from the group.

When considering a member as possibly inactive, another member of the group can send him a **challenge**, by calling a function of the smart contract which will register in the blockchain the targeted user as challenged. A challenge will target a specific user, who will have to answer to it within a specific period of time. At each given time, one challenge can be active at most. This also means that only one user can be targeted at a given time.

Sending a challenge costs to the user who sent it `penalty` ether, dynamically calculated according to this formula: $\frac{balance}{2 \times N}$, where `balance` is the amount in the shared wallet and `N` is the current number of members in group.

If the challenged user has not responded in `BLOCKS_TO_RESPOND` blocks, then he can be removed from group by any other member, by calling the appropriate function. In case of response, the user will stay in the group, $2 * penalty$ ether will be charged from the shared wallet and the challenge sender will have to wait `BLOCKS_TO_BLOCK` blocks before he can send a new challenge. `BLOCKS_TO_RESPOND` and `BLOCKS_TO_BLOCK` are two constants, defined when deploying the contract. Those two constants can be initialized to any value by the contract programmer, but we thought it would be smarter to maintain a ratio of 1 : 2 between them. For example, if `BLOCKS_TO_RESPOND` = 10, then `BLOCKS_TO_BLOCK` should be 20. We decided of this ratio because we want to block the user, but still let him send challenge in a reasonable time in the future, because we don't know if his intention was naive or malicious.

For each challenge sending, we chose to charge the penalty twice: once from the user sending the challenge (this penalty will be charged anyway), in order to prevent users from sending flooding of challenges.

The second penalty is charged only in case the challenge was answered, and this time it will be charged from the shared wallet, meaning that all the group members will be impacted from it, including the user who has been targeted. This is done in order to prevent users from faking their missing and responding a challenge when receiving it, and this way preventing the suspicious user from making the challenge sender losing his money.

Removal from group

When a user is removed from group, two cases are possible:

- if $K < N$, then group size becomes $N - 1$, and number of approvals K remains as is;
- if $K = N$, both group size N and number of approvals K become $N - 1$ and $K - 1$.

At the time of removal, all transactions remain in ledger, but some of them will become irrelevant and will never be transferred: if $N = K$ and a transaction had $K - 1$ approvals, then it will be lost forever, because even if now only $K - 1$ approvals are needed, at the time of the transaction request it needed K approvals.

Since in Solidity there is no possibility of checking that an event has occurred without triggering a function, we can't know when `BLOCKS_TO_RESPOND` has passed. For this reason we implemented the function `tryToRemoveChallengedUser`. When `BLOCKS_TO_RESPOND` has passed, a user from the group would have to call this function for removing the challenged user. In case that `BLOCKS_TO_RESPOND` has passed, no user called `tryToRemoveChallengedUser`, and the challenged user calls `respondToChallenge`, the challenged user will not be removed from the group.

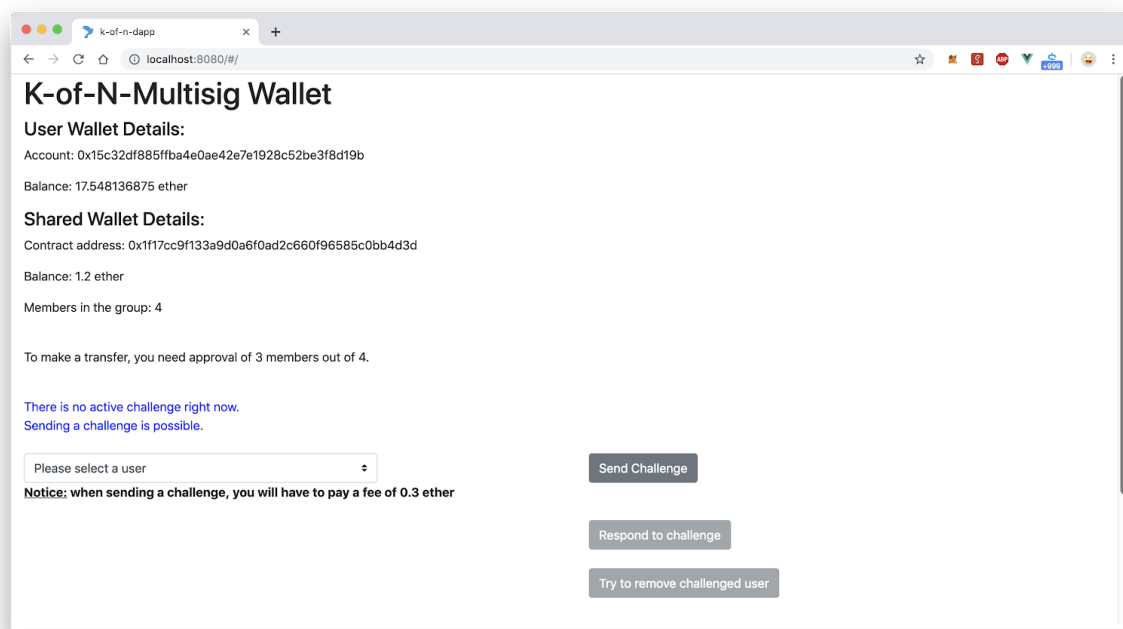
In case the group becomes empty ($N = 0$), then balance will automatically be transferred to a predefined address, so that funds will not be lost.

