Oracles Network

Abstract

In this paper we propose an open permissioned network based on Ethereum protocol with Proof of Authority consensus by independent validators.

Keywords: Ethereum, Proof of Authority, Consensus, Validators, Public Notary

Authors: Igor Barinov, Viktor Baranov, Pavel Khahulin

Introduction

Oracles Network is an open, public, permissioned blockchain based on Ethereum[1] protocol. To reach consensus on a global state, it uses a Proof of Authority consensus algorithm. PoA consensus is a straightforward and efficient form of Proof of Stake with known validators and governance-based penalty system. A list of validators is managed by a smart contract with governance by validators. During an initial ceremony, master of ceremony distributes keys to 12 independent validators. They add 12 plus one more to reach initial requirements for the consensus. To be validators on the network, a master of ceremony asks them to have an active notary public license within the United States. A concerned third party can cross-validate validators' identities using open data sources and ensure that each validator is a good actor with no criminal records. In the proposed network, identity of individual validator and trust to independent and non-affiliated participants will secure the consensus.

The network is fully compatible with Ethereum protocol. The network supports only Parity client version 1.7 and later. The network supports trusted setup, on-chain governance, and a variety of "proof of identity" oracles. We believe that Oracles Network will close a gap between private and public networks, and will become a model for open networks based on PoA consensus.

Proof of Authority

AuthorityRound (AuRa)

Aura is one of the Blockchain consensus algorithms available in Parity. It is capable of tolerating up to 50% of malicious nodes with chain reorganizations possible up to a limited depth, dependent on the number of validators, after which finality is guaranteed. This consensus requires a set of validators to be specified, which determines the list of blockchain addresses which participate in the consensus at each height. Sealing a block is the act of collecting transactions and attaching a header to produce a block.[2]

At each step the primary node is chosen that is entitled to seal and broadcast a block, specifically step modulo #_of_validatorsthe validator is chosen from the set. Blocks should be always sealed on top of the latest known block in the canonical chain. The block's header includes the step and the primary's signature of the block hash.[2:1]

Block can be verified by checking that the signature belongs to the correct primary for the given step. Finality of the chain can be achieved within at most 2 x #_of_validator steps, after more than 50% of the nodes are signed on a chain and then they are signed again on those signatures. [2:2]

History of POA

On March 6, 2017, a group of blockchain companies announced[3] new blockchain based on Ethereum protocol with Proof of Authority consensus [4]. Spam attack on the Ropsten testnet was the reason to create a new public test network[5]. This network was named Kovan, for a metro station in Singapore, where companies who founded the network are located. It is a common name convention for Ethereum test networks, for example, Morden, Ropsten, and Rinkeby are names of metro stations.

Adoption of Kovan blockchain

In the table below we show stats for Main (Homestead) and Test (Kovan) Ethereum networks.

NetworkTypeBlocks minedTx createdContract createdAccounts created

KovanTestnet3,417,5272,859,54954,38418,082

HomesteadMainnet4,203,31950,374,3591,488,0724,957,479

Large numbers of transactions, smart contracts, and accounts on the test network show adoption from the community and proven utility benefit.

Oracles Network

Validators

Independent U.S. public notaries with active commission license will be the first validators in Oracles Network. For the initial ceremony, 12 initial keys will be created by a master of ceremony. He will distribute those keys to individual validators. Each validator will change a key to a new subset of keys using a client-side DApp. After the initial distribution of licenses, an additional validator can be added through the voting process on the built-in Governance DApp. A majority of votes will be needed from validators to be accepted into the smart contract with a list of validators.

Economy

Crowdsale will take place before the launch of the main network.

Purchased coins will be included in the genesis block and will create initial liquidity for the network.

Validators will start to create blocks and generate a reward for the network security. For each generated block, a validator who created it will get one coin and all fees for transactions. Each validator has equal rights to create a block.

The network will start with 12 validators. With 12 validators active, each validator will create one block every 12 blocks. For each block one coin will be created as a reward for validators and one coin for self sustaining of the network.

A block will be generated with an average time of 5 seconds. During the first year of the network, validators will create 31,536,000 sec/5 sec per block = 6,307,200 blocks.

The emission rate for validators is 2.5% for the first year of the network. The network will use disinflation model, and emission will decrease every year. An additional 2.5% will be added to support sustainability of the network.

Therefore, 2.5% of the network supply will be generated as a reward for validators to secure the network. And 2.5% of total supply will be distributed to support sustainability. Validators will be able to propose areas of spending:

burn coins

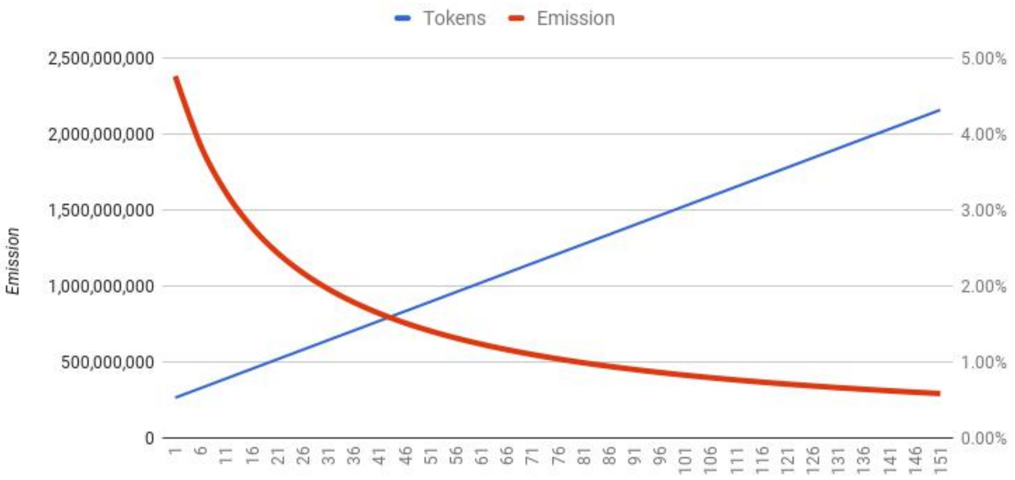
hold coins

spend on R&D Foundation

Sustainability emission will be governed by decentralized apps.

Emission rate.

X-axis - %, Y-axis - Years



Use Cases

Inexpensive Network

Oracles Network provides inexpensive consensus to secure the network. Users can run Ethereum programs on Oracles Network and spend less money on transaction fees. Overall cost of the network's security will also be cheaper due to lower market cap.

Problem

Though the issuance of ETH is in a fixed amount each year, the rate of growth of the monetary base (monetary inflation) is not constant. This monetary inflation rate decreases every year, making ETH a disinflationary currency (in terms of monetary base). Disinflation is a special case of inflation in which the amount of inflation shrinks over time.[6]

In 2017 the issuance rate of Ether is 14.75%.[7] Roughly five Ethers per block are issued. Because Ethereum rewards Uncles it means that there may be more or less than five Ethers.[7:1]

By 9/7/2017 miners generated 21,335,541.72 ETH as Mining Block Reward and 1,181,201.88 Mining Uncle Reward. For securing the network, they received a total of 22,516,743.6 ETH. Using the 9/7/2017 price of $303.86, security of the network costs 22516743.6 ETH * $303.86 = $6,841,937,710.296.

