microraiden Documentation

Release 0.2.6

brainbot labs Est.

Aug 22, 2018

Contents

CHAPTER 1

What is µRaiden?

µRaiden (read: Micro Raiden) is a payment channel framework for frequent, fast and free ERC20 token based micropayments between two parties. It comes as a set of open source libraries, documentation, and code examples for multiple use cases, ready to implement on the Ethereum mainnet. Whereas its big brother, the Raiden Network, aims to allow for multihop transfers via a network of bidirectional payment channels, µRaiden already enables to make micropayments through unidirectional payment channels.

[1] All robot icons made by Freepic from http://www.flaticon.com.

[3] All other icons from http://icomoon.io IcoMoon Icon Pack Free, licensed under a Creative Commons Attribution 4.0 International License

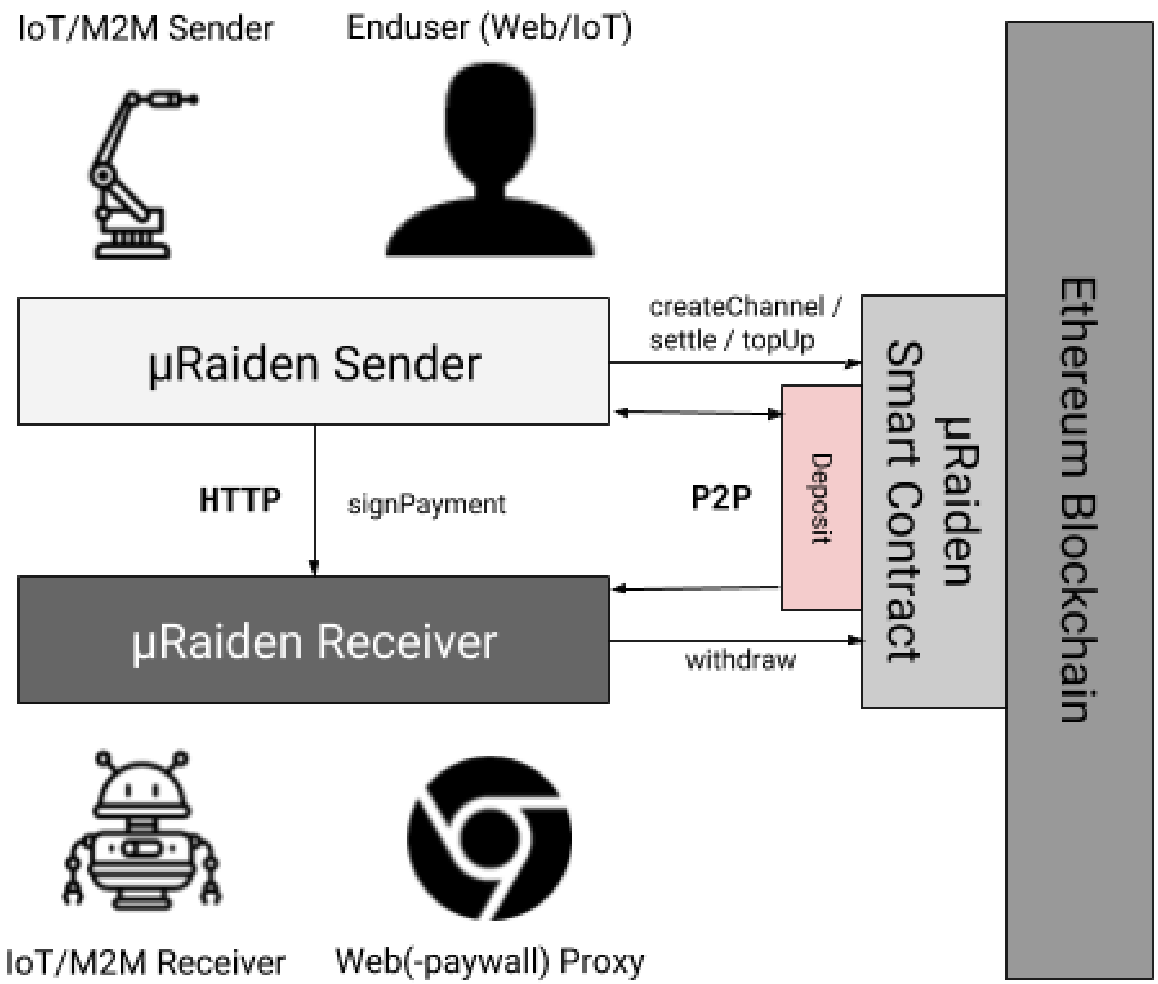


Fig. 1: Schematic overview of an exemplaric µRaiden application[13]

CHAPTER 2

Try out the demos

We have deployed some demo applications that make extensive use of µRaiden in your browser. Although you need some testnet-Ether and MetaMask, it's a very easy starting point to try out µRaiden. Just follow the instructions on https://demo.micro.raiden.network/

CHAPTER 3

Next steps

If you want to start playing with µRaiden, a good starting point is to check out the *Tutorials* section. Best you start with the *developer setup* and continue with our *Blockchain Tutorial*.

CHAPTER 4

Documentation Content

4.1 Introduction

4.1.1 Comparison: Raiden Network

µRaiden is not part of the Raiden Network. However, it was built using the same state channel idea and implements it in a less general fashion. It focuses on the concrete application of charging per-use of APIs, digital content and utilities via micropayments in Ethereum based ERC20 tokens.

The main differences between the Raiden Network and µRaiden are:

- **µRaiden** is a many-to-one unidirectional payment channel protocol.

A payment channel in the **Raiden Network** is based on the same principles as µRaiden, but is laid out bidirectionally, so that the roles of sender and receiver are mutable. Additionally it uses a special cryptographic protocol to connect the owners of those singular payment channels to form an interconnected network of channels.

This allows participants of the Raiden Network to efficiently send transfers without being forced to pay for opening new channels with people who are already in the network - it is a many-to-many payment solution.

- **µRaiden**'s off-chain transactions do not cost anything as they are only exchanged between sender and receiver, because they don't use intermediary channels.

Apart from the initial cost of opening up a channel, a µRaiden transaction doesn't cost anything, because to deliver the payment itself is as easy as putting some additional data in a http-request.

To be able to use an existing channel in an interconnected network of channels, the Raiden Network requires an additional, sophisticated application transport layer. The forwarding of payments from sender to receiver through the network is based on incentivizing intermediary users to lend their resources in a secure and automated way.

4.1.2 Sender / Receiver

Since µRaiden enables easy micropayments from one party to another, the application is structured in 2 logically separated parts:

- the *Sender* or *Client* side of a payment
- the *Receiver* or *Proxy-Server* side of a payment

The *Sender* is the one who initially deposits Ether in the µRaiden payment channel. From this point on he signs so called *balance proofs* with his private key. A balance-proof functions as a valid micropayment, once the *Receiver* gets hold of it and keeps it on his disk.

The µRaiden application has different implementations for different scenarios for the *Sender* side:

- a JavaScript client that runs in the Senders browser whenever the Sender visits the Receiver's webpage
- a Python client that runs on the Sender's machine and makes http requests to the Receiver's Proxy-Server instance

A typical use case for the **JavaScript** client would be a content provider, who wants to receive micropayments for accessing paywalled content. The content provider is the *Receiver* in this scenario and he would integrate µRaiden's *Proxy-Server* for example in his flask or Django backend. At the same time, the content provider would serve an implementation of µRaiden's JavaScript Client from his webpage. All that the consumer of the paywalled content now needs is an Ethereum account that is backed with some RDN and that is web3 accessible (for example with MetaMask). The JavaScript client will run in the consumer's browser and once it needs to sign a microtransaction the MetaMask plugin will pop up and ask for confirmation.

The **Python client** would get mainly used in Machine-to-Machine (M2M) applications or more customized applications without the use of a browser. In this scenario, the *Sender* has to actively install the *client application* and connect it to his standard blockchain-interface (like geth or parity).The client will then send out http-requests to a known *Receiver* that is running a **Proxy-Server** application. Price information on the requested resource will be sent from the *Receiver* to the *Sender* in a custom http-Header. Vice versa, once the *Sender* has processed his business-logic (like evaluating the price), he will repeat the http-request with a matching *balance proof* embedded in the custom *http-Header*. This balance proof signature represents the actual micropayment and should be followed up by the *Receiver* with the delivery of the requested resource.

Off-chain transactions

A visual description of the process can be found *here*.

The heart of the system lies in its sender -> receiver off-chain transactions. They offer a secure way to keep track of the last verified channel balance. The channel balance is calculated each time the sender pays for a resource. He is prompted to sign a so-called balance proof, i.e., a message that provably confirms the total amount of transferred tokens. This balance proof is then sent to the receiver's server. If the balance proof checks out after comparing it with the last received balance and verifying the sender's signature, the receiver replaces the old balance value with the new one.

4.1.3 Smart Contract

To be exact, there is a third party involved in µRaiden:

- the *Enforcing* or *Smart Contract* part

This is the part where the trustless nature of the Ethereum blockchain comes into play. The contract acts as the intermediary, that locks up the initial deposit from the *Sender* and enforces a possible payout of the funds based on the signed balance proofs, that the *Sender* sent out to the *Receiver* without the use of a blockchain.

Once the *Receiver* has a balance proof, it's easy for the *Receiver* to prove to the contract that the *Sender* owes him some tokens. With the balance proof, the contract now can reconstruct the public key of the *Sender* and knows with certainty that the Sender must have agreed to the updated balance.

[1] All robot icons made by Freepic from http://www.flaticon.com.

[3] All other icons from http://icomoon.io IcoMoon Icon Pack Free, licensed under a Creative Commons Attribution 4.0 International License

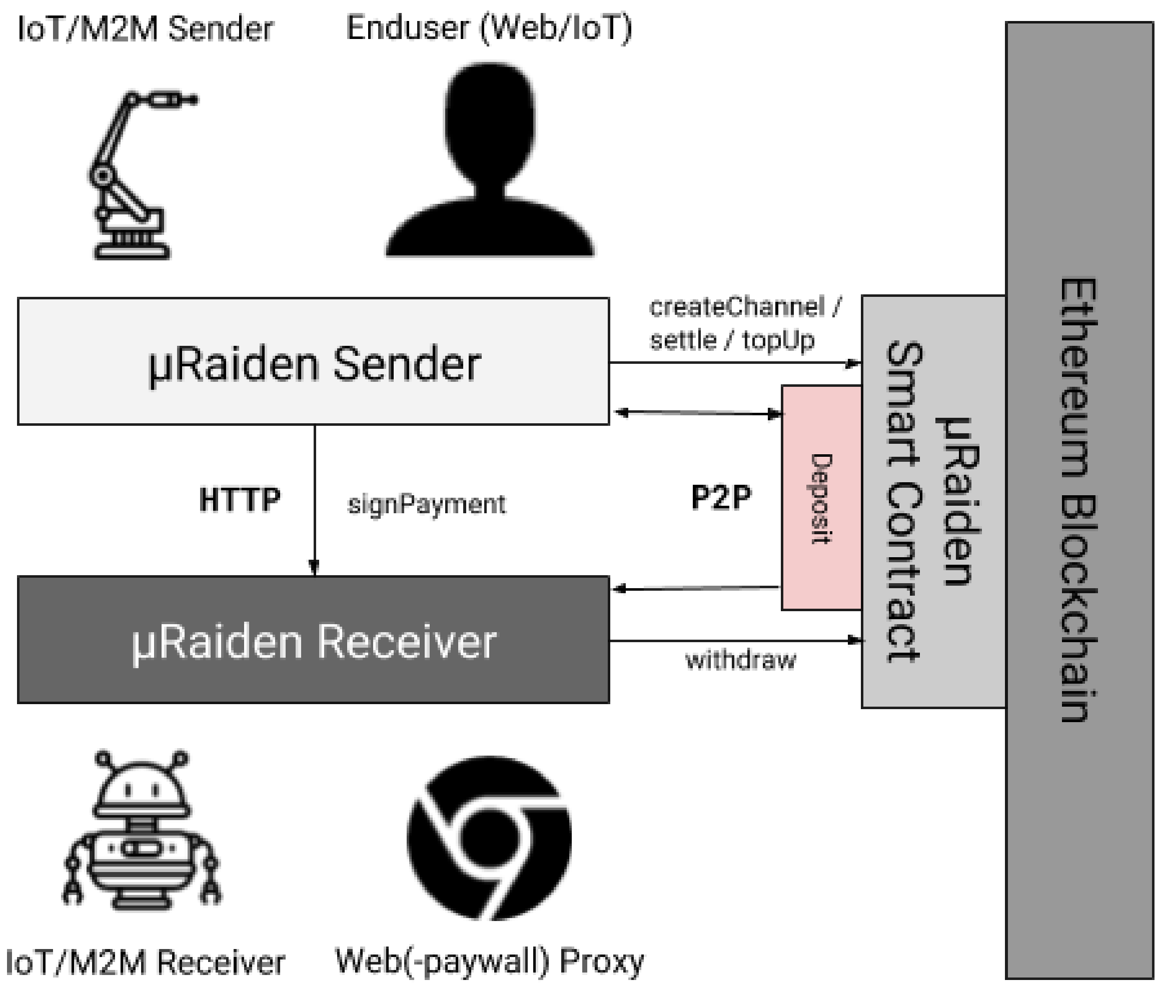


Fig. 1: Schematic overview of an exemplaric µRaiden application[13]

This means that there are only 2 transactions that have to happen on the blockchain:

- the initial *opening* of the channel with the prepaid amount the sender eventually wants to spend during the channel's lifetime
- the final *closing* of the channel, where the sender's initial deposit is paid out to the receiver and sender, based on the agreed on off-chain balances

If the channel runs low on funds before it is closed, the sender can increase the transferable amount of the channel with a *topup* transaction on-chain.

After a channel is closed, it can't be used anymore. If the business-relationship between the same sender and receiver should revive again, a new channel has to be opened.

µRaiden uses its own token for payments which is both ERC20 and ERC223 compliant.

Closing and settling channels

A visual description of the process can be found *here*.

When a sender wants to close a channel, a final balance proof is prepared and sent to the receiver for a closing signature. In the happy case, the receiver signs and sends the balance proof and his signature to the smart contract managing the channels. The channel is promptly closed and the receiver debt is settled. If there are surplus tokens left, they are returned to the sender.

In the case of an uncooperative receiver (that refuses to provide his closing signature), a sender can send his balance proof to the contract and trigger a challenge period. The channel is marked as closed, but the receiver can still close and settle the debt if he wants. If the challenge period has passed and the channel has not been closed, the sender can call the contract's settle method to quickly settle the debt and remove the channel from the contract's memory.

What happens if the sender attempts to cheat and sends a balance proof with a smaller balance? The receiver server will notice the error and automatically send a request to the channel manager contract during the challenge period to close the channel with the receiver's latest stored balance proof.

There are incentives for having a collaborative channel closing. On-chain transaction gas cost is significantly smaller when the receiver sends a single transaction with the last balance proof and his signature, to settle the debt. Also, gas cost is acceptable when the sender sends the balance proof along with the receiver's closing signature. Worst case scenario is the receiver closing the channel during the challenge period. Therefore, trustworthy sender-receiver relations are stimulated.

4.2 Tutorials

4.2.1 Development setup

Requirements

It is required that you have `pip`, `python` (version 3.5 or greater) and `git` installed.

- To install pip, see the official documentation
- To install Python download the latest version
- If you don't have git, download it here

Python environment setup

In general, it is recommended to use a virtual environment to separate your global python application from the environment (all the dependency-packages) µRaiden likes to run in:

```
python3 -m venv env
. env/bin/activate
```

A short check of the location of your python version should show the `./env/bin/python` binary.

```
which python
```

To switch back to your usual python executable, simply deactivate the *venv*:

```
deactivate
```

There are more sophisticated tools to keep track of your virtualenvs and python installations. For example, check out pyenv in combination with pyenv-virtualenv.

µRaiden installation for development

When you want to develop on the µRaiden codebase, it is best to install it in pip's editable mode. This way, you can edit the source code directly and never worry about reinstalling µRaiden - the linked application always reflects the changes you made. To install µRaiden for development, download the repository and run our install script with:

```
git clone git@github.com:raiden-network/microraiden.git
cd microraiden
make pip-install-dev
```

4.2.2 Blockchain setup

If you want to develop applications that function as the **Receiver** you have to connect to the Ethereum blockchain through one of the Ethereum node applications. There are others, but we focus on *geth*, the Go implementation.

Setup a Ropsten-synced Geth or Parity

The quick start requires that your *geth* client is synced to the Ropsten testnet. Geth should answer RPC calls on *http://localhost:8545* and have the APIs *eth*, *net*, *web3* and *personal* accessible.

Note: as of Geth version *1.8.0*, these parameters are required to start Geth in the correct mode:

```
geth --testnet --syncmode "fast" \
     --rpc --rpcapi eth,net,web3,personal \
     --cache=1024 \
     --rpcport 8545 --rpcaddr 127.0.0.1 \
     --rpccorsdomain "*"
```

If you are having trouble syncing with Geth, you may want to use Parity instead. Use the same parameter but with this code:

```
parity --chain ropsten --rpcport=8545
```

Funded Ropsten account with MetaMask

Note: You don't have to follow these steps if you have an account on Ropsten already preloaded with Ropsten Ether and know how to export the private key of the preloaded account.