Complete documentation can be found at the appendix (page 22).

Web-based application

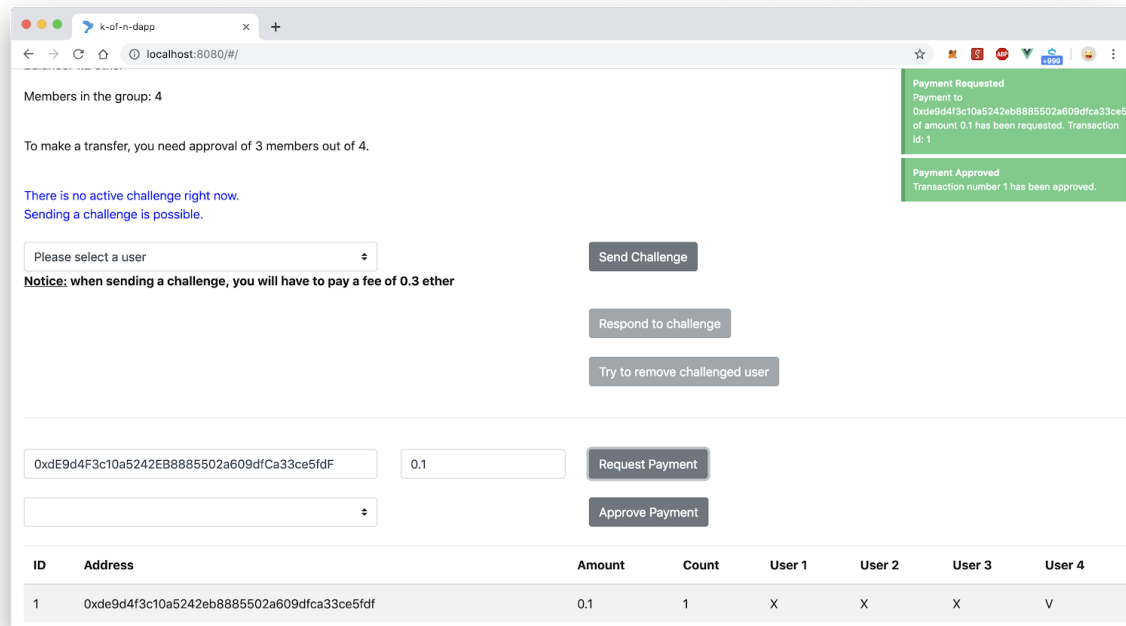
We built a user interface for interacting with our deployed smart contract on the blockchain.

When a user is connecting to the web application, the app shows user personal wallet details (account address and balance), and shared wallet details, if the user belongs to it (contract address, balance, K and N). Also, user can see if there is currently an active challenge or not, and if another user can be removed from the group. Finally, the app indicates which fee needs to be attached when sending an challenge (calculated dynamically by the formula above).



All members of the group have access to the transactions ledger, which shows for each transaction its details: destination address, amount in ether, and if there is an approval or not for each user (V or X). If the transaction has been transferred to destination, then the correspondent row in ledger will be colored in green, as shown in the screenshot below:

When contract transaction's sending has been confirmed by user, it is sent to the blockchain and calls the relevant function in the contract. On the app side, user is waiting for a confirmation whether the contract transaction has been approved or not. Once the contract transaction has been added to a block and registered in the blockchain, a notification appears in the app and updates the user on the status of the contract transaction which has been sent, as in the following screenshot:



Demo user interface can be found at <https://relaxed-engelbart-b2aacb.netlify.com/>.

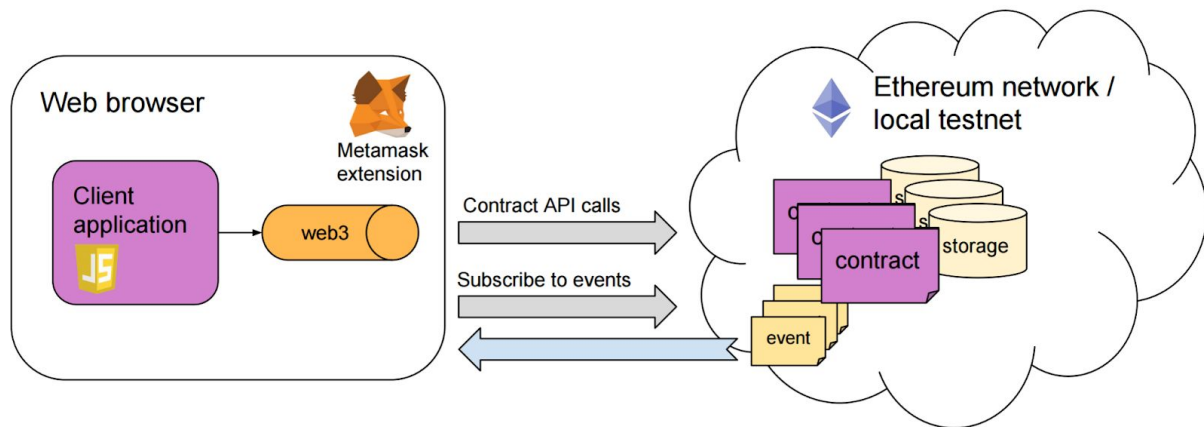
Front/Back End Relation

The communication between front-end (web interface) and back-end (smart contract) is essential to provide a seamless user experience.