There are 56,048,767 transactions on the network. Security of a transaction in the main Ethereum network costs are about $122.07 at the current rate.

Solution

In Oracles Network the issuance rate is 3.5% with future disinflation. There is no Mining Uncle Reward in the network, because consensus is not based on Proof of Work.

Validators with known identity

Each validator of the network will prove his/her identity using “proof of identity” DApps. Each block will be attributed with the identity of a validator. If a miner breaks the rules of the open network, e.g. will not accept a transaction to a specific address, participants of the network will have legal instruments to resolve that problem.

Fast network

Validators in Oracles Network create blocks every five seconds. This rate is tested on Kovan testnet and usable in the long-term. A faster network allows for building new types of applications where response rate from the distributed consensus is important.

Legally recognizable hard forks

Hard fork is a change of the software. After applying this software, old clients will not be able to work on the new network.

All validators on the network are residents of the U.S. Therefore, they are all located in the same legal system. Hard fork decisions will be signed as legal documents and will be recognizable in a court system. This will bring protections to participants of the network and will open new possibilities to decide how to deal with ongoing changes.

Model for experiments

The network is built to iterate fast. In the future many open and independent networks based on Ethereum protocol will operate and have interface for interoperability.

Security Risks

Key compromise

During the initial ceremony, validators will be required to replace their initial keys with a set of three keys. Mining keys are located on a mining node. If a node is compromised, validators will create a ballot using Governance DApp and propose replacement of the mining key. If a voting key is compromised, a validator will ask another validator to create a ballot to replace his/her voting key. If a payout key is compromised, a validator will create a ballot to replace his/her payout key. Because payout is not required, a validator can specify a new payout key on a mining node without proposing ballots.

Censoring signer

Censoring signing is an attack vector in which a signer or a group of signers attempts to censor out blocks that vote on removing them from the validators list. To work around this, we restrict the allowed minting frequency of signers to 1 out of N. This ensures that malicious signers need to control at least 51% of signing accounts, at which case it’s game over anyway.

Regulatory risks

All validators are required to have an active notary commission. Doing block validation under the name of a notary public may be considered as false advertising and a regulator may revoke the notary commission from the validator. The network will mitigate the risk by providing additional identity checks for a validator. Eventually, those unbiased checks will replace the need for a validator to have an active notary commission.

Collusion of validators

Validators may become an affiliated group even though we require them to be independent. Before distribution of initial keys, the master of ceremony will require validators to sign a non-affiliation agreement between them and the network. All validators are in the same jurisdiction, where the general public may enforce that agreement.

Deployment

We provide a deployment script for cloud installation of mining, boot, and general purpose nodes.

For a validator, setting up a node is a one-button solution. For a mining node, a validator will provide

Mining Address. The address of the mining key received at the initial ceremony.

Mining Keyfile. File with the private key of the mining key.

Mining Keypass. The password to unlock the private key of the mining key.

Admin Username. Username of admin user of the virtual machine, e.g. root.

Admin SSH public key. Content of admin’s SSH public key. We do not allow use of passwords for the VMs.

Netstats Server. Network statistics, e.g. number of Active Nodes, Last Block, Avg Block Time, Best Block, Gas Spending, Gas Limit, List of validators with parameters.

Netstats Secret. Password to the netstat server.

Decentralized apps

The term decentralized app or DApp stands for an application which works with a smart contract and can be deployed on any host and redeployed in case of attack or censorship without any harm to its functions. Oracles Network provides sets of supported DApps for identity verification, governance, and network administration.

Initial ceremony DApp

During the initial ceremony a master of ceremony creates a set of keys for each validator. He/She distributes them to validators one by one. Before each distribution of keys, he/she sends a transaction to a smart contract with a list of validators. That smart contract is used by consensus algorithm to determine if a validator has rights to participate in consensus and create blocks. The validator’s smart contracts are used by other DApps, e.g. Governance DApp and Payout DApp.

A validator generates three keys in the Initial Ceremony DApp:

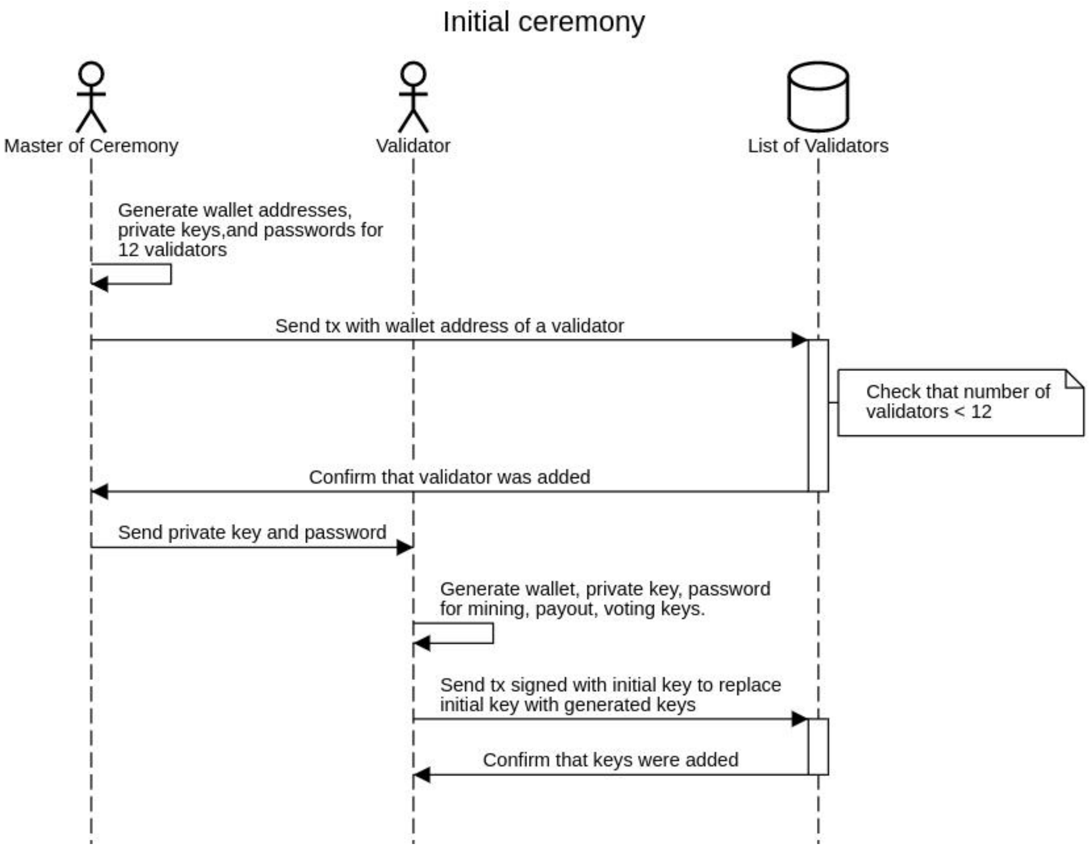
mining key, required to participate in consensus and create blocks.

voting key, required to create ballots and vote on ballots.

payout key, not required. Used in Payout DApp to send daily mined coins from the mining key to the payout key. If a mining node should be compromised, an attacker will get daily earnings or less.

