Incubed Documentation

Release 2.3

Blockchains LLC

Nov 29, 2021

Reference

CHAPTER 1

Getting Started

Incubed can be used in different ways:

Stack	Size	Code Base	Use Case
TS/JS	2.7 MB (browserified)	TypeScript	Web application (client in the browser) or mobile application
TS/JS/WASM	470 kB	C - (WASM)	Web application (client in the browser) or mobile application
C/C++	200 KB	C	IoT devices can be integrated nicely on many micro controllers (like Zephyr-supported boards (https://docs.zephyrproject.org/latest/boards/index.html)) or any other C/C++ application
Java	705 KB	C	Java implementation of a native wrapper
Docker	2.6 MB	C	For replacing existing clients with this docker and connecting to Incubed via localhost:8545 without needing to change the architecture
Bash	400 KB	C	The command-line tool can be used directly as executable within Bash script or on the shell

Other languages will be supported soon (or simply use the shared library directly).

1.1 TypeScript/JavaScript

Installing Incubed is as easy as installing any other module:

```
npm install --save in3
```

1.1.1 As Provider in Web3

The Incubed client also implements the provider interface used in the Web3 library and can be used directly.

```
// import in3-Module
import In3Client from 'in3'
import * as web3 from 'web3'

// use the In3Client as Http-Provider
const web3 = new Web3(new In3Client({
    proof         : 'standard',
    signatureCount: 1,
    requestCount  : 2,
    chainId       : 'mainnet'
}).createWeb3Provider())

// use the web3
const block = await web.eth.getBlockByNumber('latest')
...
```

1.1.2 Direct API

Incubed includes a light API, allowing the ability to not only use all RPC methods in a type-safe way but also sign transactions and call functions of a contract without the Web3 library.

For more details, see the API doc.

```
// import in3-Module
import In3Client from 'in3'

// use the In3Client
const in3 = new In3Client({
    proof         : 'standard',
    signatureCount: 1,
    requestCount  : 2,
    chainId       : 'mainnet'
})

// use the API to call a function..
const myBalance = await in3.eth.callFn(myTokenContract, 'balanceOf(address):uint',␣
↪myAccount)

// ot to send a transaction..
const receipt = await in3.eth.sendTransaction({
  to           : myTokenContract,
  method       : 'transfer(address,uint256)',
  args         : [target,amount],
  confirmations: 2,
  pk           : myKey
})

...
```

1.2 As Docker Container

To start Incubed as a standalone client (allowing other non-JS applications to connect to it), you can start the container as the following:

```
docker run -d -p 8545:8545  slockit/in3:latest -port 8545
```

1.3 C Implementation

The C implementation will be released soon!

```
#include <in3/client.h>    // the core client
#include <in3/eth_api.h>   // wrapper for easier use
#include <in3/eth_basic.h> // use the basic module
#include <in3/in3_curl.h>  // transport implementation

#include <inttypes.h>
#include <stdio.h>

int main(int argc, char* argv[]) {

  // register a chain-verifier for basic Ethereum-Support, which is enough to verify␣
↪blocks
  // this needs to be called only once
  in3_register_eth_basic();

  // use curl as the default for sending out requests
  // this needs to be called only once.
  in3_register_curl();

  // create new incubed client
  in3_t* in3 = in3_new();

  // the b lock we want to get
  uint64_t block_number = 8432424;

  // get the latest block without the transaction details
  eth_block_t* block = eth_getBlockByNumber(in3, block_number, false);

  // if the result is null there was an error an we can get the latest error message␣
↪from eth_lat_error()
  if (!block)
    printf("error getting the block : %s\n", eth_last_error());
  else {
    printf("Number of transactions in Block #%llu: %d\n", block->number, block->tx_
↪count);
    free(block);
  }

  // cleanup client after usage
  in3_free(in3);
}
```

More details coming soon. . .

1.4 Java

The Java implementation uses a wrapper of the C implemenation. This is why you need to make sure the libin3.so, in3.dll, or libin3.dylib can be found in the java.library.path. For example:

```
java -cp in3.jar:. HelloIN3.class
```

```
import java.util.*;
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;

public class HelloIN3 {
  //
  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = new IN3();

    // configure
    in3.setChainId(0x1); // set it to mainnet (which is also dthe default)

    // read the latest Block including all Transactions.
    Block latestBlock = in3.getEth1API().getBlockByNumber(Block.LATEST, true);

    // Use the getters to retrieve all containing data
    System.out.println("current BlockNumber : " + latestBlock.getNumber());
    System.out.println("minded at : " + new Date(latestBlock.getTimeStamp()) + " by "
↪+ latestBlock.getAuthor());

    // get all Transaction of the Block
    Transaction[] transactions = latestBlock.getTransactions();

    BigInteger sum = BigInteger.valueOf(0);
    for (int i = 0; i < transactions.length; i++)
      sum = sum.add(transactions[i].getValue());

    System.out.println("total Value transfered in all Transactions : " + sum + " wei
↪");
  }

}
```

1.5 Command-line Tool

Based on the C implementation, a command-line utility is built, which executes a JSON-RPC request and only delivers the result. This can be used within Bash scripts:

```
CURRENT_BLOCK = `in3 -c kovan eth_blockNumber`

#or to send a transaction

in3 -pk my_key_file.json send -to 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1 -value 0.
↪2eth
```

(continues on next page)

(continued from previous page)

```
in3 -pk my_key_file.json send -to 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1 -gas␣
↪1000000  "registerServer(string,uint256)" "https://in3.slock.it/kovan1" 0xFF
```

1.6 Supported Chains

Currently, Incubed is deployed on the following chains:

1.6.1 Mainnet

Registry-legacy: 0x2736D225f85740f42D17987100dc8d58e9e16252

Registry: 0x64abe24afbba64cae47e3dc3ced0fcab95e4edd5

ChainId: 0x1 (alias `mainnet`)

Status: https://in3.slock.it?n=mainnet

NodeList: https://in3.slock.it/mainnet/nd-3

1.6.2 Kovan

Registry-legacy: 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1

Registry: 0x33f55122c21cc87b539e7003f7ab16229bc3af69

ChainId: 0x2a (alias `kovan`)

Status: https://in3.slock.it?n=kovan

NodeList: https://in3.slock.it/kovan/nd-3

1.6.3 Evan

Registry: 0x85613723dB1Bc29f332A37EeF10b61F8a4225c7e

ChainId: 0x4b1 (alias `evan`)

Status: https://in3.slock.it?n=evan

NodeList: https://in3.slock.it/evan/nd-3

1.6.4 Görli

Registry-legacy: 0x85613723dB1Bc29f332A37EeF10b61F8a4225c7e

Registry: 0xfea298b288d232a256ae0ad5941e5c890b1db691

ChainId: 0x5 (alias `goerli`)

Status: https://in3.slock.it?n=goerli

NodeList: https://in3.slock.it/goerli/nd-3

1.6.5 IPFS

Registry: 0xf0fb87f4757c77ea3416afe87f36acaa0496c7e9

ChainId: 0x7d0 (alias `ipfs`)

Status: https://in3.slock.it?n=ipfs

NodeList: https://in3.slock.it/ipfs/nd-3

1.7 Registering an Incubed Node

If you want to participate in this network and also register a node, you need to send a transaction to the registry contract, calling `registerServer(string _url, uint _props)`.

ABI of the registry:


```
[{"constant":true,"inputs":[],"name":"totalServers","outputs":[{"name":"","type":
↪"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant
↪":false,"inputs":[{"name":"_serverIndex","type":"uint256"},{"name":"_props","type":
↪"uint256"}],"name":"updateServer","outputs":[],"payable":true,"stateMutability":
↪"payable","type":"function"},{"constant":false,"inputs":[{"name":"_url","type":
↪"string"},{"name":"_props","type":"uint256"}],"name":"registerServer","outputs":[],
↪"payable":true,"stateMutability":"payable","type":"function"},{"constant":true,
↪"inputs":[{"name":"","type":"uint256"}],"name":"servers","outputs":[{"name":"url",
↪"type":"string"},{"name":"owner","type":"address"},{"name":"deposit","type":"uint256
↪"},{"name":"props","type":"uint256"},{"name":"unregisterTime","type":"uint128"},{
↪"name":"unregisterDeposit","type":"uint128"},{"name":"unregisterCaller","type":
↪"address"}],"payable":false,"stateMutability":"view","type":"function"},{"constant
↪":false,"inputs":[{"name":"_serverIndex","type":"uint256"}],"name":
↪"cancelUnregisteringServer","outputs":[],"payable":false,"stateMutability":
↪"nonpayable","type":"function"},{"constant":false,"inputs":[{"name":"_serverIndex",
↪"type":"uint256"},{"name":"_blockhash","type":"bytes32"},{"name":"_blocknumber",
↪"type":"uint256"},{"name":"_v","type":"uint8"},{"name":"_r","type":"bytes32"},{"name
↪":"_s","type":"bytes32"}],"name":"convict","outputs":[],"payable":false,
↪"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[{"name
↪":"_serverIndex","type":"uint256"}],"name":"calcUnregisterDeposit","outputs":[{"name
↪":"","type":"uint128"}],"payable":false,"stateMutability":"view","type":"function"},
↪{"constant":false,"inputs":[{"name":"_serverIndex","type":"uint256"}],"name":
↪"confirmUnregisteringServer","outputs":[],"payable":false,"stateMutability":
↪"nonpayable","type":"function"},{"constant":false,"inputs":[{"name":"_serverIndex",
↪"type":"uint256"}],"name":"requestUnregisteringServer","outputs":[],"payable":true,
↪"stateMutability":"payable","type":"function"},{"anonymous":false,"inputs":[{
↪"indexed":false,"name":"url","type":"string"},{"indexed":false,"name":"props","type
↪":"uint256"},{"indexed":false,"name":"owner","type":"address"},{"indexed":false,
↪"name":"deposit","type":"uint256"}],"name":"LogServerRegistered","type":"event"},{
↪"anonymous":false,"inputs":[{"indexed":false,"name":"url","type":"string"},{"indexed
↪":false,"name":"owner","type":"address"},{"indexed":false,"name":"caller","type":
↪"address"}],"name":"LogServerUnregisterRequested","type":"event"},{"anonymous
↪":false,"inputs":[{"indexed":false,"name":"url","type":"string"},{"indexed":false,
↪"name":"owner","type":"address"}],"name":"LogServerUnregisterCanceled","type":"event
↪"},{"anonymous":false,"inputs":[{"indexed":false,"name":"url","type":"string"},{
↪"indexed":false,"name":"owner","type":"address"}],"name":"LogServerConvicted","type
↪":"event"},{"anonymous":false,"inputs":[{"indexed":false,"name":"url","type":"string
↪"},{"indexed":false,"name":"owner","type":"address"}],"name":"LogServerRemoved",
↪"type":"event"}]
```


To run an Incubed node, you simply use docker-compose:

```yaml
version: '2'
services:
  incubed-server:
    image: slockit/in3-server:latest
    volumes:
    - $PWD/keys:/secure                                     # directory where the private key is stored
    ports:
    - 8500:8500/tcp                                         # open the port 8500 to be accessed by the public
    command:
    - --privateKey=/secure/myKey.json                       # internal path to the key
    - --privateKeyPassphrase=dummy                          # passphrase to unlock the key
    - --chain=0x1                                           # chain (Kovan)
    - --rpcUrl=http://incubed-parity:8545                   # URL of the Kovan client
    - --registry=0xFdb0eA8AB08212A1fFfDB35aFacf37C3857083ca # URL of the Incubed registry
    - --autoRegistry-url=http://in3.server:8500             # check or register this node for this URL
    - --autoRegistry-deposit=2                              # deposit to use when registering

  incubed-parity:
    image: slockit/parity-in3:v2.2                          # parity-image with the getProof-function implemented
    command:
    - --auto-update=none                                    # do not automatically update the client
    - --pruning=archive
    - --pruning-memory=30000                                # limit storage
```

CHAPTER 2

Downloading in3

in3 is divided into two distinct components, the in3-node and in3-client. The in3-node is currently written in typescript, whereas the in3-client has a version in typescript as well as a smaller and more feature packed version written in C.

In order to compile from scratch, please use the sources from our github page or the public gitlab page. Instructions for building from scratch can be found in our documentation.

The in3-server and in3-client has been published in multiple package managers and locations, they can be found here:

	Package manager	Link	Use case
in3-node(ts)	Docker Hub	DockerHub	To run the in3-server, which the in3-client can use to connect to the in3 network
in3-client(ts)	NPM	NPM	To use with js applications
in3-client(C)	Ubuntu Launchpad	Ubuntu	It can be quickly integrated on linux systems, IoT devices or any micro controllers
	Docker Hub	DockerHub	Quick and easy way to get in3 client running
	Brew	Homebrew	Easy to install on MacOS or linux/windows subsystems
	Release page	Github	For directly playing with the binaries/deb/jar/wasm files

2.1 in3-node

2.1.1 Docker Hub

1. Pull the image from docker using `docker pull slockit/in3-node`
2. In order to run your own in3-node, you must first register the node. The information for registering a node can be found here

3. Run the in3-node image using a direct docker command or a docker-compose file, the parameters for which are explained here

2.2 in3-client (ts)

2.2.1 npm

1. Install the package by running `npm install --save in3`
2. `import In3Client from "in3"`
3. View our examples for information on how to use the module

2.3 in3-client(C)

2.3.1 Ubuntu Launchpad

There are 2 packages published to Ubuntu Launchpad: `in3` and `in3-dev`. The package `in3` only installs the binary file and allows you to use in3 via command line. The package `in3-dev` would install the binary as well as the library files, allowing you to use in3 not only via command line, but also inside your C programs by including the statically linked files.

Installation instructions for `in3`:

This package will only install the in3 binary in your system.

1. Add the slock.it ppa to your system with `sudo add-apt-repository ppa:devops-slock-it/in3`
2. Update the local sources `sudo apt-get update`
3. Install in3 with `sudo apt-get install in3`

Installation instructions for `in3-dev`:

This package will install the statically linked library files and the include files in your system.

1. Add the slock.it ppa to your system with `sudo add-apt-repository ppa:devops-slock-it/in3`
2. Update the local sources `sudo apt-get update`
3. Install in3 with `sudo apt-get install in3-dev`

2.3.2 Docker Hub

Usage instructions:

1. Pull the image from docker using `docker pull slockit/in3`
2. Run the client using: `docker run -d -p 8545:8545 slockit/in3:latest --chainId=goerli -port 8545`
3. More parameters and their descriptions can be found here.

2.3.3 Release page

Usage instructions:

1. Navigate to the in3-client release page on this github repo
2. Download the binary that matches your target system, or read below for architecture specific information:

For WASM:

1. Download the WASM binding with `npm install --save in3-wasm`
2. More information on how to use the WASM binding can be found here
3. Examples on how to use the WASM binding can be found here

For C library:

1. Download the C library from the release page or by installing the `in3-dev` package from ubuntu launchpad
2. Include the C library in your code, as shown in our examples
3. Build your code with `gcc -std=c99 -o test test.c -lin3 -lcurl`, more information can be found here

For Java:

1. Download the Java file from the release page
2. Use the java binding as show in our example
3. Build your java project with `javac -cp $IN3_JAR_LOCATION/in3.jar *.java`

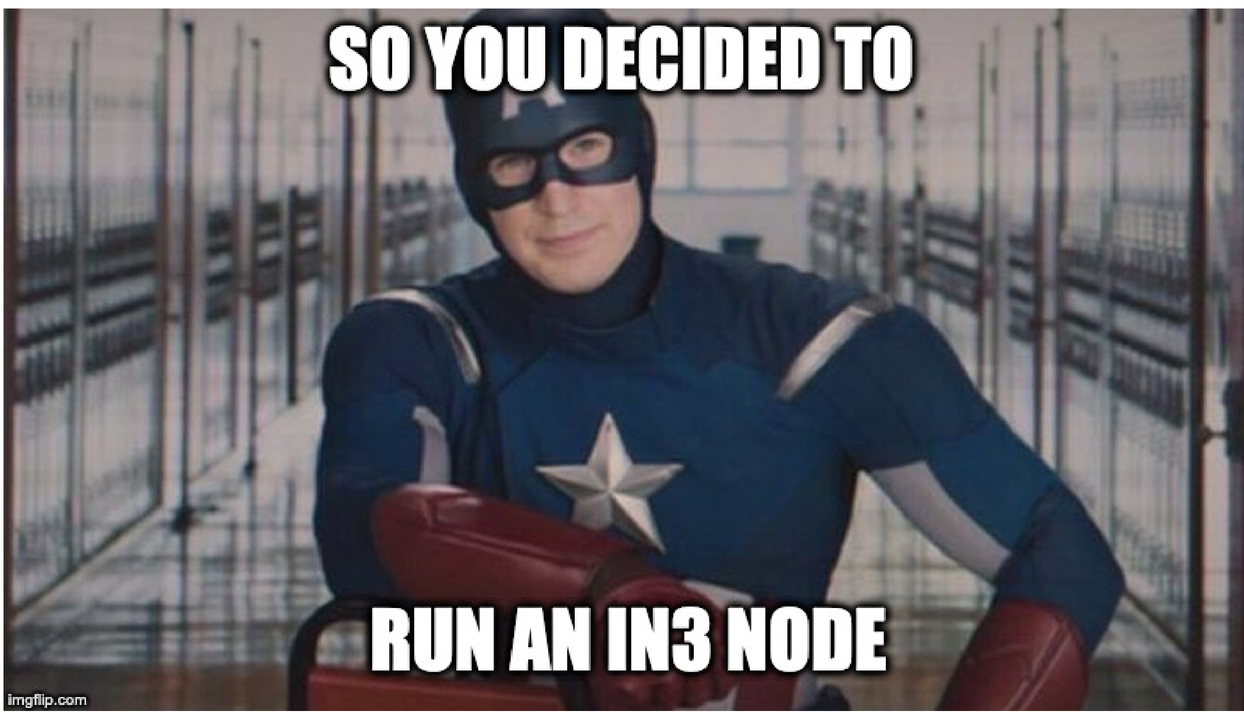
2.3.4 Brew

Usage instructions:

1. Ensure that homebrew is installed on your system
2. Add a brew tap with `brew tap slockit/in3`
3. Install in3 with `brew install in3`
4. You should now be able to use `in3` in the terminal, can be verified with `in3 eth_blockNumber`

CHAPTER 3

Running an in3 node on a VPS



img

Disclaimers: This guide is meant to give you a general idea of the steps needed to run an in3 node on a VPS, please do not take it as a definitive source for all the information. An in3 node is a public facing service that comes with all the associated security implications and complexity. This guide is meant for internal use at this time, once a target audience and depth has been defined, a public version will be made.

That being said, setup of an in3 node requires the following steps:

```
1. Generate a private key and docker-compose file from in3-setup.slock.it
2. Setup a VPS
3. Start the Ethereum RPC node using the docker-compose
```

(continues on next page)

(continued from previous page)

```
4. Assign a DNS domain, static IP (or Dynamic DNS) to the server
5. Run the in3 node docker image with the required flags
6. Register the in3 node with in3-setup.slock.it
```

1. Generate a private key and docker-compose file using in3-setup.slock.it: We will use the in3-setup tool to guide us through the process of starting an incubed node. Begin by filling up the required details, add metadata if you improve our statistics. Choose the required chain and logging level. Choose a secure private key passphrase, it is important to save it in your password manager or somewhere secure, we cannot recover it for you. Click on generate private key, this process takes some time. Download the private key and store it in the secure location.

Once the private key is downloaded, enter your Ethereum node URL in case you already have one. Generate the docker-compose file and save it in the same folder as the private key.

1. Setup a VPS:

A VPS is basically a computer away from home that offers various preselected (usually) Linux distros out of the box. You can then set it up with any service you like - for example Hetzner,Contabo,etc. ServerHunter is a good comparison portal to find a suitable VPS service.The minimum specs required for a server to host both an ethereum RPC node as well as an in3 node would be:

```
4 CPU cores
8GB of Ram
300GB SSD disk space or more
Atleast 5MBit/s up/down
Linux OS, eg: Ubuntu
```

Once the server has been provisioned, look for the IP address,SSH port and username. This information would be used to login,transfer files to the VPS.

Transfer the files to the server using a file browser or an scp command. The target directory for docker-compose.yml and exported-private.key.json file on the incubed server is the /int3 directory The scp command to transfer the files are:

```
scp docker-compose.yml user@ip-address:
scp exported-private-key.json user@ip-address:
```

If you are using windows you should use Winscp. Copy it to your home directory and thean move the files to /int3

Once the files have been transferred, we will SSH into the server with:

```
ssh username@ip-address
```

Now we will install the dependencies required to run in3. This is possible through a one step install script that can be found (here)[https://github.com/slockit/in3-server-setup-tool/blob/master/incubed_dependency_install_script.sh] or by installing each dependency individually.

If you wish to use our dependency install script, please run the following commands in your VPS, then skip to step 4 and setup your domain name:

```
curl -o incubed_dependency_install_script.sh https://raw.githubusercontent.com/
↪slockit/in3-server-setup-tool/master/incubed_dependency_install_script.sh
chmod +x incubed_dependency_install_script.sh
sudo su
./incubed_dependency_install_script.sh
```

If you wish to install each dependency individually, please follow the proceeding steps. Begin by removing older installations of docker:

```
# remove existing docker installations
sudo apt remove docker docker-engine docker.io
```

Make sure you have the necessary packages to allow the use of Docker's repository:

```
# install dependencies
sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

To verify the hashes of the docker images from dockerhub you must add Docker's GPG key:

```
# add the docker gpg key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Verify the fingerprint of the GPG key, the UID should say "Docker Release":

```
# verify the gpg key
sudo apt-key fingerprint 0EBFCD88
```

Add the stable Docker repository:

```
# add the stable Docker repository
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
↪$(lsb_release -cs) stable"
```

Update and install docker-ce:

```
# update the sources
sudo apt update
# install docker-ce
sudo apt install docker-ce
```

Add your limited Linux user account to the docker group:

```
# add your limited Linux user account to the docker group
sudo usermod -aG docker $USER
```

Verify your installation with a hello-world image:

```
docker run hello-world
```

Now we will continue to install docker-compose by downloading it and moving it to the right location:

```
# install docker-compose
sudo curl -L https://github.com/docker/compose/releases/download/1.18.0/docker-
↪compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

Set the right permissions:

```
# set the right permissions
sudo chmod +x /usr/local/bin/docker-compose
```

Verify the installation with:

```
docker-compose --version
```

1. Start the Ethereum RPC node using the docker-compose: We will use the downloaded docker-compose file to start the Ethereum RPC node.

Change directory to the created in3 folder, verify that the files exist there and then start parity with:

```
screen
docker-compose up incubed-parity
control+A and control+D to exit from screen
```

The time for the whole sync with parity is nearly 4h. The sync process starts with Block snapshots. After This is ready the block syncing starts. In order to verify the status of the syncing, run:

```
echo $((`curl --data '{"method":"eth_blockNumber","params":[],"id":1,"jsonrpc":"2.0"}
↪' -H "Content-Type: application/json" -X POST 172.15.0.3:8545 | grep -oh "\w*0x\w*
↪"`))
```

That command will return the latest block number, verify that the block number is the latest one by checking on etherscan. We recommend to go forward with Step 4. if sync is completly finished.

1. Run the in3 node docker image with the required flags Once the Ethereum RPC node has been synced, we can proceed with starting the in3-node. This can also be done with the docker-compose file that we used earlier.

```
docker-compose up incubed-server
```

Wait for the in3-server to finish starting, then run the below command to verify the functioning of the in3-server:

```
echo $((`curl --data '{"method":"eth_blockNumber","params":[],"id":1,"jsonrpc":"2.0"}
↪' -H "Content-Type: application/json" -X POST 172.15.0.2:8500 | grep -oh "\w*0x\w*
↪"`))
```

You can now type "exit" to end the SSH session, we should be done with the setup stages in the VPS.

1. Assign a DNS domain, static IP (or Dynamic DNS) to the server You need to register a DNS domain name using cloudflare or some other DNS provider. This Domain name needs to point to your server. A simple way to test it once it is up is with the following command run from your computer:

```
echo $((`curl --data '{"method":"eth_blockNumber","params":[],"id":1,"jsonrpc":"2.0"}
↪' -H "Content-Type: application/json" -X POST Domain-name:80 | grep -oh "\w*0x\w*
↪"`))
```

1. Setup https for your domain

a) Install nginx and certbot and generate certificates.

```
sudo apt-get install certbot nginx
sudo certbot certonly --standalone
# check if automatic renewal of the certificates works as expected
sudo certbot renew --dry-run
```

b) Configure nginx as a reverse proxy using SSL. Replace /etc/nginx/sites/available/default with the following content. (Comment everything else out, also the certbot generated stuff.)

```
server {
        listen 443 default_server;
        server_name Domain-name;
        ssl on;
        ssl_certificate /etc/letsencrypt/live/Domain-name/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/Domain-name/privkey.pem;
        ssl_session_cache shared:SSL:10m;

        location / {
```

(continues on next page)

(continued from previous page)

```
                proxy_pass http://localhost:80;
                proxy_set_header Host $host;

                proxy_redirect http:// https://;
        }
}
```

c) Restart nginx.

```
sudo service nginx restart
```

HTTPS should be working now. Check with:

```
echo $((`curl --data '{"method":"eth_blockNumber","params":[],"id":1,"jsonrpc":"2.0"}
↪' -H "Content-Type: application/json" -X POST Domain-name:443 | grep -oh "\w*0x\w*
↪"`))
```

1. Register the in3 node with in3-setup.slock.it Lastly, we need to head back to in3-setup.slock.it and register our new node. Enter the URL address from which the in3 node can be reached. Add the deposit amount in Ether and click on "Register in3 server" to send the transaction.

3.1 Side notes/ chat summary

1. Redirect HTTP to HTTPS

Using the above config file nginx doesn't listen on port 80, that port is already being listened to by the incubed-server image (see docker-compose file, mapping 80:8500). That way the port is open for normal HTTP requests and when registering the node one can "check" the HTTP capability. If that is unwanted one can append

```
server {
    listen 80;
    return 301 https://$host$request_uri;
}
```

to the nginx config file and change the port mapping for the incubed-server image. One also needs then to adjust the port that nginx redirects to on localhost. For example

```
        ports:
        - 8080:8500/tcp
```

In the incubed-server section in the docker compose file and

```
                proxy_pass http://localhost:8080;
```

in the nginx config. (Port 8080 also has to be closed using the firewall, e.g. `ufw deny 8080`)

1. OOM - Out of memory

If having memory issues while syncing adding some parity flags might help (need to be added in the docker-compose for incubed-parity)

```
    --pruning-history=[NUM]
        Set a minimum number of recent states to keep in memory when pruning is␣
↪active. (default: 64)
```

(continues on next page)

(continued from previous page)

```
    --pruning-memory=[MB]
        The ideal amount of memory in megabytes to use to store recent states. As␣
↪many states as possible will be kept
        within this limit, and at least --pruning-history states will always be kept.␣
↪(default: 32)
```

with appropiate values. Note that inside the docker compose file pruning-memory is set to 30000, which might exceed your RAM!

1. Saving the chaindb on disk using docker volume

To prevent the chaindb data being lost add

```
        volumes:
        - /wherever-you-want-to-store-data/:/home/parity/.local/share/io.parity.
↪ethereum/
```

to the parity section in the docker compose file.

1. Added stability/ speed while syncing

Exposing the port 30303 to the public will prevent parity having to rely on UPnP for node discovery. For this add

```
        ports:
        - 30303:30303
        - 30303:30303/udp
```

to the parity section in the docker compose file.

Increasing the database, state and queuing cache can improve the syncing speed (default is around 200MB). The needed flag for it is:

```
    --cache-size=[MB]
        Set total amount of discretionary memory to use for the entire system,␣
↪overrides other cache and queue options.
```

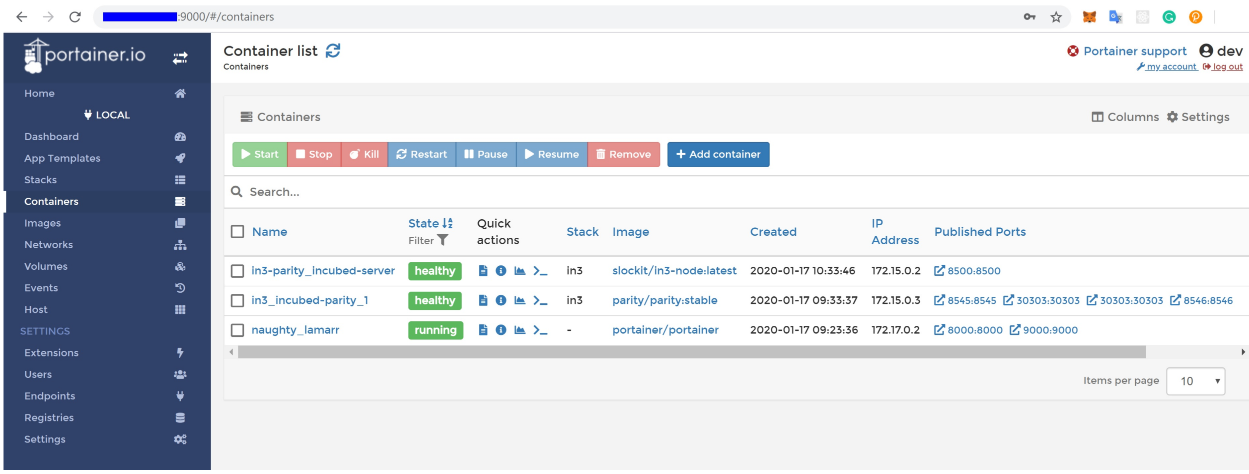
1. If you like a UI to manage and check your docker containers, please have a look at Portainer.io

Installation instructions can be found here: https://www.portainer.io/installation/.

It can be run with docker, using:

```
sudo docker run -d --restart always -p 8000:8000 -p 9000:9000 -v /var/run/docker.
↪sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer
```

After the setup, it will be availabe on port 9000. The enabled WebGUI looks like the below picture:



img

3.2 Recommendations

1. Disable SSH `PasswordAuthentication` & `RootLogin` and install `fail2ban` to protect your VPS from unauthorized access and brute-force attacks. See How To Configure SSH Key-Based Authentication on a Linux Server and How To Protect SSH with Fail2Ban.

CHAPTER 4

IN3-Protocol

This document describes the communication between a Incubed client and a Incubed node. This communication is based on requests that use extended JSON-RPC-Format. Especially for ethereum-based requests, this means each node also accepts all standard requests as defined at Ethereum JSON-RPC, which also includes handling Bulk-requests.

Each request may add an optional `in3` property defining the verification behavior for Incubed.

4.1 Incubed Requests

Requests without an `in3` property will also get a response without `in3`. This allows any Incubed node to also act as a raw ethereum JSON-RPC endpoint. The `in3` property in the request is defined as the following:

- **chainId** `string<hex>` - The requested *chainId*. This property is optional, but should always be specified in case a node may support multiple chains. In this case, the default of the node would be used, which may end up in an undefined behavior since the client cannot know the default.
- **includeCode** `boolean` - Applies only for `eth_call`-requests. If true, the request should include the codes of all accounts. Otherwise only the the codeHash is returned. In this case, the client may ask by calling eth_getCode() afterwards.
- **verifiedHashes** `string<bytes32>[]` - If the client sends an array of blockhashes, the server will not deliver any signatures or blockheaders for these blocks, but only return a string with a number. This allows the client to skip requiring signed blockhashes for blocks already verified.
- **latestBlock** `integer` - If specified, the blocknumber `latest` will be replaced by a blockNumber-specified value. This allows the Incubed client to define finality for PoW-Chains, which is important, since the `latest`-block cannot be considered final and therefore it would be unlikely to find nodes willing to sign a blockhash for such a block.
- **useRef** `boolean` - If true, binary-data (starting with a 0x) will be referred if occurring again. This decreases the payload especially for recurring data such as merkle proofs. If supported, the server (and client) will keep track of each binary value storing them in a temporary array. If the previously used value is used again, the server replaces it with `:<index>`. The client then resolves such refs by lookups in the temporary array.

- **useBinary** `boolean` - If true, binary-data will be used. This format is optimzed for embedded devices and reduces the payload to about 30%. For details see *the Binary-spec*.
- **useFullProof** `boolean` - If true, all data in the response will be proven, which leads to a higher payload. The result depends on the method called and will be specified there.
- **finality** `number` - For PoA-Chains, it will deliver additional proof to reach finality. If given, the server will deliver the blockheaders of the following blocks until at least the number in percent of the validators is reached.
- **verification** `string` - Defines the kind of proof the client is asking for. Must be one of the these values:
  - `'never`' : No proof will be delivered (default). Also no `in3`-property will be added to the response, but only the raw JSON-RPC response will be returned.
  - `'proof`' : The proof will be created including a blockheader, but without any signed blockhashes.
- **preBIP34** `boolean` - Defines if the client wants to verify blocks before BIP34 (height < 227836). If true, the `proof`-section will include data to verify the existence and correctness of *old* blocks as well (before BIP34).
- **whiteList** `address` - If specified, the incubed server will respond with `lastWhiteList`, which will indicate the last block number of whitelist contract event.
- **signers** `string<address>[]` - A list of addresses (as 20bytes in hex) requested to sign the blockhash.

A example of an Incubed request may look like this:

```
{
    "jsonrpc": "2.0",
    "id": 2,
    "method": "eth_getTransactionByHash",
    "params": ["0xf84cfb78971ebd940d7e4375b077244e93db2c3f88443bb93c561812cfed055c"],
    "in3": {
        "chainId": "0x1",
        "verification": "proof",
        "whiteList": "0x08e97ef0a92EB502a1D7574913E2a6636BeC557b",
        "signers":["0x784bfa9eb182C3a02DbeB5285e3dBa92d717E07a"]
  }
}
```

4.2 Incubed Responses

Each Incubed node response is based on JSON-RPC, but also adds the `in3` property. If the request does not contain a `in3` property or does not require proof, the response must also omit the `in3` property.

If the proof is requested, the `in3` property is defined with the following properties:

- **proof** *Proof* - The Proof-data, which depends on the requested method. For more details, see the *Proofs* section.
- **lastNodeList** `number` - The blocknumber for the last block updating the nodeList. This blocknumber should be used to indicate changes in the nodeList. If the client has a smaller blocknumber, it should update the nodeList.
- **lastValidatorChange** `number` - The blocknumber of the last change of the validatorList (only for PoA-chains). If the client has a smaller number, it needs to update the validatorlist first. For details, see *PoA Validations*
- **lastWhiteList** `number` - The blocknumber for the last block updating the whitelist nodes in whitelist contract. This blocknumber could be used to detect if there is any change in whitelist nodes. If the client has a smaller blocknumber, it should update the white list.
- **currentBlock** `number` - The current blocknumber. This number may be stored in the client in order to run sanity checks for `latest` blocks or `eth_blockNumber`, since they cannot be verified directly.

An example of such a response would look like this:


```
{
  "jsonrpc": "2.0",
  "result": {
    "blockHash": "0x2dbbac3abe47a1d0a7843d378fe3b8701ca7892f530fd1d2b13a46b202af4297",
    "blockNumber": "0x79fab6",
    "chainId": "0x1",
    "condition": null,
    "creates": null,
    "from": "0x2c5811cb45ba9387f2e7c227193ad10014960bfc",
    "gas": "0x186a0",
    "gasPrice": "0x4a817c800",
    "hash": "0xf84cfb78971ebd940d7e4375b077244e93db2c3f88443bb93c561812cfed055c",
    "input":
↪"0xa9059cbb000000000000000000000000290648fc6f2cb27a2a81dc35a429090872991b920000000000000000000000000
↪",
    "nonce": "0xa8",
    "publicKey":
↪"0x6b30c392dda89d58866bf2c1bedf8229d12c6ae3589d82d0f52ae588838a475aacda64775b7a1b376935d732bb80226
↪",
    "r": "0x4666976b528fc7802edd9330b935c7d48fce0144ce97ade8236da29878c1aa96",
    "raw":
↪"0xf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb00000000000
↪",
    "s": "0x5089dca7ecf7b061bec3cca7726aab1fcb4c8beb51517886f91c9b0ca710b09d",
    "standardV": "0x0",
    "to": "0xd3ebdaea9aeac98de723f640bce4aa07e2e44192",
    "transactionIndex": "0x3e",
    "v": "0x25",
    "value": "0x0"
  },
  "id": 2,
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block":
↪"0xf90219a03d050deecd980b16cad9752133333ccdface463cc69e784f32dd981e2e751e34a01dcc4de8dec75d7aab85b
↪",
      "merkleProof": [

↪"0xf90131a00150ff50e29f3df34b89870f183c85a82a73f21722d7e6c787e663159f165010a0b8c56f207a223067c7ae5
↪",

↪"0xf90211a0f4a5e4a1197190f910e4a026f50bd6a169716b52be42c99ddb043ad9b4da6117a09ad1def70dd1d991331d0
↪",