After successful installation of MetaMask, just follow the steps mentioned in the screenshots to create a new account, get Ropsten Ether at a faucet and export the private key of this new, Ether preloaded account. We will use the private key for Sender applications in the tutorials. Your MetaMask account will represent the **Sender** of a microtransaction.

Buy TKN on Ropsten

To be able to use the echo client, you have to get the "TKN" Token configured for the RaidenMicroTransferChannels on Ropsten.

Directly with the TKN smart contract

On the main page (https://github.com/raiden-network/microraiden) the Token addresses are listed, for Ropsten it can be found here

With our demo app

The easiest way to get some TKN on the Ropsten-network is to use our JavaScript application that we host with our µRaiden live-demos.

Altough this is part of a specific demo-application, you can just use the upper part of the dialogue and forget about the lower half.

1. Choose the browser with your Ropsten-ETH loaded MetaMask account activated
2. visit e.g. the fortune cookie demo here
3. Click on *Buy* to exchange your RopstenETH (RETH) for TKN (in 50 TKN increments)

A dialogue will pop up in MetaMask that asks for your confirmation of the generated transaction.

4. Accept the transaction

To check whether the exchange of TKN was successful, you can add TKN as a custom token to MetaMask.

5. Under the *Tokens* tab, choose *Add token* and fill in the TKN address again:

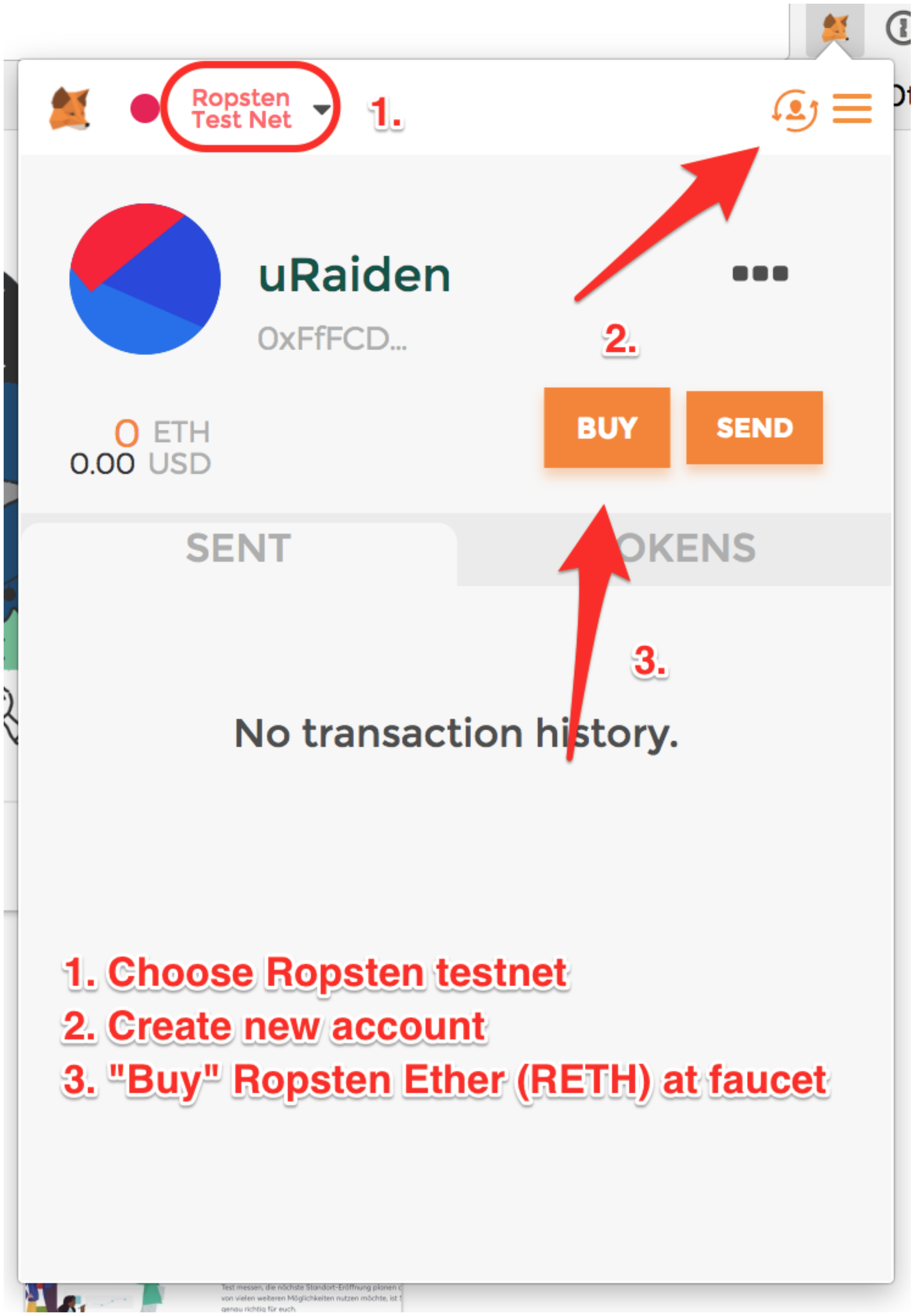
```
0xFF24D15afb9Eb080c089053be99881dd18aa1090
```

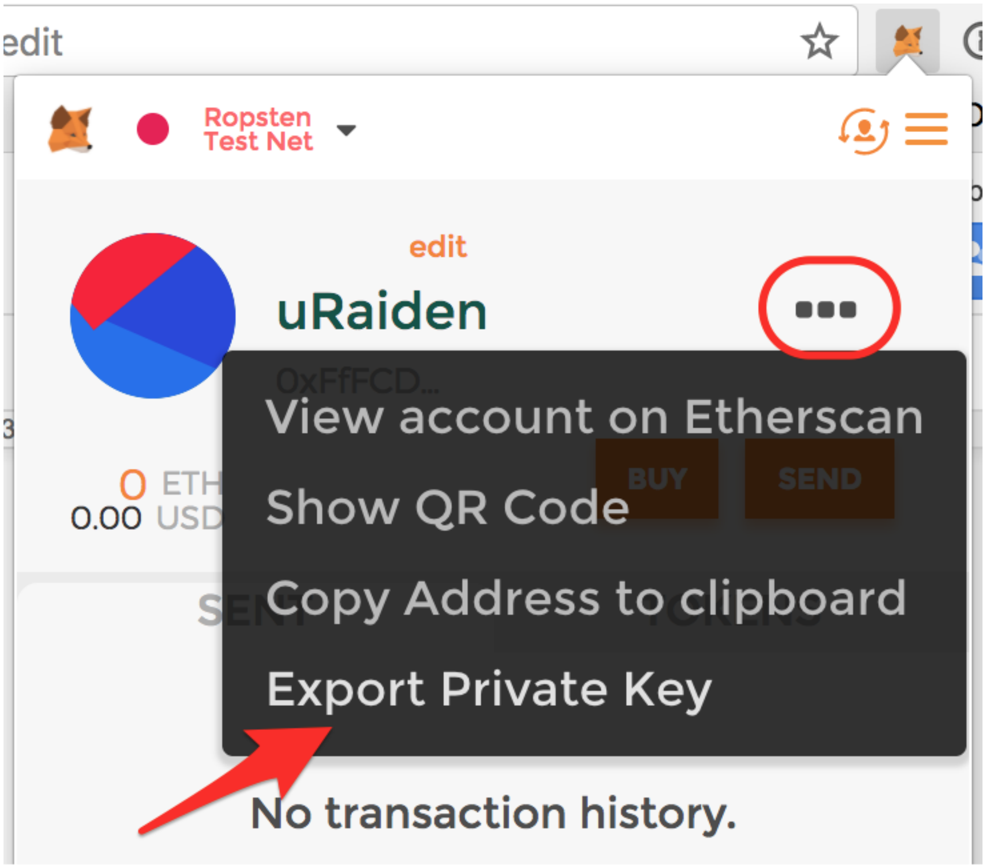
6. Once the transaction was successful, you should see your TKN balance under the *Tokens* tab

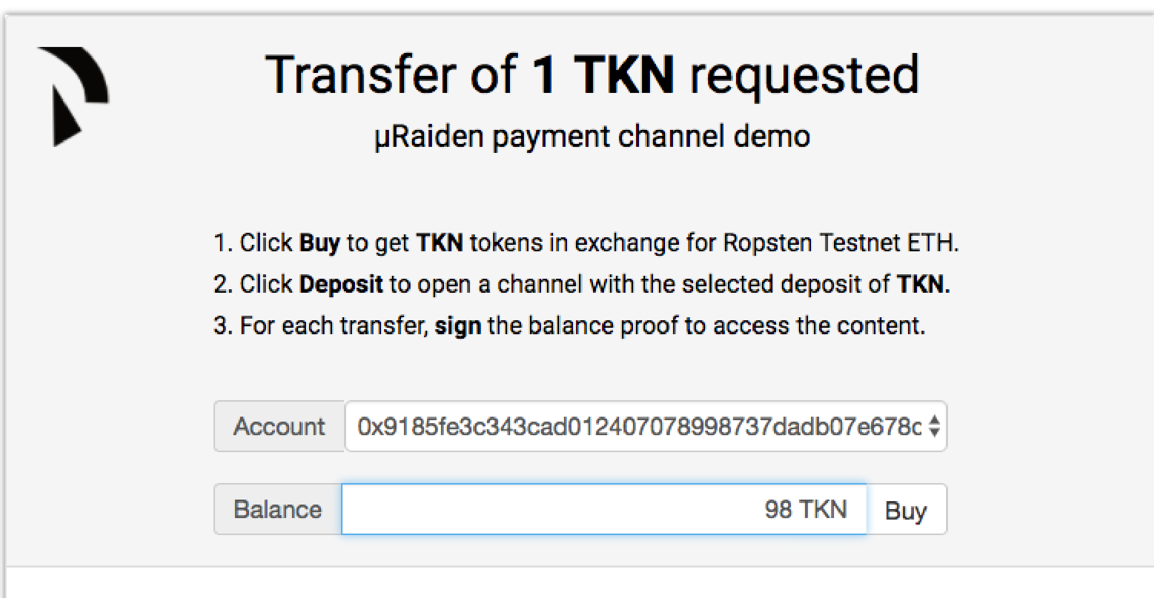
With MyEtherWallet

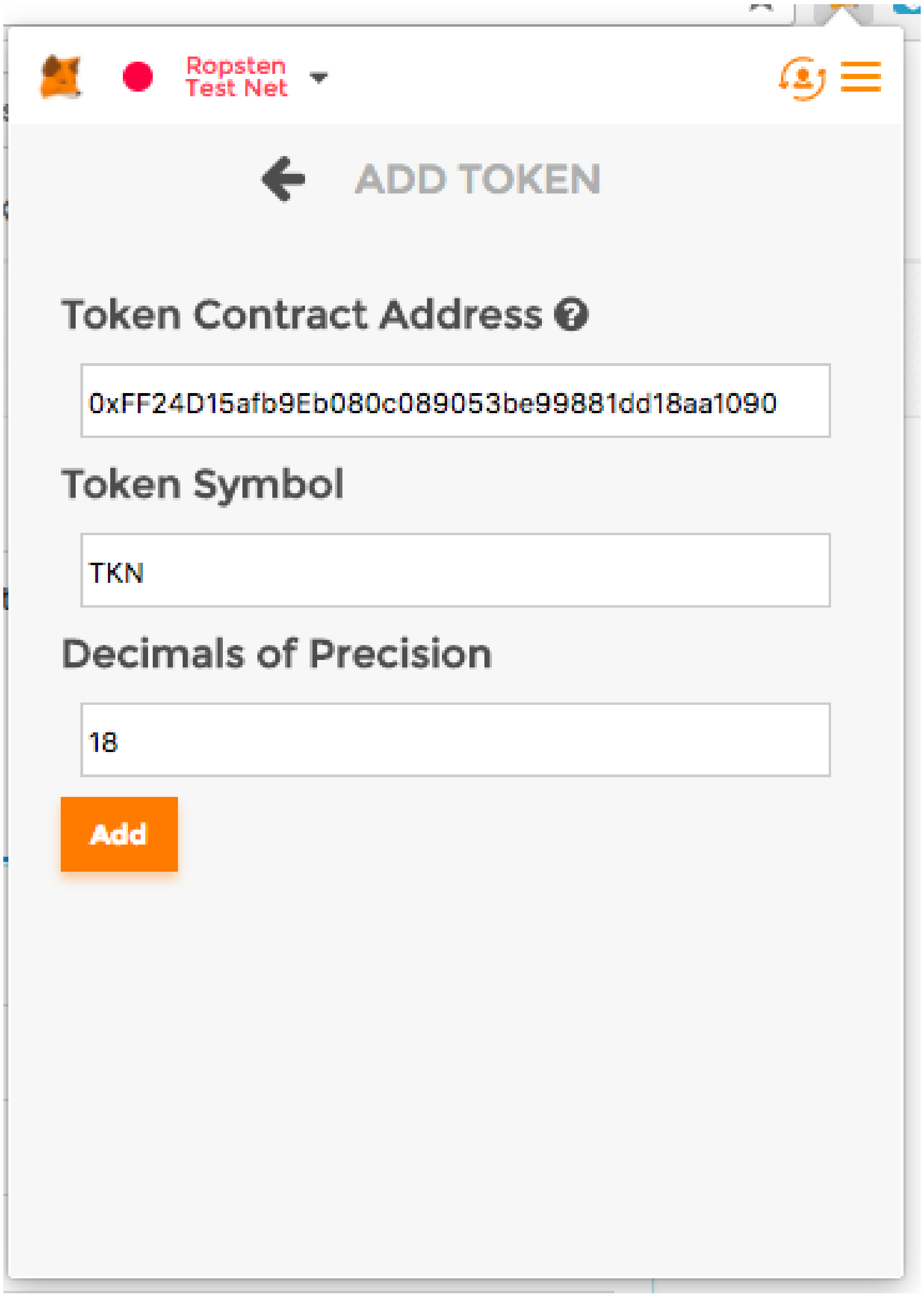
If you want to have a little bit more control over the exchange of token, you can also use MyEtherWallet to interact with the Smart Contract directly:

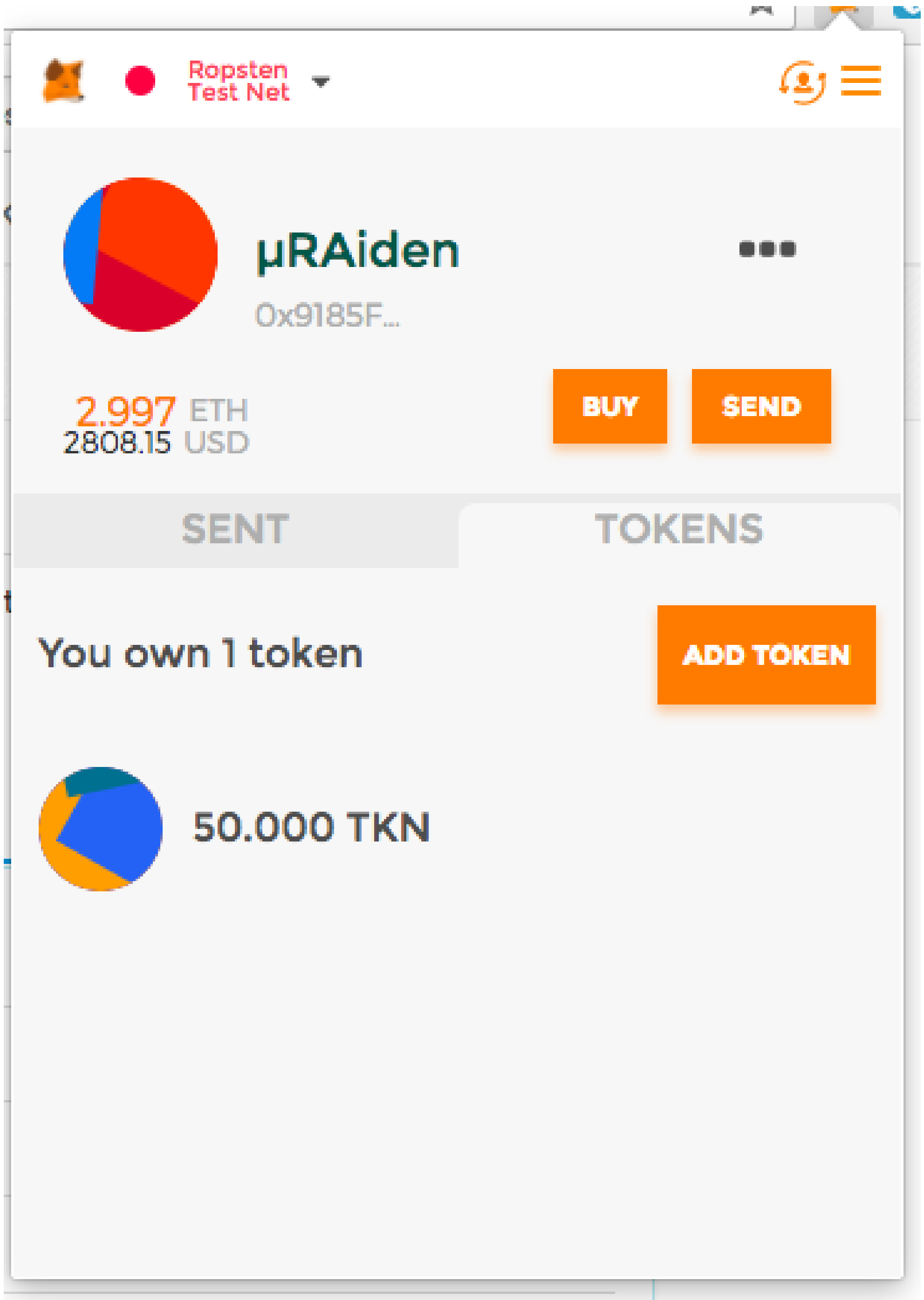
1. Choose the browser with your Ropsten-ETH loaded MetaMask account activated
2. Go to https://www.myetherwallet.com/ and go through their advice on phishing-precautions.
3. Select the *Ropsten* Network in the tab in the upper right
4. click on the Contracts tab and fill in the contract address:


Ropsten
Test Net
1.
uRaiden
0xFfFCD...
2.
0 ETH
0.00 USD
BUY
SEND
SENT
OKENS
3.
No transaction history.
1. Choose Ropsten testnet
2. Create new account
3. "Buy" Ropsten Ether (RETH) at faucet


edit
Ropsten Test Net
edit
uRaiden
0xFfFCD...
View account on Etherscan
Show QR Code
Copy Address to clipboard
Export Private Key
0 ETH
0.00 USD
BUY
SEND
No transaction history.