All keys are generated on the client side and not transmitted over the Internet without a validator’s permission and willingness. When keys are generated, the validator stores them on secure local storage, e.g. saves them to a hardware wallet and the password to a password manager. The validator signs a transaction to the validator’s contract with the initial key, provided by the master of ceremony.

Initial ceremony is a required procedure to start a new network based on Oracles Network’s ideas of independent validators.


Initial ceremony
Master of Ceremony
Validator
List of Validators
Generate wallet addresses, private keys,and passwords for 12 validators
Send tx with wallet address of a validator
Check that number of validators < 12
Confirm that validator was added
Send private key and password
Generate wallet, private key, password for mining, payout, voting keys.
Send tx signed with initial key to replace initial key with generated keys
Confirm that keys were added

Proof of Identity DApps

In Oracles Network, identity of individual validators plays a major role for selected consensus. We propose a requirement for the initial validators to have an active notary commission within one of the states of the United States, although notary commission is not an object a validator can control solely. A regulator, e.g. a Secretary of State, may revoke notarial license from a validator, and we propose additional checks of identity, performed in a decentralized way.

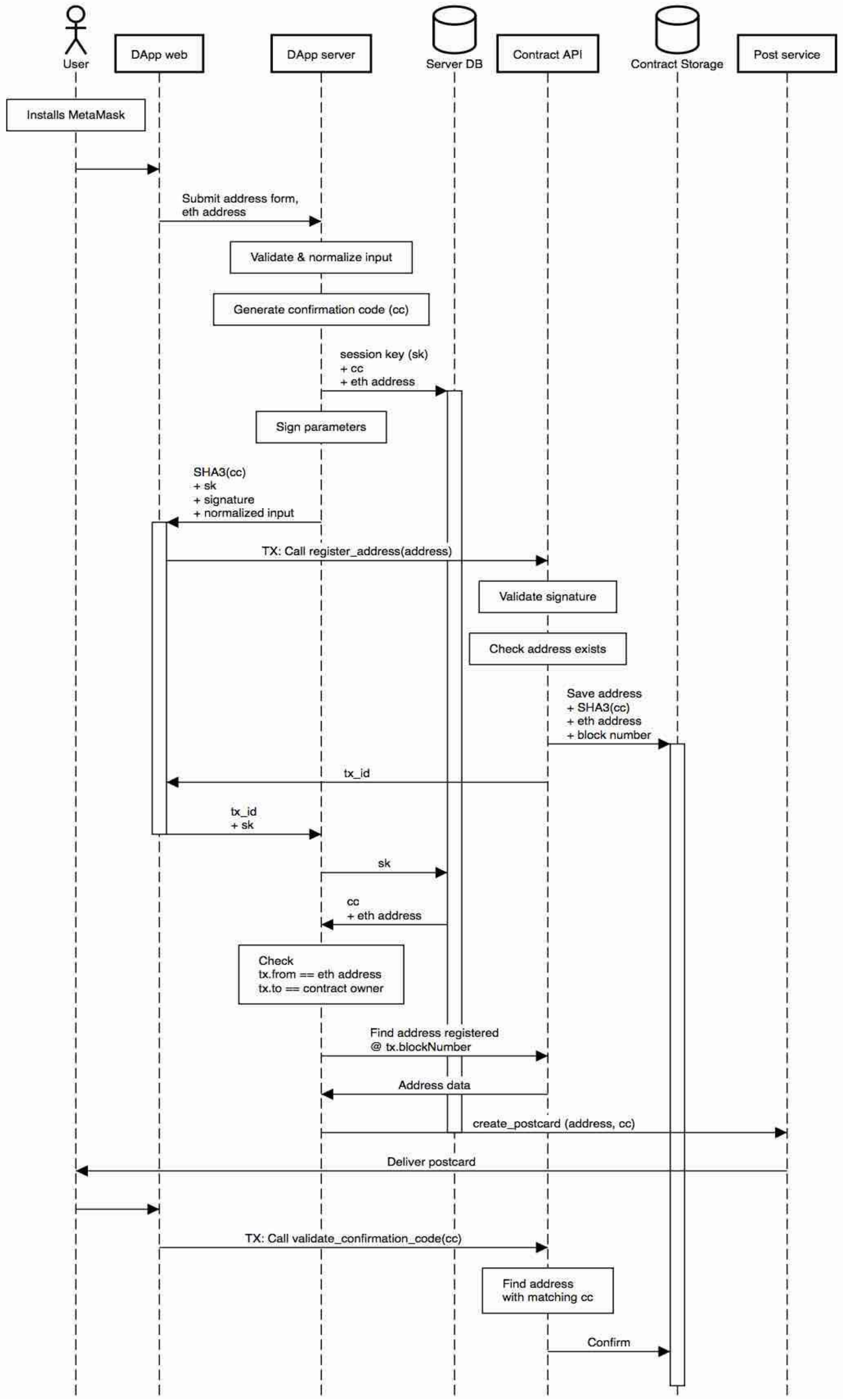
Proof of Identity DApps is a series of decentralized applications focused on connecting a user's identity to his/her wallet. Applications can be run on any Ethereum-compatible network.

Proof of Physical Address (PoPA) DApp

Using Proof of Physical Address, a user can confirm his/her physical address. It can be used to prove residency.

Typical workflow for Identity DApps on PoPA example

Proof of Physical Address



User fills out a form in DApp and submits it to the server.

Server consists of a web app and a parity node connected to the blockchain. The node is run under the ethereum account that was used to deploy the PoPA contract (contract’s owner), and this account needs to be unlocked. It shouldn’t have any ether on it though, as it doesn’t send any more transactions.

Server validates and normalizes the user’s input: removes trailing spaces, converts letters to lower case.

Then it generates a random confirmation code (alphanumeric sequence) and computes its SHA-3 (strictly speaking, keccak256[8]) hash. Also, it generates a random session code (see below), that it stores in memory/database along with the user’s eth address and plain text confirmation code.

Then the server combines input data, namely str2sign = (user's eth address + user's name + all parts of physical address + confirmation code's hash) into a string that is hashed and signed with the owner's private key. (This is why the owner's account needs to be unlocked. In the next release of web3js it will probably become possible to sign using a private key without unlocking.)

Signature, the confirmation code’s hash, the user’s normalized input, and the session code are sent back to the client. User then confirms the transaction in MetaMask and invokes the contract’s method. The contract combines input data in the same order as the server did, hashes it, and then uses the built-in function ecrecover to validate that the signature belongs to the owner. If it doesn’t, the contract rejects the transaction, otherwise it adds some metadata, most importantly the current block’s number, and saves it in the blockchain.