↪"0xf8b020b8adf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb0
↪"
      ],
      "txIndex": 62,
      "signatures": [
        {
          "blockHash":
↪"0x2dbbac3abe47a1d0a7843d378fe3b8701ca7892f530fd1d2b13a46b202af4297",
          "block": 7994038,
          "r": "0xef73a527ae8d38b595437e6436bd4fa037d50550bf3840ad0cd3c6ca641a951e",
          "s": "0x6a5815db16c12b890347d42c014d19b60e1605d2e8e64b729f89e662f9ce706b",
```


(continues on next page)

(continued from previous page)


```
          "v": 27,
          "msgHash":
↪"0xa8fc6e2564e496efc5fd7db8e70f03fd50af53e092f47c98329c84c96026fdff"
        }
      ]
    },
    "currentBlock": 7994124,
    "lastValidatorChange": 0,
    "lastNodeList": 6619795,
    "lastWhiteList": 1546354
  }
}
```


4.3 ChainId

Incubed supports multiple chains and a client may even run requests to different chains in parallel. While, in most cases, a chain refers to a specific running blockchain, chainIds may also refer to abstract networks such as ipfs. So, the definition of a chain in the context of Incubed is simply a distributed data domain offering verifiable api-functions implemented in an in3-node.

Each chain is identified by a `uint64` identifier written as hex-value (without leading zeros). Since incubed started with ethereum, the chainIds for public ethereum-chains are based on the intrinsic chainId of the ethereum-chain. See https://chainid.network.

For each chain, Incubed manages a list of nodes as stored in the *server registry* and a chainspec describing the verification. These chainspecs are held in the client, as they specify the rules about how responses may be validated.

4.4 Registry

As Incubed aims for fully decentralized access to the blockchain, the registry is implemented as an ethereum smart contract.

This contract serves different purposes. Primarily, it manages all the Incubed nodes, both the onboarding and also unregistering process. In order to do so, it must also manage the deposits: reverting when the amount of provided ether is smaller than the current minimum deposit; but also locking and/or sending back deposits after a server leaves the in3-network.

In addition, the contract is also used to secure the in3-network by providing functions to "convict" servers that provided a wrongly signed block, and also having a function to vote out inactive servers.

4.4.1 Register and Unregister of nodes

Register

There are two ways of registering a new node in the registry: either calling [`registerNode()`][registerNode] or by calling [`registerNodeFor()`][registerNodeFor]. Both functions share some common parameters that have to be provided:

- `url` the url of the to be registered node
- `props` the properties of the node

- `weight` the amount of requests per second the node is capable of handling
- `deposit` the deposit of the node in ERC20 tokens.

Those described parameters are sufficient when calling [`registerNode()`][registerNode] and will register a new node in the registry with the sender of the transaction as the owner. However, if the designated signer and the owner should use different keys, [`registerNodeFor()`][registerNodeFor] has to be called. In addition to the already described parameters, this function also needs a certain signature (i.e. `v`, `r`, `s`). This signature has to be created by hashing the url, the properties, the weight and the designated owner (i.e. `keccack256(url,properties, weight,owner)`) and signing it with the privateKey of the signer. After this has been done, the owner then can call [`registerNodeFor()`][registerNodeFor] and register the node.

However, in order for the register to succeed, at least the correct amount of deposit has to be approved by the designated owner of the node. The supported token can be received by calling [`supportedToken()`][supportedToken] the registry contract. The same approach also applied to the minimal amount of tokens needed for registering by calling [`minDeposit()`][minDeposit].

In addition to that, during the first year after deployment there is also a maximum deposit for each node. This can be received by calling [`maxDepositFirstYear()`][maxDepositFirstYear]. Providing a deposit greater then this will result in a failure when trying to register.

Unregister a node

In order to remove a node from the registry, the function [`unregisteringNode()`][unregisteringNode] can be used, but is only callable by the owner the node.

While after a successful call the node will be removed from the nodeList immediately, the deposit of the former node will still be locked for the next 40 days after this function had been called. After the timeout is over, the function [`returnDeposit()`][returnDeposit] can be called in order to get the deposit back. The reason for that decision is simple: this approach makes sure that there is enough time to convict a malicious node even after he unregistered his node.

4.4.2 Convicting a node

After a malicious node signed a wrong blockhash, he can be convicted resulting in him loosing the whole deposit while the caller receives 50% of the deposit. There are two steps needed for the process to succeed: calling [`convict()`][convict] and [`revealConvict()`][revealConvict].

calling convict

The first step for convicting a malicious node is calling the [`convict()`][convict]-function. This function will store a specific hash within the smart contract.

The hash needed for convicting requires some parameters:

- `blockhash` the wrongly blockhash that got signed the by malicious node
- `sender` the account that sends this transaction
- `v` v of the signature of the wrong block
- `r` r of the signature of the wrong block
- `s` s of the signature of the wrong block

All those values are getting hashed (`keccack256(blockhash,sender,v,r,s`) and are stored within the smart contract.

calling revealConvcit

This function requires that at least 2 blocks have passed since [`convict()`][convict] was called. This mechanic reduces the risks of successful frontrunning attacks.

In addition, there are more requirements for successfully convicting a malicious node:

- the blocknumber of the wrongly signed block has to be either within the latest 256 blocks or be stored within the BlockhashRegistry.
- the malicious node provided a signature for the wong block and it was signed by the node
- the specific hash of the convict-call can be recreated (i.e. the caller provided the very same parameters again)
- the malicious node is either currently active or did not withdraw his deposit yet

If the [`revealConvict()`][revealConvict]-call passes, the malicious node will be removed immediately from the nodeList. As a reward for finding a malicious node the caller receives 50% of the deposit of the malicious node. The remaining 50% will stay within the nodeRegistry, but nobody will be able to access/transfer them anymore.

recreating blockheaders

When a malicious node returns a block that is not within the latest 256 blocks, the BlockhashRegistry has to be used.

There are different functions to store a blockhash and its number in the registry:

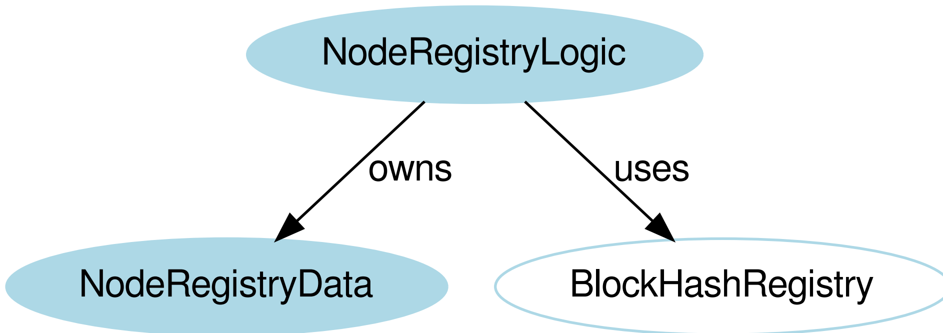
- [`snapshot`][snapshot] stores the blockhash and its number of the previous block
- [`saveBlockNumber`][saveBlockNumber] stores a blockhash and its number from the latest 256 blocks
- [`recreateBlockheaders`][recreateBlockheaders] starts from an already stored block and recreates a chain of blocks. Stores the last block at the end.

In order to reduce the costs of convicting, both [`snapshot`][snapshot] and [`saveBlockNumber`][saveBlockNumber] are the cheapest options, but are limited to the latest 256 blocks.

Recreating a chain of blocks is way more expensive, but is provides the possibility to recreate way older blocks. It requires the blocknumber of an already stored hash in the smart contract as first parameter. As a second parameter an array of serialized blockheaders have to be provided. This array has to start with the blockheader of the stored block and then the previous blockheaders in reverse order (e.g. `100,99,98`). The smart contract will try to recreate the chain by comparing both the provided (hashed) headers with the calculated parent and also by comparing the extracted blocknumber with the calculated one. After the smart contracts successfully recreates the provided chain, the blockhash of the last element gets stored within the smart contract.

4.4.3 Updating the NodeRegistry

In ethereum the deployed code of an already existing smart contract cannot be changed. This means, that as soon as the Registry smart contract gets updated, the address would change which would result in changing the address of the smart contract containing the nodeList in each client and device.



In order to solve this issue, the registry is divided between two different deployed smart contracts:

- `NodeRegistryData`: a smart contract to store the nodeList
- `NodeRegistryLogic`: a smart contract that has the logic needed to run the registry

There is a special relationship between those two smart contracts: The NodeRegistryLogic "owns" the NodeRegistryData. This means, that only he is allowed to call certain functions of the NodeRegistryData. In our case this means all writing operations, i.e. he is the only entity that is allowed to actually be allowed to store data within the smart contract. We are using this approach to make sure that only the NodeRegistryLogic can call the register, update and remove functions of the NodeRegistryData. In addition, he is the only one allowed to change the ownership to a new contract. Doing so results in the old NodeRegistryLogic to lose write access.

In the NodeRegistryLogic there are 2 special parameters for the update process:

- `updateTimeout`: a timestamp that defines when it's possible to update the registry to the new contract
- `pendingNewLogic`: the address of the already deployed new NodeRegistryLogic contract for the updated registry

When an update of the Registry is needed, the function `adminUpdateLogic` gets called by the owner of the NodeRegistryLogic. This function will set the address of the new pending contract and also set a timeout of 47 days until the new logic can be applied to the NodeRegistryData contract. After 47 days everyone is allowed to call `activateNewLogic` resulting in an update of the registry.

The timeout of accessing the deposit of a node after removing it from the nodeList is only 40 days. In case a node owner dislikes the pending registry, he has 7 days to unregister in order to be able to get his deposit back before the new update can be applied.

4.4.4 Node structure

Each Incubed node must be registered in the NodeRegistry in order to be known to the network. A node or server is defined as:

- **url** `string` - The public url of the node, which must accept JSON-RPC requests.
- **owner** `address` - The owner of the node with the permission to edit or remove the node.
- **signer** `address` - The address used when signing blockhashes. This address must be unique within the nodeList.

- **timeout** `uint64` - Timeout after which the owner is allowed to receive its stored deposit. This information is also important for the client, since an invalid blockhash-signature can only "convict" as long as the server is registered. A long timeout may provide higher security since the node can not lie and unregister right away.
- **deposit** `uint256` - The deposit stored for the node, which the node will lose if it signs a wrong blockhash.
- **props** `uint192` - A bitmask defining the capabilities of the node:
  - **proof** (`0x01`) : The node is able to deliver proof. If not set, it may only serve pure ethereum JSON/RPC. Thus, simple remote nodes may also be registered as Incubed nodes.
  - **multichain** (`0x02`) : The same RPC endpoint may also accept requests for different chains. if this is set the `chainId`-prop in the request in required.
  - **archive** (`0x04`) : If set, the node is able to support archive requests returning older states. If not, only a pruned node is running.
  - **http** (`0x08`) : If set, the node will also serve requests on standard http even if the url specifies https. This is relevant for small embedded devices trying to save resources by not having to run the TLS.
  - **binary** (`0x10`) : If set, the node accepts request with `binary:true`. This reduces the payload to about 30% for embedded devices.
  - **onion** (`0x20`) : If set, the node is reachable through onionrouting and url will be a onion url.
  - **signer** (`0x40`) : If set, the node will sign blockhashes.
  - **data** (`0x80`) : If set, the node will provide rpc responses (at least without proof).
  - **stats** (`0x100`) : If set, the node will provide and endpoint for delivering metrics, which is usually the `/metrics`- endpoint, which can be used by prometheus to fetch statistics.
  - **minBlockHeight** (`0x0100000000` - `0xFF00000000`): : The min number of blocks this node is willing to sign. if this number is low (like <6) the risk of signing unindentially a wrong blockhash because of reorgs is high. The default should be 10)

    ```
    minBlockHeight = props >> 32 & 0xFF
    ```

  More capabilities will be added in future versions.
- **unregisterTime** `uint64` - The earliest timestamp when the node can unregister itself by calling `confirmUnregisteringServer`. This will only be set after the node requests an unregister. The client nodes with an `unregisterTime` set have less trust, since they will not be able to convict after this timestamp.
- **registerTime** `uint64` - The timestamp, when the server was registered.
- **weight** `uint64` - The number of parallel requests this node may accept. A higher number indicates a stronger node, which will be used within the incentivization layer to calculate the score.

4.5 Binary Format

Since Incubed is optimized for embedded devices, a server can not only support JSON, but a special binary-format. You may wonder why we don't want to use any existing binary serialization for JSON like CBOR or others. The reason is simply: because we do not need to support all the features JSON offers. The following features are not supported:

- no escape sequences (this allows use of the string without copying it)
- no float support (at least for now)
- no string literals starting with `0x` since this is always considered as hexcoded bytes

- no propertyNames within the same object with the same key hash

Since we are able to accept these restrictions, we can keep the JSON-parser simple. This binary-format is highly optimized for small devices and will reduce the payload to about 30%. This is achieved with the following optimizations:

- All strings starting with `0x`are interpreted as binary data and stored as such, which reduces the size of the data to 50%.
- Recurring byte-values will use references to previous data, which reduces the payload, especially for merkle proofs.
- All propertyNames of JSON-objects are hashed to a 16bit-value, reducing the size of the data to a signifivant amount (depending on the propertyName).

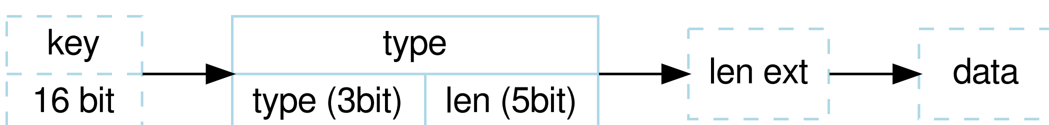
  The hash is calculated very easily like this:

```
static d_key_t key(const char* c) {
  uint16_t val = 0, l = strlen(c);
  for (; l; l--, c++) val ^= *c | val << 7;
  return val;
}
```

Note: A very important limitation is the fact that property names are stored as 16bit hashes, which decreases the payload, but does not allow for the restoration of the full json without knowing all property names!

The binary format is based on JSON-structure, but uses a RLP-encoding approach. Each node or value is represented by these four values:

- **key** `uint16_t` - The key hash of the property. This value will only pass before the property node if the structure is a property of a JSON-object.
- **type** `d_type_t` - 3 bit : defining the type of the element.
- **len** `uint32_t` - 5 bit : the length of the data (for bytes/string/array/object). For (boolean or integer) the length will specify the value.
- **data** `bytes_t` - The bytes or value of the node (only for strings or bytes).



The serialization depends on the type, which is defined in the first 3 bits of the first byte of the element:

```
d_type_t type = *val >> 5;      // first 3 bits define the type
uint8_t  len  = *val & 0x1F;    // the other 5 bits  (0-31) the length
```

The `len` depends on the size of the data. So, the last 5 bit of the first bytes are interpreted as follows:

- `0x00` - `0x1c` : The length is taken as is from the 5 bits.
- `0x1d` - `0x1f` : The length is taken by reading the big-endian value of the next `len - 0x1c` bytes (len ext).

After the type-byte and optional length bytes, the 2 bytes representing the property hash is added, but only if the element is a property of a JSON-object.

Depending on these types, the length will be used to read the next bytes:

- `0x0` : **binary data** - This would be a value or property with binary data. The `len` will be used to read the number of bytes as binary data.
- `0x1` : **string data** - This would be a value or property with string data. The `len` will be used to read the number of bytes (+1) as string. The string will always be null-terminated, since it will allow small devices to use the data directly instead of copying memory in RAM.
- `0x2` : **array** - Represents an array node, where the `len` represents the number of elements in the array. The array elements will be added right after the array-node.
- `0x3` : **object** - A JSON-object with `len` properties coming next. In this case the properties following this element will have a leading `key` specified.
- `0x4` : **boolean** - Boolean value where `len` must be either `0x1`= `true` or `0x0` = `false`. If `len > 1` this element is a copy of a previous node and may reference the same data. The index of the source node will then be `len-2`.
- `0x5` : **integer** - An integer-value with max 29 bit (since the 3 bits are used for the type). If the value is higher than `0x20000000`, it will be stored as binary data.
- `0x6` : **null** - Represents a null-value. If this value has a `len`> 0 it will indicate the beginning of data, where `len` will be used to specify the number of elements to follow. This is optional, but helps small devices to allocate the right amount of memory.

4.6 Communication

Incubed requests follow a simple request/response schema allowing even devices with a small bandwith to retrieve all the required data with one request. But there are exceptions when additional data need to be fetched.

These are:

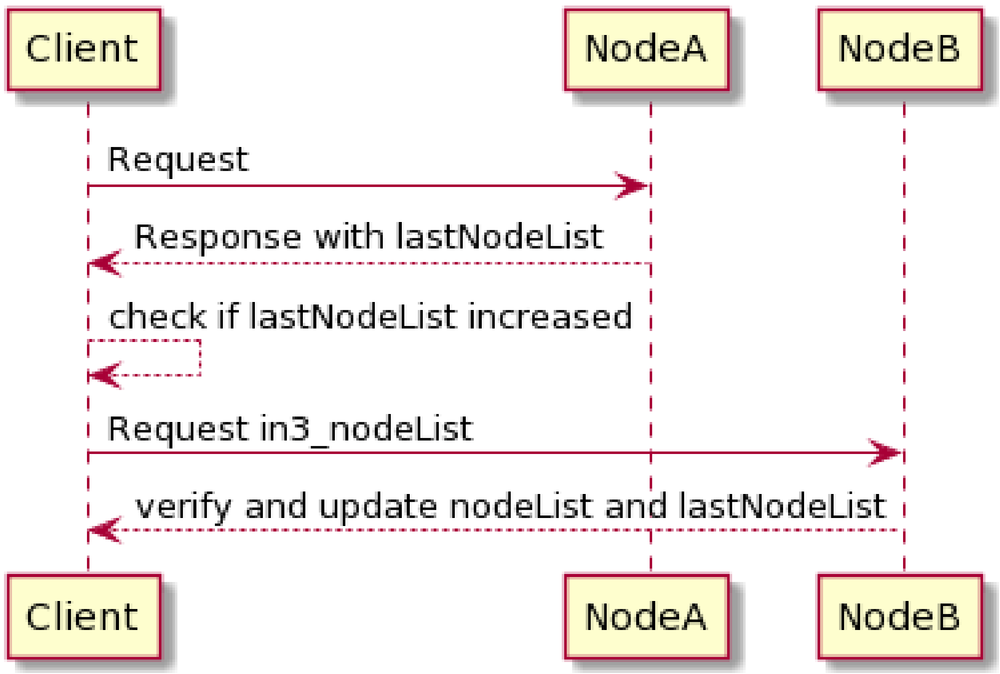
1. **Changes in the NodeRegistry**

   Changes in the NodeRegistry are based on one of the following events:

   - `LogNodeRegistered`
   - `LogNodeRemoved`
   - `LogNodeChanged`

   The server needs to watch for events from the `NodeRegistry` contract, and update the nodeList when needed.

   Changes are detected by the client by comparing the blocknumber of the latest change with the last known blocknumber. Since each response will include the `lastNodeList`, a client may detect this change after receiving the data. The client is then expected to call `in3_nodeList` to update its nodeList before sending out the next request. In the event that the node is not able to proof the new nodeList, the client may blacklist such a node.



1. **Changes in the ValidatorList**

   This only applies to PoA-chains where the client needs a defined and verified validatorList. Depending on the consensus, changes in the validatorList must be detected by the node and indicated with the `lastValidatorChange` on each response. This `lastValidatorChange` holds the last blocknumber of a change in the validatorList.

   Changes are detected by the client by comparing the blocknumber of the latest change with the last known blocknumber. Since each response will include the `lastValidatorChange` a client may detect this change after receiving the data or in case of an unverifiable response. The client is then expected to call `in3_validatorList` to update its list before sending out the next request. In the event that the node is not able to proof the new nodeList, the client may blacklist such a node.

2. **Failover**

   It is also good to have a second request in the event that a valid response is not delivered. This could happen if a node does not respond at all or the response cannot be validated. In both cases, the client may blacklist the node for a while and send the same request to another node.

CHAPTER 5

Roadmap

Incubed implements two versions:

- **TypeScript / JavaScript**: optimized for dApps, web apps, or mobile apps.
- **C**: optimized for microcontrollers and all other use cases.

In the future we will focus on one codebase, which is C. This will be ported to many platforms (like WASM).

5.1 V2.0 Stable: Q3 2019

This was the first stable release, which was published after Devcon. It contains full verification of all relevant Ethereum RPC calls (except eth_call for eWasm contracts), but there is no payment or incentivization included yet.

- **Fail-safe Connection**: The Incubed client will connect to any Ethereum blockchain (providing Incubed servers) by randomly selecting nodes within the Incubed network and, if the node cannot be reached or does not deliver verifiable responses, automatically retrying with different nodes.
- **Reputation Management**: Nodes that are not available will be temporarily blacklisted and lose reputation. The selection of a node is based on the weight (or performance) of the node and its availability.
- **Automatic NodeList Updates**: All Incubed nodes are registered in smart contracts on chain and will trigger events if the NodeList changes. Each request will always return the blockNumber of the last event so that the client knows when to update its NodeList.
- **Partial NodeList**: To support small devices, the NodeList can be limited and still be fully verified by basing the selection of nodes deterministically on a client-generated seed.
- **Multichain Support**: Incubed is currently supporting any Ethereum-based chain. The client can even run parallel requests to different networks without the need to synchronize first.
- **Preconfigured Boot Nodes**: While you can configure any registry contract, the standard version contains configuration with boot nodes for `mainnet`, `kovan`, `evan`, `tobalaba`, and `ipfs`.
- **Full Verification of JSON-RPC Methods**: Incubed is able to fully verify all important JSON-RPC methods. This even includes calling functions in smart contract and verifying their return value (`eth_call`), which means executing each opcode locally in the client to confirm the result.

- **IPFS Support**: Incubed is able to write and read IPFS content and verify the data by hashing and creating the multihash.
- **Caching Support**: An optional cache enables storage of the results of RPC requests that can automatically be used again within a configurable time span or if the client is offline. This also includes RPC requests, blocks, code, and NodeLists.
- **Custom Configuration**: The client is highly customizable. For each request, a configuration can be explicitly passed or adjusted through events (`client.on('beforeRequest',...)`). This allows the proof level or number of requests to be sent to be optimized depending on the context.
- **Proof Levels**: Incubed supports different proof levels: `none` for no verification, `standard` for verifying only relevant properties, and `full` for complete verification, including uncle blocks or previous transactions (higher payload).
- **Security Levels**: Configurable number of signatures (for PoW) and minimal deposit stored.
- **PoW Support**: For PoW, blocks are verified based on blockhashes signed by Incubed nodes storing a deposit, which they lose if this blockhash is not correct.
- **PoA Support**: (experimental) For PoA chains (using Aura and clique), blockhashes are verified by extracting the signature from the sealed fields of the blockheader and by using the Aura algorithm to determine the signer from the validatorlist (with static validatorlist or contract-based validators).
- **Finality Support**: For PoA chains, the client can require a configurable number of signatures (in percent) to accept them as final.
- **Flexible Transport Layer**: The communication layer between clients and nodes can be overridden, but the layer already supports different transport formats (JSON/CBOR/Incubed).
- **Replace Latest Blocks**: Since most applications per default always ask for the latest block, which cannot be considered final in a PoW chain, a configuration allows applications to automatically use a certain block height to run the request (like six blocks).
- **Light Ethereum API**: Incubed comes with a simple type-safe API, which covers all standard JSON-RPC requests (`in3.eth.getBalance('0x52bc44d5378309EE2abF1539BF71dE1b7d7bE3b5')`). This API also includes support for signing and sending transactions, as well as calling methods in smart contracts without a complete ABI by simply passing the signature of the method as an argument.
- **TypeScript Support**: Because Incubed is written 100% in TypeScript, you get all the advantages of a type-safe toolchain.
- **java**: java version of the Incubed client based on the C sources (using JNI)

5.2 V2.1 Incentivization: Q4 2019

This release will introduce the incentivization layer, which should help provide more nodes to create the decentralized network.

- **PoA Clique**: Supports Clique PoA to verify blockheaders.
- **Signed Requests**: Incubed supports the incentivization layer, which requires signed requests to assign client requests to certain nodes.
- **Network Balancing**: Nodes will balance the network based on load and reputation.
- **python-bindings**: integration in python
- **go-bindings**: bindings for go

5.3 V2.2 Bitcoin: Q1 2020

Multichain Support for BTC

- **Bitcoin**: Supports Verfification for Bitcoin blocks and Transactions
- **WASM**: Typescript client based on a the C-Sources compiled to wasm.

5.4 V2.3 WASM: Q3 2020

For `eth_call` verification, the client and server must be able to execute the code. This release adds the ability to support eWasm contracts.

- **eth 2.0**: Basic Support for Eth 2.0
- **eWasm**: Supports eWasm contracts in eth_call.

5.5 V2.4 Substrate: Q1 2021

Supports Polkadot or any substrate-based chains.

- **Substrate**: Framework support.
- **Runtime Optimization**: Using precompiled runtimes.

5.6 V2.5 Services: Q3 2021

Generic interface enables any deterministic service (such as docker-container) to be decentralized and verified.

CHAPTER 6

Benchmarks

These benchmarks aim to test the Incubed version for stability and performance on the server. As a result, we can gauge the resources needed to serve many clients.

6.1 Setup and Tools

- JMeter is used to send requests parallel to the server
- Custom Python scripts is used to generate lists of transactions as well as randomize them (used to create test plan)
- Link for making JMeter tests online without setting up the server: https://www.blazemeter.com/

JMeter can be downloaded from: https://jmeter.apache.org/download_jmeter.cgi

Install JMeter on Mac OS With HomeBrew

1. Open a Mac Terminal where we will be running all the commands
2. First, check to see if HomeBrew is installed on your Mac by executing this command. You can either run brew help or brew -v
3. If HomeBrew is not installed, run the following command to install HomeBrew on Mac:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
↪install/master/install)"
Once HomeBrew is installed, we can continue to install JMeter.
```

4. To install JMeter without the extra plugins, run the following command:

```
brew install jmeter
```

5. To install JMeter with all the extra plugins, run the following command:

```
brew install jmeter --with-plugins
```

6. Finally, verify the installation by executing jmeter -v
7. Run JMeter using 'jmeter' which should load the JMeter GUI

JMeter on EC2 instance CLI only (testing pending):

1. Login to AWS and navigate to the EC2 instance page
2. Create a new instance, choose an Ubuntu AMI]
3. Provision the AWS instance with the needed information, enable CloudWatch monitoring
4. Configure the instance to allow all outgoing traffic, and fine tune Security group rules to suit your need
5. Save the SSH key, use the SSH key to login to the EC2 instance
6. Install Java:

```
sudo add-apt-repository ppa:linuxuprising/java
sudo apt-get update
sudo apt-get install oracle-java11-installer
```

7. Install JMeter using:

```
sudo apt-get install jmeter
```

8. Get the JMeter Plugins:

```
wget http://jmeter-plugins.org/downloads/file/JMeterPlugins-
↪Standard-1.2.0.zip
wget http://jmeter-plugins.org/downloads/file/JMeterPlugins-
↪Extras-1.2.0.zip
wget http://jmeter-plugins.org/downloads/file/JMeterPlugins-
↪ExtrasLibs-1.2.0.zip
```

9. Move the unzipped jar files to the install location:

```
sudo unzip JMeterPlugins-Standard-1.2.0.zip -d /usr/share/jmeter/
sudo unzip JMeterPlugins-Extras-1.2.0.zip -d /usr/share/jmeter/
sudo unzip JMeterPlugins-ExtrasLibs-1.2.0.zip -d /usr/share/
↪jmeter/
```

10. Copy the JML file to the EC2 instance using:

    (On host computer)

```
scp -i <path_to_key> <path_to_local_file> <user>@<server_url>:
↪<path_on_server>
```

11. Run JMeter without the GUI:

```
jmeter -n -t <path_to_jmx> -l <path_to_output_jtl>
```

12. Copy the JTL file back to the host computer and view the file using JMeter with GUI

Python script to create test plan:

1. Navigate to the txGenerator folder in the in3-tests repo.
2. Run the main.py file while referencing the start block (-s), end block (-e) and number of blocks to choose in this range (-n). The script will randomly choose three transactions per block.

3. The transactions chosen are sent through a tumble function, resulting in a randomized list of transactions from random blocks. This should be a realistic scenario to test with, and prevents too many concurrent cache hits.
4. Import the generated CSV file into the loaded test plan on JMeter.
5. Refer to existing test plans for information on how to read transactions from CSV files and to see how it can be integrated into the requests.

6.2 Considerations

- When the Incubed benchmark is run on a new server, create a baseline before applying any changes.
- Run the same benchmark test with the new codebase, test for performance gains.
- The tests can be modified to include the number of users and duration of the test. For a stress test, choose 200 users and a test duration of 500 seconds or more.
- When running in an EC2 instance, up to 500 users can be simulated without issues. Running in GUI mode reduces this number.
- A beneficial method for running the test is to slowly ramp up the user count. Start with a test of 10 users for 120 seconds in order to test basic stability. Work your way up to 200 users and longer durations.
- Parity might often be the bottleneck; you can confirm this by using the get_avg_stddev_in3_response.sh script in the scripts directory of the in3-test repo. This would help show what optimizations are needed.

6.3 Results/Baseline

- The baseline test was done with our existing server running multiple docker containers. It is not indicative of a perfect server setup, but it can be used to benchmark upgrades to our codebase.
- The baseline for our current system is given below. This system has multithreading enabled and has been tested with ethCalls included in the test plan.

Users/duration	Number of requests	tps	getBlockByHash (ms)	getBlockByNumber (ms)	getTransactionHash (ms)	getTransactionReceipt (ms)	EthCall(ms)	eth_getStorage (ms)	Notes
10/120s									
20/120s	4800	40	580	419	521	923	449	206	
40/120s	5705	47	1020	708	902	1508	816	442	
80/120s	7970	66	1105	790	2451	3197	984	452	
100/120s	6911	57	1505	1379	2501	4310	1486	866	
110/120s	6000	50	1789	1646	4204	5662	1811	1007	
120/500s	32000	65	1331	1184	4600	5314	1815	1607	
140/500s	31000	62	1666	1425	5207	6722	1760	941	
160/500s	33000	65	1949	1615	6269	7604	1900	930	In3 -> 400ms, rpc -> 2081ms
200/500s	34000	70	1270	1031	12500	14349	1251	716	At higher loads, the RPC delay adds up. It is the bottlenecking factor. Able to handle 200 users on sustained loads.

- More benchmarks and their results can be found in the in3-tests repo

CHAPTER 7

Embedded Devices

7.1 Hardware Requirements

7.1.1 Memory

For the memory this example requires:

- Dynamic memory(DRAM) : 30 - 50kB
- Flash Memory : 150 - 200kB

7.1.2 Networking

In3 client needs to have a reliable internet connection to work properly, so your hardware must support any network interface or module that could give you access to it. i.e Bluetooth, Wifi, ethernet, etc.

7.2 Incubed with ESP-IDF

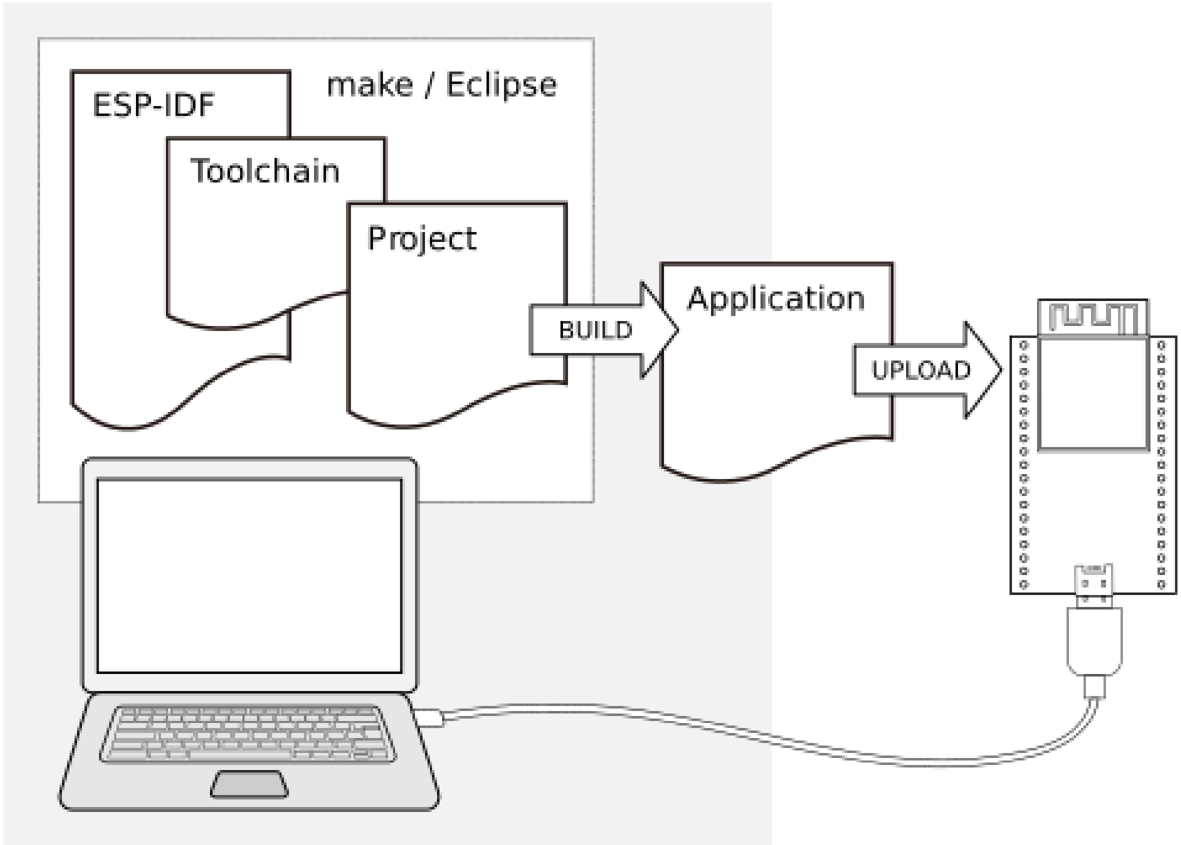
7.2.1 Use case example: Airbnb Property access

A smart door lock that grants access to a rented flat is installed on the property. It is able to connect to the Internet to check if renting is allowed and that the current user is authorized to open the lock.

The computational power of the control unit is restricted to the control of the lock. And it is also needed to maintain a permanent Internet connection.

You want to enable this in your application as an example of how in3 can help you, we will guide through the steps of doing it, from the very basics and the resources you will need

Hardware requirements



from https://docs.espressif.com/projects/esp-idf/en/stable/get-started/

- ESP32-DevKitC V4 or similar dev board
- Android phone
- Laptop MAC, Linux, Windows
- USB Cable

Software requirements

- In3 C client
- Esp-idf toolchain and sdk, (please follow this guide) and be sure on the cloning step to use `release/v4.0` branch

```
git clone -b release/v4.0 --recursive https://github.com/espressif/esp-idf.git
```

- Android Studio
- Solidity smart contract: we will control access to properties using a public smart contract, for this example, we will use the following template
- Silab USB drivers

```
pragma solidity ^0.5.1;

contract Access {
    uint8 access;
```

(continues on next page)

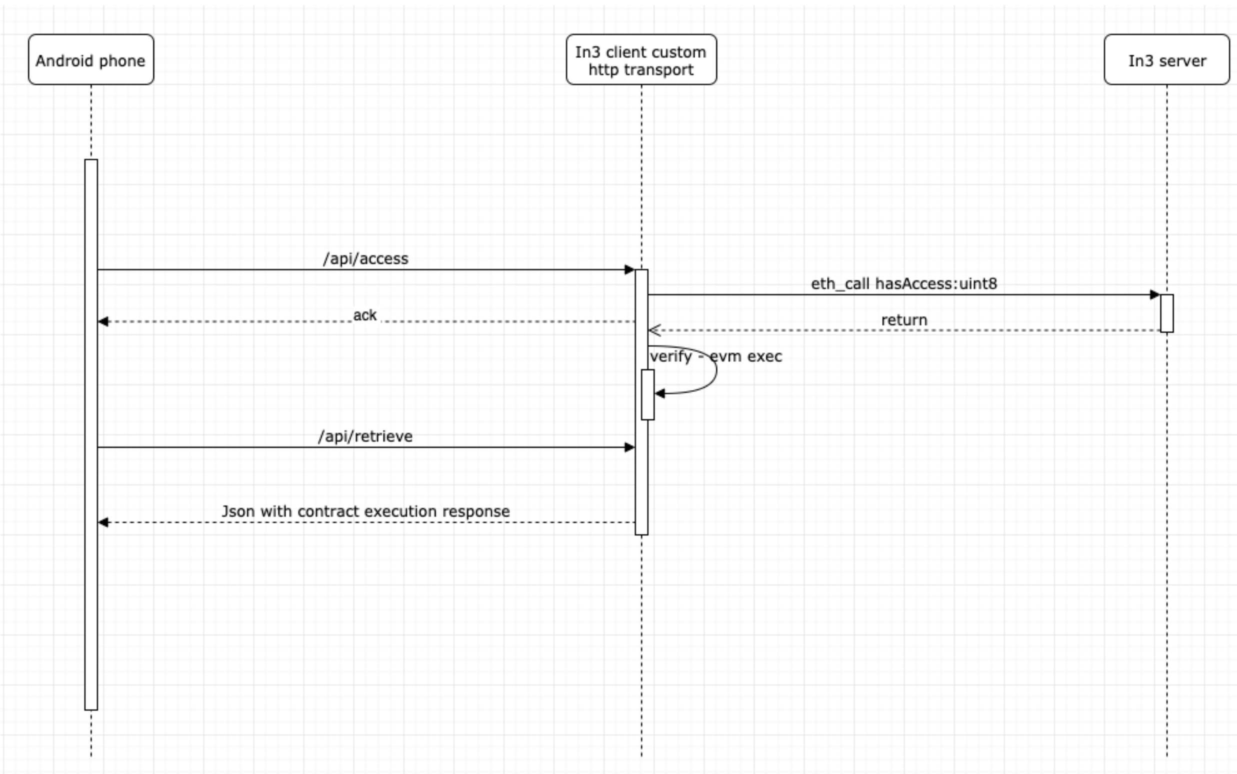
(continued from previous page)

```
    constructor() public  {
        access = 0;
    }

    function hasAccess() public view returns(uint8) {
        return access;
    }

    function setAccess(uint8 accessUpdate) public{
        access = accessUpdate;
    }
}
```

How it works



sequence diagram

In3 will support a wide range of microcontrollers, in this guide we will use well-known esp32 with freertos framework, and an example android app to interact with it via Wifi connection.