The **front-end** is everything involved with what the user sees, including design and some languages like HTML, CSS and Javascript. In our project, it will consist of the web interface of our application and the Metamask extension which calls the functions of the contract.

The **back-end**, or the "server-side", is basically how the app works, updates and changes. This refers to everything the user can't see in the browser, like databases and servers. In our case, it includes all the contract's functions and the Ethereum network.

The connection between the front-end and back-end is done by **events**, written in our contract and which can be used to "call" JavaScript callbacks in the user interface of the dApp, which listen for these events.



Project tools

Programming languages

Solidity

Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM).

Using Solidity is extremely intuitive and easy to start for people who already have experience with programming and understands the basics of Ethereum.

Solidity features:

- Each class represents and contract, and is instantiated as shown below:

```

3 contract Will {
4
5     //Insert contract logic here
6
7 }
```

- Solidity includes the basic following types: `bool`, `uint`, `enum`, `array`, `string`, etc. but also unique types:
 - `address` - holds a 20 byte value (size of an Ethereum address). Address types also have members such as `balance()` and `transfer()` and serve as a base for all contracts.
 - `mapping(_KeyType => _ValueType)` - hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's default value.
- The `payable` keyword allows the function to send and receive ether. When the contract receives ether, it will store it in its own address.
- The `require` keyword states that everything written within the following parenthesis must be equal to true or else Solidity will throw an error and the execution will stop.

Solidity is a dynamically evolving language: when we started to write the contract, last version was 0.4.24, and as of today the version is 0.5.3. Each new version adds its set of semantic and syntactic changes, which improve the security and readability of the contract code. Even though, it is always possible to compile the contract code under any released version. For example, our contract code is compiled under version 0.4.25.

Solidity is not the only programming language for smart contracts, but it is the most developed and secure nowadays. For example, Vyper is a contract-oriented, pythonic programming language also compiling on the Ethereum Virtual Machine (EVM), but it is still in development and provides less community support. For all of those reasons we chose to write our smart contract in Solidity.

Javascript/Vue

In this project we used Javascript for two different purposes: writing test units to ensure that our smart contracts work as expected, and also for the user web interface (with Vue Framework)

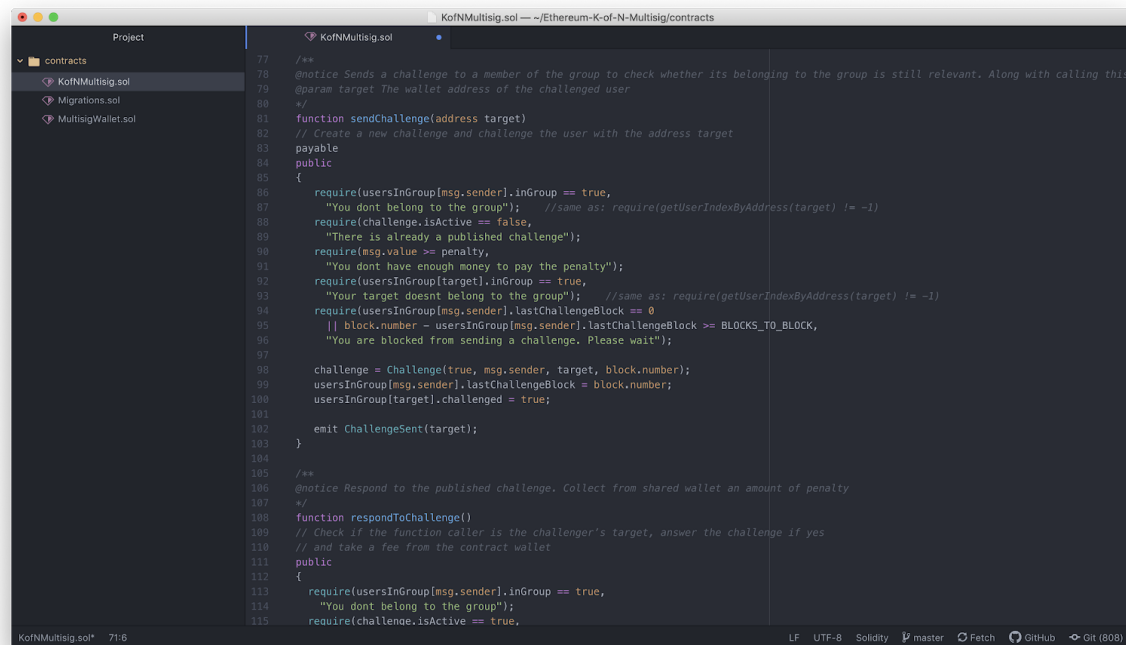
In case of writing unit tests, Javascript is an efficient language because it handles asynchronous programming, and since every contract function call waits for a response from the blockchain, it makes our code more convenient (by using keyword `await`).