When the transaction is mined, tx_id is returned to the client and then via the client to the server, along with the session code. Server queries memory by the session code and validates the user’s eth address. Then it fetches the transaction from the blockchain by tx_id. It verifies that tx.to is equal to owner and tx.from is equal to the user’s eth address. Then, using tx.blockNumber the server uses the contract’s method to find the physical address added at that blockNumber. User should be limited to registering at most one address per eth block. Since block generation time is less than a minute, it shouldn’t be too restrictive on the user.

Having fetched the address from the contract, the server calls postoffice’s api (lob.com) to create a postcard. Server uses the session code to get plain text confirmation code from memory and print it on the postcard. Then the server removes this session code from memory to prevent reuse.

When the postcard arrives, the user enters the confirmation code in DApp that invokes the contract’s method directly, without server interaction, as there doesn’t seem to be any need in signing with the owner's private key. Contract computes the confirmation code’s hash and loops over the user’s addresses to find the matching one.

Possible cheating:

user can generate his/her own confirmation code, compute all hashes and submit it to the contract, and then confirm it

This can’t be done because the user doesn’t know the owner's private key and therefore can’t compute a valid signature.

user can reuse someone else’s confirmation code, or his/her own confirmation code from one of the previously confirmed addresses

This is prevented by hashing all essential pieces of data together before signing (user’s eth address, full physical address, confirmation code) and by checking the address for duplicates in the contract.

user can submit the form, but doesn’t sign the transaction

For this reason the postcard is sent after the address is added to the blockchain and tx_id is presented to the server.

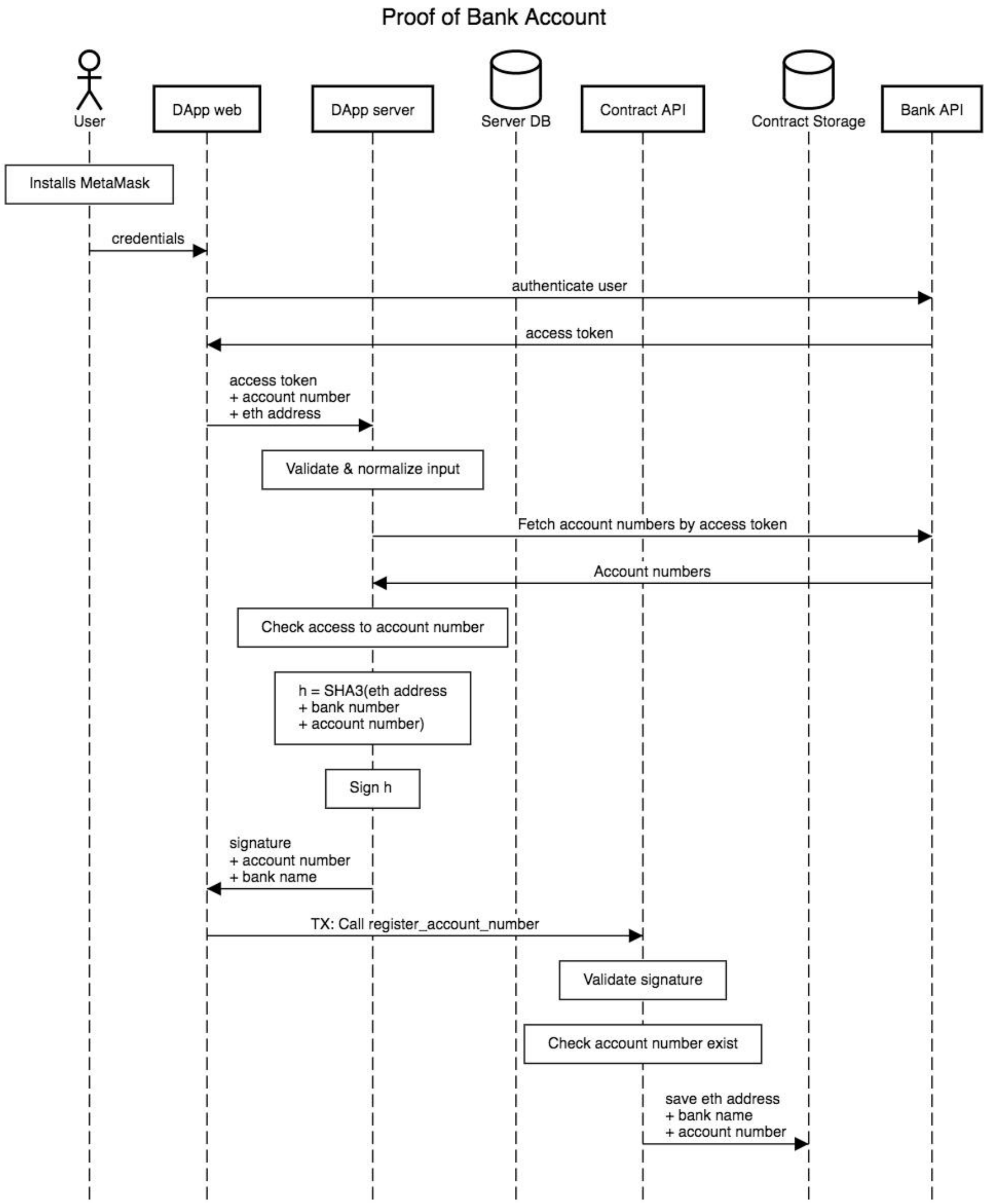
user can submit the form and sign the transaction, but sends another address to the server to send postcard to

After the first transaction is mined, the server sees for itself what address was added and fetches it from the contract instead of trusting the client. Session code is then used to retrieve the corresponding confirmation code. To simpify things, we can limit the user to only submitting a single address per block. In this case, the contract just needs to find the first record with matching creation_block.

user can resubmit the same tx_id to the server multiple times

This is prevented by removing the session code from memory after the first postcard is sent.

Proof of Bank Account DApp



In contrast to other identity DApps, PoBA is (from the contract's point of view) a one-step verification process.

DApp client and server are integrated with bank accounting API service (plaid.com).

Client side uses the service's widget (Plaid Link) to authenticate the user, and as a result of successful authentication, access_token is returned from Plaid to the client. User then fills out a form with his/her bank account number and submits it to the server alongside Plaid's access token.

Server consists of a web app and a parity node connected to the blockchain. The node is run under the ethereum account that was used to deploy the PoP contract (contract’s owner). This account needs to be unlocked.

Server validates and normalizes the user’s account number by removing trailing spaces. Then the server fetches the bank account numbers from Plaid using access_token. It checks that the account number submitted by the user is present in the list returned by Plaid.

Server then combines user's eth address + bank's name + account number into a single string and hashes it with SHA-3 function. The hash is then signed with owner's private key (this is why owner account needs to be unlocked).

Signature, normalized account number, and bank name are returned to the client. User then signs the transaction in MetaMask and invokes the contract’s method.