Instalation instructions

1. Clone the repo

```
git clone --recursive https://github.com/slockit/in3-devices-esp
```

1. Deploy the contract with your favorite tool (truffle, etc) or use our previusly deployed contract on goerli, with address `0x36643F8D17FE745a69A2Fd22188921Fade60a98B`
2. Config your SSID and password inside sdkconfig file `sdkconfig.defaults`

```
CONFIG_WIFI_SSID="YOUR SSID"
CONFIG_WIFI_PASSWORD="YOUR PWD"
```

1. Build the code `idf.py build`
2. Connect the usb cable to flash and monitor the serial output from the application.

```
idf.py flash && idf.py monitor
```

after the build finishes and the serial monitor is running you will see the configuration and init logs.

1. Configure the ip address of the example, to work with: Take a look at the inital output of the serial output of the `idf.py monitor` command, you will the ip address, as follows

```
I (2647) tcpip_adapter: sta ip: 192.168.178.64, mask: 255.255.255.0, gw: 192.168.178.1
I (2647) IN3: got ip:192.168.178.64
```

take note if your ip address which will be used in the android application example.

1. Clone the android repository, compile the android application and install the in3 demo application in your phone.

```
git clone https://github.com/slockit/in3-android-example
```

1. Modify the android source changing ip address variable inside kotlin source file `MainActivity.kt`, with the IP address found on step 6.

```
(L:20) private const val ipaddress = "http://192.168.xx.xx"
```

1. If you want to test directly without using android you can also do it with the following http curl requests:

- `curl -X GET http://slock.local/api/access`
- `curl -X GET http://slock.local/api/retrieve`

we need 2 requests as the verification process needs to be executed in asynchronous manner, first one will trigger the execution and the result could be retrieved with the second one

7.3 Incubed with Zephyr

. . . .(Comming soon)

CHAPTER 8

API RPC

This section describes the behavior for each RPC-method supported with incubed.

The core of incubed is to execute rpc-requests which will be send to the incubed nodes and verified. This means the available RPC-Requests are defined by the clients itself.

- For Ethereum : https://eth.wiki/json-rpc/API
- For Bitcoin : https://bitcoincore.org/en/doc/0.18.0/

8.1 in3

There are also some Incubed specific rpc-methods, which will help the clients to bootstrap and update the nodeLists.

The incubed client itself offers special RPC-Methods, which are mostly handled directly inside the client:

8.1.1 in3_config

changes the configuration of a client. The configuration is passed as the first param and may contain only the values to change.

Parameters:

1. `config`: config-object - a Object with config-params.

The config params support the following properties :

- **autoUpdateList** :`bool` *(optional)* - if true the nodelist will be automaticly updated if the lastBlock is newer. example: true
- **chainId** :`uint32_t` or `string (mainnet/kovan/goerli)` - servers to filter for the given chain. The chain-id based on EIP-155. example: 0x1
- **signatureCount** :`uint8_t` *(optional)* - number of signatures requested. example: 2

- **finality** :`uint16_t` *(optional)* - the number in percent needed in order reach finality (% of signature of the validators). example: 50
- **includeCode** :`bool` *(optional)* - if true, the request should include the codes of all accounts. otherwise only the the codeHash is returned. In this case the client may ask by calling eth_getCode() afterwards. example: true
- **bootWeights** :`bool` *(optional)* - if true, the first request (updating the nodelist) will also fetch the current health status and use it for blacklisting unhealthy nodes. This is used only if no nodelist is availabkle from cache. example: true
- **maxAttempts** :`uint16_t` *(optional)* - max number of attempts in case a response is rejected. example: 10
- **keepIn3** :`bool` *(optional)* - if true, requests sent to the input sream of the comandline util will be send theor responses in the same form as the server did. example: false
- **key** :`bytes32` *(optional)* - the client key to sign requests. (only available if build with `-DPK_SIGNER=true` , which is on per default) example: 0x387a8233c96e1fc0ad5e284353276177af2186e7afa85296f106336e376669f7
- **pk** :`bytes32|bytes32[]` *(optional)* - registers raw private keys as signers for transactions. (only availble if build with `-DPK_SIGNER=true` , which is on per default) example: 0x387a8233c96e1fc0ad5e284353276177af2186e7afa85296f106336e376669f7
- **useBinary** :`bool` *(optional)* - if true the client will use binary format. example: false
- **useHttp** :`bool` *(optional)* - if true the client will try to use http instead of https. example: false
- **timeout** :`uint32_t` *(optional)* - specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection. example: 100000
- **minDeposit** :`uint64_t` - min stake of the server. Only nodes owning at least this amount will be chosen.
- **nodeProps** :`uint64_t` bitmask *(optional)* - used to identify the capabilities of the node.
- **nodeLimit** :`uint16_t` *(optional)* - the limit of nodes to store in the client. example: 150
- **proof** :`string (none/standard/full)` *(optional)* - if true the nodes should send a proof of the response. example: true
- **replaceLatestBlock** :`uint8_t` *(optional)* - if specified, the blocknumber *latest* will be replaced by blockNumber- specified value. example: 6
- **requestCount** :`uint8_t` - the number of request send when getting a first answer. example: 3
- **btc** :`Object` *(optional)* - configuration for bitcoin-verification (only available if build with `-DBTC=true`, which is on per default). The config may contains the following fields:
  - **maxDAP** :`number` - max number of DAPs (Difficulty Adjustment Periods) allowed when accepting new targets.
  - **maxDiff** :`number` - max increase (in percent) of the difference between targets when accepting new targets.
- **zksync** :`Object` *(optional)* - configuration for zksync-api (only available if build with `-DZKSYNC=true`, which is off per default). The config may contains the following fields:
  - **provider_url** :`string` *(optional)* - url of the zksync-server (if not defined it will be choosen depending on the chain)
  - **account** :`address` *(optional)* - the account to be used. if not specified, the first signer will be used.
- **rpc** :`string` *(optional)* - url of one or more rpc-endpoints to use. (list can be comma seperated)
- **servers/nodes** : `collection of JSON objects with chain Id (hex string) as key` *(optional)* - the value of each JSON object defines the nodelist per chain and may contain the following fields:

- **contract** :`address` - address of the registry contract.
- **whiteListContract** :`address` *(optional, cannot be combined with whiteList)* - address of the whiteList contract.
- **whiteList** :`array of addresses` *(optional, cannot be combined with whiteListContract)* - manual whitelist.
- **registryId** :`bytes32` - identifier of the registry.
- **needsUpdate** :`bool` *(optional)* - if set, the nodeList will be updated before next request.
- **avgBlockTime** :`uint16_t` *(optional)* - average block time (seconds) for this chain.
- **verifiedHashes** :`array of JSON objects` *(optional)* - if the client sends an array of blockhashes the server will not deliver any signatures or blockheaders for these blocks, but only return a string with a number. This is automaticly updated by the cache, but can be overriden per request. MUST contain the following fields:
  * **block** :`uint64_t` - block number.
  * **hash** : `bytes32` - verified hash corresponding to block number.
- **nodeList** :`array of JSON objects` *(optional)* - manual nodeList, each JSON object may contain the following fields:
  * **url** :`string` - URL of the node.
  * **address** :`address` - address of the node.
  * **props** :`uint64_t` bitmask *(optional)* - used to identify the capabilities of the node (defaults to 65535).

Returns:

an boolean confirming that the config has changed.

Example:

Request:

```
{
        "method": "in3_config",
        "params": [{
                "chainId": "0x5",
                "maxAttempts": 4,
                "nodeLimit": 10,
                "servers": {
                        "0x1": {
                                "nodeList": [{
                                                "address":
↪"0x1234567890123456789012345678901234567890",
                                                "url": "https://mybootnode-A.com",
                                                "props": "0xFFFF"
                                        },
                                        {
                                                "address":
↪"0x1234567890123456789012345678901234567890",
                                                "url": "https://mybootnode-B.com",
                                                "props": "0xFFFF"
                                        }
                                ]
                        }
```

(continues on next page)

(continued from previous page)

```
                }
        }]
}
```

Response:

```
{
  "id": 1,
  "result": true
}
```

8.1.2 in3_abiEncode

based on the ABI-encoding used by solidity, this function encodes the values and returns it as hex-string.

Parameters:

1. `signature`: string - the signature of the function. e.g. `getBalance(uint256)`. The format is the same as used by solidity to create the functionhash. optional you can also add the return type, which in this case is ignored.
2. `params`: array - a array of arguments. the number of arguments must match the arguments in the signature.

Returns:

the ABI-encoded data as hex including the 4 byte function-signature. These data can be used for `eth_call` or to send a transaction.

Request:

```
{
    "method":"in3_abiEncode",
    "params":[
        "getBalance(address)",
        ["0x1234567890123456789012345678901234567890"]
    ]
}
```

Response:

```
{
  "id": 1,
  "result":
↪"0xf8b2cb4f0000000000000000000000001234567890123456789012345678901234567890",
}
```

8.1.3 in3_abiDecode

based on the ABI-encoding used by solidity, this function decodes the bytes given and returns it as array of values.

Parameters:

1. `signature`: string - the signature of the function. e.g. `uint256`, `(address,string,uint256)` or `getBalance(address):uint256`. If the complete functionhash is given, only the return-part will be used.
2. `data`: hex - the data to decode (usually the result of a eth_call)

Returns:

a array (if more then one arguments in the result-type) or the the value after decodeing.

Request:

```
{
    "method":"in3_abiDecode",
    "params":[
        "(address,uint256)",

↪"0x00000000000000000000000012345678901234567890123456789012345678900000000000000000000000000000000
↪"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": ["0x1234567890123456789012345678901234567890","0x05"],
}
```

8.1.4 in3_addRawKey

adds a raw private key as signer, which allows signing transactions.

Parameters:

1. `pk`: string - the 32byte long private key as hex string.

Returns:

the address of given key.

Request:

```
{
    "method":"in3_addRawKey",
    "params":[
        "0x1234567890123456789012345678901234567890123456789012345678901234"]
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0x2e988a386a799f506693793c6a5af6b54dfaabfb"
}
```

8.1.5 in3_checksumAddress

Will convert an upper or lowercase Ethereum address to a checksum address. (See EIP55)

Parameters:

1. `address`: address - the address to convert.

2. `useChainId`: boolean - if true, the chainId is integrated as well (See EIP1191)

Returns:

the address-string using the upper/lowercase hex characters.

Request:

```
{
    "method":"in3_checksumAddress",
    "params":[
        "0x1fe2e9bf29aa1938859af64c413361227d04059a",
        false
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0x1Fe2E9bf29aa1938859Af64C413361227d04059a"
}
```

8.1.6 in3_ens

resolves a ens-name. the domain names consist of a series of dot-separated labels. Each label must be a valid normalised label as described in UTS46 with the options `transitional=false` and `useSTD3AsciiRules=true`. For Javascript implementations, a library is available that normalises and checks names.

Parameters:

1. `name`: string - the domain name UTS46 compliant string.
2. `field`: string - the required data, which could be
    - `addr` - the address (default)
    - `resolver` - the address of the resolver
    - `hash` - the namehash
    - `owner` - the owner of the domain

Returns:

the address-string using the upper/lowercase hex characters.

Request:

```
{
    "method":"in3_ens",
    "params":[
        "cryptokitties.eth",
        "addr"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0x06012c8cf97bead5deae237070f9587f8e7a266d"
}
```

8.1.7 in3_toWei

converts the given value into wei.

Parameters:

1. `value`: string or integer - the value, which may be floating number as string, like '0.9'
2. `unit`: the unit of the value, which must be one of `wei`, `kwei`, `Kwei`, `babbage`, `femtoether`, `mwei`, `Mwei`, `lovelace`, `picoether`, `gwei`, `Gwei`, `shannon`, `nanoether`, `nano`, `szabo`, `microether`, `micro`, `finney`, `milliether`, `milli`, `ether`, `eth`, `kether`, `grand`, `mether`, `gether`, `tether`

Returns:

the value in wei as hex.

Request:

```
{
    "method":"in3_toWei",
    "params":[
         "20.0009123", "eth"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0x01159183c4793db800",
}
```

8.1.8 in3_pk2address

extracts the address from a private key.

Parameters:

1. `key`: hex - the 32 bytes private key as hex.

Returns:

the address-string.

Request:

```
{
    "method":"in3_pk2address",
    "params":[
        "0x0fd65f7da55d811634495754f27ab318a3309e8b4b8a978a50c20a661117435a"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0xdc5c4280d8a286f0f9c8f7f55a5a0c67125efcfd"
}
```

8.1.9 in3_pk2public

extracts the public key from a private key.

Parameters:

1. `key`: hex - the 32 bytes private key as hex.

Returns:

the public key.

Request:

```
{
    "method":"in3_pk2public",
    "params":[
        "0x0fd65f7da55d811634495754f27ab318a3309e8b4b8a978a50c20a661117435a"
    ]
}
```

Response:

```
{
  "id": 1,
  "result":
↪"0x0903329708d9380aca47b02f3955800179e18bffbb29be3a644593c5f87e4c7fa960983f78186577eccc909cec71cb5
↪"
}
```

8.1.10 in3_ecrecover

extracts the public key and address from signature.

Parameters:

1. `msg`: hex - the message the signature is based on.
2. `sig`: hex - the 65 bytes signature as hex.
3. `sigtype`: string - the type of the signature data : `eth_sign` (use the prefix and hash it), `raw` (hash the raw data), `hash` (use the already hashed data). Default: `raw`

Returns:

a object with 2 properties:

- `publicKey` : hex - the 64 byte public key
- `address` : address - the 20 byte address

Request:

```
{
    "method":"in3_ecrecover",
    "params":[
        "0x487b2cbb7997e45b4e9771d14c336b47c87dc2424b11590e32b3a8b9ab327999",

↪"0x0f804ff891e97e8a1c35a2ebafc5e7f129a630a70787fb86ad5aec0758d98c7b454dee5564310d497ddfe814839c8bab
↪",
        "hash"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": {
      "publicKey":
↪"0x94b26bafa6406d7b636fbb4de4edd62a2654eeecda9505e9a478a66c4f42e504c4481bad171e5ba6f15a5f11c26acfc6
↪",
      "address":"0xf68a4703314e9a9cf65be688bd6d9b3b34594ab4"
   }
}
```

8.1.11 in3_signData

signs the given data

Parameters:

1. `msg`: hex - the message to sign.
2. `key`: hex - the key (32 bytes) or address (20 bytes) of the signer. If the address is passed, the internal signer needs to support this address.
3. `sigtype`: string - the type of the signature data : `eth_sign` (use the prefix and hash it), `raw` (hash the raw data), `hash` (use the already hashed data). Default: `raw`

Returns:

a object with the following properties:

- `message` : hex - original message used
- `messageHash` : hex - the hash the signature is based on
- `signature`: hex - the signature (65 bytes)
- `r` : hex - the x -value of the EC-Point
- `s` : hex - the y -value of the EC-Point
- `v` : number - the sector (0|1) + 27

Request:

```
{
    "method":"in3_signData",
    "params":[
        "0x0102030405060708090a0b0c0d0e0f",
        "0xa8b8759ec8b59d7c13ef3630e8530f47ddb47eba12f00f9024d3d48247b62852",
```

(continues on next page)

(continued from previous page)

```
        "raw"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": {
      "message":"0x0102030405060708090a0b0c0d0e0f",
      "messageHash":
↪"0x1d4f6fccf1e27711667605e29b6f15adfda262e5aedfc5db904feea2baa75e67",
      "signature":
↪"0xa5dea9537d27e4e20b6dfc89fa4b3bc4babe9a2375d64fb32a2eab04559e95792264ad1fb83be70c145aec69045da798
↪",
      "r":"0xa5dea9537d27e4e20b6dfc89fa4b3bc4babe9a2375d64fb32a2eab04559e9579",
      "s":"0x2264ad1fb83be70c145aec69045da7986b95ee957fb9c5b6d315daa5c0c3e152",
      "v":27
   }
}
```

8.1.12 in3_decryptKey

decrypts a JSON Keystore file as defined in the Web3 Secret Storage Definition . The result is the raw private key.

Parameters:

1. `key`: Object - Keydata as object as defined in the keystorefile
2. `passphrase`: String - the password to decrypt it.

Returns:

a raw private key (32 bytes)

Request:

```
{
    "method": "in3_decryptKey",
    "params": [
        {
            "version": 3,
            "id": "f6b5c0b1-ba7a-4b67-9086-a01ea54ec638",
            "address": "08aa30739030f362a8dd597fd3fcde283e36f4a1",
            "crypto": {
                "ciphertext":
↪"d5c5aafdee81d25bb5ac4048c8c6954dd50c595ee918f120f5a2066951ef992d",
                "cipherparams": {
                    "iv": "415440d2b1d6811d5c8a3f4c92c73f49"
                },
                "cipher": "aes-128-ctr",
                "kdf": "pbkdf2",
                "kdfparams": {
                    "dklen": 32,
                    "salt":
↪"691e9ad0da2b44404f65e0a60cf6aabe3e92d2c23b7410fd187eeeb2c1de4a0d",
                    "c": 16384,
```

(continues on next page)

(continued from previous page)

```
                    "prf": "hmac-sha256"
                },
                "mac":
↪"de651c04fc67fd552002b4235fa23ab2178d3a500caa7070b554168e73359610"
            }
        },
        "test"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "0x1ff25594a5e12c1e31ebd8112bdf107d217c1393da8dc7fc9d57696263457546"
}
```

8.1.13 in3_cacheClear

clears the incubed cache (usually found in the .in3-folder)

Request:

```
{
    "method":"in3_cacheClear",
    "params":[]
}
```

Response:

```
{
  "id": 1,
  "result": true
}
```

8.1.14 in3_nodeList

return the list of all registered nodes.

Parameters:

all parameters are optional, but if given a partial NodeList may be returned.

1. `limit`: number - if the number is defined and >0 this method will return a partial nodeList limited to the given number.
2. `seed`: hex - This 32byte hex integer is used to calculate the indexes of the partial nodeList. It is expected to be a random value choosen by the client in order to make the result deterministic.
3. `addresses`: address[] - a optional array of addresses of signers the nodeList must include.

Returns:

an object with the following properties:

- `nodes`: Node[] - a array of node-values. Each Object has the following properties:

  - `url` : string - the url of the node. Currently only http/https is supported, but in the future this may even support onion-routing or any other protocols.
  - `address` : address - the address of the signer
  - `index`: number - the index within the nodeList of the contract
  - `deposit`: string - the stored deposit
  - `props`: string - the bitset of capabilities as described in the *Node Structure*
  - `timeout`: string - the time in seconds describing how long the deposit would be locked when trying to unregister a node.
  - `registerTime` : string - unix timestamp in seconds when the node has registered.
  - `weight` : string - the weight of a node (not used yet) describing the amount of request-points it can handle per second.
  - `proofHash`: hex - a hash value containing the above values. This hash is explicitly stored in the contract, which enables the client to have only one merkle proof per node instead of verifying each property as its own storage value. The proof hash is build :

```
return keccak256(
    abi.encodePacked(
        _node.deposit,
        _node.timeout,
        _node.registerTime,
        _node.props,
        _node.signer,
        _node.url
    )
);
```

- `contract` : address - the address of the Incubed-storage-contract. The client may use this information to verify that we are talking about the same contract or throw an exception otherwise.
- `registryId`: hex - the registryId (32 bytes) of the contract, which is there to verify the correct contract.
- `lastBlockNumber` : number - the blockNumber of the last change of the list (usually the last event).
- `totalServer` : number - the total numbers of nodes.

if proof is requested, the proof will have the type `accountProof`. In the proof-section only the storage-keys of the `proofHash` will be included. The required storage keys are calcualted :

- `0x00` - the length of the nodeList or total numbers of nodes.
- `0x01` - the registryId
- per node : `0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563 + index * 5 + 4`

The blockNumber of the proof must be the latest final block (`latest`- minBlockHeight) and always greater or equal to the `lastBlockNumber`

This proof section contains the following properties:

- `type` : constant : `accountProof`
- `block` : the serialized blockheader of the latest final block
- `signatures` : a array of signatures from the signers (if requested) of the above block.

- `accounts`: a Object with the addresses of the db-contract as key and Proof as value. The Data Structure of the Proof is exactly the same as the result of - `eth_getProof`, but it must containi the above described keys
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Request:

```
{
  "method":"in3_nodeList",
  "params":[2,"0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b",
↪[]],
  "in3":{
    "verification":"proof"
  }
}
```

Response:

```
{
  "id": 1,
  "result": {
    "totalServers": 5,
    "contract": "0x64abe24afbba64cae47e3dc3ced0fcab95e4edd5",
    "lastBlockNumber": 8669495,
    "nodes": [
      {
        "url": "https://in3-v2.slock.it/mainnet/nd-3",
        "address": "0x945F75c0408C0026a3CD204d36f5e47745182fd4",
        "index": 2,
        "deposit": "10000000000000000",
        "props": "29",
        "chainIds": [
          "0x1"
        ],
        "timeout": "3600",
        "registerTime": "1570109570",
        "weight": "2000",
        "proofHash": "27ffb9b7dc2c5f800c13731e7c1e43fb438928dd5d69aaa8159c21fb13180a4c
↪"
      },
      {
        "url": "https://in3-v2.slock.it/mainnet/nd-5",
        "address": "0xbcdF4E3e90cc7288b578329efd7bcC90655148d2",
        "index": 4,
        "deposit": "10000000000000000",
        "props": "29",
        "chainIds": [
          "0x1"
        ],
        "timeout": "3600",
        "registerTime": "1570109690",
        "weight": "2000",
        "proofHash": "d0dbb6f1e28a8b90761b973e678cf8ecd6b5b3a9d61fb9797d187be011ee9ec7
↪"
      }
    ],
    "registryId": "0x423dd84f33a44f60e5d58090dcdcc1c047f57be895415822f211b8cd1fd692e3"
  },
```

(continues on next page)

(continued from previous page)


```
  "in3": {
    "proof": {
      "type": "accountProof",
      "block": "0xf9021ca01...",
      "accounts": {
        "0x64abe24afbba64cae47e3dc3ced0fcab95e4edd5": {
          "accountProof": [
            "0xf90211a0e822...",
            "0xf90211a0f6d0...",
            "0xf90211a04d7b...",
            "0xf90211a0e749...",
            "0xf90211a059cb...",
            "0xf90211a0568f...",
            "0xf8d1a0ac2433...",
            "0xf86d9d33b981..."
          ],
          "address": "0x64abe24afbba64cae47e3dc3ced0fcab95e4edd5",
          "balance": "0xb1a2bc2ec50000",
          "codeHash":
↪"0x18e64869905158477a607a68e9c0074d78f56a9dd5665a5254f456f89d5be398",
          "nonce": "0x1",
          "storageHash":
↪"0x4386ec93bd665ea07d7ed488e8b495b362a31dc4100cf762b22f4346ee925d1f",
          "storageProof": [
            {
              "key": "0x0",
              "proof": [
                "0xf90211a0ccb6d2d5786...",
                "0xf871808080808080800...",
                "0xe2a0200decd9548b62a...05"
              ],
              "value": "0x5"
            },
            {
              "key": "0x1",
              "proof": [
                "0xf90211a0ccb6d2d5786...",
                "0xf89180a010806a37911...",
                "0xf843a0200e2d5276120...
↪423dd84f33a44f60e5d58090dcdcc1c047f57be895415822f211b8cd1fd692e3"
              ],
              "value":
↪"0x423dd84f33a44f60e5d58090dcdcc1c047f57be895415822f211b8cd1fd692e3"
            },
            {
              "key":
↪"0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e571",
              "proof": [
                "0xf90211a0ccb6d2d...",
                "0xf871a08b9ff91d8...",
                "0xf843a0206695c25...
↪27ffb9b7dc2c5f800c13731e7c1e43fb438928dd5d69aaa8159c21fb13180a4c"
              ],
              "value":
↪"0x27ffb9b7dc2c5f800c13731e7c1e43fb438928dd5d69aaa8159c21fb13180a4c"
            },
            {
```


(continues on next page)

(continued from previous page)

```
              "key":
↪"0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e57b",
              "proof": [
                "0xf90211a0ccb6d2d1...",
                "0xf851a06807310abd...",
                "0xf843a0204d807394...
↪0d0dbb6f1e28a8b90761b973e678cf8ecd6b5b3a9d61fb9797d187be011ee9ec7"
              ],
              "value":
↪"0xd0dbb6f1e28a8b90761b973e678cf8ecd6b5b3a9d61fb9797d187be011ee9ec7"
            }
          ]
        }
      }
    }
  }
}
```

Partial NodeLists

if the client requests a partial nodeList and the given limit is smaller then the total amount of nodes, the server needs to pick nodes in a deterministic way. This is done by using the given seed.

1. add all required addresses (if any) to the list.
2. iterate over the indexes until the limit is reached:

```
function createIndexes(total: number, limit: number, seed: Buffer): number[] {
  const result: number[] = []              // the result as a list of indexes
  let step = seed.readUIntBE(0, 6)         // first 6 bytes define the step size
  let pos  = seed.readUIntBE(6, 6) % total // next 6 bytes define the offset
  while (result.length < limit) {
    if (result.indexOf(pos) >= 0) {        // if the index is already part of the␣
↪result
      seed = keccak256(seed)               // we create a new seed by hashing the␣
↪seed.
      step = seed.readUIntBE(0, 6)         // and change the step-size
    }
    else
      result.push(pos)
    pos = (pos + step) % total             // use the modulo operator to␣
↪calculate the next position.
  }
  return result
}
```

8.1.15 in3_sign

requests a signed blockhash from the node. In most cases these requests will come from other nodes, because the client simply adds the addresses of the requested signers and the processing nodes will then aquire the signatures with this method from the other nodes.

Since each node has a risk of signing a wrong blockhash and getting convicted and losing its deposit, per default nodes will and should not sign blockHash of the last `minBlockHeight` (default: 6) blocks!

Parameters:

1. `blocks`: Object[] - requested blocks. Each block-object has these 2 properties:
   (a) `blockNumber` : number - the blockNumber to sign.
   (b) `hash` : hex - (optional) the expected hash. This is optional and can be used to check if the expected hash is correct, but as a client you should not rely on it, but only on the hash in the signature.

Returns:

a Object[] with the following properties for each block:

1. `blockHash` : hex - the blockhash signed.
2. `block` : number - the blockNumber
3. `r` : hex - r-value of the signature
4. `s` : hex - s-value of the signature
5. `v` : number- v-value of the signature
6. `msgHash` : the msgHash signed. This Hash is created :

```
keccak256(
    abi.encodePacked(
        _blockhash,
        _blockNumber,
        registryId
    )
)
```

Request:

```
{
  "method":"in3_sign",
  "params":[{"blockNumber":8770580}]
}
```

Response:

```
{
  "id": 1,
  "result": [
    {
      "blockHash": "0xd8189793f64567992eaadefc51834f3d787b03e9a6850b8b9b8003d8d84a76c8
↪",
      "block": 8770580,
      "r": "0x954ed45416e97387a55b2231bff5dd72e822e4a5d60fa43bc9f9e49402019337",
      "s": "0x277163f586585092d146d0d6885095c35c02b360e4125730c52332cf6b99e596",
      "v": 28,
      "msgHash": "0x40c23a32947f40a2560fcb633ab7fa4f3a96e33653096b17ec613fbf41f946ef"
    }
  ],
  "in3": {
    "lastNodeList": 8669495,
    "currentBlock": 8770590
  }
}
```

8.1.16 in3_whitelist

Returns whitelisted in3-nodes addresses. The whitelist addressed are accquired from whitelist contract that user can specify in request params.

Parameters:

1. `address`: address of whitelist contract

Returns:

- `nodes`: address[] - array of whitelisted nodes addresses.
- `lastWhiteList`: number - the blockNumber of the last change of the in3 white list event.
- `contract`: address - whitelist contract address.
- `totalServer` : number - the total numbers of whitelist nodes.
- `lastBlockNumber` : number - the blockNumber of the last change of the in3 nodes list (usually the last event).

If proof requested the proof section contains the following properties:

- `type` : constant : `accountProof`
- `block` : the serialized blockheader of the latest final block
- `signatures` : a array of signatures from the signers (if requested) of the above block.
- `accounts`: a Object with the addresses of the whitelist contract as key and Proof as value. The Data Structure of the Proof is exactly the same as the result of - `eth_getProof` and this proof is for proofHash of byte array at storage location 0 in whitelist contract. This byte array is of whitelisted nodes addresses.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Request:

```
{
    "jsonrpc": "2.0",
    "method": "in3_whiteList",
    "params": ["0x08e97ef0a92EB502a1D7574913E2a6636BeC557b"],
    "id": 2,
    "in3": {
        "chainId": "0x5",
         "verification": "proofWithSignature",
         "signatures": [
            "0x45d45e6Ff99E6c34A235d263965910298985fcFe"
        ]
    }
}
```

Response:

```
{
    "id": 2,
    "result": {
        "totalServers": 2,
        "contract": "0x08e97ef0a92EB502a1D7574913E2a6636BeC557b",
        "lastBlockNumber": 1546354,
        "nodes": [
            "0x1fe2e9bf29aa1938859af64c413361227d04059a",
```

(continues on next page)

(continued from previous page)

```
            "0x45d45e6ff99e6c34a235d263965910298985fcfe"
        ]
    },
    "jsonrpc": "2.0",
    "in3": {
        "execTime": 285,
        "lastValidatorChange": 0,
        "proof": {
            "type": "accountProof",
            "block":
↪"0xf9025ca0082a4e766b4af76b7be75818f25310cbc684ccfbd747a4ccb6cacfb4f870d06ba01dcc4de8dec75d7aab85b
↪",
            "accounts": {
                "0x08e97ef0a92EB502a1D7574913E2a6636BeC557b": {
                    "accountProof": [

↪"0xf90211a00cb35d3a4253dde597f30682518f94cbac7690d54dc51bb091f67012e606ee1ea065e37ac9eb1773bceb22c
↪",

↪"0xf90211a0d6cce0c7317d26a22e192288b47a5a34ab7aed0b301c249f27a481f5518e4013a05cc0d414a10bdb4a9f1d6
↪",

↪"0xf90211a0432a3bf286f659650359ae590aa340ce2a2a0d1f60fae509ea9d6a8b90215bfea06b2ab1984e6e8d80eac8d
↪",

↪"0xf8d1a06f998e7193562c27933250e1e72c5a2ff0bf2df556fe478b4436e8b8ac7a7900808080a0de5d6d0bab81e7a0d
↪",

↪"0xf8518080808080808080a03dd3d6e0c95682f178213fd20364be0395c9e94086eb373fd4aa13ebe4ab3ee280808080808
↪",

↪"0xf8679e39ce2fd3705a1089a91865fc977c0a778d01f4f3ba9a0fd6378abecef87ab846f8440180a0f5e650b7122ddd2
↪"
                    ],
                    "address": "0x08e97ef0a92eb502a1d7574913e2a6636bec557b",
                    "balance": "0x0",
                    "codeHash":
↪"0x640aaa823fe1752d44d83bcfd0081ec6a1dc72bb82223940a621b0ea251b52c4",
                    "nonce": "0x1",
                    "storageHash":
↪"0xf5e650b7122ddd254ecc84d87c04ea99117f12badec917985f5f3335b355cb5e",
                    "storageProof": [
                        {
                            "key": "0x0",
                            "proof": [

↪"0xf90111a05541df1966b288bce9c5b6f93d564e736f3f984cb3aa4b067ba88e4398bdc86da06483c09a5b5f8f4206d30
↪",

↪"0xf8518080808080808080a02b2bb6a045f22c77b07ecf8b1f7655f7ed4ccb826b16681ccf1965d4b72ad6df8080808
↪",

↪"0xf843a0200decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563a1a06aa7bbfbb1778efa33da1
↪"
                            ],
                            "value":
↪"0x6aa7bbfbb1778efa33da1ba032cc3a79b9ef57b428441b4de4f1c38c3f258874"
```

(continues on next page)

(continued from previous page)


```
                        }
                    ]
                }
            },
            "signatures": [
                {
                    "blockHash":
↪"0x2d775ab9b1290f487065e612942a84fc2275572e467040eea154fbbae2005c41",
                    "block": 1798342,
                    "r":
↪"0xf6036400705455c1dfb431e1c90b91f3e50815516577f1ebca9a494164b12d17",
                    "s":
↪"0x30e77bc851e02fc79deab63812203b2dfcacd7a83af14a86c8c9d26d95763cc5",
                    "v": 28,
                    "msgHash":
↪"0x7953b8a420bfe9d1c902e2090f533c9b3f73f0f825b7cec247d7d94e548bc5d9"
                }
            ]
        },
        "lastWhiteList": 1546354
    }
}
```


8.2 eth

Standard JSON-RPC calls as described in https://eth.wiki/json-rpc/API.

Whenever a request is made for a response with `verification`: `proof`, the node must provide the proof needed to validate the response result. The proof itself depends on the chain.

For ethereum, all proofs are based on the correct block hash. That's why verification differentiates between Verifying the blockhash (which depends on the user consensus) the actual result data.

There is another reason why the BlockHash is so important. This is the only value you are able to access from within a SmartContract, because the evm supports a OpCode (`BLOCKHASH`), which allows you to read the last 256 blockhashes, which gives us the chance to verify even the blockhash onchain.

Depending on the method, different proofs are needed, which are described in this document.

Proofs will add a special in3-section to the response containing a `proof`- object. Each `in3`-section of the response containing proofs has a property with a proof-object with the following properties:

- **type** `string` (required) - The type of the proof.Must be one of the these values : `'transactionProof`', `'receiptProof`', `'blockProof`', `'accountProof`', `'callProof`', `'logProof`'
- **block** `string` - The serialized blockheader as hex, required in most proofs.
- **finalityBlocks** `array` - The serialized following blockheaders as hex, required in case of finality asked (only relevant for PoA-chains). The server must deliver enough blockheaders to cover more then 50% of the validators. In order to verify them, they must be linkable (with the parentHash).
- **transactions** `array` - The list of raw transactions of the block if needed to create a merkle trie for the transactions.
- **uncles** `array` - The list of uncle-headers of the block. This will only be set if full verification is required in order to create a merkle tree for the uncles and so prove the uncle_hash.

- **merkleProof** `string[]` - The serialized merkle-nodes beginning with the root-node (depending on the content to prove).
- **merkleProofPrev** `string[]` - The serialized merkle-nodes beginning with the root-node of the previous entry (only for full proof of receipts).
- **txProof** `string[]` - The serialized merkle-nodes beginning with the root-node in order to proof the transactionIndex (only needed for transaction receipts).
- **logProof** *LogProof* - The Log Proof in case of a `eth_getLogs`-request.
- **accounts** `object` - A map of addresses and their AccountProof.
- **txIndex** `integer` - The transactionIndex within the block (for transaactions and receipts).
- **signatures** `Signature[]` - Requested signatures.

8.2.1 web3_clientVersion

Returns the underlying client version.

See web3_clientversion for spec.

No proof or verification possible.

8.2.2 web3_sha3

Returns Keccak-256 (not the standardized SHA3-256) of the given data.

See web3_sha3 for spec.

No proof returned, but the client must verify the result by hashing the request data itself.

8.2.3 net_version

Returns the current network ID.

See net_version for spec.

No proof returned, but the client must verify the result by comparing it to the used chainId.