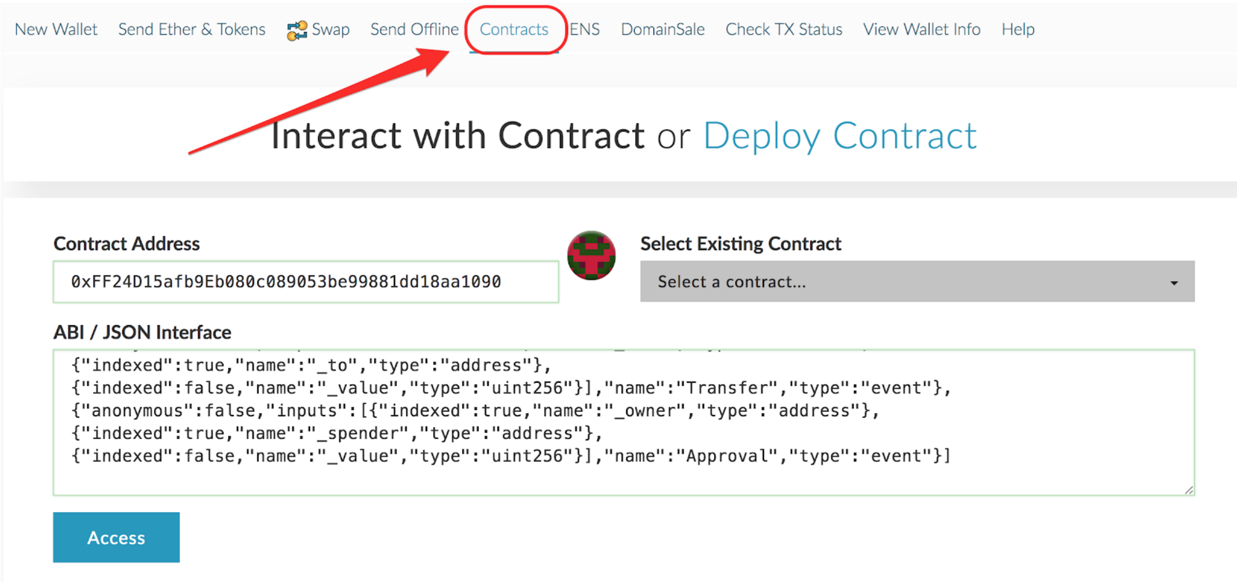

Transfer of 1 TKN requested
µRaiden payment channel demo
1. Click Buy to get TKN tokens in exchange for Ropsten Testnet ETH.
2. Click Deposit to open a channel with the selected deposit of TKN.
3. For each transfer, sign the balance proof to access the content.
Account
0x9185fe3c343cad012407078998737dadb07e678c
Balance
98 TKN
Buy


Ropsten
Test Net
ADD TOKEN
Token Contract Address
0xFF24D15afb9Eb080c089053be99881dd18aa1090
Token Symbol
TKN
Decimals of Precision
18
Add


Ropsten
Test Net
µRAiden
0x9185F...
2.997 ETH
2808.15 USD
BUY
SEND
SENT
TOKENS
You own 1 token
ADD TOKEN
50.000 TKN

```
0xFF24D15afb9Eb080c089053be99881dd18aa1090
```

5. fill in the ABI field with the data you get here:



6. Choose the *mint* function and use MetaMask to access your wallet
7. put in an amount of RopstenETH (RETH) you want to exchange for TKN (0.1 RETH will get you 50 TKN)

A dialogue will pop up in MetaMask that asks for your confirmation of the generated transaction.

8. Accept the transaction

To check whether the exchange of TKN was successful, you can add TKN as a custom token to MetaMask.

9. Under the *Tokens* tab, choose *Add token* and fill in the TKN address again:

```
0xFF24D15afb9Eb080c089053be99881dd18aa1090
```

10. Once the transaction was successful, you should see your TKN balance under the *Tokens* tab

Now you're good to go! Check out the other Tutorials and get started with µRaiden!

4.2.3 Try the echo service

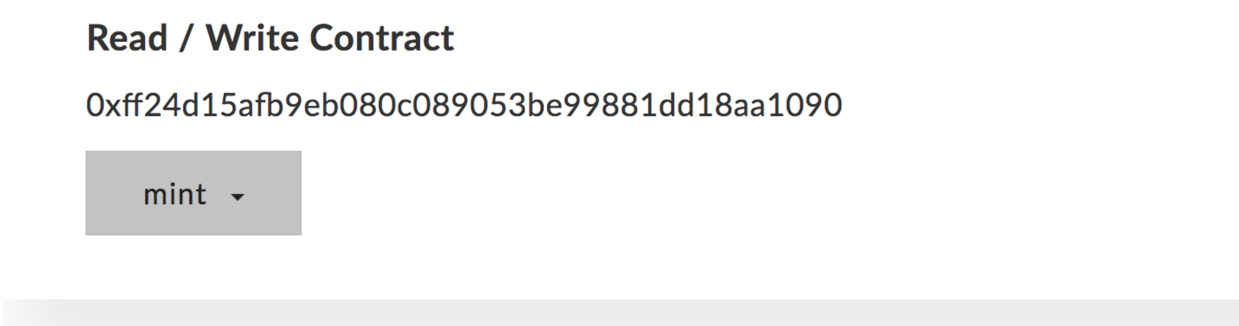
System context

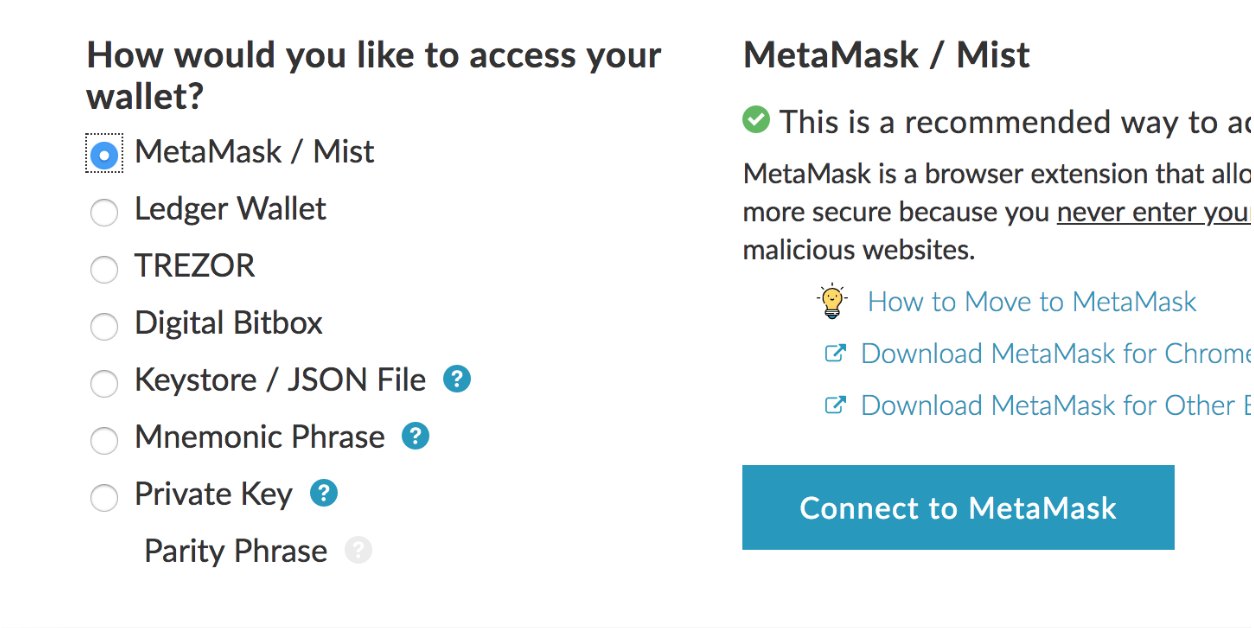
In order to get you started, we created an example application, that receives micropayments and some parameter over a http-request - and simply echos this parameter when the micropayment was valid. Please follow the *microraiden installation instructions* and the *instructions to set up geth*.

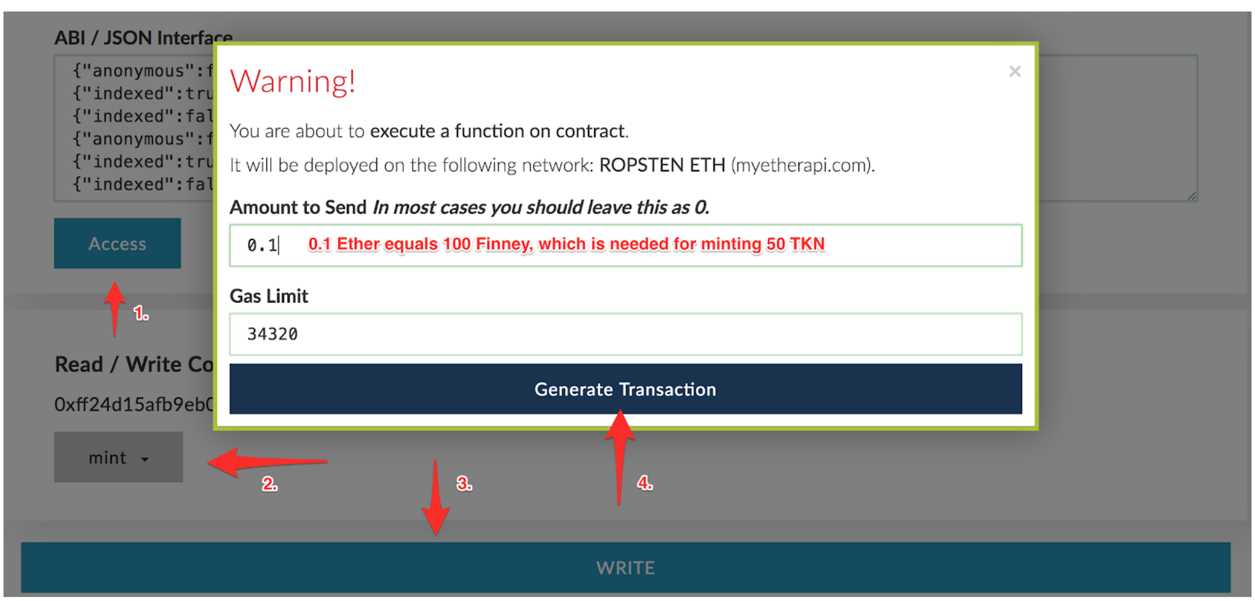
Starting the µRaiden Receiver

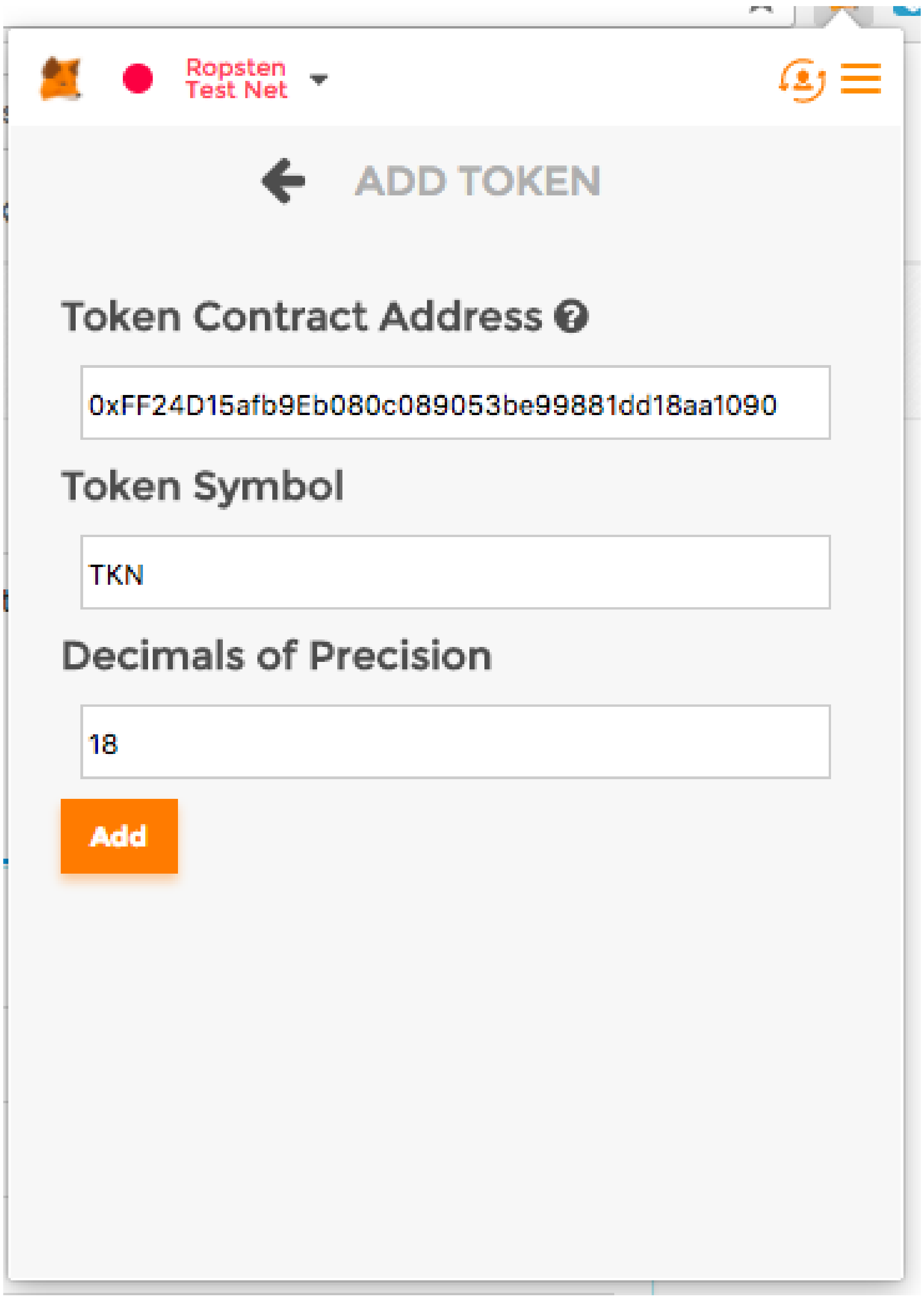
Before starting the receiver, it needs to be assigned a private key with some TKN. Navigate to `./microraiden/microraiden/examples` and create a new file containing your private key as exported in the Blockchain Setup guide by MetaMask. The file should be named `pk_tut.txt`.