Vue.js is an open-source JavaScript framework for building user interfaces and single-page applications. Since our web application is single-page only, we found that this framework was the most suitable for our needs.

Work environment

Atom

Atom is a free and open-source text and source code editor with embedded Git Control, developed by GitHub. We chose to write Solidity code in Atom because it had a plugin enabling syntax highlighting for Solidity.



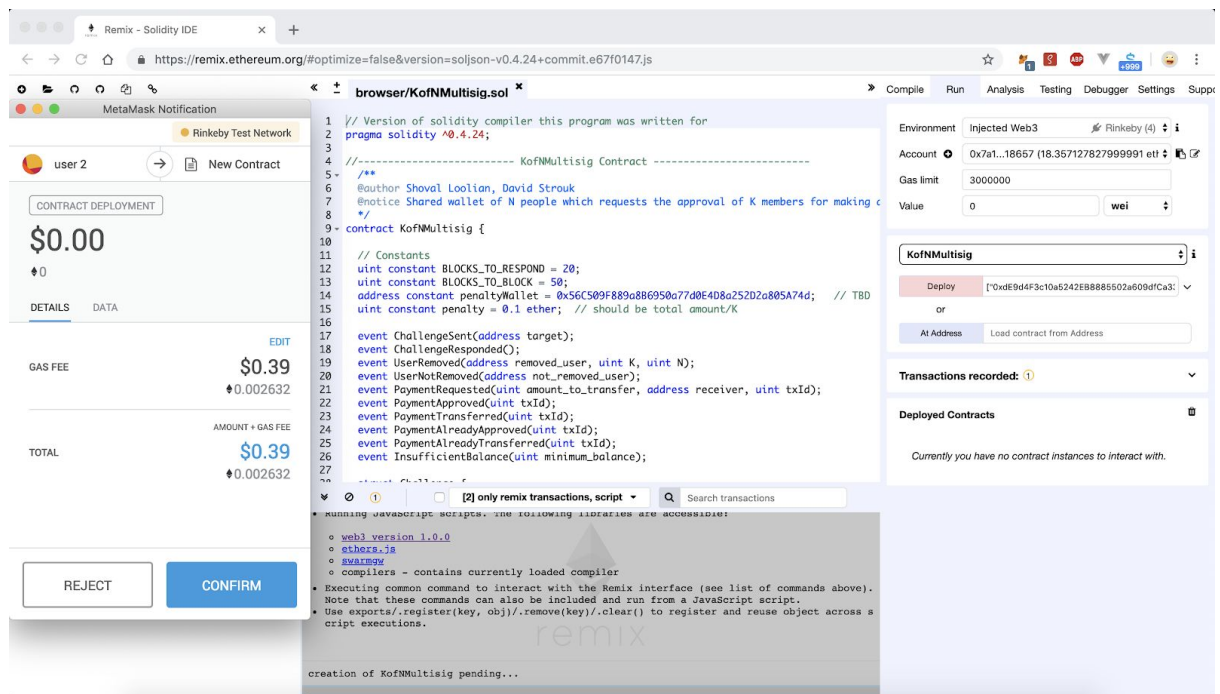
Metamask / Web3

web3.js is a collection of libraries which allow to interact with a local or remote ethereum node, using a HTTP or IPC connection. Every function call from a smart contract in Javascript is using functions from the web3 library.

MetaMask is an extension for accessing Ethereum enabled distributed applications, or "Dapps" in a browser. The extension injects the Ethereum web3 API into every website's javascript context, so that Dapps can read from the blockchain.