8.2.4 eth_accounts

returns a array of account-addresss the incubed client is able to sign with. In order to add keys, you can use *in3_addRawKey*.

8.2.5 eth_blockNumber

Returns the number of the most recent block.

See eth_blockNumber for spec.

No proof returned, since there is none, but the client should verify the result by comparing it to the current blocks returned from others. With the `blockTime` from the chainspec, including a tolerance, the current blocknumber may be checked if in the proposed range.

8.2.6 eth_getBlockByNumber

See *block based proof*

8.2.7 eth_getBlockByHash

Return the block data and proof.

See JSON-RPC-Spec

- eth_getBlockByNumber - find block by number.
- eth_getBlockByHash - find block by hash.

The `eth_getBlockBy...` methods return the Block-Data. In this case, all we need is somebody verifying the blockhash, which is done by requiring somebody who stored a deposit and would otherwise lose it, to sign this blockhash.

The verification is then done by simply creating the blockhash and comparing this to the signed one.

The blockhash is calculated by serializing the blockdata with rlp and hashing it:

```
blockHeader = rlp.encode([
  bytes32( parentHash ),
  bytes32( sha3Uncles ),
  address( miner || coinbase ),
  bytes32( stateRoot ),
  bytes32( transactionsRoot ),
  bytes32( receiptsRoot || receiptRoot ),
  bytes256( logsBloom ),
  uint( difficulty ),
  uint( number ),
  uint( gasLimit ),
  uint( gasUsed ),
  uint( timestamp ),
  bytes( extraData ),

  ... sealFields
    ? sealFields.map( rlp.decode )
    : [
      bytes32( b.mixHash ),
      bytes8( b.nonce )
    ]
])
```

For POA-chains, the blockheader will use the `sealFields` (instead of mixHash and nonce) which are already RLP-encoded and should be added as raw data when using rlp.encode.

```
if (keccak256(blockHeader) !== singedBlockHash)
  throw new Error('Invalid Block')
```

In case of the `eth_getBlockTransactionCountBy...`, the proof contains the full blockHeader already serialized plus all transactionHashes. This is needed in order to verify them in a merkle tree and compare them with the `transactionRoot`.

Requests requiring proof for blocks will return a proof of type `blockProof`. Depending on the request, the proof will contain the following properties:

- `type` : constant : `blockProof`

- `signatures` : a array of signatures from the signers (if requested) of the requested block.
- `transactions`: a array of raw transactions of the block. This is only needed the last parameter of the request (includeTransactions) is `false`, In this case the result only contains the transactionHashes, but in order to verify we need to be able to build the complete merkle-trie, where the raw transactions are needed. If the complete transactions are included the raw transactions can be build from those values.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.
- `uncles`: only if `fullProof` is requested we add all blockheaders of the uncles to the proof in order to verify the uncleRoot.

Request:

```
{
    "method": "eth_getBlockByNumber",
    "params": [
        "0x967a46",
        false
    ],
    "in3": {
      "verification":"proof"
    }
}
```

Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "author": "0x00d6cc1ba9cf89bd2e58009741f4f7325badc0ed",
        "difficulty": "0xfffffffffffffffffffffffffffffffe",
        "extraData": "0xde830201088f5061726974792d457468657265756d86312e33302e30827769
↪",
        "gasLimit": "0x7a1200",
        "gasUsed": "0x1ce0f",
        "hash": "0xfeb120ae45f1009e6c2289436d5957c58a15915288ec083658bd044101608f26",
        "logsBloom": "0x0008000...",
        "miner": "0x00d6cc1ba9cf89bd2e58009741f4f7325badc0ed",
        "number": "0x967a46",
        "parentHash":
↪"0xc591335e0cdb6b21dc9af57567a6e075fc6315aff915bd79bf78a2c8815bc657",
        "receiptsRoot":
↪"0xfa2a0b3c0715e798ae41fd4645b0261ae4bf6d2c56f29da6fcc5fbfb7c6f19f8",
        "sealFields": [
            "0x8417098353",

↪"0xb841eb80c1a0be2eb7a1c14fc38759a0f9fe9c33121d72003025160a4b35119d495d34d39a9fd7475d28ba863e35f510
↪"
        ],
        "sha3Uncles":
↪"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
        "size": "0x44e",
        "stateRoot":
↪"0xd618159b6dbd0c6213d90abbf01e06513104f0670cd79503cb2563d7ff116864",
        "timestamp": "0x5c260d4c",
        "totalDifficulty": "0x9437370000000000000000000000000484b6f390",
        "transactions": [
```

(continues on next page)

(continued from previous page)


```
            "0x16cfadb6a0a823c623788713cb1eb7d399f89f78d599d416f7b91dca44eeb804",
            "0x91458145d2c47527eee34e891879ac2915b3f8ba6f31911c5234928ae32cb191"
        ],
        "transactionsRoot":
↪"0x4f1249c6378282b1f032cc8c2562712f2450a0bed8ce20bdd2d01b6520feb75a",
        "uncles": []
    },
    "id": 77,
    "in3": {
        "proof": {
            "type": "blockProof",
            "signatures": [ ...  ],
            "transactions": [
                "0xf8ac8201158504a817c8....",
                "0xf9014c8301a3d4843b9ac....",
            ]
        },
        "currentBlock": 9866910,
        "lastNodeList": 8057063,
    }
}
```


8.2.8 eth_getBlockTransactionCountByHash

See *transaction count proof*

8.2.9 eth_getBlockTransactionCountByNumber

See *transaction count proof*

8.2.10 eth_getUncleCountByBlockHash

See *count proof*

8.2.11 eth_getUncleCountByBlockNumber

return the number of transactions or uncles.

See JSON-RPC-Spec

- eth_getBlockTransactionCountByHash - number of transaction by block hash.
- eth_getBlockTransactionCountByNumber - number of transaction by block number.
- eth_getUncleCountByBlockHash - number of uncles by block number.
- eth_getUncleCountByBlockNumber - number of uncles by block number.

Requests requiring proof for blocks will return a proof of type `blockProof`. Depending on the request, the proof will contain the following properties:

- `type` : constant : `blockProof`
- `signatures` : a array of signatures from the signers (if requested) of the requested block.

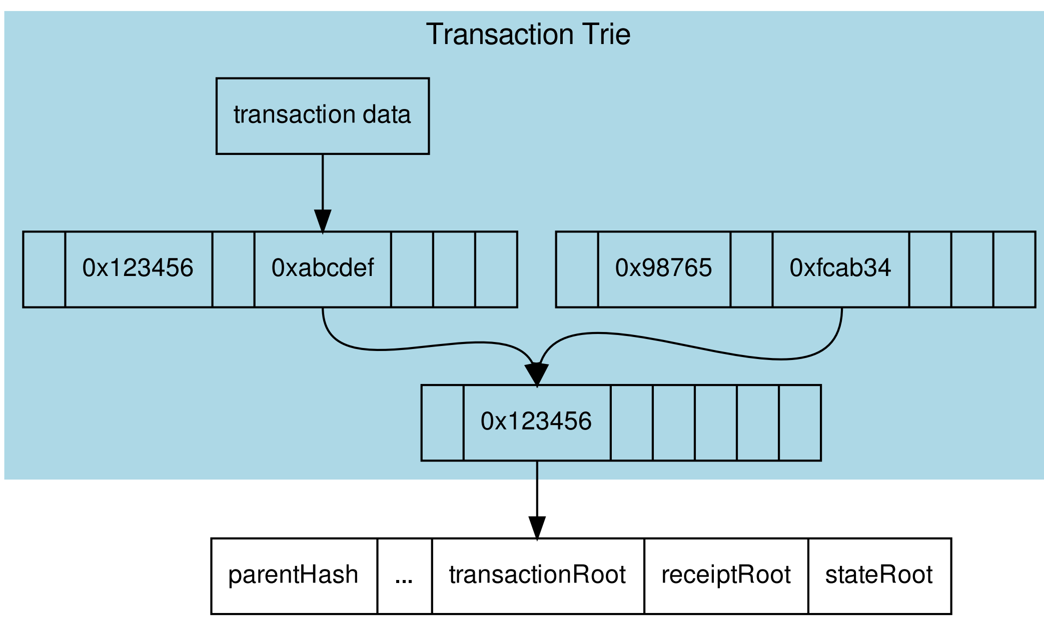
- `block` : the serialized blockheader
- `transactions`: a array of raw transactions of the block. This is only needed if the number of transactions are requested.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.
- `uncles`: a array of blockheaders of the uncles of the block. This is only needed if the number of uncles are requested.

8.2.12 eth_getTransactionByHash

return the transaction data.

See JSON-RPC-Spec

- eth_getTransactionByHash - transaction data by hash.
- eth_getTransactionByBlockHashAndIndex - transaction data based on blockhash and index
- eth_getTransactionByBlockNumberAndIndex - transaction data based on block number and index



In order to prove the transaction data, each transaction of the containing block must be serialized

```
transaction = rlp.encode([
  uint( tx.nonce ),
  uint( tx.gasPrice ),
  uint( tx.gas || tx.gasLimit ),
  address( tx.to ),
  uint( tx.value ),
  bytes( tx.input || tx.data ),
```

(continues on next page)

(continued from previous page)

```
  uint( tx.v ),
  uint( tx.r ),
  uint( tx.s )
])
```

and stored in a merkle tree with `rlp.encode(transactionIndex)` as key or path, since the blockheader only contains the `transactionRoot`, which is the root-hash of the resulting merkle tree. A merkle-proof with the transactionIndex of the target transaction will then be created from this tree.

If the request requires proof (`verification: proof`) the node will provide an Transaction Proof as part of the in3-section of the response. This proof section contains the following properties:

- `type` : constant : `transactionProof`
- `block` : the serialized blockheader of the requested transaction.
- `signatures` : a array of signatures from the signers (if requested) of the above block.
- `txIndex` : The TransactionIndex as used in the MerkleProof (not needed if the methode was `eth_getTransactionByBlock...`, since already given)
- `merkleProof`: the serialized nodes of the Transaction trie starting with the root node.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

While there is no proof for a non existing transaction, if the request was a `eth_getTransactionByBlock...` the node must deliver a partial merkle-proof to verify that this node does not exist.

Request:

```
{
  "method":"eth_getTransactionByHash",
  "params":["0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b"],
  "in3":{
    "verification":"proof"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "blockHash": "0xf1a2fd6a36f27950c78ce559b1dc4e991d46590683cb8cb84804fa672bca395b",
    "blockNumber": "0xca",
    "from": "0x7e5f4552091a69125d5dfcb7b8c2659029395bdf",
    "gas": "0x55f0",
    "gasPrice": "0x0",
    "hash": "0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b",
    "input": "0x00",
    "value": "0x3e8"
    ...
  },
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block": "0xf901e6a040997a53895b48...", // serialized blockheader
```

(continues on next page)

(continued from previous page)

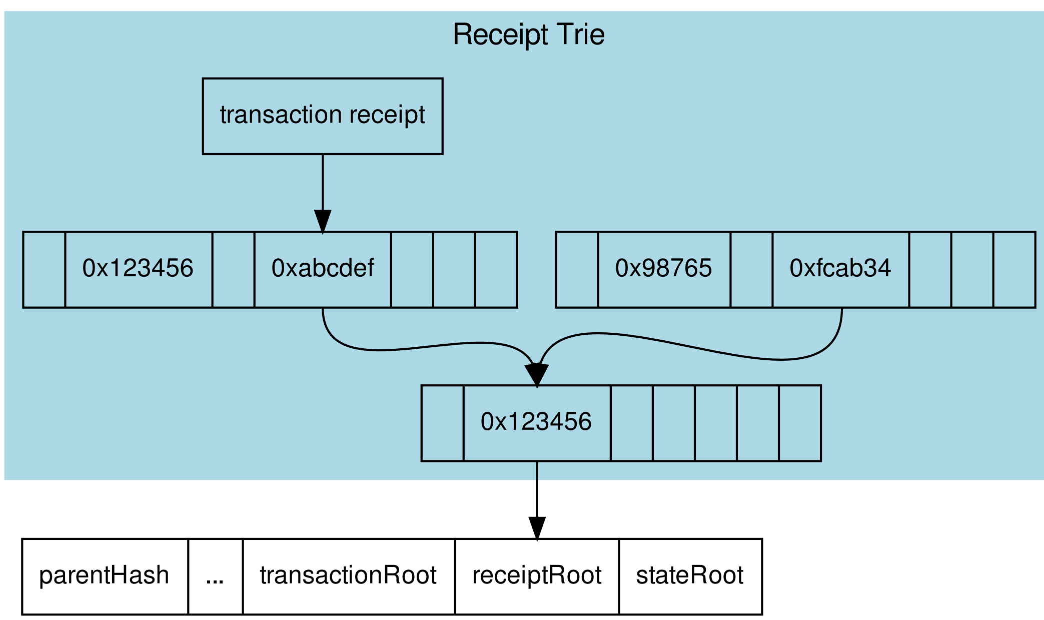
```
      "merkleProof": [  /* serialized nodes starting with the root-node */
        "0xf868822080b863f86136808255f0942b5ad5c4795c026514f8317c7a215e218dc..."
        "0xcd6cf8203e8001ca0dc967310342af5042bb64c34d3b92799345401b26713b43f..."
      ],
      "txIndex": 0,
      "signatures": [...]
    }
  }
}
```

8.2.13 eth_getTransactionReceipt

The Receipt of a Transaction.

See JSON-RPC-Spec

- eth_getTransactionReceipt - returns the receipt.



The proof works similiar to the transaction proof.

In order to create the proof we need to serialize all transaction receipts

```
transactionReceipt = rlp.encode([
  uint( r.status || r.root ),
  uint( r.cumulativeGasUsed ),
  bytes256( r.logsBloom ),
  r.logs.map(l => [
    address( l.address ),
    l.topics.map( bytes32 ),
```

(continues on next page)

(continued from previous page)

```
    bytes( l.data )
  ])
].slice(r.status === null && r.root === null ? 1 : 0))
```

and store them in a merkle tree with `rlp.encode(transactionIndex)` as key or path, since the blockheader only contains the `receiptRoot`, which is the root-hash of the resulting merkle tree. A merkle proof with the transactionIndex of the target transaction receipt will then be created from this tree.

Since the merkle proof is only proving the value for the given transactionIndex, we also need to prove that the transactionIndex matches the transactionHash requested. This is done by adding another MerkleProof for the transaction itself as described in the *Transaction Proof*.

If the request requires proof (`verification: proof`) the node will provide an Transaction Proof as part of the in3-section of the response. This proof section contains the following properties:

- `type` : constant : `receiptProof`
- `block` : the serialized blockheader of the requested transaction.
- `signatures` : a array of signatures from the signers (if requested) of the above block.
- `txIndex` : The TransactionIndex as used in the MerkleProof
- `txProof` : the serialized nodes of the Transaction trie starting with the root node. This is needed in order to proof that the required transactionHash matches the receipt.
- `merkleProof`: the serialized nodes of the Transaction Receipt trie starting with the root node.
- `merkleProofPrev`: the serialized nodes of the previous Transaction Receipt (if txInxdex>0) trie starting with the root node. This is only needed if full-verification is requested. With a verified previous Receipt we can proof the `usedGas`.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Request:

```
{
  "method": "eth_getTransactionReceipt",
  "params": [
      "0x5dc2a9ec73abfe0640f27975126bbaf14624967e2b0b7c2b3a0fb6111f0d3c5e"
  ]
  "in3":{
    "verification":"proof"
  }
}
```

Response:

```
{
    "result": {
        "blockHash":
↪"0xea6ee1e20d3408ad7f6981cfcc2625d80b4f4735a75ca5b20baeb328e41f0304",
        "blockNumber": "0x8c1e39",
        "contractAddress": null,
        "cumulativeGasUsed": "0x2466d",
        "gasUsed": "0x2466d",
        "logs": [
            {
                "address": "0x85ec283a3ed4b66df4da23656d4bf8a507383bca",
```

(continues on next page)

(continued from previous page)


```
                "blockHash":
↪"0xea6ee1e20d3408ad7f6981cfcc2625d80b4f4735a75ca5b20baeb328e41f0304",
                "blockNumber": "0x8c1e39",
                "data": "0x00000000000...",
                "logIndex": "0x0",
                "removed": false,
                "topics": [

↪"0x9123e6a7c5d144bd06140643c88de8e01adcbb24350190c02218a4435c7041f8",

↪"0xa2f7689fc12ea917d9029117d32b9fdef2a53462c853462ca86b71b97dd84af6",

↪"0x55a6ef49ec5dcf6cd006d21f151f390692eedd839c813a150000000000000000"
                ],
                "transactionHash":
↪"0x5dc2a9ec73abfe0640f27975126bbaf14624967e2b0b7c2b3a0fb6111f0d3c5e",
                "transactionIndex": "0x0",
                "transactionLogIndex": "0x0",
                "type": "mined"
            }
        ],
        "logsBloom": "0x00000000000000000000200000...",
        "root": null,
        "status": "0x1",
        "transactionHash":
↪"0x5dc2a9ec73abfe0640f27975126bbaf14624967e2b0b7c2b3a0fb6111f0d3c5e",
        "transactionIndex": "0x0"
    },
    "in3": {
        "proof": {
            "type": "receiptProof",
            "block": "0xf9023fa019e9d929ab...",
            "txProof": [
                "0xf851a083c8446ab932130..."
            ],
            "merkleProof": [
                "0xf851a0b0f5b7429a54b10..."
            ],
            "txIndex": 0,
            "signatures": [...],
            "merkleProofPrev": [
                "0xf851a0b0f5b7429a54b10..."
            ]
        },
        "currentBlock": 9182894,
        "lastNodeList": 6194869
    }
}
```


8.2.14 eth_getLogs

Proofs for logs or events.

See JSON-RPC-Spec

- eth_getLogs - returns all event matching the filter.

Since logs or events are based on the TransactionReceipts, the only way to prove them is by proving the Transaction-Receipt each event belongs to.

That's why this proof needs to provide:

- all blockheaders where these events occured
- all TransactionReceipts plus their MerkleProof of the logs
- all MerkleProofs for the transactions in order to prove the transactionIndex

The proof data structure will look like this:

```
Proof {
  type: 'logProof',
  logProof: {
    [blockNr: string]: {  // the blockNumber in hex as key
      block : string  // serialized blockheader
      receipts: {
        [txHash: string]: {  // the transactionHash as key
          txIndex: number // transactionIndex within the block
          txProof: string[] // the merkle Proof-Array for the transaction
          proof: string[] // the merkle Proof-Array for the receipts
        }
      }
    }
  }
}
```

In order to create the proof, we group all events into their blocks and transactions, so we only need to provide the blockheader once per block. The merkle-proofs for receipts are created as described in the *Receipt Proof*.

If the request requires proof (`verification: proof`) the node will provide an Transaction Proof as part of the in3-section of the response. This proof section contains the following properties:

- `type` : constant : `logProof`
- `logProof` : The proof for all the receipts. This structure contains an object with the blockNumbers as keys. Each block contains the blockheader and the receipt proofs.
- `signatures` : a array of signatures from the signers (if requested) of the above blocks.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Request:

```
{
  "method": "eth_getLogs",
  "params": [
      {
          "fromBlock": "0x7ae000",
          "toBlock": "0x7af0e4",
          "address": "0x27a37a1210df14f7e058393d026e2fb53b7cf8c1"
      }
  ],
  "in3":{
    "verification":"proof"
  }
}
```

Response:

```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "address": "0x27a37a1210df14f7e058393d026e2fb53b7cf8c1",
            "blockHash":
↪"0x12657acc9dbca74775efcc09bcd55da769e89fff27a0402e02708a6e69caa3bb",
            "blockNumber": "0x7ae16b",
            "data": "0x0000000000000...",
            "logIndex": "0x0",
            "removed": false,
            "topics": [
                "0x690cd1ace756531abc63987913dcfaf18055f3bd6bb27d3def1cc5319ebc1461"
            ],
            "transactionHash":
↪"0xddc81454b0df60fb31dbefd0fd4c5e8fe4f3daa541c879964500d876056e2976",
            "transactionIndex": "0x0",
            "transactionLogIndex": "0x0",
            "type": "mined"
        },
        {
            "address": "0x27a37a1210df14f7e058393d026e2fb53b7cf8c1",
            "blockHash":
↪"0x2410d512d12e18b2451efe195ece85723b7f39c3f5d706ea112bfcc57c0249d2",
            "blockNumber": "0x7af0e4",
            "data": "0x000000000000000...",
            "logIndex": "0x4",
            "removed": false,
            "topics": [
                "0x690cd1ace756531abc63987913dcfaf18055f3bd6bb27d3def1cc5319ebc1461"
            ],
            "transactionHash":
↪"0x30fe995d61a5491a49e8f1283b36f4cb7fa5d370927bd8784c33e702546a9daa",
            "transactionIndex": "0x4",
            "transactionLogIndex": "0x0",
            "type": "mined"
        }
    ],
    "id": 144,
    "in3": {
        "proof": {
            "type": "logProof",
            "logProof": {
                "0x7ae16b": {
                    "number": 8053099,
                    "receipts": {

↪"0xddc81454b0df60fb31dbefd0fd4c5e8fe4f3daa541c879964500d876056e2976": {
                            "txHash":
↪"0xddc81454b0df60fb31dbefd0fd4c5e8fe4f3daa541c879964500d876056e2976",
                            "txIndex": 0,
                            "proof": [
                                "0xf9020e822080b90208f..."
                            ],
                            "txProof": [
                                "0xf8f7822080b8f2f8f080..."
                            ]
```


(continues on next page)

(continued from previous page)

```
                        }
                    },
                    "block": "0xf9023ea002343274..."
                },
                "0x7af0e4": {
                    "number": 8057060,
                    "receipts": {
                        
↪"0x30fe995d61a5491a49e8f1283b36f4cb7fa5d370927bd8784c33e702546a9daa": {
                            "txHash":
↪"0x30fe995d61a5491a49e8f1283b36f4cb7fa5d370927bd8784c33e702546a9daa",
                            "txIndex": 4,
                            "proof": [
                                "0xf851a039faec6276...",
                                "0xf8b180a0ee82c377...",
                                "0xf9020c20b90208f9..."
                            ],
                            "txProof": [
                                "0xf851a09250840f6b87...",
                                "0xf8b180a04e5257328b...",
                                "0xf8f620b8f3f8f18085..."
                            ]
                        }
                    },
                    "block": "0xf9023ea03837491e4b3b..."
                }
            }
        },
        "lastValidatorChange": 0,
        "lastNodeList": 8057063
    }
}
```

8.2.15 eth_getBalance

See *account proof*

8.2.16 eth_getCode

See *account proof*

8.2.17 eth_getTransactionCount

See *account proof*

8.2.18 eth_getStorageAt

Returns account based values and proof.

See JSON-RPC-Spec

- eth_getBalance - returns the balance.

- eth_getCode - the byte code of the contract.
- eth_getTransactionCount - the nonce of the account.
- eth_getStorageAt - the storage value for the given key of the given account.

Each of these account values are stored in the account-object:

```
account = rlp.encode([
  uint( nonce),
  uint( balance),
  bytes32( storageHash || ethUtil.KECCAK256_RLP),
  bytes32( codeHash || ethUtil.KECCAK256_NULL)
])
```

The proof of an account is created by taking the state merkle tree and creating a MerkleProof. Since all of the above RPC-methods only provide a single value, the proof must contain all four values in order to encode them and verify the value of the MerkleProof.

For verification, the `stateRoot` of the blockHeader is used and `keccak(accountProof.address)` as the path or key within the merkle tree.

```
verifyMerkleProof(
 block.stateRoot, // expected merkle root
 keccak(accountProof.address), // path, which is the hashed address
 accountProof.accountProof), // array of Buffer with the merkle-proof-data
 isNotExistend(accountProof) ? null : serializeAccount(accountProof), // the expected␣
↪serialized account
)
```

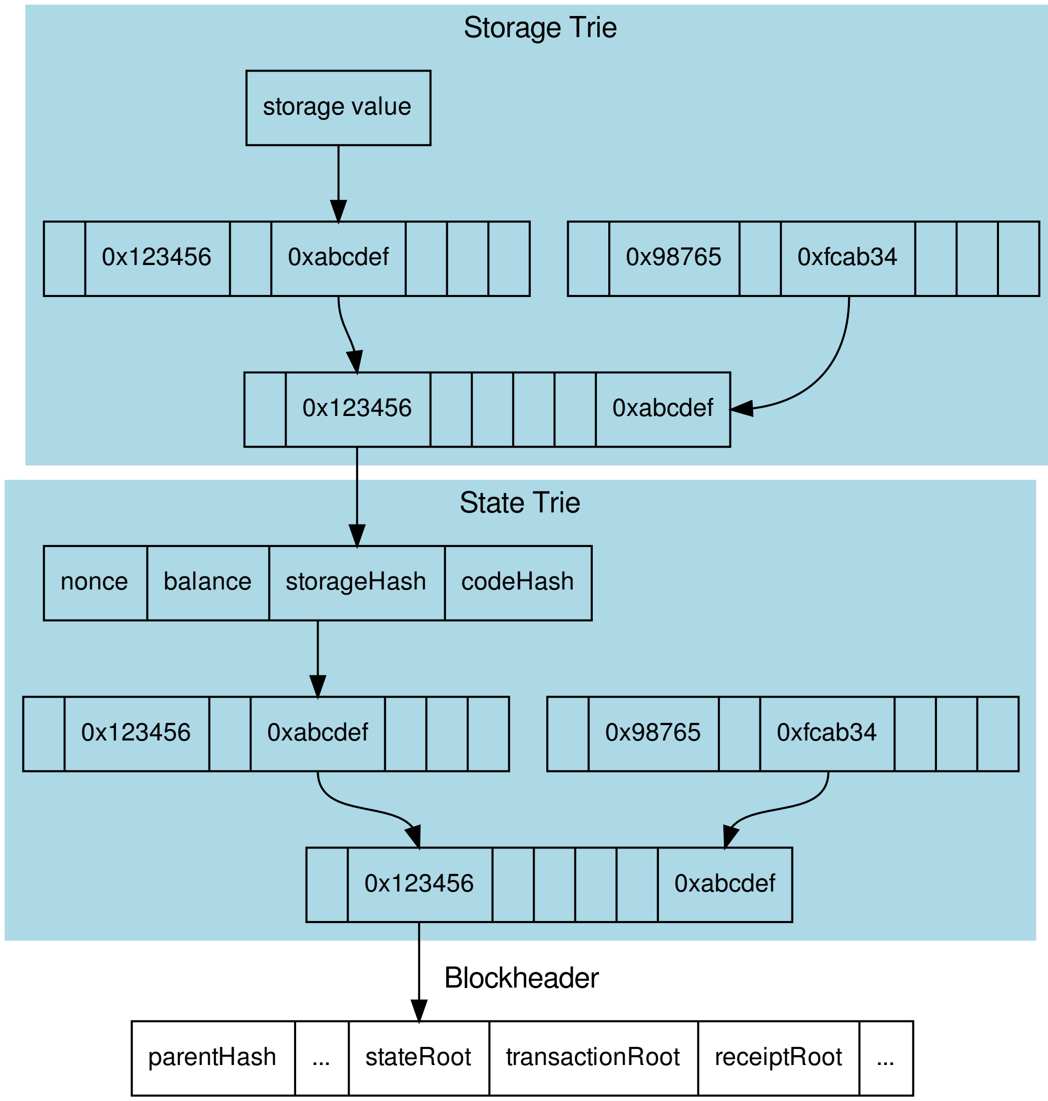
In case the account does not exist yet (which is the case if `none == startNonce` and `codeHash == '0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470'`), the proof may end with one of these nodes:

- The last node is a branch, where the child of the next step does not exist.
- The last node is a leaf with a different relative key.

Both would prove that this key does not exist.

For `eth_getStorageAt`, an additional storage proof is required. This is created by using the `storageHash` of the account and creating a MerkleProof using the hash of the storage key (`keccak(key)`) as path.

```
verifyMerkleProof(
  bytes32( accountProof.storageHash ),   // the storageRoot of the account
  keccak(bytes32(s.key)),  // the path, which is the hash of the key
  s.proof.map(bytes), // array of Buffer with the merkle-proof-data
  s.value === '0x0' ? null : util.rlp.encode(s.value) // the expected value or none␣
↪to proof non-existence
))
```



If the request requires proof (`verification: proof`) the node will provide an Account Proof as part of the in3-section of the response. This proof section contains the following properties:

- `type` : constant : `accountProof`
- `block` : the serialized blockheader of the requested block (the last parameter of the request)
- `signatures` : a array of signatures from the signers (if requested) of the above block.
- `accounts`: a Object with the addresses of all required accounts (in this case it is only one account) as key and Proof as value. The DataStructure of the Proof for each account is exactly the same as the result of - `eth_getProof`.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Example

Request:

```
{
    "method": "eth_getStorageAt",
    "params": [
        "0x27a37a1210Df14f7E058393d026e2fB53B7cf8c1",
        "0x0",
        "latest"
    ],
    "in3": {
      "verification":"proof"
    }
}
```

Response:

```
{
    "id": 77,
    "jsonrpc": "2.0",
    "result": "0x5",
    "in3": {
        "proof": {
            "type": "accountProof",
            "block": "0xf90246...",
            "signatures": [...],
            "accounts": {
                "0x27a37a1210Df14f7E058393d026e2fB53B7cf8c1": {
                    "accountProof": [
                        "0xf90211a0bf....",
                        "0xf90211a092....",
                        "0xf90211a0d4....",
                        "0xf90211a084....",
                        "0xf9019180a0...."
                    ],
                    "address": "0x27a37a1210df14f7e058393d026e2fb53b7cf8c1",
                    "balance": "0x11c37937e08000",
                    "codeHash":
↪"0x3b4e727399e02beb6c92e8570b4ccdd24b6a3ef447c89579de5975edd861264e",
                    "nonce": "0x1",
                    "storageHash":
↪"0x595b6b8bfaad7a24d0e5725ba86887c81a9d99ece3afcce1faf508184fcbe681",
                    "storageProof": [
                        {
                            "key": "0x0",
                            "proof": [
                                "0xf90191a08e....",
                                "0xf871808080....",

↪"0xe2a0200decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56305"
                            ],
                            "value": "0x5"
                        }
                    ]
                }
            }
        },
```

(continues on next page)

(continued from previous page)

```
        "currentBlock": 9912897,
        "lastNodeList": 8057063
    }
}
```

8.2.19 eth_estimateGas

See *call proof*

8.2.20 eth_call

calls a function of a contract (or simply executes the evm opcodes).

See JSON-RPC-Spec

- eth_call - executes a function and returns the result.
- eth_estimateGas - executes a function and returns the gas used.

Verifying the result of an `eth_call` is a little bit more complex because the response is a result of executing opcodes in the vm. The only way to do so is to reproduce it and execute the same code. That's why a call proof needs to provide all data used within the call. This means:

- All referred accounts including the code (if it is a contract), storageHash, nonce and balance.
- All storage keys that are used (this can be found by tracing the transaction and collecting data based on the `SLOAD`-opcode).
- All blockdata, which are referred at (besides the current one, also the `BLOCKHASH`-opcodes are referring to former blocks).

For verifying, you need to follow these steps:

1. Serialize all used blockheaders and compare the blockhash with the signed hashes. (See *BlockProof*)
2. Verify all used accounts and their storage as showed in *Account Proof*.
3. Create a new VM with a MerkleTree as state and fill in all used value in the state:

```
  // create new state for a vm
  const state = new Trie()
  const vm = new VM({ state })

  // fill in values
  for (const adr of Object.keys(accounts)) {
    const ac = accounts[adr]

    // create an account-object
    const account = new Account([ac.nonce, ac.balance, ac.stateRoot, ac.codeHash])

    // if we have a code, we will set the code
    if (ac.code) account.setCode( state, bytes( ac.code ))

    // set all storage-values
    for (const s of ac.storageProof)
      account.setStorage( state, bytes32( s.key ), rlp.encode( bytes32( s.value )))
```

(continues on next page)

(continued from previous page)

```
    // set the account data
    state.put( address( adr ), account.serialize())
  }

  // add listener on each step to make sure it uses only values found in the proof
  vm.on('step', ev => {
     if (ev.opcode.name === 'SLOAD') {
        const contract = toHex( ev.address ) // address of the current code
        const storageKey = bytes32( ev.stack[ev.stack.length - 1] ) // last element␣
↪on the stack is the key
        if (!getStorageValue(contract, storageKey))
          throw new Error(`incomplete data: missing key ${storageKey}`)
     }
     /// ... check other opcodes as well
  })

  // create a transaction
  const tx = new Transaction(txData)

  // run it
  const result = await vm.runTx({ tx, block: new Block([block, [], []]) })

  // use the return value
  return result.vm.return
```

In the future, we will be using the same approach to verify calls with ewasm.

If the request requires proof (`verification: proof`) the node will provide an Call Proof as part of the in3-section of the response. Details on how create the proof can be found in the *CallProof-Chapter*. This proof section contains the following properties:

- `type` : constant : `callProof`
- `block` : the serialized blockheader of the requested block (the last parameter of the request)
- `signatures` : a array of signatures from the signers (if requested) of the above block.
- `accounts`: a Object with the addresses of all accounts required to run the call as keys. This includes also all storage values (SLOAD) including proof used. The DataStructure of the Proof for each account is exactly the same as the result of - `eth_getProof`.
- `finalityBlocks`: a array of blockHeaders which were mined after the requested block. The number of blocks depends on the request-property `finality`. If this is not specified, this property will not be defined.

Request:

```
{
    "method": "eth_call",
    "params": [
        {
            "to": "0x2736D225f85740f42D17987100dc8d58e9e16252",
            "data":
↪"0x5cf0f3570000000000000000000000000000000000000000000000000000000000000001"
        },
        "latest"
    ],
    "in3": {
      "verification":"proof"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Response:

```
{
    "result": "0x0000000000000000000000000...",
    "in3": {
        "proof": {
            "type": "callProof",
            "block": "0xf90215a0c...",
            "signatures": [...],
            "accounts": {
                "0x2736D225f85740f42D17987100dc8d58e9e16252": {
                    "accountProof": [
                        "0xf90211a095...",
                        "0xf90211a010...",
                        "0xf90211a062...",
                        "0xf90211a091...",
                        "0xf90211a03a...",
                        "0xf901f1a0d1...",
                        "0xf8b18080808..."
                    ],
                    "address": "0x2736d225f85740f42d17987100dc8d58e9e16252",
                    "balance": "0x4fffb",
                    "codeHash":
↪"0x2b8bdc59ce78fd8c248da7b5f82709e04f2149c39e899c4cdf4587063da8dc69",
                    "nonce": "0x1",
                    "storageHash":
↪"0xbf904e79d4ebf851b2380d81aab081334d79e231295ae1b87f2dd600558f126e",
                    "storageProof": [
                        {
                            "key": "0x0",
                            "proof": [
                                "0xf901f1a0db74...",
                                "0xf87180808080...",
                                "0xe2a0200decd9....05"
                            ],
                            "value": "0x5"
                        },
                        {
                            "key":
↪"0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e569",
                            "proof": [
                                "0xf901f1a0db74...",
                                "0xf891a0795a99...",
                                "0xe2a020ab8540...43"
                            ],
                            "value": "0x43"
                        },
                        {
                            "key":
↪"0xaaab8540682e3a537d17674663ea013e92c83fdd69958f314b4521edb3b76f1a",
                            "proof": [
                                "0xf901f1a0db747...",
                                "0xf891808080808...",
                                "0xf843a0207bd5ee..."
```

(continues on next page)

(continued from previous page)


```
                              ],
                              "value":
↪"0x68747470733a2f2f696e332e736c6f636b2e69742f6d61696e6e65742f6e642d"
                         }
                    ]
               }
          }
     },
     "currentBlock": 8040612,
     "lastNodeList": 6619795
  }
}
```


8.2.21 eth_accounts

8.2.22 eth_sign

8.2.23 eth_sendTransaction

See JSON-RPC-Spec

- eth_accounts - returns the unlocked accounts.
- eth_sign - signs data with an unlocked account.
- eth_sendTransaction - signs and sends a transaction.

Signing is **not supported** since the nodes are serving a public rpc-enpoint. These methods will return a error. The client may still support those methods, but handle those requests internally.

8.2.24 eth_sendTransactionAndWait

Sends a Transaction just like `eth_sendTransaction` but instead of returning the TransactionHash it will wait until the transaction is mined and return the transaction receipt. See eth_getTransactionReceipt.

8.2.25 eth_sendRawTransaction

See JSON-RPC-Spec

- eth_sendRawTransaction - sends a prviously signed transaction.

This Method does not require any proof. (even if requested). Clients must at least verify the returned transactionHash by hashing the rawTransaction data. To know whether the transaction was actually broadcasted and mined, the client needs to run a second request `eth_getTransactionByHash` which should contain the blocknumber as soon as this is mined.

8.3 ipfs

A Node supporting IPFS must support these 2 RPC-Methods for uploading and downloading IPFS-Content. The node itself will run a ipfs-client to handle them.