Contract checks that the account number for this bank for this eth address doesn’t already exist. If it does, the contract rejects the transaction. Otherwise, it combines parameters in the same order as the server did and computes sha3 hash of them. Then it uses the built-in ecrecover function to validate that the signature belongs to the owner. If it doesn’t, the contract rejects the transaction, otherwise, it saves the information to the blockchain.

Possible cheating:

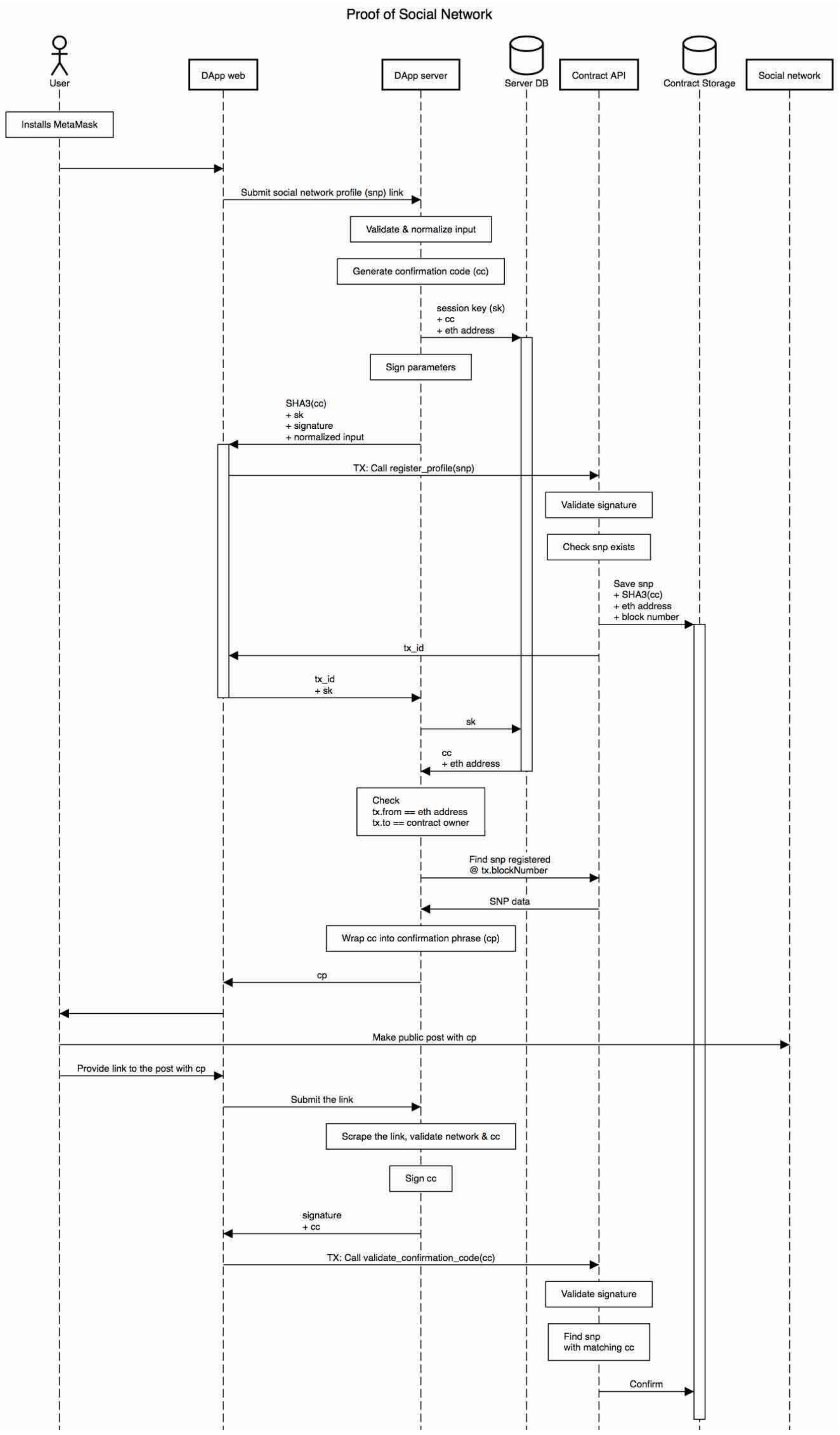
user can generate his/her own confirmation code, compute all hashes, and submit it to the contract, and then confirm it

This can’t be done because the user doesn’t know the owner's private key and hence can’t compute a valid signature.

user can use someone else’s access_token returned by Plaid and thus verify the account he/she has no real access to

This is equivalent to either hacking someone else’s computer or the account’s owner deliberately providing the user with his/her access_token. Since all communications with Plaid are via HTTPS protocol, there is no way for the user to intercept access_token sent to someone else.

Proof of Social Network DApp


Proof of Social Network
User
DApp web
DApp server
Server DB
Contract API
Contract Storage
Social network
Installs MetaMask
Submit social network profile (snp) link
Validate & normalize input
Generate confirmation code (cc)
session key (sk)
+ cc
+ eth address
Sign parameters
SHA3(cc)
+ sk
+ signature
+ normalized input
TX: Call register_profile(snp)
Validate signature
Check snp exists
Save snp
+ SHA3(cc)
+ eth address
+ block number
tx_id
tx_id
+ sk
sk
cc
+ eth address
Check
tx.from == eth address
tx.to == contract owner
Find snp registered
@ tx.blockNumber
SNP data
Wrap cc into confirmation phrase (cp)
cp
Make public post with cp
Provide link to the post with cp
Submit the link
Scrape the link, validate network & cc
Sign cc
signature
+ cc
TX: Call validate_confirmation_code(cc)
Validate signature
Find snp
with matching cc
Confirm

User fills out a form in DApp providing the link to his/her social network profile and submits it to the server.

Server consists of a web app and a parity node connected to the blockchain. The node is run under the ethereum account that was used to deploy the PoSN contract (contract's owner). This account needs to be unlocked.

Server validates and normalizes the user's profile link: removes trailing spaces, converts protocol to HTTPS if applicable, domain name to lowercase, and removes extra URL parameters.

Then it generates a random confirmation code (alphanumeric sequence) and computes its SHA-3 (strictly speaking, keccak256[8:1]) hash. Also, it generates a random session code (see below), that it stores in memory/database along with the user's eth address and plain text confirmation code.

Then server combines input data, namely str2sign = (user's eth address + user's profile link + confirmation code's hash) into a string that is hashed and signed with owner's private key (this is why owner's account needs to be unlocked).

Signature, the confirmation code's hash, the user's normalized profile link, and the session code are sent back to the client. User then confirms the transaction in MetaMask and invokes the contract's method. The contract combines input data in the same order as the server did, hashes it, and then uses the built-in function ecrecover to validate that the signature belongs to the owner. If it doesn't, the contract rejects the transaction, otherwise it adds some metadata, most importantly the current block's number, and saves it in the blockchain.

When the transaction is mined, tx_id is returned to the client and then via the client to the server along with the session code. Server queries memory by the session code and validates the user's eth address. Then it fetches the transaction from the blockchain by tx_id. It verifies that tx.to is equal to owner and tx.from is equal to the user's eth address. Then, using tx.blockNumber the server uses the contract's method to find the profile link added at that blockNumber. User should be limited to registering at most one profile link per eth block.