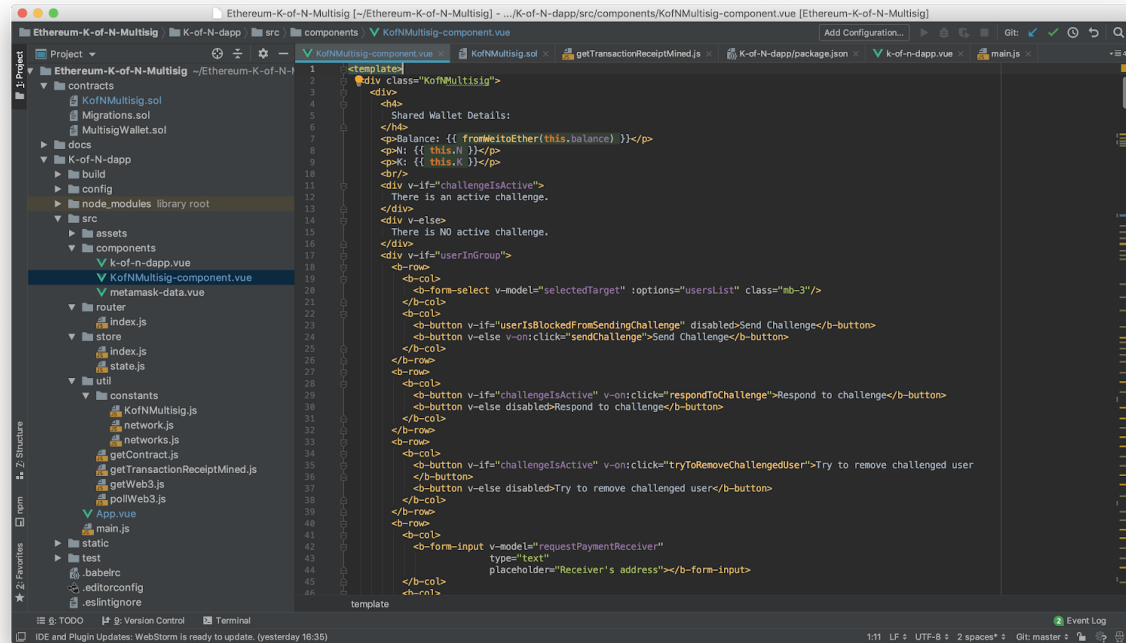
Remix IDE

Remix is an open source tool that helps write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally. It is convenient when deploying contract to Ethereum Testnet or Mainnet because it easily communicates with Metamask extension.



Webstorm

WebStorm is a cross-platform IDE primarily for web, JavaScript and TypeScript development. It supports development on Vue framework and enables debugging.



Development process

In this section we will describe the different steps for building our project. The first step was to read about blockchain, Bitcoin, Ethereum and all its definitions. Then we followed an online tutorial for learning the programming language Solidity (<https://cryptozombies.io/>).

Contract Writing

We started by designing the contract by defining its attributes and methods. In the first time, we implemented all the functions related to payment requests and transfers. We defined the different structures needed in our contract: `User` and `Transaction`, with the relevant attributes in each one of them. We also defined a ledger that contains all the transactions which belong to the contract.

Then in the second time, we implemented the structures and functions related to challenge sending and user removal. We defined a new struct `Challenge` which contains information about the challenged user and the number of blocks left to respond.

While designing the contract, we ran into some issues :

- We added `require` and `assert` conditions to every function, checking that all the parameters values and the contract's current state are valid for successfully calling the function. We had to decide when it suited best to use a condition of type `require` or `assert`, as explained in Solidity documentation:

The `require` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts. If used properly, analysis tools can evaluate your contract to identify the conditions and function calls which will reach a failing `assert`. Properly functioning code should never reach a failing `assert` statement; if this happens there is a bug in your contract which you should fix.

- We added the fallback function and added to it the keyword "payable" so users could send ether to the shared account contract's address, which will be transferred when requesting transactions (when a transaction approved, the contract's balance must be at least the correspondent amount to transfer).
- When sending a challenge, we wanted to check how much time has been elapsed from the moment the challenge was sent, but we realized it was impossible to measure time in seconds. Our solution was to keep the block number in which the challenge sending transaction was registered and use the block number of the current transaction (`block.number`) for calculating how much blocks have passed since sending the challenge and check if user is blocked or not from sending a new one. Knowing that every block takes approximately 12-15 seconds, it is possible to get an estimated evaluation of time.
- During the implementation, we investigated parameters types for deciding the most relevant type for each parameter purpose. For example, we had to think about whether we want our structures to be stored in `memory` (which is not persistent) or `storage` (where the state variables are held). In addition, we got familiar with the `mapping` type, which does not have a length or a concept of a key or value being "set", and found it very useful.
- As we were implementing the contract, we had to make a lot of changes since we noticed that some of the cases were not supported: we added missing parameters to the contract, help functions, edge cases handling, validation checkings (`require` conditions), ...

Contract Compiling

When we finished the contract writing, we copied it online to Remix IDE and compiled it according to the relevant Solidity version (0.4.25).