Fetching ipfs-content can be easily verified by creating the ipfs-hash based on the received data and comparing it to the requested ipfs-hash. Since there is no chance of manipulating the data, there is also no need to put a deposit or convict a node. That's why the registry-contract allows a zero-deposit fot ipfs-nodes.

8.3.1 ipfs_get

Fetches the data for a requested ipfs-hash. If the node is not able to resolve the hash or find the data a error should be reported.

No proof or verification needed on the server side.

Parameters:

1. `ipfshash`: string - the ipfs multi hash
2. `encoding`: the encoding used for the response. (`hex` , `base64` or `utf8`)

Returns:

the content matching the requested hash.

Request:

```
{
    "method":"ipfs_get",
    "params":[
        "QmSepGsypERjq71BSm4Cjq7j8tyAUnCw6ZDTeNdE8RUssD",
        "utf8"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "I love Incubed",
}
```

8.3.2 ipfs_put

Stores ipfs-content to the ipfs network. Important! As a client there is no garuantee that a node made this content available. (just like `eth_sendRawTransaction` will only broadcast it). Even if the node stores the content there is no gurantee it will do it forever.

Parameters:

1. `data`: string - the content encoded with the specified encoding.
2. `encoding`: the encoding used for the response. (`hex` , `base64` or `utf8`)

Returns:

the ipfs multi hash

Request:

```
{
    "method":"ipfs_put",
    "params":[
```

(continues on next page)

(continued from previous page)

```
        "I love Incubed",
        "utf8"
    ]
}
```

Response:

```
{
  "id": 1,
  "result": "QmSepGsypERjq71BSm4Cjq7j8tyAUnCw6ZDTeNdE8RUssD",
}
```

8.4 btc

For bitcoin incubed follows the specification as defined in https://bitcoincore.org/en/doc/0.18.0/. Internally the in3-server will add proofs as part of the responses. The proof data differs between the methods. You will read which proof data will be provided and how the data can be used to prove the result for each method.

Proofs will add a special `in3`-section to the response containing a `proof`- object. This object will contain parts or all of the following properties:

- **block**
- **final**
- **txIndex**
- **merkleProof**
- **cbtx**
- **cbtxMerkleProof**

8.4.1 btc_getblockheader

Returns data of block header for given block hash. The returned level of details depends on the argument verbosity.

Parameters:

1. `hash` : (string, required) The block hash
2. `verbosity` : (number or boolean, optional, default=1) 0 or false for the hex-encoded data, 1 or true for a json object
3. `in3.finality` : (number, required) defines the amount of finality headers
4. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)
5. `in3.preBIP34` : (boolean, required) defines if the client wants to verify blocks before BIP34 (height < 227836)

Returns:

- verbose `0` or `false`: a hex string with 80 bytes representing the blockheader
- verbose `1` or `true`: an object representing the blockheader:
  - `hash`: hex - the block hash (same as provided)

- `confirmations`: number - The number of confirmations, or -1 if the block is not on the main chain
- `height`: number : The block height or index
- `version`: number - The block version
- `versionHex`: hex - The block version formatted in hexadecimal
- `merkleroot`: hex - The merkle root (32 bytes)
- `time`: number - The block time in seconds since epoch (Jan 1 1970 GMT)
- `mediantime`: number - The median block time in seconds since epoch (Jan 1 1970 GMT)
- `nonce`: number - The nonce
- `bits`: hex - The bits (4 bytes as hex) representing the target
- `difficulty`: number - The difficulty
- `chainwork`: hex - Expected number of hashes required to produce the current chain (in hex)
- `nTx`: number - The number of transactions in the block.
- `previousblockhash`: hex - The hash of the previous block
- `nextblockhash`: hex - The hash of the next block

The `proof`-object contains the following properties:

- for blocks before BIP34 (height < 227,836) and `in3.preBIP34` = false
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
- for blocks before BIP34 (height < 227,836) and `in3.preBIP34` = true
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated up to the next checkpoint (maximum of 200 finality headers, since the distance between checkpoints = 200)
  - `height`: number - the height of the block (block number)
- for blocks after BIP34 (height >= 227,836), *the value of `in3.preBIP34` does not matter*
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
  - `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proofing the correctness of the cbtx.

Old blocks (height < 227,836) with `in3.preBIP34` disabled cannot be verified (proving the finality does not provide any security as explained in preBIP34 proof). Old blocks with `in.preBIP34` enabled can be verified by performing a preBIP34 proof. Verifying newer blocks requires multiple proofs. The finality headers from the `final`-field will be used to perform a finality proof. To verify the block number we are going to perform a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

Example

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getblockheader",
    "params": ["000000000000000000103b2395f6cd94221b10d02eb9be5850303c0534307220",
↪ true],
    "in3":{
        "finality":8,
        "verification":"proof"
        "preBIP34": true
    }
}
```

Response:

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "hash": "000000000000000000103b2395f6cd94221b10d02eb9be5850303c0534307220",
        "confirmations": 8268,
        "height": 624958,
        "version": 536928256,
        "versionHex": "2000e000",
        "merkleroot":
↪"d786a334ea8c65f39272d5b9be505ac3170f3904842bd52525538a9377b359cb",
        "time": 1586333924,
        "mediantime": 1586332639,
        "nonce": 1985217615,
        "bits": "17143b41",
        "difficulty": 13912524048945.91,
        "chainwork": "00000000000000000000000000000000000000000e4c88b66c5ee78deff0d494
↪",
        "nTx": 33,
        "previousblockhash":
↪"00000000000000000013cba040837778744ce66961cfcf2e7c34bb3d194c7f49",
        "nextblockhash":
↪"0000000000000000000c799dc0e36302db7fbb471711f140dc308508ef19e343"
    },
    "in3": {
        "proof": {
            "final": "0x00e0ff2720723034053c305058beb92ed010...276470",
            "cbtx": "0x0100000000010100000000000000000000000...39da2fc",
            "cbtxMerkleProof": "0x6a8077bb4ce76b71d7742ddd368770279a64667b...52e688"
        }
    }
}
```

8.4.2 btc_getblock

Returns data of block for given block hash. The returned level of details depends on the argument verbosity.

Parameters:

1. `blockhash` : (string, required) The block hash
2. `verbosity` : (number or boolean, optional, default=true) 0 or false for hex-encoded data, 1 or true for a json

object, and 2 for json object **with** transaction data

3. `in3.finality` : (number, required) defines the amount of finality headers
4. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)
5. `in3.preBIP34` : (boolean, required) defines if the client wants to verify blocks before BIP34 (height < 227836)

Returns

- verbose `0` or `false` : a string that is serialized, hex-encoded data for block hash
- verbose `1` or `true`: an object representing the block:
  - `hash`: hex - the block hash (same as provided)
  - `confirmations` : number - The number of confirmations, or -1 if the block is not on the main chain
  - `size`:
  - `strippedsize`:
  - `weight`:
  - `height`: number - The block height or index
  - `version`: number - The block version
  - `versionHex`: hex - The block version formatted in hexadecimal
  - `merkleroot`: hex - The merkle root (32 bytes)
  - `tx`: array of string - The transaction ids
  - `time`: number - The block time in seconds since epoch (Jan 1 1970 GMT)
  - `mediantime`: number - The median block time in seconds since epoch (Jan 1 1970 GMT)
  - `nonce`: number - The nonce
  - `bits`: hex - The bits (4 bytes as hex) representing the target
  - `difficulty`: number - The difficulty
  - `chainwork`: hex - Expected number of hashes required to produce the current chain (in hex)
  - `nTx`: number - The number of transactions in the block.
  - `previousblockhash`: hex - The hash of the previous block
  - `nextblockhash`: hex - The hash of the next block
- verbose `2`: an object representing the block with information about each transaction:
  - `...`: same output as `verbosity=1`
  - `tx`: array of objects - The transactions in the format of the getrawtransaction-RPC. `tx` result is different from `verbosity=1`
  - `...`: same output as `verbosity=1`

The `proof`-object contains the following properties:

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = false
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = true
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated up to the next checkpoint (maximum of 200 finality headers, since the distance between checkpoints = 200)
  - `height`: number - the height of the block (block number)
- for blocks after BIP34 (height >= 227836), *the value of* `in3.preBIP34` *does not matter*
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
  - `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proofing the correctness of the cbtx.

Old blocks (height < 227,836) with `in3.preBIP34` disabled cannot be verified (proving the finality does not provide any security as explained in preBIP34 proof). Old blocks with `in.preBIP34` enabled can be verified by performing a preBIP34 proof. Verifying newer blocks requires multiple proofs. The finality headers from the `final`-field will be used to perform a finality proof. To verify the block number we are going to perform a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

Example

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getblock",
    "params":  ["000000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7",␣
↪true],
    "in3":{
        "finality":8,
        "verification":"proof",
        "preBIP34": true
    }
}
```

Response:

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "hash": "000000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7",
        "confirmations": 8226,
        "strippedsize": 914732,
        "size": 1249337,
        "weight": 3993533,
        "height": 625000,
        "version": 1073733632,
        "versionHex": "3fffe000",
        "merkleroot":
↪"4d51591497f1d646070f9f9fdeb50dc338e2a8bb9a5cb721c55f452938165ff8",
        "tx": [
```

(continues on next page)

(continued from previous page)

```
            "d79ffc80e07fe9e0083319600c59d47afe69995b1357be6e5dba035675780290",
            ...
            "6456819bfa019ba30788620153ea9a361083cb888b3662e2ff39c0f7adf16919"
        ],
        "time": 1586364107,
        "mediantime": 1586361287,
        "nonce": 3963275925,
        "bits": "171320bc",
        "difficulty": 14715214060656.53,
        "chainwork": "000000000000000000000000000000000000000000e4eba1824303796d776922b
↪",
        "nTx": 2626,
        "previousblockhash":
↪"000000000000000000068fb1ddc43ca83bc4bfb23444f7236992cfc565d40e08",
        "nextblockhash":
↪"00000000000000000010b3d94671593da669b25fecf7005de38dc2b2fa208dc7"
    },
    "in3": {
        "proof": {
            "final": "0x00e00020d7cd60cab00255bb13bbb023fd5d85daaa...bbd60f",
            "cbtx": "0x01000000000101000000000000000000000000000000000...4ddd5c",
            "cbtxMerkleProof": "0xa22e7468d9bf239167ff6f97d066818b4a5278d29fc13dbcbd5.
↪..4b2f3a"
        }
    }
}
```

8.4.3 btc_getrawtransaction

Returns the raw transaction data. The returned level of details depends on the argument verbosity.

Parameters:

1. `txid` : (string, required) The transaction id
2. `verbosity` : (number or boolean, optional, default=1) 0 or false for the hex-encoded data for `txid`, 1 or true for a json object with information about `txid`
3. `blockhash` : (string, optional) The block in which to look for the transaction
4. `in3.finality` : (number, required) defines the amount of finality headers
5. `in3.verification`: (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)
6. `in3.preBIP34` : (boolean, required) defines if the client wants to verify blocks before BIP34 (height < 227836)

Returns:

- verbose `0` or `false`: a string that is serialized, hex-encoded data for `txid`
- verbose `1` or `false`: an object representing the transaction:
  - `in_active_chain`: boolean - Whether specified block is in the active chain or not (only present with explicit "blockhash" argument)
  - `hex`: string - The serialized, hex-encoded data for `txid`

  - `txid`: string - The transaction id (same as provided)
  - `hash`: string - The transaction hash (differs from txid for witness transactions)
  - `size`: number - The serialized transaction size
  - `vsize`: number - The virtual transaction size (differs from size for witness transactions)
  - `weight`: number - The transaction's weight (between `vsize`*4-3 and `vsize`*4)
  - `version`: number - The version
  - `locktime`: number - The lock time
  - `vin`: array of json objects
    * `txid`: number - The transaction id
    * `vout`: number
    * `scriptSig`: json object - The script
      · `asm`: string - asm
      · `hex`: string - hex
    * `sequence`: number - The script sequence number
    * `txinwitness`: array of string - hex-encoded witness data (if any)
  - `vout`: array of json objects
    * `value`: number - The value in BTC
    * `n`: number - index
    * `scriptPubKey`: json object
      · `asm`: string - asm
      · `hex`: string - hex
      · `reqSigs`: number - The required sigs
      · `type`: string - The type, eg 'pubkeyhash'
      · `addresses`: json array of strings (each representing a bitcoin adress)
  - `blockhash`: string - the block hash
  - `confirmations`: number - The confirmations
  - `blocktime`: number - The block time in seconds since epoch (Jan 1 1970 GMT)
  - `time`: number - Same as "blocktime"

The `proof`-object contains the following properties:

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = false
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `txIndex`: number - index of the transaction (`txIndex=0` for coinbase transaction, necessary to create/verify the merkle proof)

  - `merkleProof`: hex - the merkle proof of the requested transaction, proving the correctness of the transaction

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = true
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated up to the next checkpoint (maximum of 200 finality headers, since the distance between checkpoints = 200)
  - `txIndex`: number - index of the transaction (`txIndex=0` for coinbase transaction, necessary to create/verify the merkle proof)
  - `merkleProof`: hex - the merkle proof of the requested transaction, proving the correctness of the transaction
  - `height`: number - the height of the block (block number)
- for blocks after BIP34 (height >= 227836), *the value of `in3.preBIP34` does not matter*
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `txIndex`: number - index of the transaction (`txIndex=0` for coinbase transaction, necessary to create/verify the merkle proof)
  - `merkleProof`: hex - the merkle proof of the requested transaction, proving the correctness of the transaction
  - `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
  - `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proving the correctness of the `cbtx`

Transactions of old blocks (height < 227836) with `in3.preBIP34` disabled cannot be verified (proving the finality does not provide any security as explained in preBIP34 proof and relying on the merkle proof is only possible when the block is final). Transactions of old blocks with `in3.preBIP34` enabled can be verified by performing a preBIP34 proof and a merkle proof. Verifying newer blocks requires multiple proofs. The block header from the `block`-field and the finality headers from the `final`-field will be used to perform a finality proof. By doing a merkle proof using the `txIndex`-field and the `merkleProof`-field the correctness of the requested transation can be proven. Furthermore we are going to perform a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

Example

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getrawtransaction",
    "params":  ["f3c06e17b04ef748ce6604ad68e5b9f68ca96914b57c2118a1bb9a09a194ddaf",
                true,
                "000000000000000000103b2395f6cd94221b10d02eb9be5850303c0534307220"],
    "in3":{
        "finality":8,
        "verification":"proof",
```

(continues on next page)

(continued from previous page)

```
        "preBIP34": true
    }
}
```

Response:


```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "in_active_chain": true,
        "txid": "f3c06e17b04ef748ce6604ad68e5b9f68ca96914b57c2118a1bb9a09a194ddaf",
        "hash": "f3c06e17b04ef748ce6604ad68e5b9f68ca96914b57c2118a1bb9a09a194ddaf",
        "version": 1,
        "size": 518,
        "vsize": 518,
        "weight": 2072,
        "locktime": 0,
        "vin": [
            {
                "txid":
↪"0a74f6e5f99bc69af80da9f0d9878ea6afbfb5fbb2d43f1ff899bcdd641a098c",
                "vout": 0,
                "scriptSig": {
                    "asm": "30440220481f2b3a49b202e26c73ac1b7bce022e4a74aff08473228cc.
↪..254874",
                    "hex": "4730440220481f2b3a49b202e26c73ac1b7bce022e4a74aff08473228.
↪..254874"
                },
                "sequence": 4294967295
            },
            {
                "txid":
↪"869c5e82d4dfc3139c8a153d2ee126e30a467cf791718e6ea64120e5b19e5044",
                "vout": 0,
                "scriptSig": {
                    "asm": "3045022100ae5bd019a63aed404b743c9ebcc77fbaa657e481f745e4..
↪.f3255d",
                    "hex": "483045022100ae5bd019a63aed404b743c9ebcc77fbaa657e481f745..
↪.f3255d"
                },
                "sequence": 4294967295
            },
            {
                "txid":
↪"8a03d29a1b8ae408c94a2ae15bef8329bc3d6b04c063d36b2e8c997273fa8eff",
                "vout": 1,
                "scriptSig": {
                    "asm": "304402200bf7c5c7caec478bf6d7e9c5127c71505034302056d1284...
↪0045da",
                    "hex": "47304402200bf7c5c7caec478bf6d7e9c5127c71505034302056d12...
↪0045da"
                },
                "sequence": 4294967295
            }
        ],
        "vout": [
```


(continues on next page)

(continued from previous page)


```
            {
                "value": 0.00017571,
                "n": 0,
                "scriptPubKey": {
                    "asm": "OP_DUP OP_HASH160 53196749b85367db9443ef9a5aec25cf0bdceedf OP_EQUALVERIFY OP_CHECKSIG",
                    "hex": "76a91453196749b85367db9443ef9a5aec25cf0bdceedf88ac",
                    "reqSigs": 1,
                    "type": "pubkeyhash",
                    "addresses": [
                        "18aPWzBTq1nzs9o86oC9m3BQbxZWmV82UU"
                    ]
                }
            },
            {
                "value": 0.00915732,
                "n": 1,
                "scriptPubKey": {
                    "asm": "OP_HASH160 8bb2b4b848d0b6336cc64ea57ae989630f447cba OP_EQUAL",
                    "hex": "a9148bb2b4b848d0b6336cc64ea57ae989630f447cba87",
                    "reqSigs": 1,
                    "type": "scripthash",
                    "addresses": [
                        "3ERfvuzAYPPpACivh1JnwYbBdrAjupTzbw"
                    ]
                }
            }
        ],
        "hex": "01000000038c091a64ddbc99f81f3fd4b2fbb5bfafa68e8...000000"
        "blockhash": "000000000000000000103b2395f6cd94221b10d02eb9be5850303c0534307220",
        "confirmations": 15307,
        "time": 1586333924,
        "blocktime": 1586333924
    },
    "in3": {
        "proof": {
            "block": "0x00e00020497f4c193dbb347c2ecfcf6169e64c747877...045476",
            "final": "0x00e0ff2720723034053c305058beb92ed0101b2294cd...276470",
            "txIndex": 7,
            "merkleProof": "0x348d4bb04943400a80f162c4ef64b746bc4af0...52e688",
            "cbtx": "0x010000000001010000000000000000000000000000000...9da2fc",
            "cbtxMerkleProof": "0x6a8077bb4ce76b71d7742ddd368770279a...52e688"
        }
    }
}
```


8.4.4 btc_getblockcount

Returns the number of blocks in the longest blockchain.

Parameters:

1. `in3.finality` : (number, required) defines the amount of finality headers

2. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)

Returns: Since we can't prove the finality of the latest block we consider the `current block count` - `amount of finality` (set in `in3.finality`-field) as the latest block. The number of this block will be returned. Setting `in3.finality=0` will return the actual current block count.

The `proof`-object contains the following properties:

- `block`: hex - a hex string with 80 bytes representing the blockheader
- `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
- `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
- `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proving the correctness of the `cbtx`

The server is not able to prove the finality for the latest block (obviously there are no finality headers available yet). Instead the server will fetch the number of the latest block and subtracts the amount of finality headers (set in `in3.finality`-field) and returns the result to the client (the result is considered as the latest block number). By doing so the server is able to provide finality headers. The block header from the `block`-field and the finality headers from the `final`-field will be used to perform a finality proof. Having a verified block header (and therefore a verified merkle root) enables the possibility of a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

The client can set `in3.finality` equal to `0` to get the actual latest block number. **Caution**: This block is not final and could no longer be part of the blockchain later on due to the possibility of a fork. Additionally, there may already be a newer block that the server does not yet know about due to latency in the network.

Example

The actual latest block is block `#640395` and `in3.finality` is set to `8`. The server is going to calculate `640395 - 8` and returns `640387` as the latest block number to the client. The headers of block `640388`..`640395` will be returned as finality headers.

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getblockcount",
    "params": [],
    "in3":{
        "finality":8,
        "verification":"proof"
    }
}
```

Response:

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": 640387,
    "in3": {
        "proof": {
            "block": "0x0000e020bd3eecbd741522e1aa78cd7b375744590502939aef9b...9c8b18
↪",
            "final": "0x00008020f61dfcc47a6daed717b12221855196dee02d844ebb9c...774f4c
↪",
```

(continues on next page)

(continued from previous page)

```
            "cbtx": "0x02000000000101000000000000000000000000000000000000...000000
↪",
            "cbtxMerkleProof": "0xa3d607b274770911e53f06dbdb76440580ff968239...0ba297"
        }
    }
}
```

8.4.5 btc_getbestblockhash

Returns the hash of the best (tip) block in the longest blockchain.

Parameters:

1. `in3.finality` : (number, required) defines the amount of finality headers
2. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)

Returns: Since we can't prove the finality of the latest block we consider the `current block count` - `amount of finality` (set in `in3.finality`-field) as the latest block. The hash of this block will be returned. Setting `in3.finality=0` will return will return the hash of the actual latest block.

The `proof`-object contains the following properties:

- `block`: hex - a hex string with 80 bytes representing the blockheader
- `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
- `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
- `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proving the correctness of the `cbtx`

The server is not able to prove the finality for the latest block (obviously there are no finality headers available yet). Instead the server will fetch the number of the latest block and subtracts the amount of finality headers (set in `in3.finality`-field) and returns the hash of this block to the client (the result is considered as the latest block hash). By doing so the server is able to provide finality headers. The block header from the `block`-field and the finality headers from the `final`-field will be used to perform a finality proof. Having a verified block header (and therefore a verified merkle root) enables the possibility of a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

The client can set `in3.finality` equal to `0` to get the actual latest block hash. **Caution**: This block is not final and could no longer be part of the blockchain later on due to the possibility of a fork. Additionally, there may already be a newer block that the server does not yet know about due to latency in the network.

Example

The actual latest block is block `#640395` and `in3.finality` is set to `8`. The server is going to calculate `640395` - `8` and returns the hash of block `#640387` to the client. The headers of block `640388`..`640395` will be returned as finality headers.

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getbestblockhash",
    "params": [],
    "in3":{
```

(continues on next page)

(continued from previous page)

```
        "finality":8,
        "verification":"proof"
    }
}
```

Response:

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "000000000000000000039cbb4e842de0de9651852122b117d7ae6d7ac4fc1df6",
    "in3": {
        "proof": {
           "block": "0x0000e020bd3eecbd741522e1aa78cd7b375744590502939aef9b...9c8b18",
            "final": "0x00008020f61dfcc47a6daed717b12221855196dee02d844ebb9...774f4c",
            "cbtx": "0x0200000000010100000000000000000000000000000000000000...000000",
            "cbtxMerkleProof": "0xa3d607b274770911e53f06dbdb76440580ff96823...0ba297"
        }
    }
}
```

8.4.6 btc_getdifficulty

Returns the proof-of-work difficulty as a multiple of the minimum difficulty.

Parameters:

1. `blocknumber` : (string or number, optional) Can be the number of a certain block to get its difficulty. To get the difficulty of the latest block use `latest`, `earliest`, `pending` or leave `params` empty (Hint: Latest block always means `actual latest block` minus `in3.finality`)
2. `in3.finality` : (number, required) defines the amount of finality headers
3. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)
4. `in3.preBIP34` : (boolean, required) defines if the client wants to verify blocks before BIP34 (height < 227836)

Returns:

- `blocknumber` is a certain number: the difficulty of this block
- `blocknumber` is `latest`, `earliest`, `pending` or empty: the difficulty of the latest block (`actual latest block` minus `in3.finality`)

The `proof`-object contains the following properties:

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = false
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = true
  - `block`: hex - a hex string with 80 bytes representing the blockheader

  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated up to the next checkpoint (maximum of 200 finality headers, since the distance between checkpoints = 200)
  - `height`: number - the height of the block (block number)
- for blocks after BIP34 (height >= 227836), *the value of* `in3.preBIP34` *does not matter*
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
  - `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proving the correctness of the `cbtx`

In case the client requests the diffictuly of a certain block (`blocknumber` is a certain number) the `block`-field will contain the block header of this block and the `final`-field the corresponding finality headers. For old blocks (height < 227,836) with `in3.preBIP34` disabled the result cannot be verified (proving the finality does not provide any security as explained in preBIP34 proof). The result of old blocks with `in.preBIP34` enabled can be verified by performing a preBIP34 proof. In case the client requests the difficulty of the latest block the server is not able to prove the finality for this block (obviously there are no finality headers available yet). The server considers the latest block minus `in3.finality` as the latest block and returns its difficulty. The result can be verified by performing multiple proof. The block header from the `block`-field and the finality headers from the `final`-field will be used to perform a finality proof. Having a verified block header (and therefore a verified merkle root) enables the possibility of a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field).

The result itself (the difficulty) can be verified in two ways:

- by converting the difficulty into a target and check whether the block hash is lower than the target (since we proved the finality we consider the block hash as verified)
- by converting the difficulty and the bits (part of the block header) into a target and check if both targets are similar (they will not be equal since the target of the bits is not getting saved with full precision - leading bytes are equal)

Example

Request:

```
{
    "jsonrpc": "2.0",
    "id":1,
    "method": "getdifficulty",
    "params": [631910],
    "in3":{
        "finality":8,
        "verification":"proof",
        "preBIP34": true
    }
}
```

Response:

```
{
    "id": 1,
```

(continues on next page)

(continued from previous page)

```
    "jsonrpc": "2.0",
    "result": 15138043247082.88,
    "in3": {
        "proof": {
            "block": "0x00000020aa7531df9e14536f3c92fb9479cfc4025...eeb15d",
            "final": "0x0000ff3fdfdb13f86b9bce93f6c111feccecf4eb5...3c5846",
            "cbtx": "0x010000000001010000000000000000000000000000...000000",
            "cbtxMerkleProof": "0x48de085910879b0f201b320a7dbcb65...b02414"
        }
    }
}
```

8.4.7 btc_proofTarget

Whenever the client is not able to trust the changes of the target (which is the case if a block can't be found in the verified target cache *and* the value of the target changed more than the client's limit `max_diff`) he will call this method. It will return additional proof data to verify the changes of the target on the side of the client. This is not a standard Bitcoin rpc-method like the other ones, but more like an internal method.

Parameters:

1. `target_dap` : (string or number, required) the number of the difficulty adjustment period (dap) we are looking for
2. `verified_dap` : (string or number, required) the number of the closest already verified dap
3. `max_diff` : (string or number, required) the maximum target difference between 2 verified daps
4. `max_dap` : (string or number, required) the maximum amount of daps between 2 verified daps
5. `limit` : (string or number, optional) the maximum amount of daps to return (`0` = no limit) - this is important for embedded devices since returning all daps might be too much for limited memory
6. `in3.finality` : (number, required) defines the amount of finality headers
7. `in3.verification` : (string, required) defines the kind of proof the client is asking for (must be `never` or `proof`)
8. `in3.preBIP34` : (boolean, required) defines if the client wants to verify blocks before BIP34 (height < 227836)

Hints:

- difference between `target_dap` and `verified_dap` should be greater than `1`
- `target_dap` and `verified_dap` have to be greater than `0`
- `limit` will be set to `40` internaly when the parameter is equal to `0` or greater than `40`
- `max_dap` can't be equal to `0`
- `max_diff` equal to `0` means no tolerance regarding the change of the target - the path will contain every dap between `target_dap` and `verified_dap` (under consideration of `limit`)
- total possible amount of finality headers (`in3.finaliy` * `limit`) can't be greater than `1000`
- changes of a target will always be accepted if it decreased from one dap to another (i.e. difficulty to mine a block increased)

- in case a dap that we want to verify next (i.e. add it to the path) is only 1 dap apart from a verified dap (i.e. `verified_dap` or latest dap of the path) *but* not within the given limit (`max_diff`) it will still be added to the path (since we can't do even smaller steps)

Returns: A path of daps from the `verified_dap` to the `target_dap` which fulfils the conditions of `max_diff`, `max_dap` and `limit`. Each dap of the path is a `dap`-object with corresponding proof data.

The `dap`-object contains the following properties:

- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = false
  - `dap`: number - the numer of the difficulty adjustment period
  - `block`: hex - a hex string with 80 bytes representing the (always the first block of a dap)
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
- for blocks before BIP34 (height < 227836) and `in3.preBIP34` = true
  - `dap`: number - the numer of the difficulty adjustment period
  - `block`: hex - a hex string with 80 bytes representing the blockheader
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated up to the next checkpoint (maximum of 200 finality headers, since the distance between checkpoints = 200)
  - `height`: number - the height of the block (block number)
- for blocks after BIP34 (height >= 227836), *the value of `in3.preBIP34` does not matter*
  - `dap`: number - the numer of the difficulty adjustment period
  - `block`: hex - a hex string with 80 bytes representing the (always the first block of a dap)
  - `final`: hex - the finality headers, which are hexcoded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested finality (`finality`-property in the `in3`-section of the request)
  - `cbtx`: hex - the serialized coinbase transaction of the block (this is needed to get the verified block number)
  - `cbtxMerkleProof`: hex - the merkle proof of the coinbase transaction, proving the correctness of the `cbtx`

The goal is to verify the target of the `target_dap`. We will use the daps of the result to verify the target step by step starting with the `verified_dap`. For old blocks (height < 227,836) with `in3.preBIP34` disabled the target cannot be verified (proving the finality does not provide any security as explained in preBIP34 proof). For old blocks with `in.preBIP34` enabled the block header can be verified by performing a preBIP34 proof. Verifying newer blocks requires multiple proofs. The block header from the `block`-field and the finality headers from the `final`-field will be used to perform a finality proof. Having a verified block header allows us to consider the target of the block header as verified. Therefore, we have a verified target for the whole `dap`. Having a verified block header (and therefore a verified merkle root) enables the possibility of a block number proof using the coinbase transaction (`cbtx`-field) and the merkle proof for the coinbase transaction (`cbtxMerkleProof`-field). This proof is needed to verify the dap number (`dap`). Having a verified dap number allows us to verify the mapping between the target and the dap number.

Example

Request:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "btc_proofTarget",
    "params": [230,200,5,5,15],
    "in3":{
                "finality" : 8,
                "verification":"proof",
                "preBIP34": true
        }
}
```

Response:

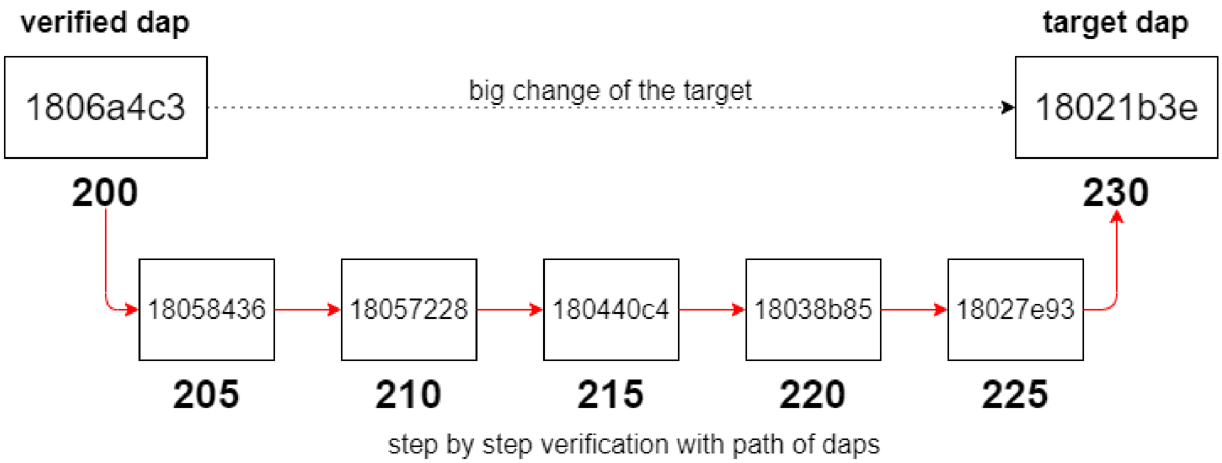
```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "dap": 205,
            "block": "0x04000000e62ef28cb9793f4f9cd2a67a58c1e7b593129b9b...0ab284",
            "final": "0x04000000cc69b68b702321adf4b0c485fdb1f3d6c1ddd140...090a5b",
            "cbtx": "0x01000000...1485ce370573be63d7cc1b9efbad3489eb57c8...000000",
            "cbtxMerkleProof": "0xc72dffc1cb4cbeab960d0d2bdb80012acf7f9c...affcf4"
        },
        {
            "dap": 210,
            "block": "0x0000003021622c26a4e62cafa8e434c7e083f540bccc8392...b374ce",
            "final": "0x00000020858f8e5124cd516f4d5e6a078f7083c12c48e8cd...308c3d",
            "cbtx": "0x01000000...c075061b4b6e434d696e657242332d50314861...000000",
            "cbtxMerkleProof": "0xf2885d0bac15fca7e1644c1162899ecd43d52b...93761d"
        },
        {
            "dap": 215,
            "block": "0x000000202509b3b8e4f98290c7c9551d180eb2a463f0b978...f97b64",
            "final": "0x0000002014c7c0ed7c33c59259b7b508bebfe3974e1c99a5...eb554e",
            "cbtx": "0x01000000...90133cf94b1b1c40fae077a7833c0fe0ccc474...000000",
            "cbtxMerkleProof": "0x628c8d961adb157f800be7cfb03ffa1b53d3ad...ca5a61"
        },
        {
            "dap": 220,
            "block": "0x00000020ff45c783d09706e359dcc76083e15e51839e4ed5...ddfe0e",
            "final": "0x0000002039d2f8a1230dd0bee50034e8c63951ab812c0b89...5670c5",
            "cbtx": "0x01000000...b98e79fb3e4b88aefbc8ce59e82e99293e5b08...000000",
            "cbtxMerkleProof": "0x16adb7aeec2cf254db0bab0f4a5083fb0e0a3f...63a4f4"
        },
        {
            "dap": 225,
            "block": "0x02000020170fad0b6b1ccbdc4401d7b1c8ee868c6977d6ce...1e7f8f",
            "final": "0x0400000092945abbd7b9f0d407fcccbf418e4fc20570040c...a9b240",
            "cbtx": "0x01000000...cf6e8f930acb8f38b588d76cd8c3da3258d5a7...000000",
            "cbtxMerkleProof": "0x25575bcaf3e11970ccf835e88d6f97bedd6b85...bfdf46"
        }
    ],
    "in3": {
        "lastNodeList": 3101668,
        "execTime": 2760,
```

(continues on next page)

(continued from previous page)

```
        "rpcTime": 172,
        "rpcCount": 1,
        "currentBlock": 3101713,
        "version": "2.1.0"
    }
}
```

This graph shows the usage of this method and visualizes the result from above. The client is not able to trust the changes of the target due to his limits (`max_diff` and `max_dap`). This method provides a path of daps in which the limits are fulfilled from dap to another. The client is going to trust the target of the target dap since he is able to perform a step by step verification of the target by using the path of daps.



8.5 zksync

the zksync-plugin is able to handle operations to use zksync like deposit transfer or withdraw. Also see the #in3-config on how to configure the zksync-server or account.

Also in order to sign messages you need to set a signer!