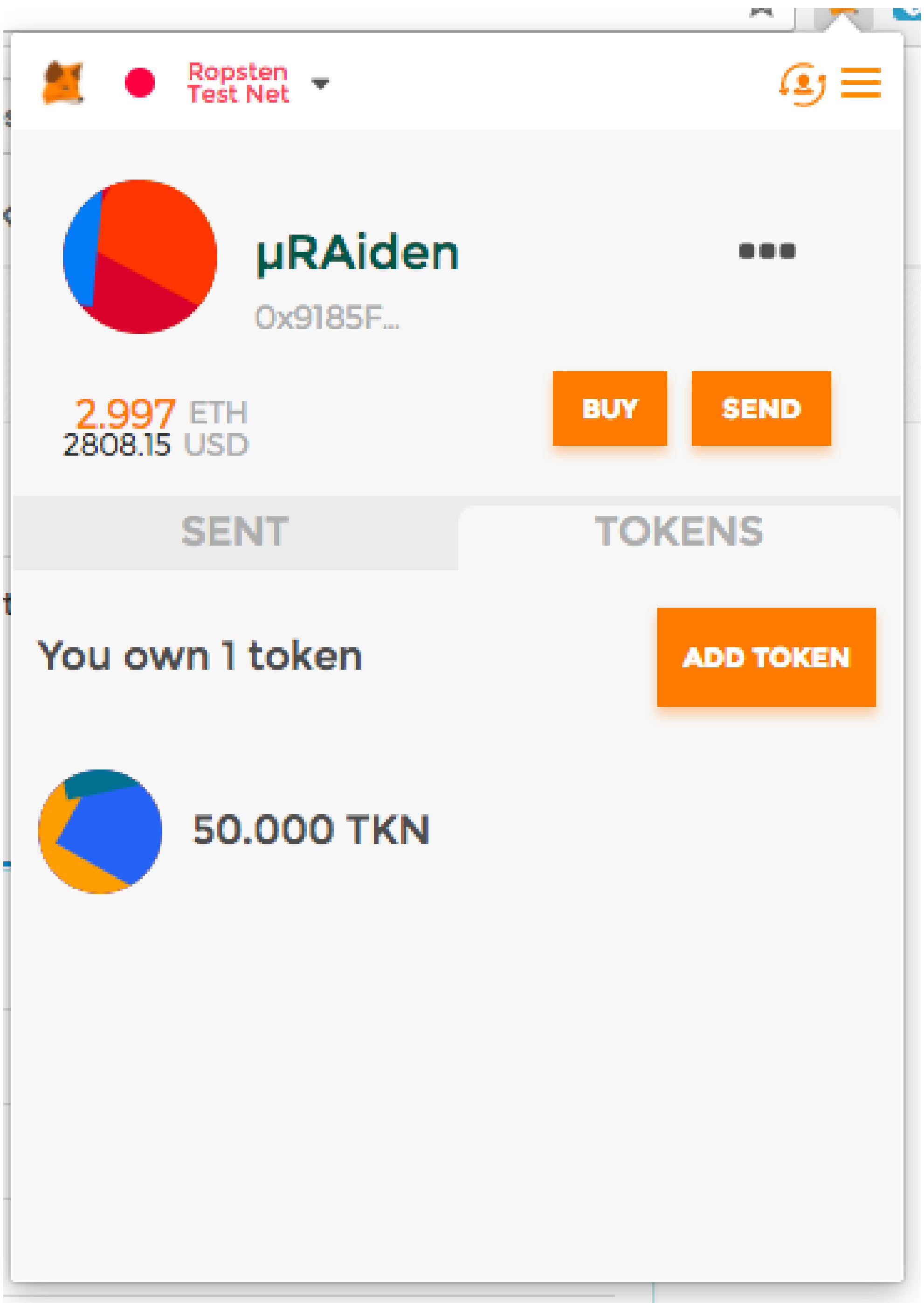
[3] All other icons from http://icomoon.io IcoMoon Icon Pack Free, licensed under a Creative Commons Attribution 4.0 International License


Read / Write Contract
0xff24d15afb9eb080c089053be99881dd18aa1090
mint


How would you like to access your wallet?
MetaMask / Mist
Ledger Wallet
TREZOR
Digital Bitbox
Keystore / JSON File
Mnemonic Phrase
Private Key
Parity Phrase
MetaMask / Mist
This is a recommended way to ac
MetaMask is a browser extension that allo
more secure because you never enter you
malicious websites.
How to Move to MetaMask
Download MetaMask for Chrome
Download MetaMask for Other E
Connect to MetaMask


ABI / JSON Interface
Warning!
You are about to execute a function on contract.
It will be deployed on the following network: ROPSTEN ETH (myetherapi.com).
Amount to Send In most cases you should leave this as 0.
0.1
0.1 Ether equals 100 Finney, which is needed for minting 50 TKN
Gas Limit
34320
Generate Transaction
Access
1.
Read / Write Co
mint
2.
3.
4.
WRITE


Ropsten
Test Net
ADD TOKEN
Token Contract Address
0xFF24D15afb9Eb080c089053be99881dd18aa1090
Token Symbol
TKN
Decimals of Precision
18
Add


Ropsten
Test Net
µRAiden
0x9185F...
2.997 ETH
2808.15 USD
BUY
SEND
SENT
TOKENS
You own 1 token
ADD TOKEN
50.000 TKN

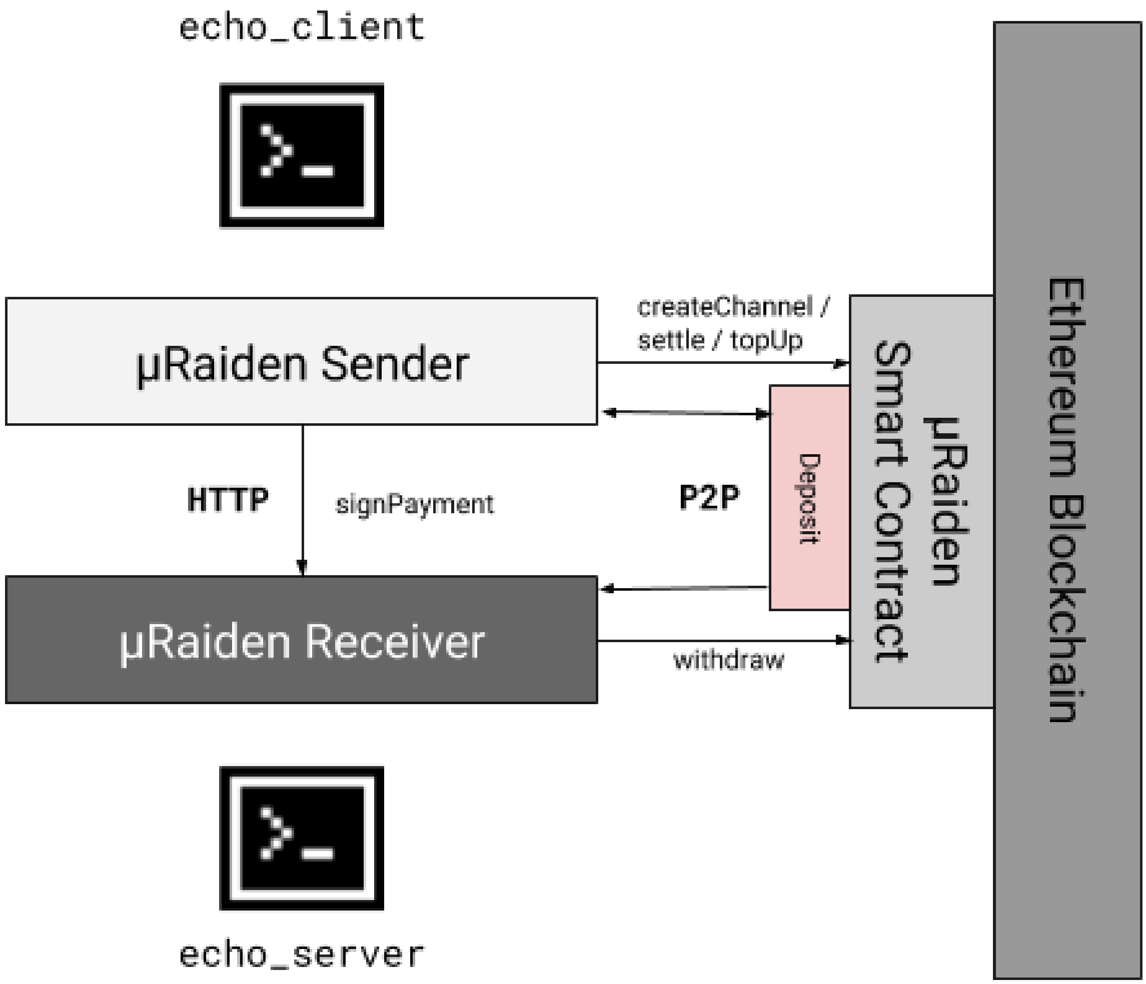


Fig. 2: Schematic overview of a machine-to-machine µRaiden application[3]

From the root directory of µRaiden, start:

```
python microraiden/examples/echo_server.py --private-key microraiden/examples/pk_tut.
↪txt
```

Starting the µRaiden Sender

To actually start the request for resource `/hello`, we will fire up the µRaiden client with the prefunded account.

While the Receiver is still running (in another terminal window for example), execute this command from the µRaiden root folder:

```
python microraiden/examples/echo_client.py --private-key microraiden/examples/pk_tut.
↪txt --resource /echofix/hello
```

After some seconds, you should get the output

```
INFO:root:Got the resource /echofix/hello type=text/html; charset=utf-8:
hello
```

This means:

- a channel has been created
- a deposit of 50 aTKN has been escrowed
- a micropayment of 1 aTKN has been transferred to the receiver
- the receiver returned the requested resource (the "hello" parameter in this simple case) for this payment

Congratulations, you just performed your first micropayment!

4.2.4 Create a paywall

Before starting, please follow the *microraiden installation instructions* and the *instructions to set up geth*.

Introduction

In this tutorial we will create a simple paywalled server that will echo a requested path paramter, payable with a custom token. You can find example code for this tutorial in `microraiden/examples/echo_server.py`.

Requirements

Please refer to README.md to install all required dependencies. You will also need a Chrome browser with the MetaMask plugin.

Setting up the proxy

Initialization

For initialization you will have to supply the following parameters:

- The **private** key of the account receiving the payments (to extract it from a keystore file you can use MyEtherWallet's "View Wallet Info" functionality).

- A **file** in which the proxy stores off-chain balance proofs. Set this to a path writable by the user that starts the server.

```
from microraiden.make_helpers import make_paywalled_proxy
app = make_paywalled_proxy(private_key, state_file_name)
```

`microraiden.make_helpers.make_paywalled_proxy()` is a helper that handles the setup of the channel manager and returns a `microraiden.proxy.paywalled_proxy.PaywalledProxy` instance. Microraiden also includes other helpers that parse common commandline options. We are not using them in this example - for a quick overview how to use them, refer to i.e. `microraiden.examples.demo_proxy.__main__()`

The channel manager will start syncing with the blockchain immediately.

Resource types

Now we will create a custom resource class that simply echoes a path-parameter of the user's request for a fixed price. The workflow is the same as with the Flask-restful: Subclass `microraiden.proxy.resources.Expensive` and implement the HTTP methods you want to expose.

```
from microraiden.proxy.resources import Expensive

class StaticPriceResource(Expensive):
    def get(self, url: str, param: str):
            log.info('Resource requested: {} with param "{}"'.format(request.url,␣
↪param))
            return param
```

We add one static resource to our `PaywalledProxy` instance. The *url* argument will comply with standard flask routing rules.

```
app.add_paywalled_resource(
    cls=StaticPriceResource,
    url="/echofix/<string:param>",
    price=5
)
```

The resource will then be available for example at the URI `/echofix/foo`. Only after a payment of 5 tokens, the proxy will send the `foo` parameter back to the user and will set the Content-Type header appropriately. Without payment, the server responds with `402 Payment Required`.

A probably more useful paywalled resource is a URL. This is useful to fetch content from a remote CDN:

```
from microraiden.proxy.content import PaywalledProxyUrl

app.add_paywalled_resource(
    cls=PaywalledProxyUrl,
    url="cdn\/.*",
    resource_class_kwargs={"domain": 'http://cdn.myhost.com:8000/resource42'}
)
```

Note, that the kwargs for the constructor of the resource-class (here our `PaywalledProxyUrl`) have to be passed as a dict with the `resource_class_kwargs` argument. In this case, the `domain` kwarg is the remote URL specifying where to fetch the content from.

Setting a price for the resource dynamically

We can also construct the Resource in a way that the price will be dynamically calculated, e.g. based on the requests parameters.

```
class DynamicPriceResource(Expensive):
    def get(self, url: str, param: str):
            log.info('Resource requested: {} with param "{}"'.format(request.url, 
↪param))
            return param

    def price(self):
            return len(request.view_args['param'])


app.add_paywalled_resource(
    cls=DynamicPriceResource,
    url="/echodyn/<string:param>",
)
```

Here, the price to be paid is the length of the requested string. A request of the `/echodyn/foo` resource, would therefore require a payment of 3 tokens.

Starting/stopping the proxy

You start proxy by calling `run()` method. This call is non-blocking – the proxy is started as a WSGI greenlet. Use `join()` to sync with the task. This will block until proxy has stopped. To terminate the server, call stop() from another greenlet.

```
app.run(debug=True)
app.join()
```

Accessing the content

Browser

To access the content with your browser, navigate to the URL of the resource you'd like to get. You'll be faced with a paywall – a site requesting you to pay for the resource. To do so, you first have to open a new channel. If you have the MetaMask extension installed, you can set the amount to be deposited to the channel. After confirming the deposit, you can navigate and payments will be done automatically.

Side notes

Proxy state file

Off-chain transactions are stored in a sqlite database. You should do regular backups of this file – it contains balance signatures of the client, and if you lose them, you will have no way of proving that the client is settling the channel using less funds than he has actually paid to the proxy.

4.2.5 Setup μRaiden Raspberry Pi in local network

Prerequisites

- Install the go-ethereum client called geth
- Install the raspberry pi with the RASPBIAN STRETCH WITH DESKTOP OS.
- Set-up the raspberry pi an explanation can be found here .
- Setup the wifi referring to this official guide.
- Login through ssh client(like putty or moba xterm) on windows or a standard terminal if you are on a linux based system.

```
$ ssh pi@192.168.1.105
```

- We highly recommend using a virtual environment with virtualenvwrapper

`sudo pip install virtualenv virtualenvwrapper`

`export WORKON_HOME=~/Envs`

`mkdir -p $WORKON_HOME`

`source /usr/local/bin/virtualenvwrapper.sh`

`mkvirtualenv -p /usr/bin/python3.5 uRaiden`

- Clone and setup microraiden

`git clone https://github.com/raiden-network/microraiden.git`

`cd microraiden/`

```
sudo apt-get install libffi-dev libtool python-dev libssl-dev python-setuptools build-
↪essential automake pkg-config libgmp-dev
```

`pip install -r requirements.txt`

`python setup.py develop`

Running μRaiden Client and Server

As the next step, you are going to setup the raspberry pi (*Sender*) as the μRaiden client or the and our PC as as the μRaiden proxy server (*Receiver*) as well as the web3 provider running a geth node synced to Ropsten testnet. Next, you will run the `echo_server` and the `echo_client` examples from the `microraiden/examples` folder. The `echo_client` on the raspberry pi, and the `echo_server` on your PC.

Please make sure that the raspberry pi and your PC are in the same network.

Running geth on the PC

Start geth with these flags(**run this command on PC**)

```
geth --testnet --syncmode "fast" --rpc --rpcapi eth,net,web3,personal --rpcaddr 0.0.0.
↪0 --rpcport 8545 --rpccorsdomain "*" --cache 256
```

[1] All robot icons made by Freepic from http://www.flaticon.com.
[2] Raspberry PI Pictograms by TinkTank.club

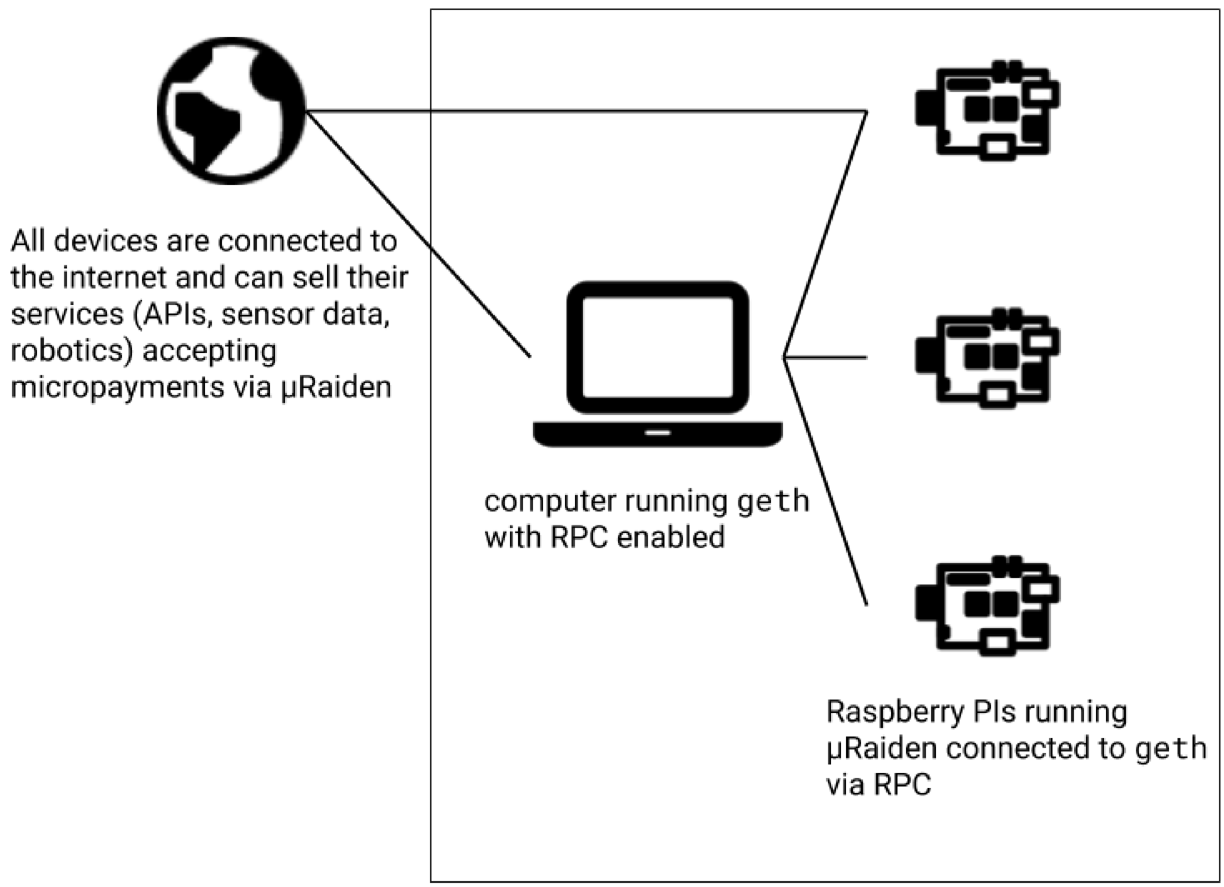


Fig. 3: Networking topology of a machine-to-machine application with RaspberryPi μRaiden-nodes[12]

The *rpcaddr* as **0.0.0.0** means that a given socket is listening on all the available IP addresses the computer has. This is important so that µRaiden client on the raspberry can use it as a **web3** provider.