Then the server uses the session code to get plain text confirmation code from memory and enclose it into a predefined meaningful text, e.g.:

My oracles identity confirmation code is

(As a side note, it'd be funny if the confirmation code was a random quote from a random book.) Then the server sends this confirmation phrase back to the client and removes the session code from memory to prevent reuse.

User must create a publicly available post where the confirmation phrase would appear alone, on a separate line (there may be other text in this post, on other lines).

Then the user returns to the DApp and submits the link to his/her post. Server needs to scrape this post, find a line starting with the predefined text and extract the confirmation code from it. Server then calculates SHA-3 of the confirmation code and signs it with the owner's private key. Hash of the confirmation code and signature is returned to the client.

User then confirms the transaction in MetaMask, which invokes the contract's method. Contract first of all uses ecrecover to verify that the signature belongs to the owner. If it doesn't, the contract rejects the transaction, otherwise it computes the confirmation code's hash and loops through the user's profile links to find a matching one. Server must also double-check that post is on the same network that is in the profile link in the contract's data.

Possible cheating:

user can generate his/her own confirmation code, compute all hashes, and submit it to the contract, and then confirm it

This can't be done because the user doesn't know the owner's private key and therefore can't compute a valid signature.

user can reuse someone else's confirmation code, or his/her own confirmation code from one of the previously confirmed profile links

This is prevented by hashing all essential pieces of data together before signing (user's eth address, profile link, confirmation code) and by checking the profile link for duplicates in the contract.

user can submit the form, but doesn't sign the transaction

For this reason confirmation phrase is sent to the client after the profile link is added to the blockchain and tx_id presented to the server.

since user knows confirmation code right from the start (cf. PoPA DApp), he/she can avoid posting the confirmation phrase and just call the contract's method directly

Link to the post should be presented to the server, which scrapes it, extracts the confirmation code, and then signs it with the owner's private key.

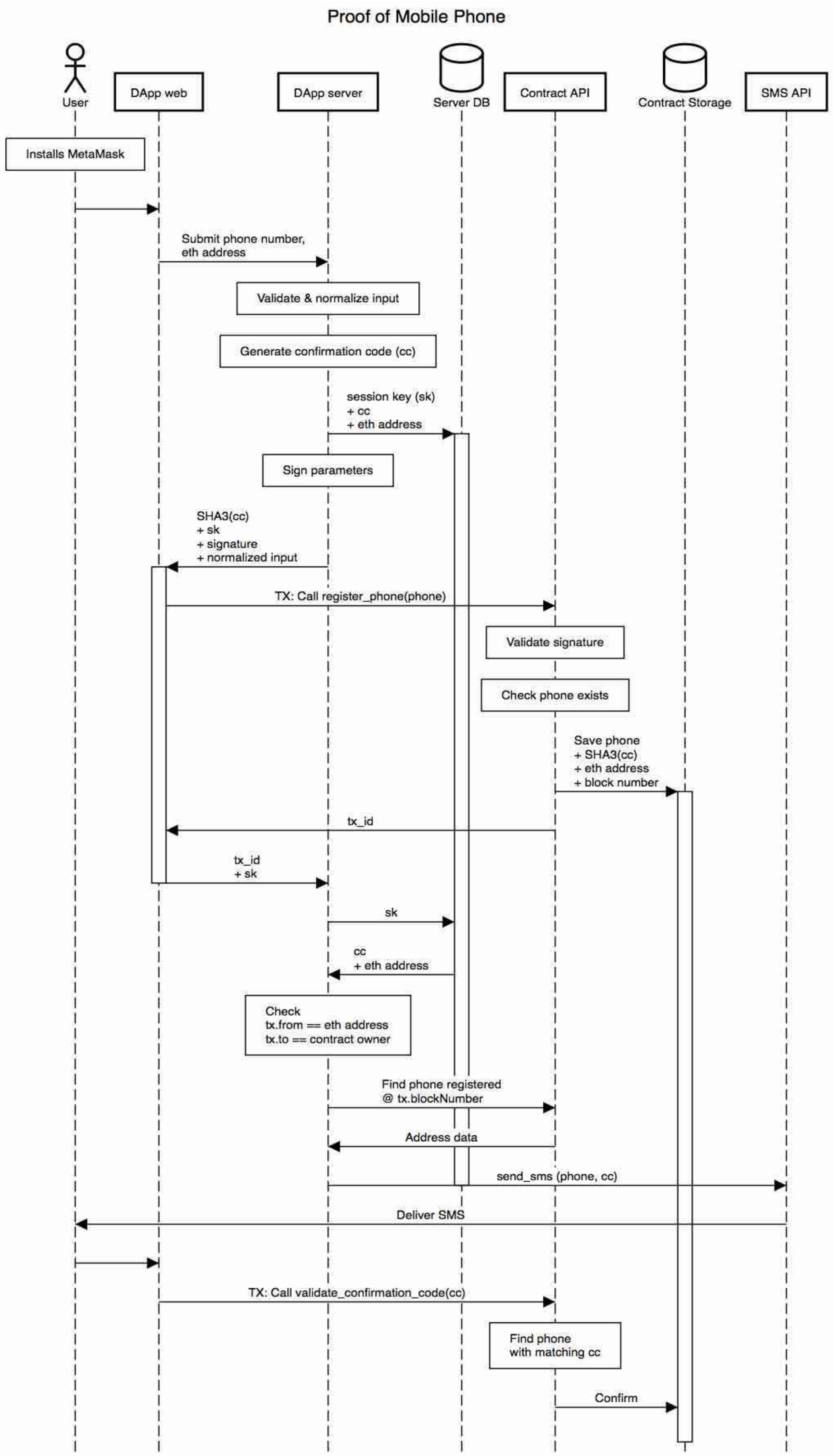
user can post the confirmation phrase on some other social network or website

Server should double-check that the post is on the same network as the profile link from the contract’s data.

user can resubmit the same tx_id to the server multiple times

This is prevented by removing the session code from memory after the first postcard is sent.

Proof of Phone Number DApp


Proof of Mobile Phone
User
DApp web
DApp server
Server DB
Contract API
Contract Storage
SMS API
Installs MetaMask
Submit phone number,
eth address
Validate & normalize input
Generate confirmation code (cc)
session key (sk)
+ cc
+ eth address
Sign parameters
SHA3(cc)
+ sk
+ signature
+ normalized input
TX: Call register_phone(phone)
Validate signature
Check phone exists
Save phone
+ SHA3(cc)
+ eth address
+ block number
tx_id
tx_id
+ sk
sk
cc
+ eth address
Check
tx.from == eth address
tx.to == contract owner
Find phone registered
@ tx.blockNumber
Address data
send_sms (phone, cc)
Deliver SMS
TX: Call validate_confirmation_code(cc)
Find phone
with matching cc
Confirm

User fills out a form in DApp providing his/her phone number and submits it to the server.