8.5.1 zksync_contract_address

params: none

returns the contract address

```
in3 zksync_contract_address | jq
```

```
{
  "govContract": "0x34460C0EB5074C29A9F6FE13b8e7E23A0D08aF01",
  "mainContract": "0xaBEA9132b05A70803a4E85094fD0e1800777fBEF"
}
```

8.5.2 zksync_tokens

params: none

returns the list of available tokens

```
in3 zksync_tokens | jq
```


```
{
  "BAT": {
    "address": "0x0d8775f648430679a709e98d2b0cb6250d2887ef",
    "decimals": 18,
    "id": 8,
    "symbol": "BAT"
  },
  "BUSD": {
    "address": "0x4fabb145d64652a948d72533023f6e7a623c7c53",
    "decimals": 18,
    "id": 6,
    "symbol": "BUSD"
  },
  "DAI": {
    "address": "0x6b175474e89094c44da98b954eedeac495271d0f",
    "decimals": 18,
    "id": 1,
    "symbol": "DAI"
  },
  "ETH": {
    "address": "0x0000000000000000000000000000000000000000",
    "decimals": 18,
    "id": 0,
    "symbol": "ETH"
  }
}
```


8.5.3 zksync_account_info

params:

- address (optional, if the pk is set it will be taken from there)

returns account_info from the server

Example:

```
in3  -pk 0xe41d2489571d322189246dafa5ebde1f4699f498000000000000000000000000 zksync_
↪account_info | jq
```


```
{
  "address": "0x3b2a1bd631d9d7b17e87429a8e78dbbd9b4de292",
  "committed": {
    "balances": {},
    "nonce": 0,
    "pubKeyHash": "sync:0000000000000000000000000000000000000000"
  },
  "depositing": {
    "balances": {}
  },
  "id": null,
  "verified": {
    "balances": {},
```


(continues on next page)

(continued from previous page)

```
    "nonce": 0,
    "pubKeyHash": "sync:0000000000000000000000000000000000000000"
  }
}
```

8.5.4 zksync_tx_info

params:

- the txHash

returns the state or receipt of the the zksync-tx

```
in3 zksync_tx_info  "sync-
↪tx:e41d2489571d322189246dafa5ebde1f4699f498000000000000000000000000" | jq
```

```
{
  "block": null,
  "executed": false,
  "failReason": null,
  "success": null
}
```

8.5.5 zksync_setKey

params: none

sets the signerkey based on the current pk

Example:

```
in3  -pk 0xe41d2489571d322189246dafa5ebde1f4699f498000000000000000000000000 zksync_
↪setKey
```

```
"sync:e41d2489571d322189246dafa5ebde1f4699f498"
```

8.5.6 zksync_ethop_info

params:

- the opId

returns the state or receipt of the the PriorityOperation

8.5.7 zksync_get_token_price

params:

- the token-address

returns current token-price

```
in3 zksync_get_token_price WBTC
```

```
11320.002167
```

8.5.8 zksync_get_tx_fee

params:

- txType ("Withdraw" or "Transfer")
- address
- token

returns fee for a transaction

```
in3 zksync_get_tx_fee Transfer 0xabea9132b05a70803a4e85094fd0e1800777fbef BAT | jq
```

```
{
  "feeType": "TransferToNew",
  "gasFee": "47684047990828528",
  "gasPriceWei": "116000000000",
  "gasTxAmount": "350",
  "totalFee": "66000000000000000",
  "zkpFee": "18378682992117666"
}
```

8.5.9 zksync_syncKey

params: none

returns private key used for signing zksync-transactions

8.5.10 zksync_deposit

params: (passed as array in this order or as array with one JSON-Object, with those props)

- amount
- token
- approveDepositAmountForERC20
- account (if not given it will be taken from the current signer)

sends a deposit-transaction and returns the opId, which can be used to tradck progress.

```
in3 -pk <MY_PK> zksync_deposit 1000 WBTC false
```

8.5.11 zksync_transfer

params:

- to

- amount
- token
- account (if not given it will be taken from the current signer)

sends a zksync-transaction and returns data including the transactionHash.

```
in3 -pk <MY_PK> zksync_transfer  0xabea9132b05a70803a4e85094fd0e1800777fbef 100 WBTC
```

8.5.12 zksync_withdraw

params:

- ethAddress
- amount
- token
- account (if not given it will be taken from the current signer)

withdraws the amount to the given `ethAddress` for the given token.

```
in3 -pk <MY_PK> zksync_withdraw  0xabea9132b05a70803a4e85094fd0e1800777fbef 100 WBTC
```

8.5.13 zksync_emergencyWithdraw

params:

- token

withdraws all tokens for the specified token as a onchain-transaction. This is useful in case the zksync-server is offline or tries to be malicious.

```
in3 -pk <MY_PK> zksync_emergencyWithdraw WBTC
```

CHAPTER 9

API Reference C

9.1 Overview

The C implementation of the Incubed client is prepared and optimized to run on small embedded devices. Because each device is different, we prepare different modules that should be combined. This allows us to only generate the code needed and reduce requirements for flash and memory.

9.1.1 Why C?

We have been asked a lot, why we implemented Incubed in C and not in Rust. When we started Incubed we began with a feasibility test and wrote the client in TypeScript. Once we confirmed it was working, we wanted to provide a minimal verifaction client for embedded devices. And yes, we actually wanted to do it in Rust, since Rust offers a lot of safety-features (like the memory-management at compiletime, thread-safety, . . .), but after considering a lot of different aspects we made a pragmatic desicion to use C.

These are the reasons why:

Support for embedded devices.

As of today almost all toolchain used in the embedded world are build for C. Even though Rust may be able to still use some, there are a lot of issues. Quote from rust-embedded.org:

Integrating Rust with an RTOS such as FreeRTOS or ChibiOS is still a work in progress; especially calling RTOS functions from Rust can be tricky.

This may change in the future, but C is so dominant, that chances of Rust taking over the embedded development completly is low.

Portability

C is the most portable programming language. Rust actually has a pretty admirable selection of supported targets for a new language (thanks mostly to LLVM), but it pales in comparison to C, which runs on almost everything. A new

CPU architecture or operating system can barely be considered to exist until it has a C compiler. And once it does, it unlocks access to a vast repository of software written in C. Many other programming languages, such as Ruby and Python, are implemented in C and you get those for free too.

Most programing language have very good support for calling c-function in a shared library (like ctypes in python or cgo in golang) or even support integration of C code directly like android studio does.

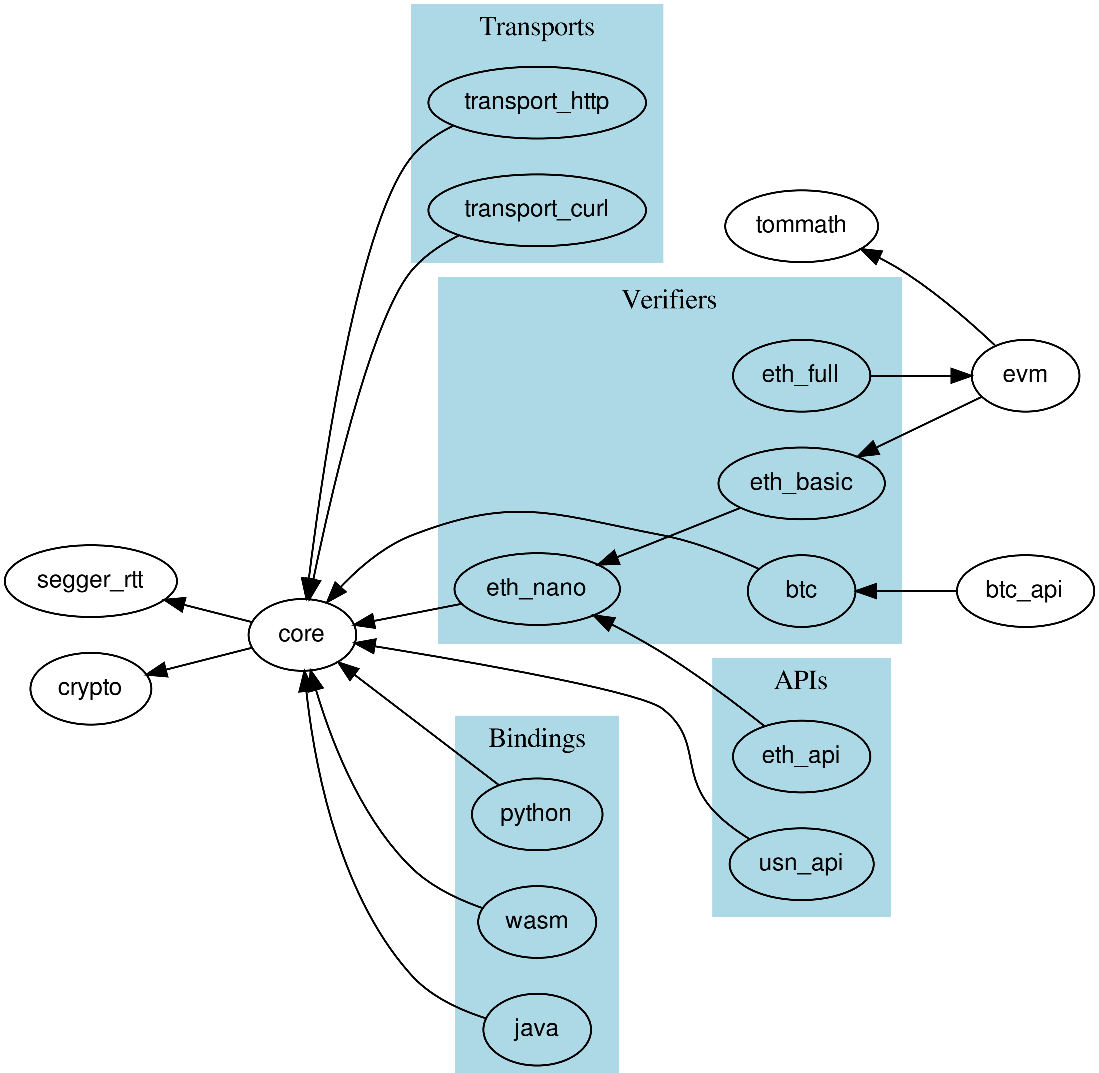
Integration in existing projects

Since especially embedded systems are usually written in C/C++, offering a pure C-Implementation makes it easy for these projects to use Incubed, since they do not have to change their toolchain.

Even though we may not be able to use a lot of great features Rust offers by going with C, it allows to reach the goal to easily integrate with a lot of projects. For the future we might port the incubed to Rust if we see a demand or chance for the same support as C has today.

9.1.2 Modules

Incubed consists of different modules. While the core module is always required, additional functions will be prepared by different modules.



Verifier

Incubed is a minimal verification client, which means that each response needs to be verifiable. Depending on the expected requests and responses, you need to carefully choose which verifier you may need to register. For Ethereum, we have developed three modules:

1. *eth_nano*: a minimal module only able to verify transaction receipts (`eth_getTransactionReceipt`).
2. *eth_basic*: module able to verify almost all other standard RPC functions (except `eth_call`).
3. *eth_full*: module able to verify standard RPC functions. It also implements a full EVM to handle `eth_call`.
4. *btc*: module able to verify bitcoin or bitcoin based chains.
5. *ipfs*: module able to verify ipfs-hashes

Depending on the module, you need to register the verifier before using it. This is done by calling the `in3_register...` function like *in3_register_eth_full()*.

Transport

To verify responses, you need to be able to send requests. The way to handle them depends heavily on your hardware capabilities. For example, if your device only supports Bluetooth, you may use this connection to deliver the request to a device with an existing internet connection and get the response in the same way, but if your device is able to use a direct internet connection, you may use a curl-library to execute them. This is why the core client only defines function pointer *in3_transport_send*, which must handle the requests.

At the moment we offer these modules; other implementations are supported by different hardware modules.

1. *transport_curl*: module with a dependency on curl, which executes these requests and supports HTTPS. This module runs a standard OS with curl installed.
2. *transport_http*: module with no dependency, but a very basic http-implementation (no https-support)

API

While Incubed operates on JSON-RPC level, as a developer, you might want to use a better-structured API to prepare these requests for you. These APIs are optional but make life easier:

1. ***eth***: This module offers all standard RPC functions as described in the Ethereum JSON-RPC Specification. In addition, it allows you to sign and encode/decode calls and transactions.
2. ***usn***: This module offers basic USN functions like renting, event handling, and message verification.
3. ***btc***: Collection of Bitcoin-functions to access blocks and transactions.
4. ***ipfs***: Simple Ipfs-functions to get and store ipfs-content

9.2 Building

While we provide binaries, you can also build from source:

9.2.1 requirements

- cmake
- curl : curl is used as transport for command-line tools, but you can also compile it without curl (`-DUSE_CURL=false -DCMD=false`), if you want to implement your own transport.

Incubed uses cmake for configuring:

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release .. && make
make install
```

9.2.2 CMake options

When configuring cmake, you can set a lot of different incubed specific like `cmake -DEVM_GAS=false ..`.

ASMJS

compiles the code as asm.js.

Default-Value: `-DASMJS=OFF`

ASSERTIONS

includes assertions into the code, which help track errors but may cost time during runtime

Default-Value: `-DASSERTIONS=OFF`

BTC

if true, the bitcoin verifiers will be build

Default-Value: `-DBTC=ON`

BUILD_DOC

generates the documenation with doxygen.

Default-Value: `-DBUILD_DOC=OFF`

CMD

build the comandline utils

Default-Value: `-DCMD=ON`

CODE_COVERAGE

Builds targets with code coverage instrumentation. (Requires GCC or Clang)

Default-Value: `-DCODE_COVERAGE=OFF`

COLOR

Enable color codes for debug

Default-Value: `-DCOLOR=ON`

DEV_NO_INTRN_PTR

(*dev option*) if true the client will NOT include a void pointer (named internal) for use by devs)

Default-Value: `-DDEV_NO_INTRN_PTR=ON`

ESP_IDF

include support for ESP-IDF microcontroller framework

Default-Value: `-DESP_IDF=OFF`

ETH_BASIC

build basic eth verification.(all rpc-calls except eth_call)

Default-Value: `-DETH_BASIC=ON`

ETH_FULL

build full eth verification.(including eth_call)

Default-Value: `-DETH_FULL=ON`

ETH_NANO

build minimal eth verification.(eth_getTransactionReceipt)

Default-Value: `-DETH_NANO=ON`

EVM_GAS

if true the gas costs are verified when validating a eth_call. This is a optimization since most calls are only interessted in the result. EVM_GAS would be required if the contract uses gas-dependend op-codes.

Default-Value: `-DEVM_GAS=ON`

FAST_MATH

Math optimizations used in the EVM. This will also increase the filesize.

Default-Value: `-DFAST_MATH=OFF`

GCC_ANALYZER

GCC10 static code analyses

Default-Value: `-DGCC_ANALYZER=OFF`

IN3API

build the USN-API which offer better interfaces and additional functions on top of the pure verification

Default-Value: `-DIN3API=ON`

IN3_LIB

if true a shared anmd static library with all in3-modules will be build.

Default-Value: `-DIN3_LIB=ON`

IN3_SERVER

support for proxy server as part of the cmd-tool, which allows to start the cmd-tool with the -p option and listens to the given port for rpc-requests

Default-Value: `-DIN3_SERVER=OFF`

IN3_STAGING

if true, the client will use the staging-network instead of the live ones

Default-Value: `-DIN3_STAGING=OFF`

IPFS

build IPFS verification

Default-Value: `-DIPFS=ON`

JAVA

build the java-binding (shared-lib and jar-file)

Default-Value: `-DJAVA=OFF`

LEDGER_NANO

include support for nano ledger

Default-Value: `-DLEDGER_NANO=OFF`

LOGGING

if set logging and human readable error messages will be inculded in th executable, otherwise only the error code is used. (saves about 19kB)

Default-Value: `-DLOGGING=ON`

MULTISIG

add capapbility to sign with a multig. Currrently only gnosis safe is supported

Default-Value: `-DMULTISIG=OFF`

PAY_ETH

support for direct Eth-Payment

Default-Value: `-DPAY_ETH=OFF`

PKG_CONFIG_EXECUTABLE

pkg-config executable

Default-Value: `-DPKG_CONFIG_EXECUTABLE=/opt/local/bin/pkg-config`

POA

support POA verification including validatorlist updates

Default-Value: `-DPOA=OFF`

SEGGER_RTT

Use the segger real time transfer terminal as the logging mechanism

Default-Value: `-DSEGGER_RTT=OFF`

TAG_VERSION

the tagged version, which should be used

Default-Value: `-DTAG_VERSION=OFF`

TEST

builds the tests and also adds special memory-management, which detects memory leaks, but will cause slower performance

Default-Value: `-DTEST=OFF`

TRANSPORTS

builds transports, which may require extra libraries.

Default-Value: `-DTRANSPORTS=ON`

USE_CURL

if true the curl transport will be built (with a dependency to libcurl)

Default-Value: `-DUSE_CURL=ON`

USE_PRECOMPUTED_EC

if true the secp256k1 curve uses precompiled tables to boost performance. turning this off makes ecrecover slower, but saves about 37kb.

Default-Value: `-DUSE_PRECOMPUTED_EC=ON`

USE_SCRYPT

integrate scrypt into the build in order to allow decrypt_key for scrypt encoded keys.

Default-Value: `-DUSE_SCRYPT=ON`

WASM

Includes the WASM-Build. In order to build it you need emscripten as toolchain. Usually you also want to turn off other builds in this case.

Default-Value: `-DWASM=OFF`

WASM_EMBED

embedds the wasm as base64-encoded into the js-file

Default-Value: `-DWASM_EMBED=ON`

WASM_EMMALLOC

use ther smaller EMSCRIPTEN Malloc, which reduces the size about 10k, but may be a bit slower

Default-Value: `-DWASM_EMMALLOC=ON`

WASM_SYNC

intiaializes the WASM synchronisly, which allows to require and use it the same function, but this will not be supported by chrome (4k limit)

Default-Value: `-DWASM_SYNC=OFF`

ZKSYNC

add RPC-functioin to handle zksync-payments

Default-Value: `-DZKSYNC=OFF`

9.3 Examples

9.3.1 btc_transaction

source : in3-c/c/examples/btc_transaction.c

checking a Bitcoin transaction data

```
#include <in3/btc_api.h>  // we need the btc-api
#include <in3/client.h>   // the core client
#include <in3/in3_init.h> // this header will make sure we initialize the default␣
↪verifiers and transports
#include <in3/utils.h>    // helper functions
```

(continues on next page)

(continued from previous page)

```
#include <stdio.h>

int main() {
  // create new incubed client for BTC
  in3_t* in3 = in3_for_chain(CHAIN_ID_BTC);

  // the hash of transaction that we want to get
  bytes32_t tx_id;
  hex_to_bytes("c41eee1c2d97f6158ea3b3aeba0a5271a2174067a38d089ccc1eefbc796706e0", -1,
↪ tx_id, 32);

  // fetch and verify the transaction
  btc_transaction_t* tx = btc_get_transaction(in3, tx_id);

  if (!tx)
    // if the result is null there was an error an we can get the latest error␣
↪message from btc_last_error()
    printf("error getting the tx : %s\n", btc_last_error());
  else {
    // we loop through the tx outputs
    for (int i = 0; i < tx->vout_len; i++)
      // and prrint the values
      printf("Transaction vout #%d : value: %llu\n", i, tx->vout[i].value);

    // don't forget the clean up!
    free(tx);
  }

  // cleanup client after usage
  in3_free(in3);
}
```

9.3.2 call_a_function

source : in3-c/c/examples/call_a_function.c

This example shows how to call functions on a smart contract eiither directly or using the api to encode the arguments

```

#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <inttypes.h>
#include <stdio.h>

static in3_ret_t call_func_rpc(in3_t* c);
static in3_ret_t call_func_api(in3_t* c, address_t contract);

int main() {
  in3_ret_t ret = IN3_OK;

  // Remove log prefix for readability
  in3_log_set_prefix("");

```

(continues on next page)

(continued from previous page)

```
  // create new incubed client
  in3_t* c = in3_for_chain(CHAIN_ID_MAINNET);

  // define a address (20byte)
  address_t contract;

  // copy the hexcoded string into this address
  hex_to_bytes("0x2736D225f85740f42D17987100dc8d58e9e16252", -1, contract, 20);

  // call function using RPC
  ret = call_func_rpc(c);
  if (ret != IN3_OK) goto END;

  // call function using API
  ret = call_func_api(c, contract);
  if (ret != IN3_OK) goto END;

END:
  // clean up
  in3_free(c);
  return 0;
}

in3_ret_t call_func_rpc(in3_t* c) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      c,                                                                      //  the configured client
      "eth_call",                                                             // the rpc-method you want to call.
      "[{\"to\":\"0x2736d225f85740f42d17987100dc8d58e9e16252\", \"data\":\"0x15625c5e\"}, \"latest\"]", // the signed raw txn, same as the one used in the API example
      &result,                                                                // the reference to a pointer which will hold the result
      &error);                                                                // the pointer which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Result: \n%s\n", result);
    free(result);
    return 0;
  } else {
    printf("Error sending tx: \n%s\n", error);
    free(error);
    return IN3_EUNKNOWN;
  }
}

in3_ret_t call_func_api(in3_t* c, address_t contract) {
  // ask for the number of servers registered
  json_ctx_t* response = eth_call_fn(c, contract, BLKNUM_LATEST(), "totalServers():uint256");
  if (!response) {
```

(continues on next page)

(continued from previous page)

```
    printf("Could not get the response: %s", eth_last_error());
    return IN3_EUNKNOWN;
  }

  // convert the response to a uint32_t,
  uint32_t number_of_servers = d_int(response->result);

  // clean up resources
  json_free(response);

  // output
  printf("Found %u servers registered : \n", number_of_servers);

  // read all structs ...
  for (uint32_t i = 0; i < number_of_servers; i++) {
    response = eth_call_fn(c, contract, BLKNUM_LATEST(), "servers(uint256):(string,
↪address,uint,uint,uint,address)", to_uint256(i));
    if (!response) {
      printf("Could not get the response: %s", eth_last_error());
      return IN3_EUNKNOWN;
    }

    char*    url     = d_get_string_at(response->result, 0); // get the first item of␣
↪the result (the url)
    bytes_t* owner   = d_get_bytes_at(response->result, 1);  // get the second item␣
↪of the result (the owner)
    uint64_t deposit = d_get_long_at(response->result, 2);   // get the third item of␣
↪the result (the deposit)

    printf("Server %i : %s owner = %02x%02x...", i, url, owner->data[0], owner->
↪data[1]);
    printf(", deposit = %" PRIu64 "\n", deposit);

    // free memory
    json_free(response);
  }
  return 0;
}
```

9.3.3 get_balance

source : in3-c/c/examples/get_balance.c

get the Balance with the API and also as direct RPC-call

```
#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <in3/utils.h>
#include <stdio.h>

static void get_balance_rpc(in3_t* in3);
static void get_balance_api(in3_t* in3);
```

(continues on next page)

(continued from previous page)

```
int main() {
  // create new incubed client
  in3_t* in3 = in3_for_chain(CHAIN_ID_MAINNET);

  // get balance using raw RPC call
  get_balance_rpc(in3);

  // get balance using API
  get_balance_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void get_balance_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,                                                              //  the␣
↪configured client
      "eth_getBalance",                                                 // the rpc-
↪method you want to call.
      "[\"0xc94770007dda54cF92009BFF0dE90c06F603a09f\", \"latest\"]", // the␣
↪arguments as json-string
      &result,                                                          // the␣
↪reference to a pointer whill hold the result
      &error);                                                          // the pointer␣
↪which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Balance: \n%s\n", result);
    free(result);
  } else {
    printf("Error getting balance: \n%s\n", error);
    free(error);
  }
}

void get_balance_api(in3_t* in3) {
  // the address of account whose balance we want to get
  address_t account;
  hex_to_bytes("0xc94770007dda54cF92009BFF0dE90c06F603a09f", -1, account, 20);

  // get balance of account
  long double balance = as_double(eth_getBalance(in3, account, BLKNUM_EARLIEST()));

  // if the result is null there was an error an we can get the latest error message␣
↪from eth_lat_error()
  balance ? printf("Balance: %Lf\n", balance) : printf("error getting the balance :
↪%s\n", eth_last_error());
}
```

9.3.4 get_block

source : in3-c/c/examples/get_block.c

using the basic-module to get and verify a Block with the API and also as direct RPC-call

```
#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions

#include <inttypes.h>
#include <stdio.h>

static void get_block_rpc(in3_t* in3);
static void get_block_api(in3_t* in3);

int main() {
  // create new incubed client
  in3_t* in3 = in3_for_chain(CHAIN_ID_MAINNET);

  // get block using raw RPC call
  get_block_rpc(in3);

  // get block using API
  get_block_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void get_block_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,                    //  the configured client
      "eth_getBlockByNumber", // the rpc-method you want to call.
      "[\"latest\",true]",    // the arguments as json-string
      &result,                // the reference to a pointer whill hold the result
      &error);                // the pointer which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Latest block : \n%s\n", result);
    free(result);
  } else {
    printf("Error verifing the Latest block : \n%s\n", error);
    free(error);
  }
}

void get_block_api(in3_t* in3) {
  // get the block without the transaction details
  eth_block_t* block = eth_getBlockByNumber(in3, BLKNUM(8432424), false);
```

(continues on next page)

(continued from previous page)

```
  // if the result is null there was an error an we can get the latest error message␣
↪from eth_lat_error()
  if (!block)
    printf("error getting the block : %s\n", eth_last_error());
  else {
    printf("Number of transactions in Block #%llu: %d\n", block->number, block->tx_
↪count);
    free(block);
  }
}
```

9.3.5 get_logs

source : in3-c/c/examples/get_logs.c

fetching events and verify them with eth_getLogs

```

#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <inttypes.h>
#include <stdio.h>

static void get_logs_rpc(in3_t* in3);
static void get_logs_api(in3_t* in3);

int main() {
  // create new incubed client
  in3_t* in3     = in3_for_chain(CHAIN_ID_MAINNET);
  in3->chain_id = CHAIN_ID_KOVAN;

  // get logs using raw RPC call
  get_logs_rpc(in3);

  // get logs using API
  get_logs_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void get_logs_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,           //  the configured client
      "eth_getLogs", // the rpc-method you want to call.
      "[{}]",        // the arguments as json-string
      &result,       // the reference to a pointer whill hold the result
      &error);       // the pointer which may hold a error message

```

(continues on next page)

(continued from previous page)

```
  // check and print the result or error
  if (res == IN3_OK) {
    printf("Logs : \n%s\n", result);
    free(result);
  } else {
    printf("Error getting logs : \n%s\n", error);
    free(error);
  }
}

void get_logs_api(in3_t* in3) {
  // Create filter options
  char b[30];
  sprintf(b, "{\"fromBlock\":\"0x%" PRIx64 "\"}", eth_blockNumber(in3) - 2);
  json_ctx_t* jopt = parse_json(b);

  // Create new filter with options
  size_t fid = eth_newFilter(in3, jopt);

  // Get logs
  eth_log_t* logs = NULL;
  in3_ret_t  ret  = eth_getFilterLogs(in3, fid, &logs);
  if (ret != IN3_OK) {
    printf("eth_getFilterLogs() failed [%d]\n", ret);
    return;
  }

  // print result
  while (logs) {
    eth_log_t* l = logs;
    printf("--------------------------------------------------------------------------------
↪------\n");
    printf("\tremoved: %s\n", l->removed ? "true" : "false");
    printf("\tlogId: %lu\n", l->log_index);
    printf("\tTxId: %lu\n", l->transaction_index);
    printf("\thash: ");
    ba_print(l->block_hash, 32);
    printf("\n\tnum: %" PRIu64 "\n", l->block_number);
    printf("\taddress: ");
    ba_print(l->address, 20);
    printf("\n\tdata: ");
    b_print(&l->data);
    printf("\ttopics[%lu]: ", l->topic_count);
    for (size_t i = 0; i < l->topic_count; i++) {
      printf("\n\t");
      ba_print(l->topics[i], 32);
    }
    printf("\n");
    logs = logs->next;
    free(l->data.data);
    free(l->topics);
    free(l);
  }
  eth_uninstallFilter(in3, fid);
  json_free(jopt);
}
```

9.3.6 get_transaction

source : in3-c/c/examples/get_transaction.c

checking the transaction data

```
#include <in3/client.h> // the core client
#include <in3/eth_api.h>
#include <in3/in3_curl.h> // transport implementation
#include <in3/in3_init.h>
#include <in3/utils.h>
#include <stdio.h>

static void get_tx_rpc(in3_t* in3);
static void get_tx_api(in3_t* in3);

int main() {
  // create new incubed client
  in3_t* in3 = in3_for_chain(CHAIN_ID_MAINNET);

  // get tx using raw RPC call
  get_tx_rpc(in3);

  // get tx using API
  get_tx_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void get_tx_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,                                                                                // 
↪the configured client
      "eth_getTransactionByHash",                                                         //
↪the rpc-method you want to call.
      "[\"0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e\"]", //
↪the arguments as json-string
      &result,                                                                            //
↪the reference to a pointer which will hold the result
      &error);                                                                            //
↪the pointer which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Latest tx : \n%s\n", result);
    free(result);
  } else {
    printf("Error verifing the Latest tx : \n%s\n", error);
    free(error);
  }
}
```

(continues on next page)

(continued from previous page)

```
void get_tx_api(in3_t* in3) {
  // the hash of transaction that we want to get
  bytes32_t tx_hash;
  hex_to_bytes("0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e", -
↪1, tx_hash, 32);

  // get the tx by hash
  eth_tx_t* tx = eth_getTransactionByHash(in3, tx_hash);

  // if the result is null there was an error an we can get the latest error message␣
↪from eth_last_error()
  if (!tx)
    printf("error getting the tx : %s\n", eth_last_error());
  else {
    printf("Transaction #%d of block #%llx", tx->transaction_index, tx->block_number);
    free(tx);
  }
}
```

9.3.7 get_transaction_receipt

source : in3-c/c/examples/get_transaction_receipt.c

validating the result or receipt of an transaction

```

#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <in3/utils.h>
#include <inttypes.h>
#include <stdio.h>

static void get_tx_receipt_rpc(in3_t* in3);
static void get_tx_receipt_api(in3_t* in3);

int main() {
  // create new incubed client
  in3_t* in3 = in3_for_chain(CHAIN_ID_MAINNET);

  // get tx receipt using raw RPC call
  get_tx_receipt_rpc(in3);

  // get tx receipt using API
  get_tx_receipt_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void get_tx_receipt_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;
```

(continues on next page)

(continued from previous page)

```
  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,                                                                    // ␣
↪the configured client
      "eth_getTransactionReceipt",                                            //␣
↪the rpc-method you want to call.
      "[\"0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e\"]", //␣
↪the arguments as json-string
      &result,                                                                //␣
↪the reference to a pointer which will hold the result
      &error);                                                                //␣
↪the pointer which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Transaction receipt: \n%s\n", result);
    free(result);
  } else {
    printf("Error verifing the tx receipt: \n%s\n", error);
    free(error);
  }
}

void get_tx_receipt_api(in3_t* in3) {
  // the hash of transaction whose receipt we want to get
  bytes32_t tx_hash;
  hex_to_bytes("0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e", -
↪1, tx_hash, 32);

  // get the tx receipt by hash
  eth_tx_receipt_t* txr = eth_getTransactionReceipt(in3, tx_hash);

  // if the result is null there was an error an we can get the latest error message␣
↪from eth_last_error()
  if (!txr)
    printf("error getting the tx : %s\n", eth_last_error());
  else {
    printf("Transaction #%d of block #%llx, gas used = %" PRIu64 ", status = %s\n",␣
↪txr->transaction_index, txr->block_number, txr->gas_used, txr->status ? "success" :
↪"failed");
    eth_tx_receipt_free(txr);
  }
}
```

9.3.8 ipfs_put_get

source : in3-c/c/examples/ipfs_put_get.c

using the IPFS module

```
#include <in3/client.h>   // the core client
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/ipfs_api.h> // access ipfs-api
#include <in3/log.h>      // logging functions
```

(continues on next page)

(continued from previous page)

```
#include <stdio.h>

#define LOREM_IPSUM "Lorem ipsum dolor sit amet"
#define return_err(err)                                \
  do {                                                 \
    printf(__FILE__ ":%d::Error %s\n", __LINE__, err); \
    return;                                            \
  } while (0)

static void ipfs_rpc_example(in3_t* c) {
  char *result, *error;
  char  tmp[100];

  in3_ret_t res = in3_client_rpc(
      c,
      "ipfs_put",
      "[\"" LOREM_IPSUM "\", \"utf8\"]",
      &result,
      &error);
  if (res != IN3_OK)
    return_err(in3_errmsg(res));

  printf("IPFS hash: %s\n", result);
  sprintf(tmp, "[%s, \"utf8\"]", result);
  free(result);
  result = NULL;

  res = in3_client_rpc(
      c,
      "ipfs_get",
      tmp,
      &result,
      &error);
  if (res != IN3_OK)
    return_err(in3_errmsg(res));
  res = strcmp(result, "\"" LOREM_IPSUM "\"");
  if (res) return_err("Content mismatch");
}

static void ipfs_api_example(in3_t* c) {
  bytes_t b         = {.data = (uint8_t*) LOREM_IPSUM, .len = strlen(LOREM_IPSUM)};
  char*   multihash = ipfs_put(c, &b);
  if (multihash == NULL)
    return_err("ipfs_put API call error");
  printf("IPFS hash: %s\n", multihash);

  bytes_t* content = ipfs_get(c, multihash);
  free(multihash);
  if (content == NULL)
    return_err("ipfs_get API call error");

  int res = strncmp((char*) content->data, LOREM_IPSUM, content->len);
  b_free(content);
  if (res)
    return_err("Content mismatch");
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  // create new incubed client
  in3_t* c = in3_for_chain(CHAIN_ID_IPFS);

  // IPFS put/get using raw RPC calls
  ipfs_rpc_example(c);

  // IPFS put/get using API
  ipfs_api_example(c);

  // cleanup client after usage
  in3_free(c);
  return 0;
}
```

9.3.9 ledger_sign

source : in3-c/c/examples/ledger_sign.c

```
#include <in3/client.h>  // the core client
#include <in3/eth_api.h> // functions for direct api-access
#include <in3/ethereum_apdu_client.h>
#include <in3/in3_init.h>      // if included the verifier will automaticly be␣
↪initialized.
#include <in3/ledger_signer.h> //to invoke ledger nano device for signing
#include <in3/log.h>           // logging functions
#include <in3/utils.h>
#include <stdio.h>

static void send_tx_api(in3_t* in3);

int main() {
  // create new incubed client
  uint8_t bip_path[5] = {44, 60, 0, 0, 0};
  in3_t*  in3         = in3_for_chain(CHAIN_ID_MAINNET);
  in3_log_set_level(LOG_DEBUG);
  // setting ledger nano s to be the default signer for incubed client
  // it will cause the transaction or any msg to be sent to ledger nanos device for␣
↪siging
  eth_ledger_set_signer_txn(in3, bip_path);
  // eth_ledger_set_signer(in3, bip_path);

  // send tx using API
  send_tx_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void send_tx_api(in3_t* in3) {
  // prepare parameters
  address_t to, from;
  hex_to_bytes("0xC51fBbe0a68a7cA8d33f14a660126Da2A2FAF8bf", -1, from, 20);
  hex_to_bytes("0xd46e8dd67c5d32be8058bb8eb970870f07244567", -1, to, 20);
```

(continues on next page)

(continued from previous page)

```
  bytes_t* data = hex_to_new_bytes("0x00", 0);
  // send the tx
  bytes_t* tx_hash = eth_sendTransaction(in3, from, to, OPTIONAL_T_VALUE(uint64_t,␣
↪0x96c0), OPTIONAL_T_VALUE(uint64_t, 0x9184e72a000), OPTIONAL_T_VALUE(uint256_t, to_
↪uint256(0x9184e72a)), OPTIONAL_T_VALUE(bytes_t, *data), OPTIONAL_T_UNDEFINED(uint64_
↪t));

  // if the result is null there was an error and we can get the latest error message␣
↪from eth_last_error()
  if (!tx_hash)
    printf("error sending the tx : %s\n", eth_last_error());
  else {
    printf("Transaction hash: ");
    b_print(tx_hash);
    b_free(tx_hash);
  }
  b_free(data);
}
```

9.3.10 send_transaction

source : in3-c/c/examples/send_transaction.c

sending a transaction including signing it with a private key

```
#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <in3/signer.h>   // default signer implementation
#include <in3/utils.h>
#include <stdio.h>

// fixme: This is only for the sake of demo. Do NOT store private keys as plaintext.
#define ETH_PRIVATE_KEY
↪"0x8da4ef21b864d2cc526dbdb2a120bd2874c36c9d0a1fb7f8c63d7f7a8b41de8f"

static void send_tx_rpc(in3_t* in3);
static void send_tx_api(in3_t* in3);

int main() {
  // create new incubed client
  in3_t* in3 = in3_for_chain(CHAIN_ID_MAINNET);

  // convert the hexstring to bytes
  bytes32_t pk;
  hex_to_bytes(ETH_PRIVATE_KEY, -1, pk, 32);

  // create a simple signer with this key
  eth_set_pk_signer(in3, pk);

  // send tx using raw RPC call
  send_tx_rpc(in3);
```

(continues on next page)

(continued from previous page)

```
  // send tx using API
  send_tx_api(in3);

  // cleanup client after usage
  in3_free(in3);
}

void send_tx_rpc(in3_t* in3) {
  // prepare 2 pointers for the result.
  char *result, *error;

  // send raw rpc-request, which is then verified
  in3_ret_t res = in3_client_rpc(
      in3,                        //  the configured client
      "eth_sendRawTransaction", // the rpc-method you want to call.
      "[\"0xf892808609184e72a0008296c094d46e8dd67c5d32be8058bb8eb970870f0724456"

↪"7849184e72aa9d46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb9"

↪"70870f07244567526a06f0103fccdcae0d6b265f8c38ee42f4a722c1cb36230fe8da40315acc3051"
      "9a8a06252a68b26a5575f76a65ac08a7f684bc37b0c98d9e715d73ddce696b58f2c72\"]", //␣
↪the signed raw txn, same as the one used in the API example
      &result,                                                                //␣
↪the reference to a pointer which will hold the result
      &error);                                                                //␣
↪the pointer which may hold a error message

  // check and print the result or error
  if (res == IN3_OK) {
    printf("Result: \n%s\n", result);
    free(result);
  } else {
    printf("Error sending tx: \n%s\n", error);
    free(error);
  }
}

void send_tx_api(in3_t* in3) {
  // prepare parameters
  address_t to, from;
  hex_to_bytes("0x63FaC9201494f0bd17B9892B9fae4d52fe3BD377", -1, from, 20);
  hex_to_bytes("0xd46e8dd67c5d32be8058bb8eb970870f07244567", -1, to, 20);

  bytes_t* data = hex_to_new_bytes(
↪"d46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675
↪", 82);

  // send the tx
  bytes_t* tx_hash = eth_sendTransaction(in3, from, to, OPTIONAL_T_VALUE(uint64_t,␣
↪0x96c0), OPTIONAL_T_VALUE(uint64_t, 0x9184e72a000), OPTIONAL_T_VALUE(uint256_t, to_
↪uint256(0x9184e72a)), OPTIONAL_T_VALUE(bytes_t, *data), OPTIONAL_T_UNDEFINED(uint64_
↪t));

  // if the result is null there was an error and we can get the latest error message␣
↪from eth_last_error()
  if (!tx_hash)
    printf("error sending the tx : %s\n", eth_last_error());
```

(continues on next page)

(continued from previous page)

```
  else {
    printf("Transaction hash: ");
    b_print(tx_hash);
    b_free(tx_hash);
  }
  b_free(data);
}
```

9.3.11 usn_device

source : in3-c/c/examples/usn_device.c

a example how to watch usn events and act upon it.