Before running both the client or the server make sure that both the sender and receiver addresses have .:doc:*TKN balances for opening channels <blockchain>*.

Running the µRaiden Proxy Server

In the `~/microraiden/microraiden/examples` folder go to the `echo_server.py` and go to the part where we start the server.(These set of actions are performed on **your PC**)

`cd ~/microraiden/microraiden/examples`

```
# Start the app. proxy is a WSGI greenlet, so you must join it properly.
app.run(debug=True)
```

Change the *app.run* to include arguments for the *host* and *port*

`app.run(host="192.168.1.104", port=5000, debug=True)`

`192.168.1.104` This IP could be different your set-up. Include the IP address of the interface on your PC that is connected to the raspberry pi.

```
$ python -m  echo_server --private-key ~/.ethereum/testnet/keystore/UTC--2016-07-
↪27T07-40-38.092883212Z--9d80d905bc1e106d5bd0637c12b893c5ab60cb41
 Enter the private key password:
 INFO:filelock:Lock 139916998263696 acquired on ~/.config/microraiden/echo_server.db.
↪lock
 INFO:blockchain:starting blockchain polling (interval 2s)
```

Running the µRaiden client on the raspberry pi

Navigate to the cloned microraiden repository and modify the following files on the **raspberry pi**.

`cd ~/microraiden/microraiden`

1. In the `microraiden/constants.py` file change the **WEB3_PROVIDER_DEFAULT** value to `"http://192.168.1.104:8545"` where `192.168.1.104` is the IP address of the PC where we started the geth node and the µRaiden `echo_server`.

`sudo nano microraiden/constants.py`

2. In the `microraiden/examples/echo_client.py` change `endpoint_url` parameter of the *run* function definition which looks like this

```
def run(
        private_key: str,
        password_path: str,
        resource: str,
        channel_manager_address: str = None,
        web3: Web3 = None,
        retry_interval: float = 5,
        endpoint_url: str = 'http://localhost:5000'
):
```

to the interface of the PC like this `endpoint_url:  str = 'http://192.168.1.104:5000'`. This enables the raspberry to make a request to the server.

```
sudo nano microraiden/examples/echo_client.py
```

Now we run the *echo_client.py* like this

```
(uRaiden) pi@raspberrypi:~/microraiden/microraiden $ python -m  microraiden.examples.
↪echo_client --private-key             ~/.ethereum/testnet/keystore/UTC--2018-02-
↪12T08-35-34.437506909Z--9a7d8c3116258c1f50f3c8ac67d120af58a46ceb --resource       /
↪echofix/hello
Enter the private key password:
INFO:microraiden.client.client:Creating channel to␣
↪0x9d80D905bc1E106d5bd0637c12B893c5Ab60CB41 with an initial deposit of    50 @2684938
WARNING:microraiden.client.session:Newly created channel does not have enough␣
↪confirmations yet. Retrying in 5 seconds.
INFO:root:Got the resource /echofix/hello type=text/html; charset=utf-8:
hello
```

You should get an output like above.The server should also give an output like this showing the requested resource

```
INFO:channel_manager:unconfirmed channel event received (sender␣
↪0x9A7d8c3116258C1F50f3c8ac67d120af58a46CeB, block_number 2684940)
192.168.1.109 - - [2018-02-20 00:41:05] "GET //echofix/hello HTTP/1.1" 402 391 0.
↪010679
INFO:channel_manager:new channel opened (sender␣
↪0x9A7d8c3116258C1F50f3c8ac67d120af58a46CeB, block number 2684940)
INFO:__main__:Resource requested: http://192.168.1.104:5000/echofix/hello with param
↪"hello"
192.168.1.109 - - [2018-02-20 00:41:10] "GET //echofix/hello HTTP/1.1" 200 120 0.
↪060261
```

Through this example we hope developers can develop their own machine to machine clients and their respective server to use microraiden for micropayments according to their respective use cases, using these resources.

1. microraiden **Session** Library (source microraiden/microraiden/client/session.py)
2. microraiden **Requests** Library (source microraiden/microraiden/requests/__init__.py)
3. microraiden **Client** Library (microraiden/microraiden/client/client.py)

Troubleshooting

Failed building wheel for secp256k1.

> If you encounter this problem its mostly your openssl not being compatible with the libsecp256k1 library. secp256k1 is the python binding for this library.

To check whether libsecp256k1 is installed do the following:

```
(uRaiden) pi@raspberrypi:~ $ apt list --installed *secp256k1*
Listing... Done
(uRaiden) pi@raspberrypi:~ $ apt list  *secp256k1*
Listing... Done
libsecp256k1-0/stable 0.1~20161228-1 armhf
libsecp256k1-dev/stable 0.1~20161228-1 armhf
```

The `installed` option tells us whether the package is installed. Since we have none it does not print anything. Later we list the packages which exists in raspbian repository of packages. We install both the packages.

```
sudo apt-get install libsecp256k1-0 libsecp256k1-dev
```

After this we go to the releases page of secp256k1 and download the tar.gz of *0.13.2.4* (version as of writing of this tutorial) like this.

```
wget https://github.com/ludbb/secp256k1-py/archive/0.13.2.4.tar.gz
```

From the current folder we install tar.gz package of *secp256k1* like this.

```
pip install 0.13.2.4.tar.gz
```

After this again install **requirements.txt**

```
pip install -r requirements.txt
```

For Transferring file from your machine to the Raspberry pi refer to this documentation

https://www.raspberrypi.org/documentation/remote-access/ssh/sftp.md

You could download and use filezilla.

References

- http://digitalatomindustry.com/install-ethereum-blockchain-on-raspberry-pi/
- http://raspnode.com/diyEthereumGeth.html
- https://golang.org/dl/
- https://geth.ethereum.org/downloads/
- https://ethereum.stackexchange.com/questions/31610/how-to-install-geth-on-rpi-3b
- https://owocki.com/install-ethereum-geth-raspberry-pi-b/

4.3 Applications

This section covers the documentation of the individual parts of our µRaiden framework. We deliver Python implementations for the payment-receiver side in general as well as for the payment-sender side in a M2M (Machine-to-Machine) or IoT (Internet-of-Things) setup. For user-facing web applications, we provide a JavaScript implementation to directly serve the payment-senders logic to the customer.

4.3.1 Proxy-Server (python)

The Proxy-Server is a Python application that runs on the payment-receivers machine. It intercepts all calls to a HTTP Resource with a Request for Payment HTTP Response. The Proxy-Server application can be used for any kind of pay-per-use use-case, such as for example video-on-demand, pay per API use or pay per newspaper article.

Installation

Prerequisites

It is required that you have `pip` and `python` (version 3.5) installed. You can visit the official pip documentation and install pip before you proceed.

It is recommended to install µRaiden in a separate virtual environment

Library installation

If you plan to use µRaiden as a Python library in your own project, you can fetch the latest release of µRaiden directly from the Python package index and install it automatically:

```
pip install microraiden
```

Development installation

If you plan to change the source-code of the server-framework or if you want to have the highest flexibility for discovering our tutorials, please refer to our *development installation guide*.

Usage

There are several examples that demonstrate how to serve custom content. To try them, run one of the following commands from a Python environment containing a successful µRaiden installation:

```
python3 -m microraiden.examples.demo_proxy --private-key <path_to_private_key_file>␣
↪start
```

or

```
python3 -m microraiden.examples.wikipaydia --private-key <path_to_private_key_file> --
↪private-key-password-file <path_to_password_file> start
```

By default, the web server listens on `0.0.0.0:5000`. The private key file should be in the JSON format produced by Geth/Parity and must be readable and writable only by the owner to be accepted (`-rw-------`).

A `--private-key-password-file` option can be specified, containing the password for the private key in the first line of the file. If it's not provided, the password will be prompted interactively. With the above commands, an Ethereum nodes RPC interface is expected to respond on http://localhost:8545.

If you want to specify a different endpoint for this, use the `--rpc-provider` commandline option.

For more command line options, have a look *here*. To setup `geth`, please refer to our *Blockchain setup tutorial*.

Command line options

When invoking microraiden from the commandline, you have several options:

microraiden

```
microraiden [OPTIONS] COMMAND [ARGS]...
```

Options

--channel-manager-address `<channel_manager_address>`
Ethereum address of the channel manager contract.

--state-file `<state_file>`
State file of the proxy

--private-key <private_key>
Path to private key file of the proxy [required]

--private-key-password-file <private_key_password_file>
Path to file containing password for the JSON-encoded private key

--ssl-cert <ssl_cert>
Cerfificate of the server (cert.pem or similar)

--gas-price <gas_price>
Gas price of outbound transactions

--rpc-provider <rpc_provider>
Address of the Ethereum RPC provider

--ssl-key <ssl_key>
SSL key of the server (key.pem or similar)

--paywall-info <paywall_info>
Directory where the paywall info is stored. The directory should contain an index.html file with the payment info/webapp. Content of the directory (js files, images..) is available on the "js/" endpoint.

start

```
microraiden start [OPTIONS]
```

Options

--host <host>
Address of the proxy

--port <port>
Port of the proxy

Components overview

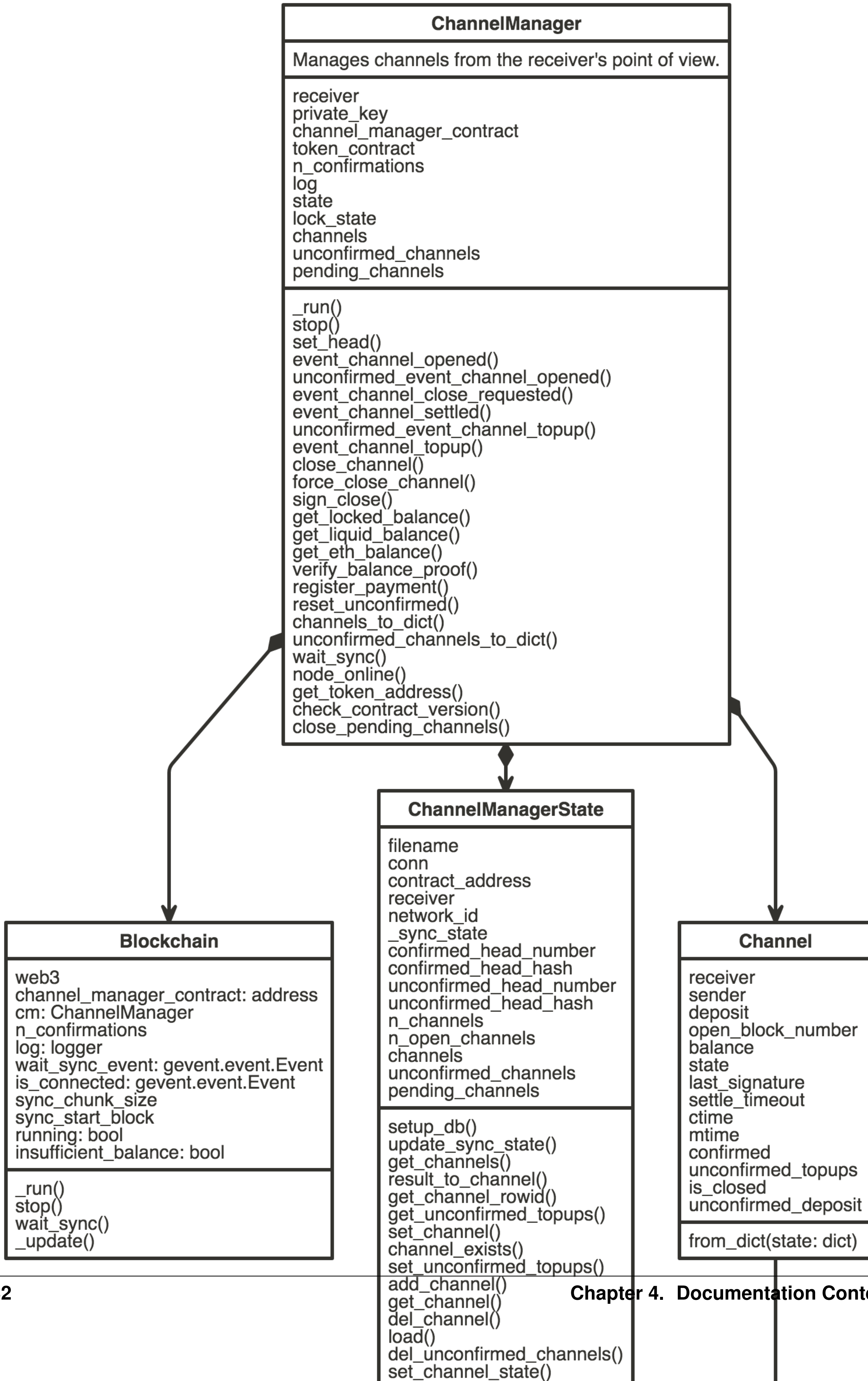
This section will give you an overview of the architecture of the µRaiden server. The illustrations are inspired by UML even though they might not be exact standard.

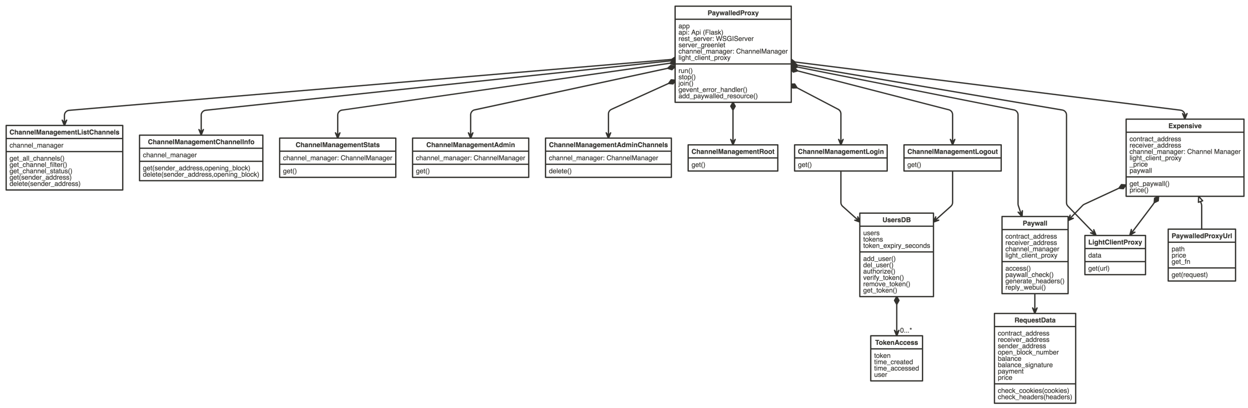
Channel manager

This component interacts with the blockchain and keeps track of the state of all the channels from the transactions Receivers point of view. For more information, look at the API for the `ChannelManager`

Proxy

This component interacts with the blockchain and keeps track of the state of all the channels from the transactions Receivers point of view. For more information, look at the API for the `PaywalledProxy`


ChannelManager
Manages channels from the receiver's point of view.
receiver
private_key
channel_manager_contract
token_contract
n_confirmations
log
state
lock_state
channels
unconfirmed_channels
pending_channels
_run()
stop()
set_head()
event_channel_opened()
unconfirmed_event_channel_opened()
event_channel_close_requested()
event_channel_settled()
unconfirmed_event_channel_topup()
event_channel_topup()
close_channel()
force_close_channel()
sign_close()
get_locked_balance()
get_liquid_balance()
get_eth_balance()
verify_balance_proof()
register_payment()
reset_unconfirmed()
channels_to_dict()
unconfirmed_channels_to_dict()
wait_sync()
node_online()
get_token_address()
check_contract_version()
close_pending_channels()
ChannelManagerState
filename
conn
contract_address
receiver
network_id
_sync_state
confirmed_head_number
confirmed_head_hash
unconfirmed_head_number
unconfirmed_head_hash
n_channels
n_open_channels
channels
unconfirmed_channels
pending_channels
setup_db()
update_sync_state()
get_channels()
result_to_channel()
get_channel_rowid()
get_unconfirmed_topups()
set_channel()
channel_exists()
set_unconfirmed_topups()
add_channel()
get_channel()
del_channel()
load()
del_unconfirmed_channels()
set_channel_state()
Blockchain
web3
channel_manager_contract: address
cm: ChannelManager
n_confirmations
log: logger
wait_sync_event: gevent.event.Event
is_connected: gevent.event.Event
sync_chunk_size
sync_start_block
running: bool
insufficient_balance: bool
_run()
stop()
wait_sync()
_update()
Channel
receiver
sender
deposit
open_block_number
balance
state
last_signature
settle_timeout
ctime
mtime
confirmed
unconfirmed_topups
is_closed
unconfirmed_deposit
from_dict(state: dict)



Python API

Here you can find the API of some of the important classes and methods of the µRaiden server framework. This documentation is auto-generated from the docstrings and type-hints in the source code, so if you have further questions please consult the source-code first.