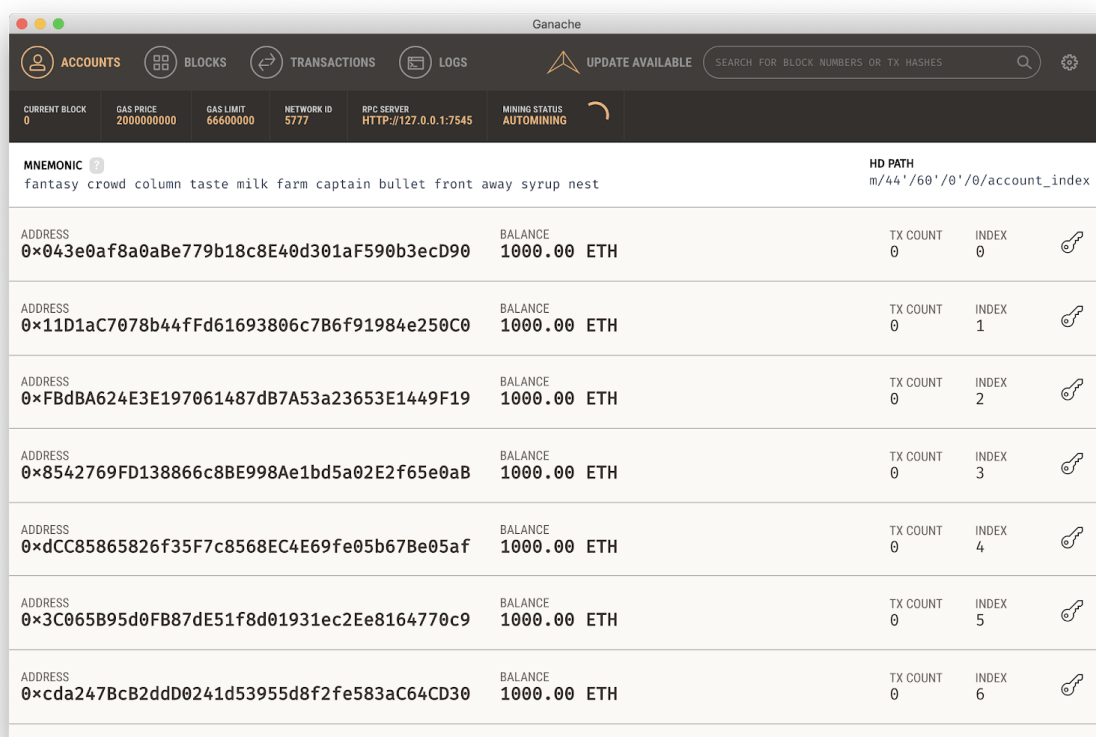
When building the contract, the compiler indicated us if our code was not correctly written by displaying errors and warnings.

We called the different functions with basic tests and verified that contract calls were working.

Test units

For the purposes of our tests, we needed to create some accounts with ether in each one of them, and we wanted to repeat this action for each one of our tests. Of course we could have done this manually, but this would have been less efficient.

To resolve this issue we used Ganache, a personal Ethereum blockchain which can be used to run tests, execute commands, and inspect the state while controlling how the blockchain operates. When launching a private blockchain on Ganache, it initializes a predefined number of wallets (in our case 15) with an account default balance for each one of them (in our case 1000 ether).

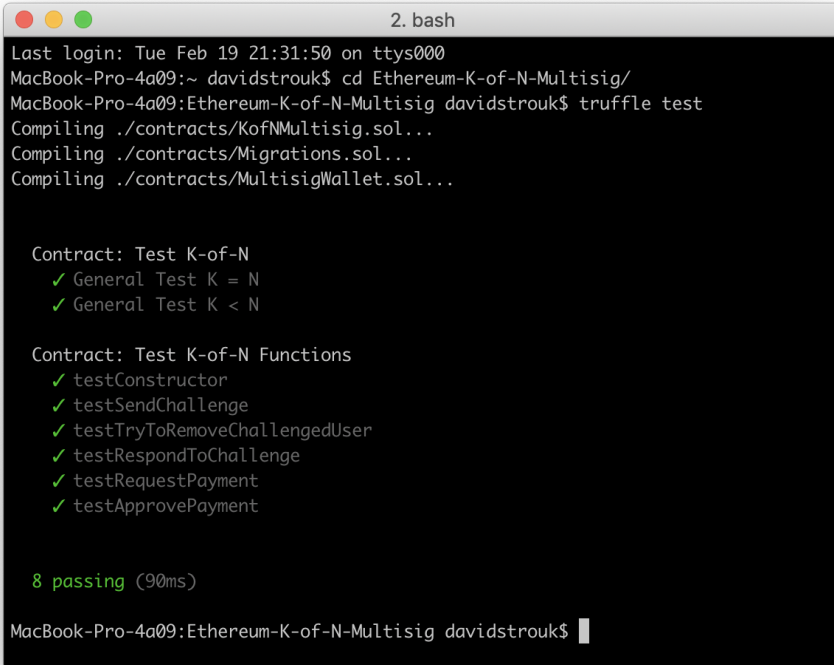


Then we wrote our tests in Javascript, when we ran a test function for each function of the contract, and also general tests which simulate real-life scenarios, when including all sorts of edge cases. A complete description of all the tests can be found at the appendix (page 25).

For comparing actual and expected values of each one of the parameters in the contract, we added help "get functions" which returns parameter values in the contract. For example, the help function "getK()" will return the current number of needed approvals K.

Finally, we used the Truffle framework to run tests on the private blockchain created by Ganache, by calling the command `truffle test`.

For the tests to be run we had to install locally on our computer the same specific Solidity version as of the contract, but since the newest Solidity version was not compatible with our contract, we had to install the previous version and make it retroactively compatible with the Truffle framework, something that was not so evident.