```

#include <in3/client.h>   // the core client
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/log.h>      // logging functions
#include <in3/signer.h>   // signer-api
#include <in3/usn_api.h>
#include <in3/utils.h>
#include <inttypes.h>
#include <stdio.h>
#include <time.h>
#if defined(_WIN32) || defined(WIN32)
#include <windows.h>
#else
#include <unistd.h>
#endif

static int handle_booking(usn_event_t* ev) {
  printf("\n%s Booking timestamp=%" PRIu64 "\n", ev->type == BOOKING_START ? "START"␣
↪: "STOP", ev->ts);
  return 0;
}

int main(int argc, char* argv[]) {
  // create new incubed client
  in3_t* c = in3_for_chain(CHAIN_ID_MAINNET);

  // switch to goerli
  c->chain_id = 0x5;

  // setting up a usn-device-config
  usn_device_conf_t usn;
  usn.booking_handler    = handle_booking;                                        //
↪ this is the handler, which is called for each rent/return or start/stop
  usn.c                  = c;                                                     //
↪ the incubed client
  usn.chain_id           = c->chain_id;                                           //
↪ the chain_id
  usn.devices            = NULL;                                                  //
↪ this will contain the list of devices supported
  usn.len_devices        = 0;                                                     //
↪ and length of this list
```

(continues on next page)

(continued from previous page)

```
  usn.now                   = 0;                                                    //
↪ the current timestamp
  unsigned int wait_time = 5;                                                       //
↪ the time to wait between the internval
  hex_to_bytes("0x85Ec283a3Ed4b66dF4da23656d4BF8A507383bca", -1, usn.contract, 20); //
↪ address of the usn-contract, which we copy from hex

  // register a usn-device
  usn_register_device(&usn, "office@slockit");

  // now we run en endless loop which simply wait for events on the chain.
  printf("\n start watching...\n");
  while (true) {
    usn.now               = time(NULL);                               // update the␣
↪timestamp, since this is running on embedded devices, this may be depend on the␣
↪hardware.
    unsigned int timeout = usn_update_state(&usn, wait_time) * 1000; // this will now␣
↪check for new events and trigger the handle_booking if so.

    // sleep
#if defined(_WIN32) || defined(WIN32)
    Sleep(timeout);
#else
    nanosleep((const struct timespec[]){{0, timeout * 1000000L}}, NULL);
#endif
  }

  // clean up
  in3_free(c);
  return 0;
}
```

9.3.12 usn_rent

source : in3-c/c/examples/usn_rent.c

how to send a rent transaction to a usn contract usinig the usn-api.

```

#include <in3/api_utils.h>
#include <in3/eth_api.h>  // functions for direct api-access
#include <in3/in3_init.h> // if included the verifier will automaticly be initialized.
#include <in3/signer.h>   // signer-api
#include <in3/usn_api.h>  // api for renting
#include <in3/utils.h>
#include <inttypes.h>
#include <stdio.h>

void unlock_key(in3_t* c, char* json_data, char* passwd) {
  // parse the json
  json_ctx_t* key_data = parse_json(json_data);
  if (!key_data) {
    perror("key is not parseable!\n");
    exit(EXIT_FAILURE);
  }
```

(continues on next page)

(continued from previous page)

```
  // decrypt the key
  uint8_t* pk = malloc(32);
  if (decrypt_key(key_data->result, passwd, pk) != IN3_OK) {
    perror("wrong password!\n");
    exit(EXIT_FAILURE);
  }

  // free json
  json_free(key_data);

  // create a signer with this key
  eth_set_pk_signer(c, pk);
}

int main(int argc, char* argv[]) {
  // create new incubed client
  in3_t* c = in3_for_chain(CHAIN_ID_GOERLI);

  // address of the usn-contract, which we copy from hex
  address_t contract;
  hex_to_bytes("0x85Ec283a3Ed4b66dF4da23656d4BF8A507383bca", -1, contract, 20);

  // read the key from args - I know this is not safe, but this is just a example.
  if (argc < 3) {
    perror("you need to provide a json-key and password to rent it");
    exit(EXIT_FAILURE);
  }
  char* key_data = argv[1];
  char* passwd   = argv[2];
  unlock_key(c, key_data, passwd);

  // rent it for one hour.
  uint32_t renting_seconds = 3600;

  // allocate 32 bytes for the resulting tx hash
  bytes32_t tx_hash;

  // start charging
  if (usn_rent(c, contract, NULL, "office@slockit", renting_seconds, tx_hash))
    printf("Could not start charging\n");
  else {
    printf("Charging tx successfully sent... tx_hash=0x");
    for (int i = 0; i < 32; i++) printf("%02x", tx_hash[i]);
    printf("\n");

    if (argc == 4) // just to include it : if you want to stop earlier, you can call
      usn_return(c, contract, "office@slockit", tx_hash);
  }

  // clean up
  in3_free(c);
  return 0;
}
```

9.3.13 Building

In order to run those examples, you only need a c-compiler (gcc or clang) and curl installed.

```
./build.sh
```

will build all examples in this directory. You can build them individually by executing:

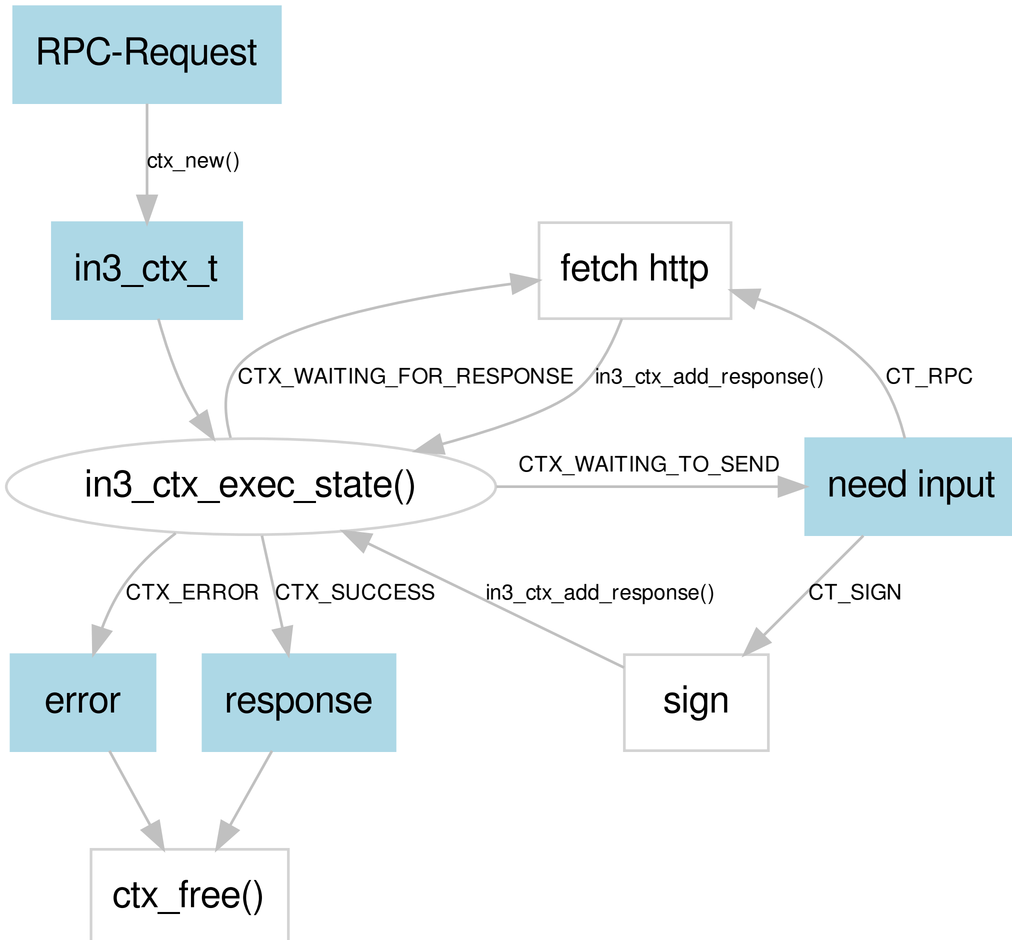
```
gcc -o get_block_api get_block_api.c -lin3 -lcurl
```

9.4 How it works

The core of incubed is the processing of json-rpc requests by fetching data from the network and verifying them. This is why in the `core`-module it is all about rpc-requests and their responses.

9.4.1 the statemachine

Each request is represented internally by the `in3_ctx_t` -struct. This context is responsible for trying to find a verifyable answer to the request and acts as a statemachine.



In order to process a request we follow these steps.

1. `ctx_new` which creates a new context by parsing a JSON-RPC request.
2. `in3_ctx_exec_state` this will try to process the state and returns the new state, which will be one of he following:

- `CTX_SUCCESS` - we have a response
- `CTX_ERROR` - we stop because of an unrecoverable error
- `CTX_WAITING_TO_SEND` - we need input and need to send out a request. By calling `in3_create_request()` the ctx will switch to the state to `CTX_WAITING_FOR_RESPONSE` until all the needed responses are repoorted. While it is possible to fetch all responses and add them before calling `in3_ctx_exec_state()`, but it would be more efficient if can send all requests out, but then create a response-queue and set one response add a time so we can return as soon as we have the first verifiable response.
- `CTX_WAITING_FOR_RESPONSE` - the request has been send, but no verifieable response is available. Once the next (or more) responses have been added, we call `in3_ctx_exec_state()` again, which will verify

all available responses. If we could verify it, we have a respoonse, if not we may either wait for more responses (in case we send out multiple requests -> CTX_WAITING_FOR_RESPONSE) or we send out new requests (CTX_WAITING_TO_SEND)

the in3_send_ctx-function will executly this:

```
in3_ret_t in3_send_ctx(in3_ctx_t* ctx) {
  ctx_req_transports_t transports = {0};
  while (true) {
    switch (in3_ctx_exec_state(ctx)) {
      case CTX_ERROR:
      case CTX_SUCCESS:
        transport_cleanup(ctx, &transports, true);
        return ctx->verification_state;

      case CTX_WAITING_FOR_RESPONSE:
        in3_handle_rpc_next(ctx, &transports);
        break;

      case CTX_WAITING_TO_SEND: {
        in3_ctx_t* last = in3_ctx_last_waiting(ctx);
        switch (last->type) {
          case CT_SIGN:
            in3_handle_sign(last);
            break;
          case CT_RPC:
            in3_handle_rpc(last, &transports);
        }
      }
    }
  }
}
```

9.4.2 sync calls with in3_send_ctx

This statemachine can be used to process requests synchronously or asynchronously. The in3_send_ctx function, which is used in most convinience-functions will do this synchronously. In order to get user input it relies on 2 callback-functions:

- to sign : *in3_signer_t* struct including its callback function is set in the in3_t configuration.
- to fetch data : a *in3_transport_send* function-pointer will be set in the in3_t configuration.

signing

For signing the client expects a *in3_signer_t* struct to be set. Setting should be done by using the *in3_set_signer()* function. This function expects 3 arguments (after the client config itself):

- sign - this is a function pointer to actual signing-function. Whenever the incubed client needs a signature it will prepare a signing context *in3_sign_ctx_t*, which holds all relevant data, like message and the address for signing. The result will always be a signature which you need to copy into the signature-field of this context. The return value must signal the success of the execution. While IN3_OK represents success, IN3_WAITINGcan be used to indicate that we need to execute again since there may be a sub-request that needs to finished up before being able to sign. In case of an error *ctx_set_error* should be used to report the details of the error including returning the IN3_E... as error-code.

- `prepare_tx`- this function is optional and gives you a chance to change the data before signing. For example signing with a mutisig would need to do manipulate the data and also the target in order to redirect it to the multisig contract.
- `wallet` - this is a optional `void*` which will be set in the signing context. It can be used to point to any data structure you may need in order to sign.

As a example this is the implemantation of the signer-function for a simple raw private key:

```
in3_ret_t eth_sign_pk_ctx(in3_sign_ctx_t* ctx) {
  uint8_t* pk = ctx->wallet;
  switch (ctx->type) {
    case SIGN_EC_RAW:
      return ec_sign_pk_raw(ctx->message.data, pk, ctx->signature);
    case SIGN_EC_HASH:
      return ec_sign_pk_hash(ctx->message.data, ctx->message.len, pk, hasher_sha3k,
↪ctx->signature);
    default:
      return IN3_ENOTSUP;
  }
  return IN3_OK;
}
```

The pk-signer uses the wallet-pointer to point to the raw 32 bytes private key and will use this to sign.

transport

The transport function is a function-pointer set in the client configuration (`in3_t`) which will be used in the `in3_send_ctx()` function whenever data are required to get from the network. the function will get a *`request_t`* object as argument.

The main responsibility of this function is to fetch the requested data and the call *`in3_ctx_add_response`* to report this to the context. if the request only sends one request to one url, this is all you have to do. But if the user uses a configuration of `request_count` >1, the `request` object will contain a list of multiples urls. In this case transport function still has 3 options to accomplish this:

1. send the payload to each url sequentially. This is **NOT** recommented, since this increases the time the user has to wait for a response. Especially if some of the request may run into a timeout.
2. send the all in parallel and wait for all the finish. This is better, but it still means, we may have to wait until the last one responses even though we may have a verifiable response already reported.
3. send them all in parallel and return as soon as we have the first response. This increases the performance since we don't have to wait if we have one. But since we don't know yet whether this response is also correct, we must be prepared to also read the other responses if needed, which means the transport would be called multiple times for the same request. In order to process multiple calls to the same resouces the request-object contains two fields:

- `cptr` - a custom `void*` which can be set in the first call pointing to recources you may need to continue in the subsequent calls.
- `action` - This value is enum (*`#in3_req_action_t`*), which indicates these current state

So only if you need to continue your call later, because you don't want to and can't set all the responses yet, you need set the `cptr` to a non NULL value. And only in this case `in3_send_ctx()` will follow this process with these states:

- `REQ_ACTION_SEND` - this will always be set in the first call.

- REQ_ACTION_RECEIVE - a call with this state indicates that there was a send call prior but since we do not have all responses yet, the transport should now set the next reponse. So this call may be called multiple times until either we have found a verifieable response or the number of urls is reached. Important during this call the urls field of the request will be NULL since this should not send a new request.
- REQ_ACTION_CLEANUP - this will only be used if the cptr was set before. Here the transport should only clean up any allocated resources. This will also be called if not all responses were used.

While there are of course existing implementations for the transport-function (as default we use in3_curl_c), especially for embedded devices you may even implement your own.

9.4.3 async calls

While for sync calls you can just implement a transport function, you can also take full control of the process which allows to execute it completly async. The basic process is the same layed out in the *state machine*.

For the js for example the main-loop is part of a async function.

```
async sendRequest(rpc) {

    // create the context
    const r = in3w.ccall('in3_create_request_ctx', 'number', ['number', 'string'],
↪[this.ptr, JSON.stringify(rpc)]);

    // hold a queue for responses for the different request contexts
    let responses = {}

    try {
      // main async loop
      while (true) {

          // execute and fetch the new state ( in this case the ctx_execute-function
↪will return the status including the created request as json)
          const state = JSON.parse(call_string('ctx_execute', r))
          switch (state.status) {
              // CTX_ERROR
              case 'error':
                  throw new Error(state.error || 'Unknown error')

              // CTX_SUCCESS
              case 'ok':
                  return state.result

              // CTX_WAITING_FOR_RESPONSE
              case 'waiting':
                  // await the promise for the next response ( the state.request
↪contains the context-pointer to know which queue)
                  await getNextResponse(responses, state.request)
                  break

              // CTX_WAITING_TO_SEND
              case 'request': {
                  // the request already contains the type, urls and payload.
                  const req = state.request
                  switch (req.type) {
                      case 'sign':
                          try {
```

(continues on next page)

(continued from previous page)

```
                                    // get the message and account from the request
                                    const [message, account] = Array.isArray(req.payload) ?␣
↪req.payload[0].params : req.payload.params;
                                    // check if can sign
                                    if (!(await this.signer.canSign(account))) throw new␣
↪Error('unknown account ' + account)

                                    // and set the signature (65 bytes) as response.
                                    setResponse(req.ctx, toHex(await this.signer.
↪sign(message, account, true, false)), 0, false)
                                } catch (ex) {
                                    // or set the error
                                    setResponse(req.ctx, ex.message || ex, 0, true)
                                }
                                break;

                            case 'rpc':
                                // here we will send a new request, which puts its␣
↪responses in a queue
                                await getNextResponse(responses, req)
                        }
                    }
                }
            }
        }
    finally {
        // we always need to cleanup
        in3w.ccall('in3_request_free', 'void', ['number'], [r])
    }
}
```

9.5 Plugins

While the core is kept as small as possible, we defined actions, which can be implemented by plugins. The core alone would not be able to do any good. While the in3-c repository already provides default implementations for all actions, as a developer you can always extend or replace those. There are good reasons to do so:

- optimizing by using a smaller plugin (like replacing the nodelist handling)
- allowing custom rpc-commands
- changing behavior . . .

9.5.1 What is a plugin?

Each plugin needs to define those 3 things:

1. **Actions** - Which actions do I want handle. This is a bitmask with the actions set. You can use any combination.
2. **Custom data** - This optional data object may contain configurations or other data. If you don't need to hold any data, you may pass `NULL`
3. **Exec-function** - This is a function pointer to a function which will be called whenever the plugin is used.

With these 3 things you can register a plugin with the `in3_plugin_register()` -function:

```
return in3_plugin_register("myplugin"          // the plugin name
         c,                                    // the client
         PLGN_ACT_TERM | PLGN_ACT_RPC_HANDLE,   // the actions to register for
         handle_rpc,                           // the plugin-function
         cutom_data,                           // the custom data (if needed)
         false);                               // a bool indicating whether it␣
↪should always add or replace a plugin with the exact same actions.
```

The Plugin-function

Each Plugin must provide a PLugin-function to execute with the following signature:

```
in3_ret_t handle(
  void*              custom_data,  // the custom data as passed in the register-function
  in3_plugin_act_t action,         // the action to execute
  void*              arguments);   // the arguments (depending on the action)
```

While the `custom_data` is just the pointer to your data-object, the `arguments` contain a pointer to a context object. This object depends on the action you are reacting.

All plugins are stored in a linked list and when we want to trigger a specific actions we will loop through all, but only execute the function if the required action is set in the bitmask. Except for `PLGN_ACT_TERM` we will loop until the first plugin handles it. The handle-function must return a return code indicating this:

- `IN3_OK` - the plugin handled it and it was succesful
- `IN3_WAITING` - the plugin handled the action, but is waiting for more data, which happens in a sub context added. As soon as this was resolved, the plugin will be called again.
- `IN3_EIGNORE` - the plugin did **NOT** handle the action and we should continue with the other plugins.
- `IN3_E...` - the plugin did handle it, but raised a error and returned the error-code. In addition you should always use the current `in3_ctx_t`to report a detailed error-message (using `ctx_set_error()`)

9.5.2 Lifecycle

PLGN_ACT_TERM

This action will be triggered during `in3_free` and must be used to free up resources which were allocated.

`arguments` : `in3_t*` - the in3-instance will be passed as argument.

9.5.3 Transport

For Transport implementations you should always register for those 3 `PLGN_ACT_TRANSPORT_SEND` | `PLGN_ACT_TRANSPORT_RECEIVE` | `PLGN_ACT_TRANSPORT_CLEAN`. This is why you can also use the macro combining those as `PLGN_ACT_TRANSPORT`

PLGN_ACT_TRANSPORT_SEND

Send will be triggered only if the request is executed synchron, whenever a new request needs to be send out. This request may contain multiple urls, but the same payload.

`arguments` : `in3_request_t*` - a request-object holding the following data:

```
typedef struct in3_request {
  char*           payload;  // the payload to send
  char**          urls;     // array of urls
  uint_fast16_t   urls_len; // number of urls
  in3_ctx_t*      ctx;      // the current context
  void*           cptr;     // a custom ptr to hold information during
} in3_request_t;
```

It is expected that a plugin will send out http-requests to each (iterating until `urls_len`) url from `urls` with the `payload`. if the payload is NULL or empty the request is a `GET`-request. Otherwise, the plugin must use send it with HTTP-Header `Content-Type: application/json` and attach the `payload`.

After the request is send out the `cptr` may be set in order to fetch the responses later. This allows us the fetch responses as they come in instead of waiting for the last response before continuing.

Example:

```
in3_ret_t transport_handle(void* custom_data, in3_plugin, in3_plugin_act_t action,␣
↪void* arguments) {
  switch (action) {

    case PLGN_ACT_TRANSPORT_SEND: {
      in3_request_t* req = arguments; // cast it to in3_request_t*

      // init the cptr
      in3_curl_t* c = _malloc(sizeof(in3_curl_t));
      c->cm         = curl_multi_init(); // init curl
      c->start      = current_ms();      // keep the staring time
      req->cptr     = c;                 // set the cptr

      // define headers
      curl_multi_setopt(c->cm, CURLMOPT_MAXCONNECTS, (long) CURL_MAX_PARALLEL);
      struct curl_slist* headers = curl_slist_append(NULL, "Accept: application/json
↪");
      if (req->payload && *req->payload)
        headers = curl_slist_append(headers, "Content-Type: application/json");
      headers    = curl_slist_append(headers, "charsets: utf-8");
      c->headers = curl_slist_append(headers, "User-Agent: in3 curl " IN3_VERSION);

      // send out requests in parallel
      for (unsigned int i = 0; i < req->urls_len; i++)
        readDataNonBlocking(c->cm, req->urls[i], req->payload, c->headers, req->ctx->
↪raw_response + i, req->ctx->client->timeout);

      return IN3_OK;
    }

    // handle other actions ...
  }
}
```

PLGN_ACT_TRANSPORT_RECEIVE

This will only triggered if the previously triggered `PLGN_ACT_TRANSPORT_SEND`

- was successfull (IN3_OK)
- if the responses were not all set yet.

- if a `cptr` was set

`arguments : in3_request_t*` - a request-object holding the data. (the payload and urls may not be set!)

The plugin needs to wait until the first response was received (or runs into a timeout). To report, please use ‘in3_req_add_response()‘‘

```
void in3_req_add_response(
    in3_request_t* req,      //  the the request
    int            index,    //  the index of the url, since this request could go␣
↪out to many urls
    bool           is_error, //  if true this will be reported as error. the message␣
↪should then be the error-message
    const char*    data,     //  the data or the the string of the response
    int            data_len, //  the length of the data or the the string (use -1 if␣
↪data is a null terminated string)
    uint32_t       time      //  the time (in ms) this request took in ms or 0 if not␣
↪possible (it will be used to calculate the weights)
);
```

In case of a succesful response:

```
in3_req_add_response(request, index, false, response_data, -1, current_ms() - start);
```

in case of an error, the data is the error message itself:

```
in3_req_add_response(request, index, true, "Timeout waiting for a response", -1, 0);
```

PLGN_ACT_TRANSPORT_CLEAN

If a previous `PLGN_ACT_TRANSPORT_SEND` has set a `cptr` this will be triggered in order to clean up memory.

`arguments : in3_request_t*` - a request-object holding the data. (the payload and urls may not be set!)

9.5.4 Signing

For Signing we have three different action. While `PLGN_ACT_SIGN` should alos react to `PLGN_ACT_SIGN_ACCOUNT`, `PLGN_ACT_SIGN_PREPARE` can also be completly independent.

PLGN_ACT_SIGN

This action is triggered as a request to sign data.

`arguments : in3_sign_ctx_t*` - the sign context will hold those data:

```
typedef struct sign_ctx {
  uint8_t            signature[65]; // the resulting signature needs to be writte␣
↪into these bytes
  d_signature_type_t type;          // the type of signature
  in3_ctx_t*         ctx;           // the context of the request in order report␣
↪errors
  bytes_t            message;       // the message to sign
  bytes_t            account;       // the account to use for the signature  (if set)
} in3_sign_ctx_t;
```

The signature must be 65 bytes and in the format , where v must be the recovery byte and should only be 1 or 0.

```
r[32]|s[32]|v[1]
```

Currently there are 2 types of sign-request:

- SIGN_EC_RAW : the data is already 256bits and may be used directly
- SIGN_EC_HASH : the data may be any kind of message, and need to be hashed first. As hash we will use Keccak.

Example:

```
in3_ret_t eth_sign_pk(void* data, in3_plugin_act_t action, void* args) {
  // the data are our pk
  uint8_t* pk = data;

  switch (action) {

    case PLGN_ACT_SIGN: {
      // cast the context
      in3_sign_ctx_t* ctx = args;

      // if there is a account set, we only sign if this matches our account
      // this way we allow multiple accounts to added as plugin
      if (ctx->account.len == 20) {
        address_t adr;
        get_address(pk, adr);
        if (memcmp(adr, ctx->account.data, 20))
           return IN3_EIGNORE; // does not match, let someone else handle it
      }

      // sign based on sign type
      switch (ctx->type) {
        case SIGN_EC_RAW:
          return ec_sign_pk_raw(ctx->message.data, pk, ctx->signature);
        case SIGN_EC_HASH:
          return ec_sign_pk_hash(ctx->message.data, ctx->message.len, pk, hasher_
↪sha3k, ctx->signature);
        default:
          return IN3_ENOTSUP;
      }
    }

    case PLGN_ACT_SIGN_ACCOUNT: {
      // cast the context
      in3_sign_account_ctx_t* ctx = args;

      // generate the address from the key
      get_address(pk, ctx->account);
      return IN3_OK;
    }

    default:
      return IN3_ENOTSUP;
  }
}


in3_ret_t eth_set_pk_signer(in3_t* in3, bytes32_t pk) {
```

(continues on next page)

(continued from previous page)

```
  // we register for both ACCOUNT and SIGN
  return plugin_register(in3, PLGN_ACT_SIGN_ACCOUNT | PLGN_ACT_SIGN, eth_sign_pk, pk,␣
↪false);
}
```

PLGN_ACT_SIGN_ACCOUNT

if we are about to sign data and need to know the address of the account abnout to sign, this action will be triggered in order to find out. This is needed if you want to send a transaction without specifying the `from` address, we will still need to get the nonce for this account before signing.

`arguments` : `in3_sign_account_ctx_t*` - the account context will hold those data:

```
typedef struct sign_account_ctx {
  in3_ctx_t* ctx;      // the context of the request in order report errors
  address_t  account; // the account to use for the signature
} in3_sign_account_ctx_t;
```

The implementation should return a status code ´IN3_OK‘ if it successfully wrote the address of the account into the content:

Example:

```
in3_ret_t eth_sign_pk(void* data, in3_plugin_act_t action, void* args) {
  // the data are our pk
  uint8_t* pk = data;

  switch (action) {

    case PLGN_ACT_SIGN_ACCOUNT: {
      // cast the context
      in3_sign_account_ctx_t* ctx = args;

      // generate the address from the key
      // and write it into account
      get_address(pk, ctx->account);
      return IN3_OK;
    }

    // handle other actions ...

    default:
      return IN3_ENOTSUP;
  }
}
```

PLGN_ACT_SIGN_PREPARE

The Prepare-action is triggered before signing and gives a plugin the chance to change the data. This is needed if you want to send a transaction through a multisig. Here we have to change the `data` and `to` address.

`arguments` : `in3_sign_prepare_ctx_t*` - the prepare context will hold those data:

```
typedef struct sign_prepare_ctx {
  struct in3_ctx* ctx;      // the context of the request in order report errors
  address_t       account; // the account to use for the signature
  bytes_t         old_tx;  // the data to sign
  bytes_t         new_tx;  // the new data to be set

} in3_sign_prepare_ctx_t;
```

the tx-data will be in a form ready to sign, which means those are rlp-encoded data of a transaction without a signature, but the chain-id as v-value.

In order to decode the data you must use rlp.h:

```
#define decode(data,index,dst,msg) if (rlp_decode_in_list(data, index, dst) != 1)␣
↪return ctx_set_error(ctx, "invalid" msg "in txdata", IN3_EINVAL);

in3_ret_t decode_tx(in3_ctx_t* ctx, bytes_t raw, tx_data_t* result) {
  decode(&raw, 0, &result->nonce    , "nonce");
  decode(&raw, 1, &result->gas_price, "gas_price");
  decode(&raw, 2, &result->gas      , "gas");
  decode(&raw, 3, &result->to       , "to");
  decode(&raw, 4, &result->value    , "value");
  decode(&raw, 5, &result->data     , "data");
  decode(&raw, 6, &result->v        , "v");
  return IN3_OK;
}
```

and of course once the data has changes you need to encode it again and set it as 'nex_tx''

9.5.5 RPC Handling

PLGN_ACT_RPC_HANDLE

Triggered for each rpc-request in order to give plugins a chance to directly handle it. If no onoe handles it it will be send to the nodes.

`arguments` : `in3_rpc_handle_ctx_t*` - the rpc_handle context will hold those data:

```
typedef struct {
  in3_ctx_t*       ctx;      // Request context.
  d_token_t*       request;  // request
  in3_response_t** response; // the response which a prehandle-method should set
} in3_rpc_handle_ctx_t;
```

the steps to add a new custom rpc-method will be the following.

1. get the method and params:

```
char* method      = d_get_stringk(rpc->request, K_METHOD);
d_token_t* params = d_get(rpc->request, K_PARAMS);
```

1. check if you can handle it
2. handle it and set the result

```
in3_rpc_handle_with_int(rpc,result);
```

for setting the result you should use one of the `in3_rpc_handle_...` methods. Those will create the response and build the JSON-string with the result. While most of those expect the result as a sngle value you can also return a complex JSON-Object. In this case you have to create a string builder:

```
sb_t* writer = in3_rpc_handle_start(rpc);
sb_add_chars(writer, "{\"raw\":\"");
sb_add_escaped_chars(writer, raw_string);
// ... more data
sb_add_chars(writer, "}");
return in3_rpc_handle_finish(rpc);
```

1. In case of an error, simply set the error in the context, with the right message and error-code:

```
if (d_len(params)<1) return ctx_set_error(rpc->ctx, "Not enough parameters", IN3_
↪EINVAL);
```

If the reequest needs additional subrequests, you need to follow the pattern of sending a request asynchron in a state machine:

```
// we want to get the nonce.....
uint64_t  nonce =0;

// check if a request is already existing
in3_ctx_t* ctx = ctx_find_required(rpc->ctx, "eth_getTransactionCount");
if (ctx) {
  // found one - so we check if it is ready.
  switch (in3_ctx_state(ctx)) {
    // in case of an error, we report it back to the parent context
    case CTX_ERROR:
      return ctx_set_error(rpc->ctx, ctx->error, IN3_EUNKNOWN);
    // if we are still waiting, we stop here and report it.
    case CTX_WAITING_FOR_RESPONSE:
    case CTX_WAITING_TO_SEND:
      return IN3_WAITING;

    // if it is useable, we can now handle the result.
    case CTX_SUCCESS: {
      // check if the response contains a error.
      TRY(ctx_check_response_error(ctx, 0))

      // read the nonce
      nonce = d_get_longk(ctx->responses[0], K_RESULT);
    }
  }
}
else {
  // no required context found yet, so we create one:

  // since this is a subrequest it will be freed when the parent is freed.
  // allocate memory for the request-string
  char* req = _malloc(strlen(params) + 200);
  // create it
  sprintf(req, "{\"method\":\"eth_getTransactionCount\",\"jsonrpc\":\"2.0\",\"id\":1,\
↪"params\":[\"%s\",\"latest\"]}", account_hex_string);
  // and add the request context to the parent.
  return ctx_add_required(parent, ctx_new(parent->client, req));
}
```

(continues on next page)

(continued from previous page)

```
// continue here and use the nonce....
```

Here is a simple Example how to register a plugin hashing data:

```
static in3_ret_t handle_intern(void* pdata, in3_plugin_act_t action, void* args) {
  UNUSED_VAR(pdata);

  // cast args
  in3_rpc_handle_ctx_t* rpc = args;

  swtch (action) {
    case PLGN_ACT_RPC_HANDLE: {
      // get method and params
      char*                         method   = d_get_stringk(rpc->request, K_METHOD);
      d_token_t*                    params   = d_get(rpc->request, K_PARAMS);

      // do we support it?
      if (strcmp(method, "web3_sha3") == 0) {
        // check the params
        if (!params || d_len(params) != 1) return ctx_set_error(rpc->ctx, "invalid␣
↪params", IN3_EINVAL);
        bytes32_t hash;
        // hash the first param
        keccak(d_to_bytes(d_get_at(params,0)), hash);
        // return the hash as resut.
        return in3_rpc_handle_with_bytes(ctx, bytes(hash, 32));
      }

      // we don't support this method, so we ignore it.
      return IN3_EIGNORE;
    }

    default:
      return IN3_ENOTSUP;
  }
}

in3_ret_t in3_register_rpc_handler(in3_t* c) {
  return plugin_register(c, PLGN_ACT_RPC_HANDLE, handle_intern, NULL, false);
}
```

PLGN_ACT_RPC_VERIFY

This plugin reprresents a verifier. It will be triggered after we have received a response from a node.

`arguments` : `in3_vctx_t*` - the verification context will hold those data:

```
typedef struct {
  in3_ctx_t*   ctx;                   // Request context.
  in3_chain_t* chain;                 // the chain definition.
  d_token_t*   result;                // the result to verify
  d_token_t*   request;               // the request sent.
  d_token_t*   proof;                 // the delivered proof.
  in3_t*       client;                // the client.
  uint64_t     last_validator_change; // Block number of last change of the validator␣
↪list
```

(continues on next page)

(continued from previous page)

```
  uint64_t       currentBlock;          // Block number of latest block
  int            index;                 // the index of the request within the bulk
} in3_vctx_t;
```

Example:

```
in3_ret_t in3_verify_ipfs(void* pdata, in3_plugin_act_t action, void* args) {
  if (action!=PLGN_ACT_RPC_VERIFY) return IN3_ENOTSUP;
  UNUSED_VAR(pdata);

  // we want this verifier to handle ipfs-chains
  if (vc->chain->type != CHAIN_IPFS) return IN3_EIGNORE;


  in3_vctx_t* vc     = args;
  char*       method = d_get_stringk(vc->request, K_METHOD);
  d_token_t*  params = d_get(vc->request, K_PARAMS);

  // did we ask for proof?
  if (in3_ctx_get_proof(vc->ctx, vc->index) == PROOF_NONE) return IN3_OK;

  // do we have a result? if not it is a vaslid error-response
  if (!vc->result)
    return IN3_OK;

  if (strcmp(method, "ipfs_get") == 0)
    return ipfs_verify_hash(d_string(vc->result),
                            d_get_string_at(params, 1) ? d_get_string_at(params, 1) :
↪"base64",
                            d_get_string_at(params, 0));

  // could not verify, so we hope some other plugin will
  return IN3_EIGNORE;
}

in3_ret_t in3_register_ipfs(in3_t* c) {
  return plugin_register(c, PLGN_ACT_RPC_VERIFY, in3_verify_ipfs, NULL, false);
}
```

9.5.6 Cache/Storage

For Cache implementations you also need to register all 3 actions.

PLGN_ACT_CACHE_SET

This action will be triggered whenever there is something worth putting in a cache. If no plugin picks it up, it is ok, since the cache is optional.

`arguments` : `in3_cache_ctx_t*` - the cache context will hold those data:

```
typedef struct in3_cache_ctx {
  in3_ctx_t* ctx;      // the request context
  char*      key;      // the key to fetch
```

(continues on next page)

(continued from previous page)

```
  bytes_t*    content; // the content to set
} in3_cache_ctx_t;
```

in the case of `CACHE_SET` the content will point to the bytes we need to store somewhere. If for whatever reason the item can not be stored, a `IN3_EIGNORE` should be send, since to indicate that no action took place.

```
Example:

```c
in3_ret_t handle_storage(void* data, in3_plugin_act_t action, void* arg) {
  in3_cache_ctx_t* ctx = arg;
  switch (action) {
    case PLGN_ACT_CACHE_GET: {
       ctx->content = storage_get_item(data, ctx->key);
       return ctx->content ? IN3_OK : IN3_EIGNORE;
    }
    case PLGN_ACT_CACHE_SET: {
      storage_set_item(data, ctx->key, ctx->content);
      return IN3_OK;
    }
    case PLGN_ACT_CACHE_CLEAR: {
      storage_clear(data);
      return IN3_OK;
    }
    default: return IN3_EINVAL;
  }
}

in3_ret_t in3_register_file_storage(in3_t* c) {
  return plugin_register(c, PLGN_ACT_CACHE, handle_storage, NULL, true);
}
```

PLGN_ACT_CACHE_GET

This action will be triggered whenever we access the cache in order to get values.