Channel manager handles channel state changes on a low (blockchain) level.

class `microraiden.channel_manager.manager.`**ChannelManager**(*web3*, *channel_manager_contract*, *token_contract*, *private_key*, *state_filename=None*, *n_confirmations=1*)

Manages channels from the receiver's point of view.

channels

channels_to_dict()

Export all channels as a dictionary.

check_contract_version()

Compare version of the contract to the version of the library. Only major and minor version is used in the comparison.

close_channel(*sender*, *open_block_number*)

Close and settle a channel. Params:

sender (str): sender address open_block_number (int): block the channel was open in

close_pending_channels()

Close all channels that are in CLOSE_PENDING state. This state happens if the receiver's eth balance is not enough to

close channel on-chain.

event_channel_close_requested(*sender*, *open_block_number*, *balance*, *settle_timeout*)

Notify the channel manager that a the closing of a channel has been requested. Params:

settle_timeout (int): settle timeout in blocks

event_channel_opened(*sender*, *open_block_number*, *deposit*)

Notify the channel manager of a new confirmed channel opening.

event_channel_settled(*sender*, *open_block_number*)

Notify the channel manager that a channel has been settled.

event_channel_topup(*sender*, *open_block_number*, *txhash*, *added_deposit*)
Notify the channel manager that the deposit of a channel has been topped up.

force_close_channel(*sender*, *open_block_number*)
Forcibly remove a channel from our channel state

get_eth_balance()
Get eth balance of the receiver

get_liquid_balance()
Get the balance of the receiver in the token contract (not locked in channels).

get_locked_balance()
Get the balance in all channels combined.

get_token_address()

node_online()

pending_channels

register_payment(*sender*, *open_block_number*, *balance*, *signature*)
Register a payment. Method will try to reconstruct (verify) balance update data with a signature sent by the client. If verification is succesfull, an internal payment state is updated. :type sender: `str` :param sender: sender of the balance proof :type sender: str :type open_block_number: `int` :param open_block_number: block the channel was opened in :type open_block_number: int :type balance: `int` :param balance: updated balance :type balance: int :type signature: `str` :param signature: balance proof to verify :type signature: str

reset_unconfirmed()
Forget all unconfirmed channels and topups to allow for a clean resync.

set_head(*unconfirmed_head_number*, *unconfirmed_head_hash*, *confirmed_head_number*, *confirmed_head_hash*)
Set the block number up to which all events have been registered.

sign_close(*sender*, *open_block_number*, *balance*)
Sign an agreement for a channel closing. :returns: a signature that can be used client-side to close

the channel by directly calling contract's close method on-chain.

Return type channel close signature (str)

stop()

unconfirmed_channels

unconfirmed_channels_to_dict()
Export all unconfirmed channels as a dictionary.

unconfirmed_event_channel_opened(*sender*, *open_block_number*, *deposit*)
Notify the channel manager of a new channel opening that has not been confirmed yet.

unconfirmed_event_channel_topup(*sender*, *open_block_number*, *txhash*, *added_deposit*)
Notify the channel manager of a topup with not enough confirmations yet.

verify_balance_proof(*sender*, *open_block_number*, *balance*, *signature*)
Verify that a balance proof is valid and return the sender.

This method just verifies if the balance proof is valid - no state update is performed.

Returns Channel, if it exists

wait_sync()

class microraiden.channel_manager.channel.**Channel**(*receiver*, *sender*, *deposit*, *open_block_number*)

__init__(*receiver*, *sender*, *deposit*, *open_block_number*)
A channel between two parties.

Parameters

- **receiver** (*str*) – receiver address
- **sender** (*str*) – sender address
- **deposit** (*int*) – channel deposit
- **open_block_number** (*int*) – block the channel was created in

classmethod from_dict(*state*)

is_closed
Returns – bool: True if channel is closed

Return type bool

to_dict()

Returns Channel object serialized as a dict

Return type dict

unconfirmed_deposit
Returns – int: sum of all deposits, including unconfirmed ones

class microraiden.channel_manager.channel.**ChannelState**
An enumeration.

CLOSED = 1

CLOSE_PENDING = 2

OPEN = 0

UNDEFINED = 100

class microraiden.proxy.paywalled_proxy.**PaywalledProxy**(*channel_manager*, *flask_app=None*, *paywall_html_dir=None*, *paywall_js_dir=None*)

add_paywalled_resource(*cls*, *url*, *price=None*, **args*, ***kwargs*)

static gevent_error_handler(*context*, *exc_info*)

join()
Block until server greenlet/Proxy stops

run(*host='localhost'*, *port=5000*, *debug=False*, *ssl_context=None*)
Start the proxy/WSGI server

stop()

4.3.2 M2M-Client (python)

The M2M-Client is a Python framework for building applications that run on the payment-senders machine. The client interacts with the blockchain to handle channel creation and closing when needed. It communicates with a payment-receivers http-endpoint (e.g. implemented with our *Python server framework*) and handles the signing of off-chain transactions.

In order to implement an automated machine-to-machine interaction, the framework provides event-handler methods, that allow to easily write extension classes to handle the clients business-logic without user interaction.

Installation

Prerequisites

It is required that you have `pip` and `python` (version 3.5) installed. You can visit the official pip documentation and install pip before you proceed.

It is recommended to install µRaiden in a separate virtual environment

Library installation

If you plan to use µRaiden as a Python library in your own project, you can fetch the latest release of µRaiden directly from the Python package index and install it automatically:

```
pip install microraiden
```

Development installation

If you plan to change the source-code of the client-framework or if you want to have the highest flexibility for discovering our tutorials, please refer to our *development installation guide*.

Usage

The µRaiden client can be used as a standalone library, to make machine-to-machine payments easily available.

An example use of the library would import the `Client` *class*:

```
from microraiden import Client

client = Client('<hex-encoded private key>')
```

Alternatively you can specify a path to a JSON private key, optionally specifying a file containing the password. If it's not provided, it'll be prompted interactively.

```
client = Client(key_path='<path to private key file>', key_password_file='<path to password file>')
```

This client object allows interaction with the blockchain and offline-signing of transactions/balance-proofs.

An example lifecycle of a `Client` object could look like this:

```
from microraiden import Client

receiver = '0xb6b79519c91edbb5a0fc95f190741ad0c4b1bb4d'
privkey = '0x55e58f57ec2177ea681ee461c6d2740060fd03109036e7e6b26dcf0d16a28169'

# 'with' statement to cleanly release the client's file lock in the end.
```

(continues on next page)

(continued from previous page)

```
with Client(privkey) as client:

    channel = client.get_suitable_channel(receiver, 10)
    channel.create_transfer(3)
    channel.create_transfer(4)

    print(
        'Current balance proof:\n'
        'From: {}\n'
        'To: {}\n'
        'Channel opened at block: #{}\n'  # used to uniquely identify this channel
        'Balance: {}\n'                   # total: 7
        'Signature: {}\n'                 # valid signature for a balance of 7 on␣
↪this channel
        .format(
            channel.sender, channel.receiver, channel.block, channel.balance, channel.
↪balance_sig
        )
    )

    channel.topup(5)                      # total deposit: 15

    channel.create_transfer(5)            # total balance: 12

    channel.close()

    # Wait for settlement period to end.

    channel.settle()

    # Instead of requesting a close and waiting for the settlement period to end, you␣
↪can also perform
    # a cooperative close, provided that you have a receiver-signed balance proof␣
↪that matches your
    # current channel balance.

    channel.close_cooperatively(closing_sig)
```

Components overview

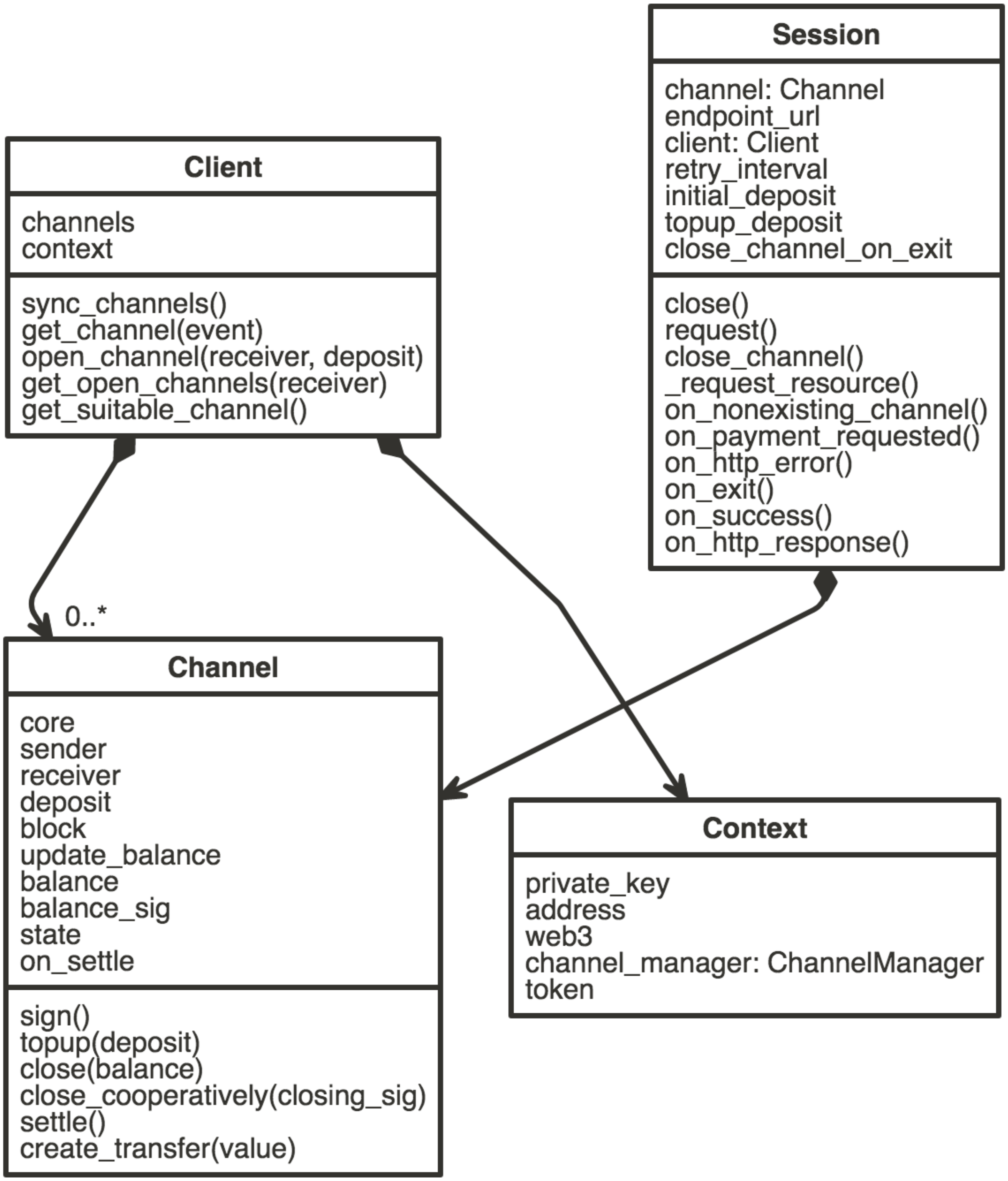
This section will give you an overview of the architecture of the µRaiden M2M-client. The illustrations are inspired by UML even though they might not be exact standard.

Client

Python API

Here you can find the API of some of the important classes and methods of the µRaiden M2M-client framework. This documentation is auto-generated from the docstrings and type-hints in the source code, so if you have further questions please consult the source-code first.

class microraiden.client.client.**Client**(*private_key=None*, *key_password_path=None*, *channel_manager_address=None*, *web3=None*)


Session
channel: Channel
endpoint_url
client: Client
retry_interval
initial_deposit
topup_deposit
close_channel_on_exit
close()
request()
close_channel()
_request_resource()
on_nonexisting_channel()
on_payment_requested()
on_http_error()
on_exit()
on_success()
on_http_response()
Client
channels
context
sync_channels()
get_channel(event)
open_channel(receiver, deposit)
get_open_channels(receiver)
get_suitable_channel()
0..*
Channel
core
sender
receiver
deposit
block
update_balance
balance
balance_sig
state
on_settle
sign()
topup(deposit)
close(balance)
close_cooperatively(closing_sig)
settle()
create_transfer(value)
Context
private_key
address
web3
channel_manager: ChannelManager
token

`__init__`(*private_key=None*, *key_password_path=None*, *channel_manager_address=None*, *web3=None*)

Parameters

- **`private_key`** (`Optional[str]`) –
- **`key_password_path`** (`Optional[str]`) –
- **`channel_manager_address`** (`Optional[str]`) –
- **`web3`** (`Optional[Web3]`) –

Return type `None`

`get_open_channels`(*receiver=None*)

Returns all open channels to the given receiver. If no receiver is specified, all open channels are returned.

Return type `List[Channel]`

`get_suitable_channel`(*receiver*, *value*, *initial_deposit=<function Client.<lambda>>*, *topup_deposit=<function Client.<lambda>>*)

Searches stored channels for one that can sustain the given transfer value. If none is found, a possibly open channel is topped up using the topup callable to determine its topup value. If both attempts fail, a new channel is created based on the initial deposit callable. Note: In the realistic case that only one channel is opened per (sender,receiver) pair, this method usually performs like this: 1. Directly return open channel if sufficiently funded. 2. Topup existing open channel if insufficiently funded. 3. Create new channel if no open channel exists. If topping up or creating fails, this method returns None. Channels are topped up just enough so that their remaining capacity equals topup_deposit(value).

Return type `Channel`

`open_channel`(*receiver_address*, *deposit*)

Open a channel with a receiver and deposit

Attempts to open a new channel to the receiver with the given deposit. Blocks until the creation transaction is found in a pending block or timeout is reached.

Parameters

- **`receiver_address`** (`str`) – the partner with whom the channel should be opened
- **`deposit`** (`int`) – the initial deposit for the channel (Should be > 0)

Return type `Optional[Channel]`

Returns The opened channel, if successful. Otherwise *None*

`sync_channels`()

Merges locally available channel information, including their current balance signatures, with channel information available on the blockchain to make up for local data loss. Naturally, balance signatures cannot be recovered from the blockchain.

class `microraiden.client.channel.`**`Channel`**(*core*, *sender*, *receiver*, *block*, *deposit=0*, *balance=0*, *state=<State.open: 1>*, *on_settle=<function Channel.<lambda>>*)

class State

An enumeration.

closed = 3

open = 1

settling = 2

balance

balance_sig

close (*balance=None*)
Attempts to request close on a channel. An explicit balance can be given to override the locally stored balance signature. Blocks until a confirmation event is received or timeout.

close_cooperatively (*closing_sig*)
Attempts to close the channel immediately by providing a hash of the channel's balance proof signed by the receiver. This signature must correspond to the balance proof stored in the passed channel state.

create_transfer (*value*)
Updates the given channel's balance and balance signature with the new value. The signature is returned and stored in the channel state.

is_suitable (*value*)

is_valid ()

Return type `bool`

key

Return type `bytes`

settle ()
Attempts to settle a channel that has passed its settlement period. If a channel cannot be settled yet, the call is ignored with a warning. Blocks until a confirmation event is received or timeout.

sign ()

topup (*deposit*)
Attempts to increase the deposit in an existing channel. Block until confirmation.

update_balance (*value*)

class `microraiden.client.session.`**Session** (*client=None, endpoint_url=None, retry_interval=5, initial_deposit=<function Session.<lambda>>, topup_deposit=<function Session.<lambda>>, close_channel_on_exit=False, **client_kwargs*)

close ()
Closes all adapters and as such the session

close_channel (*endpoint_url=None*)

on_cooperative_close_denied (*response=None*)

on_exit (*method, url, response, **kwargs*)

on_http_error (*method, url, response, **kwargs*)

Return type `bool`

on_http_response (*method, url, response, **kwargs*)
Called whenever server returns a reply. Return False to abort current request.

Return type `bool`

on_init (*method, url, **kwargs*)

on_insufficient_confirmations (*method, url, response, **kwargs*)

Return type `bool`

on_invalid_amount (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

on_invalid_balance_proof (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

on_invalid_contract_address (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

on_nonexisting_channel (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

on_payment_requested (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

on_success (*method*, *url*, *response*, ***kwargs*)

Return type `bool`

request (*method*, *url*, ***kwargs*)

Constructs a `Request`, prepares it and sends it. Returns `Response` object.

Parameters

- **method** (`str`) – method for the new `Request` object.
- **url** (`str`) – URL for the new `Request` object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*`float or tuple`*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*`bool`*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol or protocol and hostname to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. Defaults to `True`.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Return type requests.Response

class `microraiden.client.context.`**Context** (*private_key*, *web3*, *channel_manager_address*)

4.3.3 Web-Client (JavaScript)

For a quick overview on how to use the Javascript client library please refer to the README

API documentation

The autodocumented API reference currently also resides in our GitHub.

4.4 Smart Contract

Smart Contracts, unit-tests and infrastructure for the enforcing *Blockchain component* of µRaiden.

4.4.1 Installation

Prerequisites

It is required that you have `pip` and `python` (version 3.6) installed. You can visit the official pip documentation and install pip before you proceed.

It is recommended to install the Smart Contract of µRaiden in a separate virtual environment

For the installation of the Smart Contracts it is **not** necessary to install the server or client components of µRaiden.