A terminal window titled '2. bash' showing the execution of 'truffle test'. It displays the compilation of three Solidity files and the successful execution of 8 tests. The tests are grouped into two categories: 'Contract: Test K-of-N' and 'Contract: Test K-of-N Functions'.

```
2. bash
Last login: Tue Feb 19 21:31:50 on ttys000
MacBook-Pro-4a09:~ davidstrouk$ cd Ethereum-K-of-N-Multisig/
MacBook-Pro-4a09:Ethereum-K-of-N-Multisig davidstrouk$ truffle test
Compiling ./contracts/KofNMultisig.sol...
Compiling ./contracts/Migrations.sol...
Compiling ./contracts/MultisigWallet.sol...

Contract: Test K-of-N
  ✓ General Test K = N
  ✓ General Test K < N

Contract: Test K-of-N Functions
  ✓ testConstructor
  ✓ testSendChallenge
  ✓ testTryToRemoveChallengedUser
  ✓ testRespondToChallenge
  ✓ testRequestPayment
  ✓ testApprovePayment

8 passing (90ms)
MacBook-Pro-4a09:Ethereum-K-of-N-Multisig davidstrouk$
```

Documentation Generation

We used Docusaurus to automatically generate a website with our contracts documentation, according to this online tutorial: <https://github.com/OpenZeppelin/solidity-docgen>.

We published the generated documentation on the following website:

<https://boring-euler-80133b.netlify.com>.

This documentation is also available at the appendix (page 22).

Contract Deployment

A **testnet** is an alternative blockchain, to be used for testing. Testnet coins are separate and distinct from coins, and are never supposed to have any value. This allows application developers to experiment, without having to use real coins or worrying about breaking the main blockchain.

When all of our tests passed successfully, we deployed our contract on the Rinkeby Testnet. We opened the online Remix IDE, copied our contract code, compiled it with the relevant version and deploy it using Metamask.

For deploying and calling functions on the Rinkeby Testnet, we had to request “test” ether from a faucet. We found a Rinkeby Faucet on the internet (<https://faucet.rinkeby.io/>) which gave us 18.75 ether when publishing a post on Facebook/Twitter with our Ethereum address. After completing this action, we can use this “test” ether to call functions on the Rinkeby Testnet.

User Interface Building

For a simple user experience, we decided to build a web app communicating with our contract. We wrote this app in Javascript with the framework Vue.

After following a tutorial of DApp implementation in Vue (which can be found at the address: <https://itnext.io/create-your-first-ethereum-dapp-with-web3-and-vue-js-c7221af1ed82>) we started to design our webapp.

We added events to the contract in Solidity to communicate between the contract and the user interface. Every time an event is emitted in the contract, it will be transferred to the user interface and trigger the appropriate action. For example, if the contract emits the `PaymentApproved` event (`emit PaymentApproved(...)`), then in the web user interface a notification will appear, the relevant transaction in the ledger will be colored in green and every field impacted by the change will be updated.

Using Javascript and the library Web3.js was very difficult since every function call had to be asynchronous (because of network calls), and the web3 library does not implement native asynchronous calls, so we had to write wrapper functions that overcome with this issue. In addition, we had to get familiar with asynchronous principles in code and its implementation in Javascript (`async/await` functions, `Promises` use, ...)

Conclusion

During this project, we learned a lot of knowledge related to blockchain and smart contracts. First we got familiar with new notions, we learned new programming languages: Solidity, Javascript, also with the framework Vue. We understood how to use a blockchain network, and how to connect between frontend and backend.

For achieving this project we had to learn a lot of new skills, and some of them were quite difficult to assimilate. However, the project was very interesting and enriching, and the final product is very satisfying.

Appendix

Documentation

This documentation can also be found here: <https://boring-euler-80133b.netlify.com>

MultisigWallet

contract MultisigWallet

Create new shared wallets.

Source: [MultisigWallet.sol](#)

Author: Shoval Loolian, David Strouk

Functions

constructor

```
function () public
```

Initialize MultisigWallet contract.

addGroup

```
function addGroup(address[] wallets, uint k) public returns (KofNMultisig)
```

Create new shared wallet of type "KofNMultisig" between N users.

Parameters:

wallets - The wallets addresses of the N users

k - The size of required approvals

Returns:

The address of the new shared wallet contract

KofNMultisig

contract KofNMultisig

Shared wallet of N people which requests the approval of K members for making a payment. If a member is inactive, another member of the group can send him a challenge which costs `penalty` ether. If the challenged user has not responded in `BLOCKS_TO_RESPOND` blocks, then he can be removed from group. In case of response, the user will stay in the group, `penalty` ether will be charged from the shared wallet and the challenge sender will have to wait `BLOCKS_TO_BLOCK` blocks before he can send a new challenge.

`penalty` is calculated dynamically by the following rule : $\text{balance}/N$, where `balance` is the amount in the shared wallet and N is the current number of members in group.

Source: [KofNMultisig.sol](#)

Author: Shoval Loolian, David Strouk

Functions

constructor

```
constructor(address[] wallets, uint K) public
```

Initialize K-of-N-Multisig contract with N users and K approvals.

Parameters:

`wallets` - The wallets addresses of the N users

K - The size of required approvals

approvePayment

```
function approvePayment(uint txId) public
```

Give user's approval to transfer a payment requested by the `requestPayment` function.

When reaching K approvals for transaction `txId`, check if balance of shared wallet is enough to make the transfer.