Server consists of a web app and a parity node connected to the blockchain. The node is run under the ethereum account that was used to deploy the PoP contract (contract's owner). This account needs to be unlocked.

Server validates and normalizes the user's phone number: removes trailing spaces, converts it to international format.

Then it generates a random confirmation code (alphanumeric sequence) and computes its SHA-3 (strictly speaking, keccak256[8:2]) hash. Also, it generates a random session code (see below) that it stores in memory/database along with the user's eth address and plain text confirmation code.

Then the server combines input data, namely str2sign = (user's eth address + user's phone number + confirmation code's hash) into a string that is hashed and signed with the owner's private key (this is why owner's account needs to be unlocked).

Signature, the confirmation code's hash, the user's normalized phone number, and the session code are sent back to the client. User then confirms the transaction in MetaMask and invokes the contract's method. The contract combines input data in the same order as the server did, hashes it, and then uses the built-in function ecrecover to validate that the signature belongs to the owner. If it doesn't, the contract rejects the transaction, otherwise it adds some metadata, most importantly the current block's number, and saves it in the blockchain.

When the transaction is mined, tx_id is returned to the client and then via the client to the server along with session code. Server queries memory by the session code and validates the user's eth address. Then it fetches the transaction from the blockchain by tx_id. It verifies that tx.to is equal to owner and tx.from is equal to the user's eth address. Then, using tx.blockNumber the server uses the contract's method to find the phone number added at that blockNumber. User should be limited to registering at most one phone number per eth block.

Then the server uses the session code to get plain text confirmation code from memory and send it via SMS service (twilio.com) to the user's phone number. Then the server removes the session code from memory to prevent reuse.

Having received SMS with verification code, the user returns to the DApp and confirms the transaction in MetaMask, which sends confirmation code to the contract's method directly, without calling the server. There doesn't seem to be any need for signing this transaction with the owner's private key. Contract computes the confirmation code’s hash and loops over the user’s phone numbers to find a matching one.

Possible cheating:

user can generate his/her own confirmation code, compute all hashes and submit it to the contract, and then confirm it

This can’t be done because the user doesn’t know the owner's private key and therefore can’t compute a valid signature.

user can reuse someone else’s confirmation code, or his/her own confirmation code from one of the previously confirmed phone numbers

This is prevented by hashing all essential pieces of data together before signing (user’s eth address, phone number, confirmation code) and by checking the phone number for duplicates in the contract.

user can submit the form, but doesn’t sign the transaction

For this reason, SMS is sent after the phone number is added to the blockchain and tx_id is presented to the server.

user can submit the form and sign the transaction, but sends another phone number to the server to send SMS to

After the first transaction is mined, the server sees for itself what phone number was added and fetches it from the contract instead of trusting the client. Session code is then used to retrieve the corresponding confirmation code. To simpify things, we can limit the user to only submitting a single phone number per block. In this case the contract just needs to find the first record with matching creation_block.

user can resubmit the same tx_id to the server multiple times

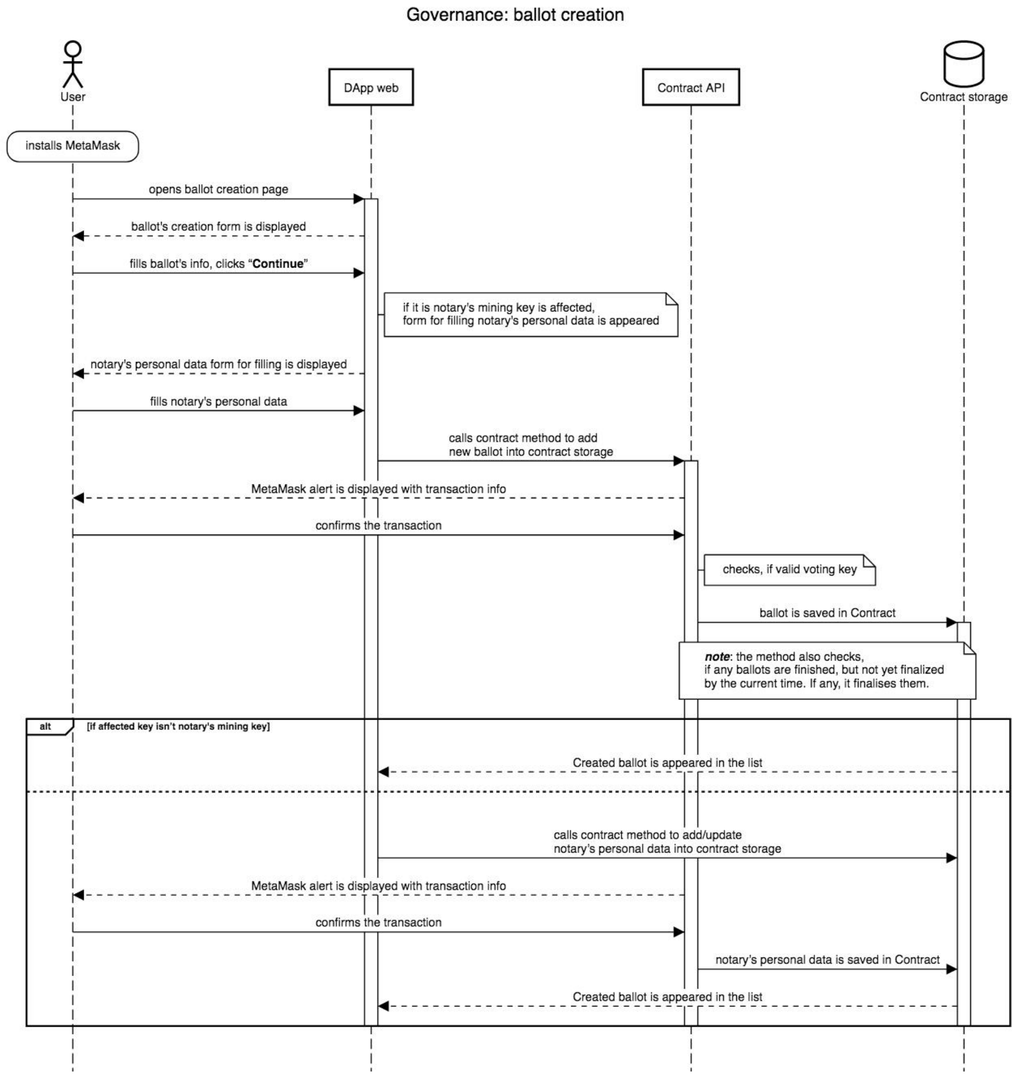
This is prevented by removing the session code from memory after the first postcard was sent.

Governance DApp

This client-side DApp provides the list of existing ballots with the ability of filtering by active, unanswered, and expired ballots, and gives the opportunity to create new ballots and to vote for or against notaries.

The governance is available only with a valid voting key that should be selected in the MetaMask Google Chrome plugin.