4.4.2 Setup

From the root directory of µRaiden, execute the following to run the install script:

```
cd contracts
make install
```

4.4.3 Deployment

Chain setup for testing

Note - you can change RPC/IPC chain connection, timeout parameters etc. in contracts/project.json

privtest

1. Start the geth-node from the commandline:

```
geth --ipcpath="~/Library/Ethereum/privtest/geth.ipc" \
     --datadir="~/Library/Ethereum/privtest" \
    --dev \
    ---rpc --rpccorsdomain '\*' --rpcport 8545 \
    --rpcapi eth,net,web3,personal \
    --unlock 0xf590ee24CbFB67d1ca212e21294f967130909A5a \
    --password ~/password.txt

# the geth console will show up
```

(continues on next page)

(continued from previous page)

```
# you have to mine yourself:
miner.start()
geth attach ipc:~/Library/Ethereum/privtest/geth.ipc
```

The `--unlock` argument specifies which geth account should be unlocked for RPC access. This assumes that `0xf590ee24CbFB67d1ca212e21294f967130909A5a` is your account's address, and has to be changed accordingly. More info can be found here.

The `--password` argument specifies the file that contains the passphrase the geth account has been locked with. The `~/password.txt` file has to be changed accordingly to your password file location. More info can be found here.

kovan

1. Get some testnet-Ether at the kovan-faucet
2. Modify the project.json to change the default account
3. Start the Parity node from the commandline:

```
parity --geth \
       --chain kovan \
       --force-ui --reseal-min-period 0 \
       --jsonrpc-cors http://localhost \
       --jsonrpc-apis web3,eth,net,parity,traces,rpc,personal \
       --unlock 0x5601Ea8445A5d96EEeBF89A67C4199FbB7a43Fbb \
       --password ~/password.txt \
       --author 0x5601Ea8445A5d96EEeBF89A67C4199FbB7a43Fbb
```

The `--unlock` argument specifies which parity account should be unlocked for RPC access. This assumes that `0x5601Ea8445A5d96EEeBF89A67C4199FbB7a43Fbb` is your account's address, and has to be changed accordingly. More info can be found here.

The `--password` argument specifies the file that contains the passphrase the geth account has been locked with. The `~/password.txt` file has to be changed accordingly to your password file location. More info can be found here.

The `--author` argument specifies what the *coinbase* address should be. Set this to the same address as with the `--unlock` argument. This assumes that `0x5601Ea8445A5d96EEeBF89A67C4199FbB7a43Fbb` is your account's address, and has to be changed accordingly. More info can be found here.

ropsten

1. Get some testnet-Ether at the ropsten-faucet
2. Modify the project.json to change the default account
3. Start the geth node from the commandline:

```
geth --testnet \
     --rpc --rpcport 8545 \
     --unlock 0xf590ee24CbFB67d1ca212e21294f967130909A5a \
     --password ~/password.txt
```

The `--unlock` argument specifies which geth account should be unlocked for RPC access. This assumes that `0xf590ee24CbFB67d1ca212e21294f967130909A5a` is your account's address, and has to be changed accordingly. More info can be found here.

The `--password` argument specifies the file that contains the passphrase the geth account has been locked with. The `~/password.txt` file has to be changed accordingly to your password file location. More info can be found here.

rinkeby

1. Get some testnet-Ether at the rinkeby-faucet
2. Modify the /contracts/project.json to change the default account

Fast deployment

There are some scripts to provide you with convenient ways to setup a quick deployment.

```
# Fast deploy on kovan | ropsten | rinkeby | tester | privtest

cd microraiden/contracts

# Following two calls are equivalent
python -m deploy.deploy_testnet  # --owner is web.eth.accounts[0]
python -m deploy.deploy_testnet \
  --chain kovan \
  --owner `0xf590ee24CbFB67d1ca212e21294f967130909A5a` \
  --challenge-period 500 \
  --token-name CustomToken --token-symbol TKN \
  --supply 10000000 --token-decimals 18

# Provide an already deployed, custom token:
python -m deploy.deploy_testnet --token-address TOKEN_ADDRESS
```

Apart from the `--owner` argument, above are the default values. The script provides the following command-line options:

python -m deploy.deploy_testnet

```
python -m deploy.deploy_testnet [OPTIONS]
```

Options

--chain `<chain>`
: Chain to deploy on: kovan | ropsten | rinkeby | tester | privtest

--owner `<owner>`
: Contracts owner, default: web3.eth.accounts[0]

--challenge-period `<challenge_period>`
: Challenge period in number of blocks.

--supply `<supply>`
: Token contract supply (number of total issued tokens).

--token-name `<token_name>`
: Token contract name.

--token-decimals <token_decimals>
Token contract number of decimals.

--token-symbol <token_symbol>
Token contract symbol.

--token-address <token_address>
Already deployed token address.

4.4.4 Usage

- from root/contracts:

```
# compilation
populus compile

# tests
pytest
pytest -p no:warnings -s
pytest tests/test_uraiden.py -p no:warnings -s

# Recommended for speed:
# you have to comment lines in tests/conftest.py to use this
pip install pytest-xdist==1.17.1
pytest -p no:warnings -s -n NUM_OF_CPUs
```

4.4.5 API

Generated docs

There is an Auto-Generated-API, that is compiled with *soldocs*.

Prerequisites

```
pip install soldocs
populus compile
soldocs --input build/contracts.json --output docs/contract/
↪RaidenMicroTransferChannels.md --contracts RaidenMicroTransferChannels
```

Opening a transfer channel

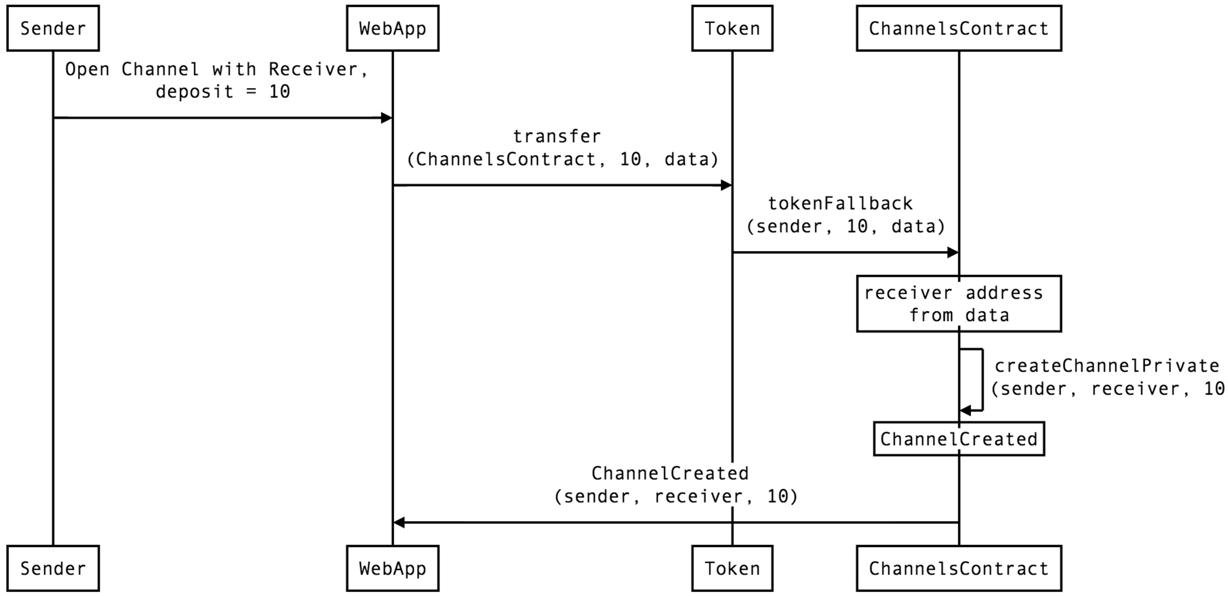
ERC223 compatible (recommended)

Sender sends tokens to the Contract, with a payload for calling `createChannelPrivate`.

```
Token.transfer(_to, _value, _data)
```

Gas cost (testing): 88976

- `_to` = `Contract.address`
- `_value` = deposit value (number of tokens)
- `_data` contains the Sender and Receiver addresses encoded in 20 bytes
- in python `_data = bytes.fromhex(sender_address[2:] + receiver_address[2:])`

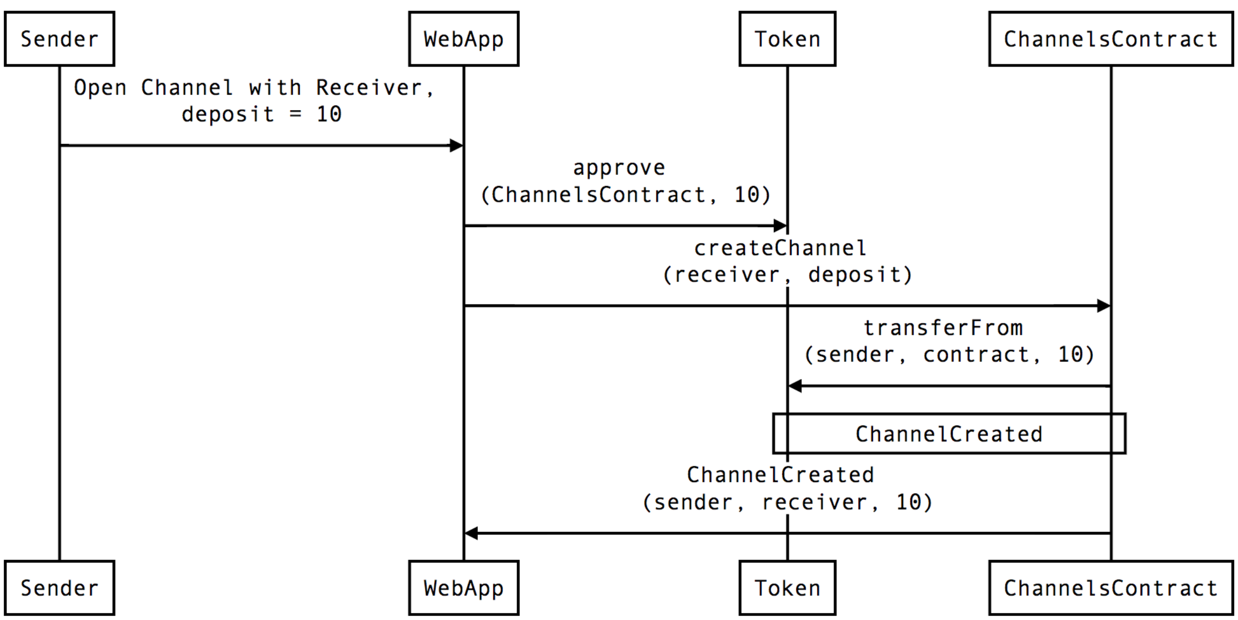


ERC20 compatible

```
# approve token transfers to the contract from the Sender's behalf
Token.approve(contract, deposit)

Contract.createChannel(receiver_address, deposit)
```

Gas cost (testing): 120090



Topping up a channel

Adding tokens to an already opened channel.

ERC223 compatible (recommended)

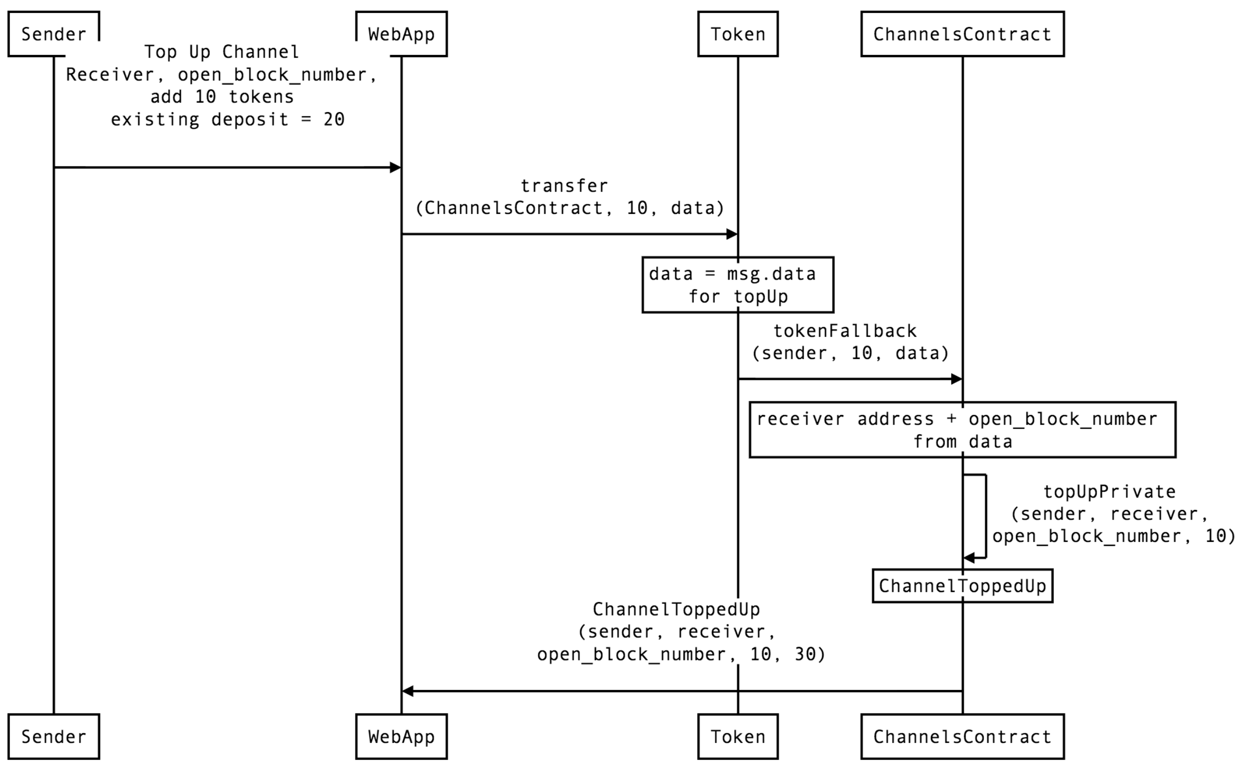
Sender sends tokens to the Contract, with a payload for calling `topUp`.

```
Token.transfer(_to, _value, _data)
```

Gas cost (testing): 54885

- `_to` = `Contract.address`
- `_value` = deposit value (number of tokens)
- `_data` contains the Sender and Receiver addresses encoded in 20 bytes + the open_block_number in 4 bytes
- in python

```
_data = sender_address[2:] + receiver_address[2:] + hex(open_block_number)[2:].
↪zfill(8)
_data = bytes.fromhex(_data)
```

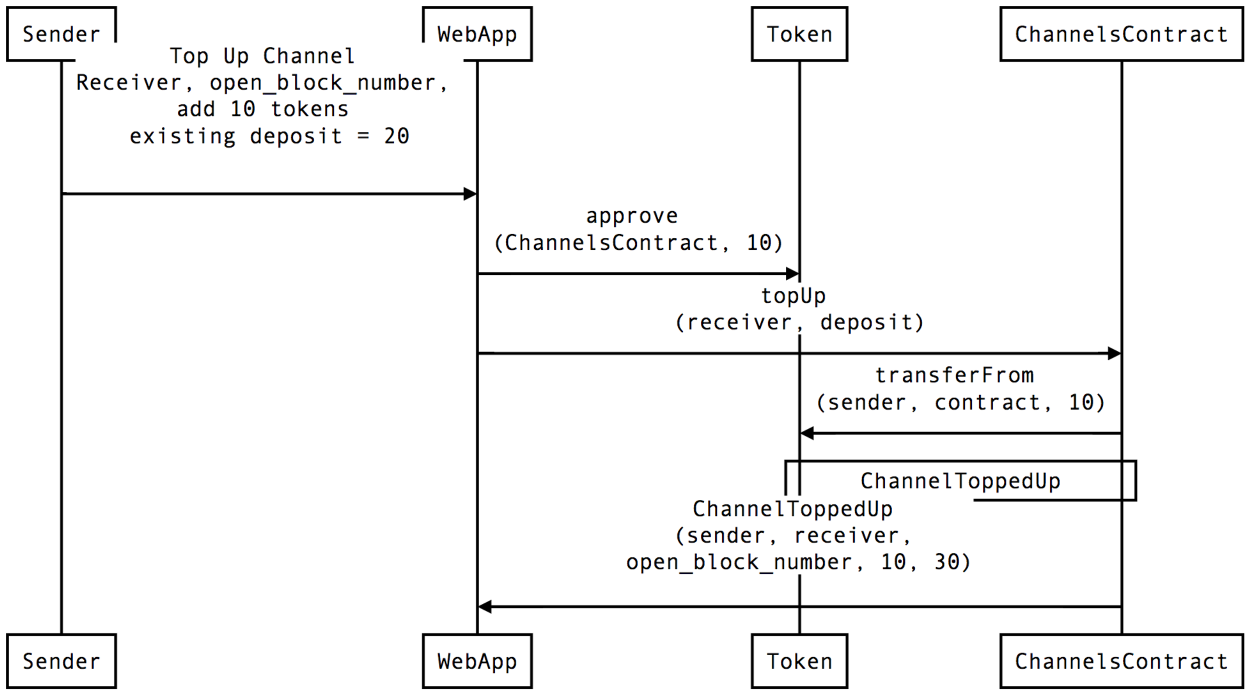


ERC20 compatible

```
#approve token transfers to the contract from the Sender's behalf
Token.approve(contract, added_deposit)

# open_block_number = block number at which the channel was opened
Contract.topUp(receiver_address, open_block_number, added_deposit)
```