Minimum balance to allow the transfer is calculated according to the following condition: if a challenge is active, balance should be at least

`amount_to_transfer + penalty`, else only `amount_to_transfer`.

There is no retroactively verification of the approval's number, hence removal of a user from the group cannot influence confirmation of previous transactions.

Parameters:

txId - The id of the transaction which the user is willing to give approval

requestPayment

```
function requestPayment(uint amount, address receiver) public
```

Request to transfer a certain amount from the shared wallet to address receiver.

Transfer will be allowed only when reached K confirmations from users in group by calling approvePayment function.

The user which requests the payment automatically approves it, hence he does not have to call approvePayment function.

Parameters:

amount - The requested amount of Wei to transfer

receiver - The destination address of the payment

respondToChallenge

```
function respondToChallenge() public
```

Respond to the challenge published by sendChallenge. If the challenged user has responded in less than BLOCKS_TO_RESPOND blocks, collect from shared wallet an amount of penalty ether.

sendChallenge

```
function sendChallenge(address target) public payable
```

Sends a challenge to a member of the group (target) to check whether its belonging to the group is still relevant. Along with calling this function user has to send penalty ether, calculated as mentioned before.

Parameters:

target - The wallet address of the challenged user

tryToRemoveChallengedUser

```
function tryToRemoveChallengedUser() public
```

Try to remove the challenged user. If the challenged user has not answered after `BLOCKS_TO_RESPOND` blocks, then he will be removed from group. Calling this function in less `BLOCKS_TO_RESPOND` blocks since challenge sending will do nothing.

fallback

```
function () external payable
```

Send any amount of ether to the shared wallet.

Tests Detailed Description

Multisig Wallet

Since this contract contains only one function `addGroup` which creates an instance of a `K-of-N Multisig` contract, we tested it online and did not run unit tests for it.

K-of-N Multisig

- **testConstructor**
 - Check edge cases: $K = 0$, $K > N$, empty list.
 - Check that all parameters are initialized.
- **testSendChallenge**
 - Check edge cases: challenge sender/target not in group, challenge already published, sending transaction with not enough money, challenge sender is blocked.
 - Sending a challenge and check that all parameters are valid.
 - Sending a challenge, try to send a new one before `BLOCKS_TO_BLOCK` blocks.
 - Sending a challenge, try to send a new one after `BLOCKS_TO_BLOCK` blocks.
- **testTryToRemoveChallengedUser**
 - Check edge cases: function caller not in group, there is no challenge currently.
 - Calling the function after `BLOCKS_TO_RESPOND`, leads to removal of a user only once.
 - Calling the function before `BLOCKS_TO_RESPOND`, does not remove any user.

- Calling the function leads to removal of a user, until there is only one user left.

- **testRespondToChallenge**

- Check edge cases: function caller not in group/not challenged, there is no challenge currently.
- Calling the function before `BLOCKS_TO_RESPOND`, and check that all parameters are valid, including balance update with sent penalty.

- **testRequestPayment**

- Check edge cases: function caller not in group, amount = 0, amount is negative.
- Calling the function once and check that all parameters are updated.
- Calling the function in the second time and check that the transactions number is updated, and that every request is independent.

- **testApprovePayment**

- Check edge cases: function caller not in group, wrong transaction number, not enough money in shared wallet to send the approved transaction, with and without a challenge.
- Calling the function once and check that all parameters are updated, including balance.
- Check the functionality in case of $K = N = 1$ (the user doesn't need to call the `approvePayment` function since it is called within the `requestPayment` function)

- **General Test $K < N$, General Test $K=N$**

In this two tests we checked some possible scenarios which simulates realistic flows in the contract:

- Two different users are calling `tryToRemoveChallengedUser`.
- Transaction becomes irrelevant after removing a user from the group, in case of a user removal which causes the number of approvals to be reached.
- Only one user in the group.
- User who removes himself from the group.
- Request and approve transactions in case of active challenge.
- Publish a challenge by any user while another user in the group is blocking from sending a challenge.
- Each transaction can be approved by any user only once.

Bibliography

- Bitcoin book: <https://github.com/bitcoinbook/bitcoinbook>
- Ethereum book: <https://github.com/ethereumbook>
- Solidity documentation: <https://solidity.readthedocs.io/en/v0.4.25/>
- Cryptozombies Solidity tutorial: <https://cryptozombies.io/>
- Docusaurus installation: <https://docusaurus.io/docs/en/installation>
- Solidity Documentation Generator: <https://github.com/OpenZeppelin/solidity-docgen>
- Tutorial "Create your first Ethereum dAPP with Web3 and Vue.JS":
 - Part 1:
<https://itnext.io/create-your-first-ethereum-dapp-with-web3-and-vue-js-c7221af1ed82>
 - Part 2:
<https://itnext.io/create-your-first-ethereum-dapp-with-web3-and-vue-js-part-2-52248a74d58a>
 - Part 3:
<https://itnext.io/create-your-first-ethereum-dapp-with-web3-and-vue-js-part-3-dc4f82fba4b4>