Creating a new ballot


Governance: ballot creation
User
DApp web
Contract API
Contract storage
installs MetaMask
opens ballot creation page
ballot's creation form is displayed
fills ballot's info, clicks "Continue"
if it is notary's mining key is affected,
form for filling notary's personal data is appeared
notary's personal data form for filling is displayed
fills notary's personal data
calls contract method to add
new ballot into contract storage
MetaMask alert is displayed with transaction info
confirms the transaction
checks, if valid voting key
ballot is saved in Contract
note: the method also checks,
if any ballots are finished, but not yet finalized
by the current time. If any, it finalises them.
alt
[if affected key isn't notary's mining key]
Created ballot is appeared in the list
calls contract method to add/update
notary's personal data into contract storage
MetaMask alert is displayed with transaction info
confirms the transaction
notary's personal data is saved in Contract
Created ballot is appeared in the list

Valid notary of the Oracles Network fills out a form in DApp providing:

mining key - mining key of a new or existing notary, which will be voted on

affected key type - key type (mining, payout, or voting key) of a new or existing notary, which will be voted on

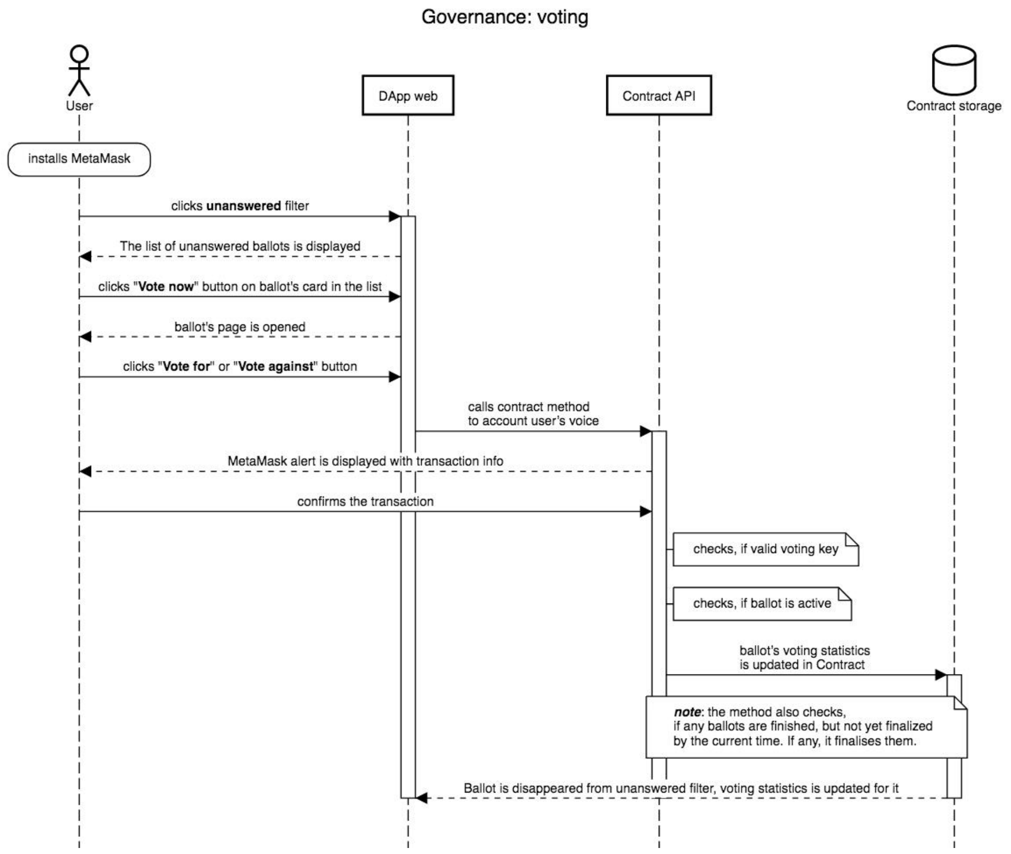
memo - brief information about notary, which will be voted on

action - add affected key to the network or remove it from the network

If the affected key type is mining key, the user will be asked to provide personal data of the notary (owner of this mining key) such as full name, physical address, U.S. state name, zip code, notary license ID, and notary license expiration date.

At the final step, one transaction to create a new ballot in Oracles contract will be pushed to the blockchain to add a new ballot after the user presses “Continue” button. It should be noted, that in case of a mining key, it will be two consistent transactions: to add personal data of a notary and a new ballot to contract. User will see MetaMask popups equal to the number of transactions. After the confirmation and successful mining of the transaction by existing validators, the user will see the created ballot in the list and be able to vote on it.

Voting on a ballot


Governance: voting
User
DApp web
Contract API
Contract storage
installs MetaMask
clicks **unanswered** filter
The list of unanswered ballots is displayed
clicks "**Vote now**" button on ballot's card in the list
ballot's page is opened
clicks "**Vote for**" or "**Vote against**" button
calls contract method
to account user's voice
MetaMask alert is displayed with transaction info
confirms the transaction
checks, if valid voting key
checks, if ballot is active
ballot's voting statistics
is updated in Contract
note: the method also checks,
if any ballots are finished, but not yet finalized
by the current time. If any, it finalises them.
Ballot is disappeared from unanswered filter, voting statistics is updated for it

The user can see all his/her unanswered ballots by clicking on the self-titled button on the filtering panel.

The list of unanswered ballots will be displayed after filtering, and the “Vote now” button will be enabled for any item in the list. After clicking on this button, a preview of the ballot will be opened with the notary’s personal data, statistics of voting, and time to ballot’s ending. Two buttons will be enabled here: “Vote for” and “Vote against”. After clicking on any of them, the transaction to account the user’s voice will be generated, and a MetaMask popup will be shown with the transaction information. After the confirmation and successful mining of the transaction by existing validators, the user will see the updated statistics with his/her voice, and the ballot will disappear from the unanswered ballots filter.

Possible cheating:

user can create ballot or vote with his/her own dummy key

It is impossible, because only a valid payout key can govern. It is checked on the contract side.

same user can vote for or against a notary twice

It is restricted at the contract side.

user can vote after ballot's time has ended

It is restricted at the contract side.

notary with counterfeit license can become a member of the network

It is impossible in practice, because any of the voters can check public information about every notary before voting.

user can govern other notaries alone

It is impossible, because the minimal amount of voices for a ballot is equal to 3.

user can manage the time of a ballot

Duration of a ballot is constant and equal to 48 hours. It is set in the contract.

Summary

We believe that such networks with Proof of Authority consensus algorithms will be a trend in public blockchains in the coming years. On-demand systems with trusted validators will play a major role in creating specialized open networks based on Ethereum's protocol. Our goal is to be a model for the generation of networks connected by inter-ledger protocols, such as Polkadot[9]and Cosmos.

Acknowledgments

We would like to express our gratitude to our mentors, advisors and to the many people in the Ethereum community that have been so welcoming and generous with their knowledge.

We would also like to thank the organizers and community members that we’ve met at the Silicon Valley and SF Ethereum Meetups including Roman Storm, Joseph Chow, Martin Koppelmann, Grant Hummer, Tom Ding, Chris Peel, Jeff Flowers, and many others.