Gas cost (testing): 85414



Generating and validating a balance proof

(to be updated post EIP712)

```
# Sender has to provide a balance proof to the Receiver when making a micropayment
# The contract implements some helper functions for that

# Balance message
bytes32 balance_message_hash = keccak256(
    keccak256(
        'string message_id',
        'address receiver',
        'uint32 block_created',
        'uint192 balance',
        'address contract'
    ),
    keccak256(
        'Sender balance proof signature',
        _receiver_address,
        _open_block_number,
        _balance,
        address(this)
    )
);

# balance_message_hash is signed by the Sender with MetaMask
balance_msg_sig

# Data is sent to the Receiver (receiver, open_block_number, balance, balance_msg_sig)
```

Generating and validating a closing agreement

```
from eth_utils import encode_hex

# Sender has to provide a balance proof to the Contract and
# a closing agreement proof from Receiver (closing_sig)
# closing_sig is created in the same way as balance_msg_sig, but it is signed by the␣
↪Receiver

# Closing signature message
bytes32 balance_message_hash = keccak256(
    keccak256(
        'string message_id',
        'address sender',
        'uint32 block_created',
        'uint192 balance',
        'address contract'
    ),
    keccak256(
        'Receiver closing signature',
        _sender_address,
        _open_block_number,
        _balance,
        address(this)
    )
);

# balance_message_hash is signed by the Sender with MetaMask
balance_msg_sig

# balance_msg_sig is signed by the Receiver inside the microraiden code
closing_sig

# Send to the Contract (example of collaborative closing, transaction sent by Sender)
Contract.transact({ "from": Sender }).cooperativeClose(
    _receiver_address,
    _open_block_number,
    _balance,
    _balance_msg_sig,
    _closing_sig
)
```

Balance proof / closing agreement signature verification:

```
sender_address = Contract.call().extractBalanceProofSignature(receiver_address, open_
↪block_number, balance, balance_msg_sig)

receiver_address = Contract.call().extractClosingSignature(sender_address, open_block_
↪number, balance, closing_sig)
```

Closing a channel

```
# 1. Receiver calls Contract with the sender's signed balance message = instant close␣
↪& settle
# 2. Client calls Contract with receiver's closing signature = instant close & settle
# Gas cost (testing): 71182
Contract.cooperativeClose(receiver_address, open_block_number, balance, balance_msg_
↪sig, closing_sig)

# 3. Client calls Contract without receiver's closing signature = challenge period␣
↪starts, channel is not settled yet
# Gas cost (testing): 53876
Contract.uncooperativeClose(receiver_address, open_block_number, balance)

# 3.a. During the challenge period, 1. can happen.

# 3.b. Client calls Contract after settlement period ends
# Gas cost (testing): 40896
Contract.settle(receiver_address, open_block_number)
```

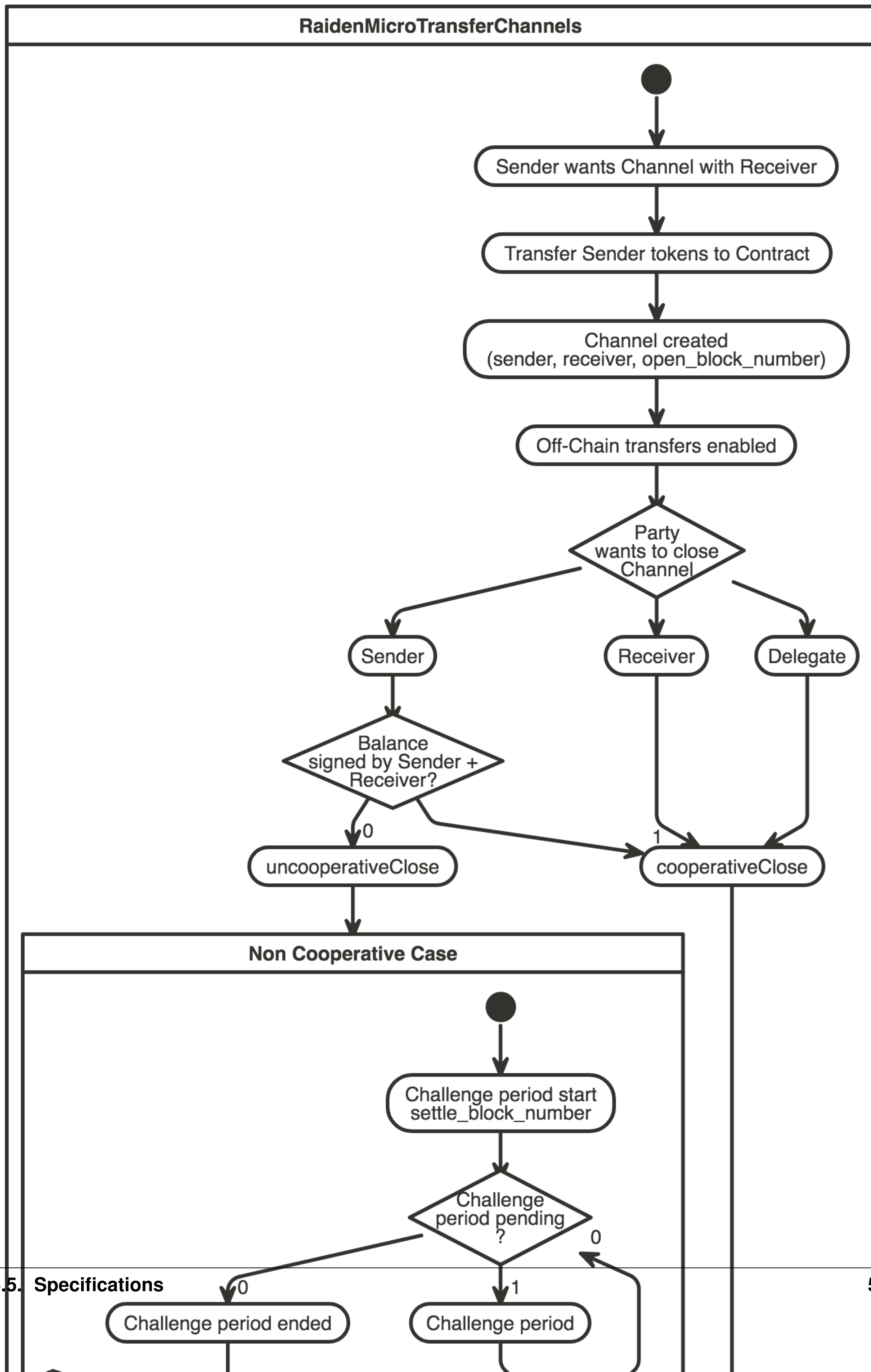
4.5 Specifications

4.5.1 HTTP Headers

Response Headers

200 OK

Headers	Type	Description
RDN-Gateway-Path	bytes	Path root of the channel management app
RDN-Receiver-Address	address	Address of the Merchant
RDN-Contract-Address	address	Address of RaidenMicroTransferChannels contract
RDN-Token-Address	address	Address of the Token contract
RDN-Price	uint	Resource price
RDN-Sender-Address	address	Address of the Client
RDN-Sender-Balance	uint	Balance of the Channel


RaidenMicroTransferChannels
Sender wants Channel with Receiver
Transfer Sender tokens to Contract
Channel created
(sender, receiver, open_block_number)
Off-Chain transfers enabled
Party
wants to close
Channel
Sender
Receiver
Delegate
Balance
signed by Sender +
Receiver?
0
1
uncooperativeClose
cooperativeClose
Non Cooperative Case
Challenge period start
settle_block_number
Challenge
period pending
?
0
0
1
Challenge period ended
Challenge period

402 Payment Required

Headers	Type	Description
RDN-Gateway-Path	bytes	Path root of the channel management app
RDN-Receiver-Address	address	Address of the Merchant
RDN-Contract-Address	address	Address of RaidenMicroTransferChannels contract
RDN-Token-Address	address	Address of the Token contract
RDN-Price	uint	Resource price
RDN-Sender-Address	address	Address of the Client
RDN-Sender-Balance	uint	Balance of the Channel
RDN-Balance-Signature	bytes	Optional. Last saved balance proof from the sender.
		+ one of the following:
RDN-Insufficient-Confirmations	uint	Failure - not enough confirmations after the channel creation. Client should wait and retry.
RDN-Nonexisting-Channel	string	Failure - channel does not exist or was closed.
RDN-Invalid-Balance-Proof	uint	Failure - Balance must not be greater than deposit or The balance must not decrease.
RDN-Invalid-Amount	uint	Failure - wrong payment value

409

- ValueError

502

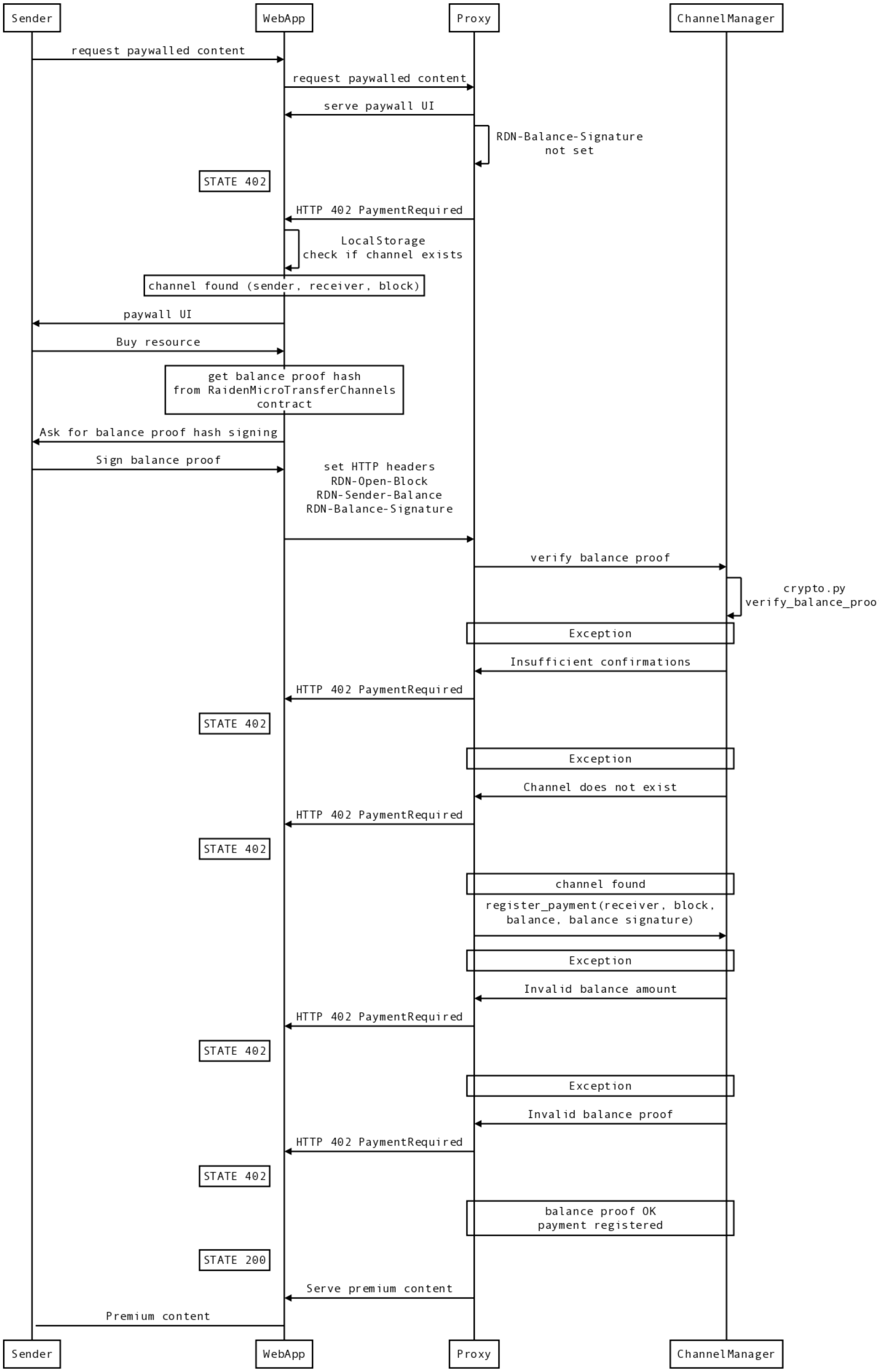
- Ethereum node is not responding
- Channel manager ETH balance is below limit

Request Headers

Headers	Type	Description
RDN-Contract-Address	address	Address of MicroTransferChannels contract
RDN-Receiver-Address	address	Address of the Merchant
RDN-Sender-Address	address	Address of the Client
RDN-Payment	uint	Amount of the payment
RDN-Sender-Balance	uint	Balance of the Channel
RDN-Balance-Signature	bytes	Signature from the Sender, signing the balance (post payment)
RDN-Open-Block	uint	Opening block number of the channel required for unique identification

4.5.2 Off-Chain sequence

(not-so-standard sequence diagram) For a better overview, also check out how the smart contract does a *balance-proof validation*.


Sender
WebApp
Proxy
ChannelManager
request paywalled content
request paywalled content
serve paywall UI
RDN-Balance-Signature not set
STATE 402
HTTP 402 PaymentRequired
LocalStorage check if channel exists
channel found (sender, receiver, block)
paywall UI
Buy resource
get balance proof hash from RaidenMicroTransferChannels contract
Ask for balance proof hash signing
Sign balance proof
set HTTP headers RDN-Open-Block RDN-Sender-Balance RDN-Balance-Signature
verify balance proof
crypto.py verify_balance_proof
Exception
Insufficient confirmations
HTTP 402 PaymentRequired
STATE 402
Exception
Channel does not exist
HTTP 402 PaymentRequired
STATE 402
channel found
register_payment(receiver, block, balance, balance signature)
Exception
Invalid balance amount
HTTP 402 PaymentRequired
STATE 402
Exception
Invalid balance proof
HTTP 402 PaymentRequired
STATE 402
balance proof OK payment registered
STATE 200
Serve premium content
Premium content
Sender
WebApp
Proxy
ChannelManager

4.5.3 REST API

µRaiden exposes a Restful API to provide insight into a channel state, balances, and it allows proxy operator to close and settle the channels.

Proxy endpoints

Getting the status of the proxy

`/api/1/stats`

Return proxy status: balances, open channels, contract ABI etc.

- **deposit_sum** - sum of all open channel deposits
- **open_channels** - count of all open channels
- **pending_channels** - count of all closed, but not yet settled channels
- **balance_sum** - sum of all spent, but not yet settled funds
- **unique_senders** - count of all unique addresses that have channels open
- **liquid_balance** - amount of tokens that are settled and available to the receiver
- **token_address** - token contract address
- **contract_address** - channel manager contract address
- **receiver_address** - server's ethereum address
- **manager_abi** - ABI of the channel manager contract
- **token_abi** - ABI of the token contract

Example Request

`GET /api/1/stats`

Example Response

`200 OK` and

```
{
  "deposit_sum": "268",
  "open_channels": "33",
  "pending_channels": "15",
  "balance_sum": "12",
  "unique_senders": "6",
  "liquid_balance": "334",
  "token_address" : "0x8227a53130c90d32e0294cdde576411379138ba8",
  "contract_address": "0x69f8b894d89fb7c4f6f082f4eb84b2b2c3311605",
  "receiver_address": "0xe67104491127e419064335ea5bf714622a209660",
  "manager_abi": "{ ... }",
  "token_abi": "{ ... }",
}
```

Channel endpoints

Getting all open channels

`/api/1/channels/`

Return a list of all open channels.

Example Request

`GET /api/1/channels`

Example Response

`200 OK` and

```
[
{
    "sender_address" : "0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb",
    "open_block"   : "3241462",
    "balance"      : "0",
    "deposit"      : "10",
},
{
    "sender_address" : "0x5176305093fff279697d3fc9b6bc09574303edb4",
    "open_block"   : "32654234",
    "balance"      : "0",
    "deposit"      : "25",
},
]
```

Getting all open channels for a given sender

`/api/1/channels/<sender_address>`

Return a list of all open channels for the sender specified in the second argument of the URL.

Example Request

`GET /api/1/channels/0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb`

Example Response

`200 OK` and

```
[
{
    "sender_address" : "0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb",
    "open_block"   : "3241462",
    "balance"      : "0",
```

(continues on next page)

(continued from previous page)

```
    "deposit"      : "10",
    "state"        : "open",
},
]
```

Getting a single channel info

`/api/1/channels/<sender_address>/<open_block>`

Return an info about the channel, identified by sender and open block id.

Example Request

`GET /api/1/channels/0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb/3241462`

Example Response

`200 OK` and

```
{
    "sender_address" : "0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb",
    "open_block"   : "3241462",
    "balance"      : "0",
    "deposit"      : "10",
    "state"        : "open",
}
```

Cooperatively closing a channel

`/api/1/channels/<sender_address>/<open_block>`

Return a receiver's signature that can be used to settle the channel immediately (by calling contract's *cooperativeClose()* function).

Example Request

`DELETE /api/1/channels/0x5601ea8445a5d96eeebf89a67c4199fbb7a43fbb/3241642`

with payload balance - last balance of the channel

```
{
    "balance": 13000,
}
```

Example Response

`200 OK` and

```
{
    "close_signature" :
↪"0xb30809f9a32e4f5012a3e7a7275e4f0f96eaff49f7a34747507abc3147a0975c31cf9f9aa318d1f9675d6e39f062a565
↪",
}
```

Possible Responses

HTTP Code	Condition
200 OK	For a successful coop-close
500 Server Error	Internal Raiden node error
400 Bad request	Invalid address, signature, or channel doesn't exist.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

M

P