`arguments` : `in3_cache_ctx_t*` - the cache context will hold those data:

```
typedef struct in3_cache_ctx {
  in3_ctx_t* ctx;      // the request context
  char*      key;      // the key to fetch
  bytes_t*   content; // the content to set
} in3_cache_ctx_t;
```

in the case of `CACHE_GET` the content will be NULL and needs to be set to point to the found values. If we did not find it in the cache, we must return `IN3_EIGNORE`.

```
Example:

```c
ctx->content = storage_get_item(data, ctx->key);
return ctx->content ? IN3_OK : IN3_EIGNORE;
```

PLGN_ACT_CACHE_CLEAR

This action will clear all stored values in the cache.

`arguments` :NULL - so no argument will be passed.

9.5.7 Configuration

For Configuration there are 2 actions for getting and setting. You should always implement both.

Example:

```
static in3_ret_t handle_btc(void* custom_data, in3_plugin_act_t action, void* args) {
  btc_target_conf_t* conf = custom_data;
  switch (action) {
    // clean up
    case PLGN_ACT_TERM: {
      if (conf->data.data) _free(conf->data.data);
      _free(conf);
      return IN3_OK;
    }

    // read config
    case PLGN_ACT_CONFIG_GET: {
      in3_get_config_ctx_t* cctx = args;
      sb_add_chars(cctx->sb, ",\"maxDAP\":");
      sb_add_int(cctx->sb, conf->max_daps);
      sb_add_chars(cctx->sb, ",\"maxDiff\":");
      sb_add_int(cctx->sb, conf->max_diff);
      return IN3_OK;
    }

    // configure
    case PLGN_ACT_CONFIG_SET: {
      in3_configure_ctx_t* cctx = args;
      if (cctx->token->key == key("maxDAP"))
        conf->max_daps = d_int(cctx->token);
      else if (cctx->token->key == key("maxDiff"))
        conf->max_diff = d_int(cctx->token);
      else
        return IN3_EIGNORE;
      return IN3_OK;
    }

    case PLGN_ACT_RPC_VERIFY:
      return in3_verify_btc(conf, pctx);

    default:
      return IN3_ENOTSUP;
  }
}


in3_ret_t in3_register_btc(in3_t* c) {
  // init the config with defaults
  btc_target_conf_t* tc = _calloc(1, sizeof(btc_target_conf_t));
  tc->max_daps            = 20;
```

(continues on next page)

(continued from previous page)

```
  tc->max_diff          = 10;
  tc->dap_limit         = 20;

  return plugin_register(c, PLGN_ACT_RPC_VERIFY | PLGN_ACT_TERM | PLGN_ACT_CONFIG_GET␣
↪| PLGN_ACT_CONFIG_SET, handle_btc, tc, false);
}
```

PLGN_ACT_CONFIG_GET

This action will be triggered during `in3_get_config()` and should dump all config from all plugins.

`arguments` : `in3_get_config_ctx_t*` - the config context will hold those data:

```
typedef struct in3_get_config_ctx {
  in3_t* client;
  sb_t*  sb;
} in3_get_config_ctx_t;
```

if you are using any configuration you should use the `sb` field and add your values to it. Each property must start with a comma.

```
in3_get_config_ctx_t* cctx = args;
sb_add_chars(cctx->sb, ",\"maxDAP\":");
sb_add_int(cctx->sb, conf->max_daps);
sb_add_chars(cctx->sb, ",\"maxDiff\":");
sb_add_int(cctx->sb, conf->max_diff);
```

PLGN_ACT_CONFIG_SET

This action will be triggered during the configuration-process. While going through all config-properties, it will ask the plugins in case a config was not handled. So this action may be triggered multiple times. And the plugin should only return `IN3_OK` if it was handled. If no plugin handles it, a error will be thrown.

`arguments` : `in3_configure_ctx_t*` - the cache context will hold those data:

```
typedef struct in3_configure_ctx {
  in3_t*     client; // the client to configure
  d_token_t* token;  // the token not handled yet
} in3_configure_ctx_t;
```

In order to check if the token is relevant for you, you simply check the name of the property and handle its value:

```
in3_configure_ctx_t* cctx = pctx;
if (cctx->token->key == key("maxDAP"))
  conf->max_daps = d_int(cctx->token);
else if (cctx->token->key == key("maxDiff"))
  conf->max_diff = d_int(cctx->token);
else
  return IN3_EIGNORE;
return IN3_OK;
```

9.5.8 Payment

PLGN_ACT_PAY_PREPARE

PLGN_ACT_PAY_FOLLOWUP

PLGN_ACT_PAY_HANDLE

PLGN_ACT_PAY_SIGN_REQ

this will be triggered in order to sign a request. It will provide a request_hash and expects a signature.

`arguments` : `in3_pay_sign_req_ctx_t*` - the sign context will hold those data:

```
typedef struct {
  in3_ctx_t* ctx;
  d_token_t* request;
  bytes32_t  request_hash;
  uint8_t    signature[65];
} in3_pay_sign_req_ctx_t;
```

It is expected that the plugin will create a signature and write it into the context.

Example:

```
in3_pay_sign_req_ctx_t* ctx = args;
return ec_sign_pk_raw(ctx->request_hash, pk->key, ctx->signature);
```

9.5.9 Nodelist

PLGN_ACT_NL_PICK_DATA

PLGN_ACT_NL_PICK_SIGNER

PLGN_ACT_NL_PICK_FOLLOWUP

9.6 Integration of Ledger Nano S

1. Ways to integrate Ledger Nano S
2. Build incubed source with ledger nano module
3. Start using ledger nano s device with Incubed

9.6.1 Ways to integrate Ledger Nano S

Currently there are two ways to integrate Ledger Nano S with incubed for transaction and message signing:

1. Install Ethereum app from Ledger Manager
2. Setup development environment and install incubed signer app on your Ledger device

Option 1 is the convinient choice for most of the people as incubed signer app is not available to be installed from Ledger Manager and it will take efforts to configure development environment for ledger manager. The main differences in above approaches are following:

If you are confortable with Option 1 , all you need to do is setup you Ledger device as per usual instructions and install Ethereum app form Ledger Manager store. Otherwise if you are interested in Option 2 Please follow all the instructions given in "Setup development environment for ledger nano s" section .

```
Ethereum official Ledger app requires rlp encoded transactions for  signing and there
↪is not much scope for customization.Currently we have support for following
↪operations with Ethereum app:
1. Getting public key
2. Sign Transactions
3. Sign Messages

Incubed signer app required just hash , so it is better option if you are looking to
↪integrate incubed in such a way that you would manage all data formation on your
↪end and use just hash to get signture from Ledger Nano S and use the signature as
↪per your wish.
```

Setup development environment for ledger nano s

Setting up dev environment for Ledger nano s is one time activity and incubed signer application will be available to install directly from Ledger Manager in future. Ledger nano applications need linux System (recommended is Ubuntu) to build the binary to be installed on Ledger nano devices

Download Toolchains and Nanos ledger SDK (As per latest Ubuntu LTS)

Download the Nano S SDK in bolos-sdk folder

```
$ git clone https://github.com/ledgerhq/nanos-secure-sdk
```

```
Download a prebuild gcc and move it to bolos-sdk folder
        Ref: https://launchpad.net/gcc-arm-embedded/+milestone/5-2016-q1-update

Download a prebuild clang and rename the folder to clang-arm-fropi then move it to
↪bolos-sdk folder
        Ref: http://releases.llvm.org/download.html#4.0.0
```

Add environment variables:

```
sudo -H gedit /etc/environment
```

```
ADD PATH TO BOLOS SDK:
BOLOS_SDK="<path>/nanos-secure-sdk"

ADD GCCPATH VARIABLE
GCCPATH="<path>/gcc-arm-none-eabi-5_3-2016q1/bin/"

ADD CLANGPATH
CLANGPATH="<path>/clang-arm-fropi/bin/"
```

Download and install ledger python tools

Installation prerequisites :

```
$ sudo apt-get install libudev-dev <
$ sudo apt-get install libusb-1.0-0-dev
$ sudo apt-get install python-dev (python 2.7)
$ sudo apt-get install virtualenv
```

Installation of ledgerblue:

```
$ virtualenv ledger
$ source ledger/bin/activate
$ pip install ledgerblue
```

Ref: https://github.com/LedgerHQ/blue-loader-python

Download and install ledger udev rules

run script from the above download

```

```

Open new terminal and check for following installations

```
$ sudo apt-get install gcc-multilib
$ sudo apt-get install libc6-dev:i386
```

Install incubed signer app

Once you complete all the steps, go to folder "c/src/signer/ledger-nano/firmware" and run following command , It will ask you to enter pin for approve installation on ledger nano device. follow all the steps and it will be done.

```
make load
```

9.6.2 Build incubed source with ledger nano module

To build incubed source with ledger nano:-

1. Open root CMakeLists file and find LEDGER_NANO option
2. Turn LEDGER_NANO option ON which is by default OFF
3. Build incubed source

```
cd build
cmake  .. && make
```

9.6.3 Start using ledger nano s device with Incubed

Open the application on your ledger nano s usb device and make signing requests from incubed.

Following is the sample command to sendTransaction from command line utility:-

```
bin/in3 send -to 0xd46e8dd67c5d32be8058bb8eb970870f07244567  -gas 0x96c0  -value␣
↪0x9184e72a  -path 0x2c3c000000 -debug
```

-path points to specific public/private key pair inside HD wallet derivation path . For Ethereum the default path is m/44'/60'/0'/0 , which we can pass in simplified way as hex string i.e [44,60,00,00,00] => 0x2c3c000000

If you want to use apis to integrate ledger nano support in your incubed application , feel free to explore apis given following header files:-

```
ledger_signer.h : It contains APIs to integrate ledger nano device with incubed␣
↪signer app.
ethereum_apdu_client.h : It contains APIs to integrate ledger nano device with␣
↪Ethereum ledger app.
```

9.7 Module api

9.7.1 btc_api.h

BTC API.

This header-file defines easy to use function, which are preparing the JSON-RPC-Request, which is then executed and verified by the incubed-client.

File: c/src/api/btc/btc_api.h

btc_last_error ()

< The current error or null if all is ok

```
#define btc_last_error () api_last_error()
```

btc_transaction_in_t

the tx in

The stuct contains following fields:

uint32_t	**vout**	the tx index of the output
bytes32_t	**txid**	the tx id of the output
uint32_t	**sequence**	the sequence
bytes_t	**script**	the script
bytes_t	**txinwitness**	witnessdata (if used)

btc_transaction_out_t

the tx out

The stuct contains following fields:

`uint64_t`	**value**	the value of the tx
`uint32_t`	**n**	the index
bytes_t	**script_pubkey**	the script pubkey (or signature)

btc_transaction_t

a transaction

The stuct contains following fields:

`bool`	**in_active_chain**	true if it is part of the active chain
bytes_t	**data**	the serialized transaction-data
bytes32_t	**txid**	the transaction id
bytes32_t	**hash**	the transaction hash
`uint32_t`	**size**	raw size of the transaction
`uint32_t`	**vsize**	virtual size of the transaction
`uint32_t`	**weight**	weight of the tx
`uint32_t`	**version**	used version
`uint32_t`	**locktime**	locktime
*btc_transaction_in_t **	**vin**	array of transaction inputs
*btc_transaction_out_t **	**vout**	array of transaction outputs
`uint32_t`	**vin_len**	number of tx inputs
`uint32_t`	**vout_len**	number of tx outputs
bytes32_t	**blockhash**	hash of block containing the tx
`uint32_t`	**confirmations**	number of confirmations or blocks mined on top of the containing block
`uint32_t`	**time**	unix timestamp in seconds since 1970
`uint32_t`	**blocktime**	unix timestamp in seconds since 1970

btc_blockheader_t

the blockheader

The stuct contains following fields:

bytes32_t	**hash**	the hash of the blockheader
`uint32_t`	**confirmations**	number of confirmations or blocks mined on top of the containing block
`uint32_t`	**height**	block number
`uint32_t`	**version**	used version
bytes32_t	**merkleroot**	merkle root of the trie of all transactions in the block
`uint32_t`	**time**	unix timestamp in seconds since 1970
`uint32_t`	**nonce**	nonce-field of the block
`uint8_t`	**bits**	bits (target) for the block
bytes32_t	**chainwork**	total amount of work since genesis
`uint32_t`	**n_tx**	number of transactions in the block
bytes32_t	**previous_hash**	hash of the parent blockheader
bytes32_t	**next_hash**	hash of the next blockheader
`uint8_t`	**data**	raw serialized header-bytes

btc_block_txdata_t

a block with all transactions including their full data

The stuct contains following fields:

btc_blockheader_t	**header**	the blockheader
`uint32_t`	**tx_len**	number of transactions
btc_transaction_t *	**tx**	array of transactiondata

btc_block_txids_t

a block with all transaction ids

The stuct contains following fields:

btc_blockheader_t	**header**	the blockheader
`uint32_t`	**tx_len**	number of transactions
bytes32_t *	**tx**	array of transaction ids

btc_get_transaction_bytes

```
bytes_t* btc_get_transaction_bytes(in3_t *in3, bytes32_t txid);
```

gets the transaction as raw bytes or null if it does not exist.

You must free the result with b_free() after use!

arguments:

in3_t *	**in3**	the in3-instance
bytes32_t	**txid**	the txid

returns: `bytes_t *`

btc_get_transaction

```
btc_transaction_t* btc_get_transaction(in3_t *in3, bytes32_t txid);
```

gets the transaction as struct or null if it does not exist.

You must free the result with free() after use!

arguments:

in3_t *	**in3**	the in3-instance
bytes32_t	**txid**	the txid

returns: `btc_transaction_t *`

btc_get_blockheader

```
btc_blockheader_t* btc_get_blockheader(in3_t *in3, bytes32_t blockhash);
```

gets the blockheader as struct or null if it does not exist.

You must free the result with free() after use!

arguments:

*in3_t **	**in3**	the in3-instance
bytes32_t	**blockhash**	the block hash

returns: *btc_blockheader_t **

btc_get_blockheader_bytes

```
bytes_t* btc_get_blockheader_bytes(in3_t *in3, bytes32_t blockhash);
```

gets the blockheader as raw serialized data (80 bytes) or null if it does not exist.

You must free the result with b_free() after use!

arguments:

*in3_t **	**in3**	the in3-instance
bytes32_t	**blockhash**	the block hash

returns: *bytes_t **

btc_get_block_txdata

```
btc_block_txdata_t* btc_get_block_txdata(in3_t *in3, bytes32_t blockhash);
```

gets the block as struct including all transaction data or null if it does not exist.

You must free the result with free() after use!

arguments:

*in3_t **	**in3**	the in3-instance
bytes32_t	**blockhash**	the block hash

returns: *btc_block_txdata_t **

btc_get_block_txids

```
btc_block_txids_t* btc_get_block_txids(in3_t *in3, bytes32_t blockhash);
```

gets the block as struct including all transaction ids or null if it does not exist.

You must free the result with free() after use!

arguments:

*in3_t **	**in3**	the in3-instance
bytes32_t	**blockhash**	the block hash

returns: `btc_block_txids_t *`

btc_get_block_bytes

```
bytes_t* btc_get_block_bytes(in3_t *in3, bytes32_t blockhash);
```

gets the block as raw serialized block bytes including all transactions or null if it does not exist.

You must free the result with b_free() after use!

arguments:

*in3_t **	**in3**	the in3-instance
bytes32_t	**blockhash**	the block hash

returns: `bytes_t *`

btc_d_to_tx

```
btc_transaction_t* btc_d_to_tx(d_token_t *t);
```

Deserialization helpers.

arguments:

*d_token_t **	**t**

returns: `btc_transaction_t *`

btc_d_to_blockheader

```
btc_blockheader_t* btc_d_to_blockheader(d_token_t *t);
```

Deserializes a `btc_transaction_t` type.

You must free the result with free() after use!

arguments:

*d_token_t **	**t**

returns: `btc_blockheader_t *`

btc_d_to_block_txids

```
btc_block_txids_t* btc_d_to_block_txids(d_token_t *t);
```

Deserializes a `btc_blockheader_t` type.

You must free the result with free() after use!

arguments:

*d_token_t **	**t**

returns: *btc_block_txids_t **

btc_d_to_block_txdata

```
btc_block_txdata_t* btc_d_to_block_txdata(d_token_t *t);
```

Deserializes a `btc_block_txids_t` type.

You must free the result with free() after use!

arguments:

*d_token_t **	**t**

returns: *btc_block_txdata_t **

9.7.2 eth_api.h

Ethereum API.

This header-file defines easy to use function, which are preparing the JSON-RPC-Request, which is then executed and verified by the incubed-client.

File: c/src/api/eth1/eth_api.h

BLKNUM (blk)

Initializer macros for eth_blknum_t.

```
#define BLKNUM (blk) ((eth_blknum_t){.u64 = blk, .is_u64 = true})
```

BLKNUM_LATEST ()

```
#define BLKNUM_LATEST () ((eth_blknum_t){.def = BLK_LATEST, .is_u64 = false})
```

BLKNUM_EARLIEST ()

```
#define BLKNUM_EARLIEST () ((eth_blknum_t){.def = BLK_EARLIEST, .is_u64 = false})
```

BLKNUM_PENDING ()

The current error or null if all is ok.

```
#define BLKNUM_PENDING () ((eth_blknum_t){.def = BLK_PENDING, .is_u64 = false})
```

eth_last_error ()

```
#define eth_last_error () api_last_error()
```

eth_blknum_def_t

Abstract type for holding a block number.

The enum type contains the following values:

BLK_LATEST	0
BLK_EARLIEST	1
BLK_PENDING	2

eth_tx_t

A transaction.

The stuct contains following fields:

bytes32_t	**hash**	the blockhash
bytes32_t	**block_hash**	hash of ther containnig block
`uint64_t`	**block_number**	number of the containing block
address_t	**from**	sender of the tx
`uint64_t`	**gas**	gas send along
`uint64_t`	**gas_price**	gas price used
bytes_t	**data**	data send along with the transaction
`uint64_t`	**nonce**	nonce of the transaction
address_t	**to**	receiver of the address 0x0000. . -Address is used for contract creation.
uint256_t	**value**	the value in wei send
`int`	**transaction_index**	the transaction index
`uint8_t`	**signature**	signature of the transaction

eth_block_t

An Ethereum Block.

The stuct contains following fields:

uint64_t	**number**	the blockNumber
bytes32_t	**hash**	the blockhash
uint64_t	**gasUsed**	gas used by all the transactions
uint64_t	**gasLimit**	gasLimit
address_t	**author**	the author of the block.
uint256_t	**difficulty**	the difficulty of the block.
bytes_t	**extra_data**	the extra_data of the block.
uint8_t	**logsBloom**	the logsBloom-data
bytes32_t	**parent_hash**	the hash of the parent-block
bytes32_t	**sha3_uncles**	root hash of the uncle-trie
bytes32_t	**state_root**	root hash of the state-trie
bytes32_t	**receipts_root**	root of the receipts trie
bytes32_t	**transaction_root**	root of the transaction trie
int	**tx_count**	number of transactions in the block
eth_tx_t *	**tx_data**	array of transaction data or NULL if not requested
bytes32_t *	**tx_hashes**	array of transaction hashes or NULL if not requested
uint64_t	**timestamp**	the unix timestamp of the block
bytes_t *	**seal_fields**	sealed fields
int	**seal_fields_count**	number of seal fields

eth_log_t

A linked list of Ethereum Logs

The stuct contains following fields:

bool	**removed**	true when the log was removed, due to a chain reorganization. false if its a valid log
size_t	**log_index**	log index position in the block
size_t	**transaction_index**	transactions index position log was created from
bytes32_t	**transaction_hash**	hash of the transactions this log was created from
bytes32_t	**block_hash**	hash of the block where this log was in
uint64_t	**block_number**	the block number where this log was in
address_t	**address**	address from which this log originated
bytes_t	**data**	non-indexed arguments of the log
bytes32_t *	**topics**	array of 0 to 4 32 Bytes DATA of indexed log arguments
size_t	**topic_count**	counter for topics
eth_logstruct , *	**next**	pointer to next log in list or NULL

eth_tx_receipt_t

A transaction receipt.

The stuct contains following fields:

bytes32_t	**transaction_hash**	the transaction hash
int	**transaction_index**	the transaction index
bytes32_t	**block_hash**	hash of ther containnig block
uint64_t	**block_number**	number of the containing block
uint64_t	**cumulative_gas_used**	total amount of gas used by block
uint64_t	**gas_used**	amount of gas used by this specific transaction
bytes_t *	**contract_address**	contract address created (if the transaction was a contract creation) or NULL
bool	**status**	1 if transaction succeeded, 0 otherwise.
eth_log_t *	**logs**	array of log objects, which this transaction generated

DEFINE_OPTIONAL_T

```
DEFINE_OPTIONAL_T(uint64_t);
```

Optional types.

arguments:

uint64_t

returns: ‘‘

DEFINE_OPTIONAL_T

```
DEFINE_OPTIONAL_T(bytes_t);
```

arguments:

bytes_t

returns: ‘‘

DEFINE_OPTIONAL_T

```
DEFINE_OPTIONAL_T(address_t);
```

arguments:

address_t

returns: ‘‘

DEFINE_OPTIONAL_T

```
DEFINE_OPTIONAL_T(uint256_t);
```

arguments:

uint256_t

returns: ‘‘

eth_getStorageAt

```
uint256_t eth_getStorageAt(in3_t *in3, address_t account, bytes32_t key, eth_blknum_t␣
↪block);
```

Returns the storage value of a given address.

arguments:

*in3_t **	**in3**
address_t	**account**
bytes32_t	**key**
eth_blknum_t	**block**

returns: `uint256_t`

eth_getCode

```
bytes_t eth_getCode(in3_t *in3, address_t account, eth_blknum_t block);
```

Returns the code of the account of given address.

(Make sure you free the data-point of the result after use.)

arguments:

*in3_t **	**in3**
address_t	**account**
eth_blknum_t	**block**

returns: `bytes_t`

eth_getBalance

```
uint256_t eth_getBalance(in3_t *in3, address_t account, eth_blknum_t block);
```

Returns the balance of the account of given address.

arguments:

*in3_t **	**in3**
address_t	**account**
eth_blknum_t	**block**

returns: `uint256_t`

eth_blockNumber

```
uint64_t eth_blockNumber(in3_t *in3);
```

Returns the current blockNumber, if bn==0 an error occured and you should check eth_last_error()

arguments:

*in3_t **	**in3**

returns: `uint64_t`

eth_gasPrice

```
uint64_t eth_gasPrice(in3_t *in3);
```

Returns the current price per gas in wei.

arguments:

*in3_t **	**in3**

returns: `uint64_t`

eth_getBlockByNumber

```
eth_block_t* eth_getBlockByNumber(in3_t *in3, eth_blknum_t number, bool include_tx);
```

Returns the block for the given number (if number==0, the latest will be returned).

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
eth_blknum_t	**number**
`bool`	**include_tx**

returns: `eth_block_t *`

eth_getBlockByHash

```
eth_block_t* eth_getBlockByHash(in3_t *in3, bytes32_t hash, bool include_tx);
```

Returns the block for the given hash.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
bytes32_t	**hash**
`bool`	**include_tx**

returns: *eth_block_t **

eth_getLogs

```
eth_log_t* eth_getLogs(in3_t *in3, char *fopt);
```

Returns a linked list of logs.

If result is null, check eth_last_error()! otherwise make sure to free the log, its topics and data after using it!

arguments:

*in3_t **	**in3**
`char *`	**fopt**

returns: *eth_log_t **

eth_newFilter

```
in3_ret_t eth_newFilter(in3_t *in3, json_ctx_t *options);
```

Creates a new event filter with specified options and returns its id (>0) on success or 0 on failure.

arguments:

*in3_t **	**in3**
*json_ctx_t **	**options**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_newBlockFilter

```
in3_ret_t eth_newBlockFilter(in3_t *in3);
```

Creates a new block filter with specified options and returns its id (>0) on success or 0 on failure.

arguments:

*in3_t **	**in3**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_newPendingTransactionFilter

```
in3_ret_t eth_newPendingTransactionFilter(in3_t *in3);
```

Creates a new pending txn filter with specified options and returns its id on success or 0 on failure.

arguments:

*in3_t **	**in3**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_uninstallFilter

```
bool eth_uninstallFilter(in3_t *in3, size_t id);
```

Uninstalls a filter and returns true on success or false on failure.

arguments:

*in3_t **	**in3**
`size_t`	**id**

returns: `bool`

eth_getFilterChanges

```
in3_ret_t eth_getFilterChanges(in3_t *in3, size_t id, bytes32_t **block_hashes, eth_
↪log_t **logs);
```

Sets the logs (for event filter) or blockhashes (for block filter) that match a filter; returns <0 on error, otherwise no.

of block hashes matched (for block filter) or 0 (for log filter)

arguments:

*in3_t **	**in3**
`size_t`	**id**
*bytes32_t ***	**block_hashes**
*eth_log_t ***	**logs**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_getFilterLogs

```
in3_ret_t eth_getFilterLogs(in3_t *in3, size_t id, eth_log_t **logs);
```

Sets the logs (for event filter) or blockhashes (for block filter) that match a filter; returns <0 on error, otherwise no. of block hashes matched (for block filter) or 0 (for log filter)

arguments:

*in3_t **	**in3**
`size_t`	**id**
*eth_log_t ***	**logs**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_chainId

```
uint64_t eth_chainId(in3_t *in3);
```

Returns the currently configured chain id.

arguments:

*in3_t **	**in3**

returns: `uint64_t`

eth_getBlockTransactionCountByHash

```
uint64_t eth_getBlockTransactionCountByHash(in3_t *in3, bytes32_t hash);
```

Returns the number of transactions in a block from a block matching the given block hash.

arguments:

*in3_t **	**in3**
bytes32_t	**hash**

returns: `uint64_t`

eth_getBlockTransactionCountByNumber

```
uint64_t eth_getBlockTransactionCountByNumber(in3_t *in3, eth_blknum_t block);
```

Returns the number of transactions in a block from a block matching the given block number.

arguments:

in3_t *	**in3**
eth_blknum_t	**block**

returns: `uint64_t`

eth_call_fn

```
json_ctx_t* eth_call_fn(in3_t *in3, address_t contract, eth_blknum_t block, char *fn_
↪sig,...);
```

Returns the result of a function_call.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it with json_free()!

arguments:

in3_t *	**in3**
address_t	**contract**
eth_blknum_t	**block**
`char *`	**fn_sig**
`...`	

returns: `json_ctx_t *`

eth_estimate_fn

```
uint64_t eth_estimate_fn(in3_t *in3, address_t contract, eth_blknum_t block, char *fn_
↪sig,...);
```

Returns the result of a function_call.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it with json_free()!

arguments:

in3_t *	**in3**
address_t	**contract**
eth_blknum_t	**block**
`char *`	**fn_sig**
`...`	

returns: `uint64_t`

eth_getTransactionByHash

```
eth_tx_t* eth_getTransactionByHash(in3_t *in3, bytes32_t tx_hash);
```

Returns the information about a transaction requested by transaction hash.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
bytes32_t	**tx_hash**

returns: *eth_tx_t **

eth_getTransactionByBlockHashAndIndex

```
eth_tx_t* eth_getTransactionByBlockHashAndIndex(in3_t *in3, bytes32_t block_hash,␣
↪size_t index);
```

Returns the information about a transaction by block hash and transaction index position.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
bytes32_t	**block_hash**
`size_t`	**index**

returns: *eth_tx_t **

eth_getTransactionByBlockNumberAndIndex

```
eth_tx_t* eth_getTransactionByBlockNumberAndIndex(in3_t *in3, eth_blknum_t block,␣
↪size_t index);
```

Returns the information about a transaction by block number and transaction index position.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
eth_blknum_t	**block**
`size_t`	**index**

returns: *eth_tx_t **

eth_getTransactionCount

```
uint64_t eth_getTransactionCount(in3_t *in3, address_t address, eth_blknum_t block);
```

Returns the number of transactions sent from an address.

arguments:

*in3_t **	**in3**
address_t	**address**
eth_blknum_t	**block**

returns: `uint64_t`

eth_getUncleByBlockNumberAndIndex

```
eth_block_t* eth_getUncleByBlockNumberAndIndex(in3_t *in3, eth_blknum_t block, size_t␣
↪index);
```

Returns information about a uncle of a block by number and uncle index position.

If result is null, check eth_last_error()! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
eth_blknum_t	**block**
`size_t`	**index**

returns: `eth_block_t *`

eth_getUncleCountByBlockHash

```
uint64_t eth_getUncleCountByBlockHash(in3_t *in3, bytes32_t hash);
```

Returns the number of uncles in a block from a block matching the given block hash.

arguments:

*in3_t **	**in3**
bytes32_t	**hash**

returns: `uint64_t`

eth_getUncleCountByBlockNumber

```
uint64_t eth_getUncleCountByBlockNumber(in3_t *in3, eth_blknum_t block);
```

Returns the number of uncles in a block from a block matching the given block number.

arguments:

*in3_t **	**in3**
eth_blknum_t	**block**

returns: `uint64_t`

eth_sendTransaction

```
bytes_t* eth_sendTransaction(in3_t *in3, address_t from, address_t to, OPTIONAL_
↪T(uint64_t) gas, OPTIONAL_T(uint64_t) gas_price, OPTIONAL_T(uint256_t) value,␣
↪OPTIONAL_T(bytes_t) data, OPTIONAL_T(uint64_t) nonce);
```

Creates new message call transaction or a contract creation.

Returns (32 Bytes) - the transaction hash, or the zero hash if the transaction is not yet available. Free result after use with b_free().

arguments:

*in3_t **	**in3**
address_t	**from**
address_t	**to**
OPTIONAL_T(uint64_t)	**gas**
OPTIONAL_T(uint64_t)	**gas_price**
`(,)`	**value**
`(,)`	**data**
OPTIONAL_T(uint64_t)	**nonce**

returns: `bytes_t *`

eth_sendRawTransaction

```
bytes_t* eth_sendRawTransaction(in3_t *in3, bytes_t data);
```

Creates new message call transaction or a contract creation for signed transactions.

Returns (32 Bytes) - the transaction hash, or the zero hash if the transaction is not yet available. Free after use with b_free().

arguments:

*in3_t **	**in3**
bytes_t	**data**

returns: `bytes_t *`

eth_getTransactionReceipt

```
eth_tx_receipt_t* eth_getTransactionReceipt(in3_t *in3, bytes32_t tx_hash);
```

Returns the receipt of a transaction by transaction hash.

Free result after use with eth_tx_receipt_free()

arguments:

*in3_t **	**in3**
bytes32_t	**tx_hash**

returns: `eth_tx_receipt_t *`

eth_wait_for_receipt

```
char* eth_wait_for_receipt(in3_t *in3, bytes32_t tx_hash);
```

Waits for receipt of a transaction requested by transaction hash.

arguments:

*in3_t **	**in3**
bytes32_t	**tx_hash**

returns: `char *`

eth_log_free

```
void eth_log_free(eth_log_t *log);
```

Frees a eth_log_t object.

arguments:

*eth_log_t **	**log**

eth_tx_receipt_free

```
void eth_tx_receipt_free(eth_tx_receipt_t *txr);
```

Frees a eth_tx_receipt_t object.

arguments:

*eth_tx_receipt_t **	**txr**

string_val_to_bytes

```
int string_val_to_bytes(char *val, char *unit, bytes32_t target);
```

reades the string as hex or decimal and converts it into bytes.

the value may also contains a suffix as unit like '1.5eth' which will convert it into wei. the target-pointer must be at least as big as the strlen. The length of the bytes will be returned or a negative value in case of an error.

arguments:

`char *`	**val**
`char *`	**unit**
bytes32_t	**target**

returns: `int`

in3_register_eth_api

```
in3_ret_t in3_register_eth_api(in3_t *c);
```

this function should only be called once and will register the eth-API verifier.

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

9.7.3 ipfs_api.h

IPFS API.

This header-file defines easy to use function, which are preparing the JSON-RPC-Request, which is then executed and verified by the incubed-client.

File: c/src/api/ipfs/ipfs_api.h

ipfs_put

```
char* ipfs_put(in3_t *in3, const bytes_t *content);
```

Returns the IPFS multihash of stored content on success OR NULL on error (check api_last_error()).

Result must be freed by caller.

arguments:

*in3_t **	**in3**
*bytes_tconst , **	**content**

returns: `char *`

ipfs_get

```
bytes_t* ipfs_get(in3_t *in3, const char *multihash);
```

Returns the content associated with specified multihash on success OR NULL on error (check api_last_error()).

Result must be freed by caller.

arguments:

*in3_t **	**in3**
`const char *`	**multihash**

returns: `bytes_t *`

9.7.4 usn_api.h

USN API.

This header-file defines easy to use function, which are verifying USN-Messages.

File: c/src/api/usn/usn_api.h

usn_msg_type_t

The enum type contains the following values:

USN_ACTION	0
USN_REQUEST	1
USN_RESPONSE	2

usn_event_type_t

The enum type contains the following values:

BOOKING_NONE	0
BOOKING_START	1
BOOKING_STOP	2

usn_booking_handler

```
typedef int(* usn_booking_handler) (usn_event_t *)
```

returns: `int(*`

usn_verify_message

```
usn_msg_result_t usn_verify_message(usn_device_conf_t *conf, char *message);
```

arguments:

*usn_device_conf_t **	**conf**
`char *`	**message**

returns: *usn_msg_result_t*

usn_register_device

```
in3_ret_t usn_register_device(usn_device_conf_t *conf, char *url);
```

arguments:

*usn_device_conf_t **	**conf**
`char *`	**url**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

usn_parse_url

```
usn_url_t usn_parse_url(char *url);
```

arguments:

char *	**url**

returns: *usn_url_t*

usn_update_state

```
unsigned int usn_update_state(usn_device_conf_t *conf, unsigned int wait_time);
```

arguments:

*usn_device_conf_t **	**conf**
unsigned int	**wait_time**

returns: unsigned int

usn_update_bookings

```
in3_ret_t usn_update_bookings(usn_device_conf_t *conf);
```

arguments:

*usn_device_conf_t **	**conf**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

usn_remove_old_bookings

```
void usn_remove_old_bookings(usn_device_conf_t *conf);
```

arguments:

*usn_device_conf_t **	**conf**

usn_get_next_event

```
usn_event_t usn_get_next_event(usn_device_conf_t *conf);
```

arguments:

*usn_device_conf_t **	**conf**

returns: `usn_event_t`

usn_rent

```
in3_ret_t usn_rent(in3_t *c, address_t contract, address_t token, char *url, uint32_t␣
↪seconds, bytes32_t tx_hash);
```

arguments:

*in3_t **	**c**
address_t	**contract**
address_t	**token**
`char *`	**url**
`uint32_t`	**seconds**
bytes32_t	**tx_hash**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

usn_return

```
in3_ret_t usn_return(in3_t *c, address_t contract, char *url, bytes32_t tx_hash);
```

arguments:

*in3_t **	**c**
address_t	**contract**
`char *`	**url**
bytes32_t	**tx_hash**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

usn_price

```
in3_ret_t usn_price(in3_t *c, address_t contract, address_t token, char *url, uint32_
↪t seconds, address_t controller, bytes32_t price);
```

arguments:

*in3_t **	**c**
address_t	**contract**
address_t	**token**
`char *`	**url**
`uint32_t`	**seconds**
address_t	**controller**
bytes32_t	**price**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`*)*

9.7.5 api_utils.h

Ethereum API utils.

This header-file helper utils for use with API modules.

File: c/src/api/utils/api_utils.h

set_error_fn

function to set error.

Will only be called internally. default implementation is NOT MT safe!

```
typedef void(* set_error_fn) (int err, const char *msg)
```

get_error_fn

function to get last error message.

default implementation is NOT MT safe!

```
typedef char*(* get_error_fn) (void)
```

returns: `char *(*`

as_double

```
long double as_double(uint256_t d);
```

Converts a uint256_t in a long double.

Important: since a long double stores max 16 byte, there is no guarantee to have the full precision.

Converts a uint256_t in a long double.

arguments:

uint256_t	**d**

returns: `long double`

as_long

```
uint64_t as_long(uint256_t d);
```

Converts a uint256_t in a long .

Important: since a long double stores 8 byte, this will only use the last 8 byte of the value.

Converts a uint256_t in a long .

arguments:

uint256_t	**d**

returns: `uint64_t`

to_uint256

```
uint256_t to_uint256(uint64_t value);
```

Converts a uint64_t into its uint256_t representation.

arguments:

`uint64_t`	**value**

returns: *`uint256_t`*

decrypt_key

```
in3_ret_t decrypt_key(d_token_t *key_data, char *password, bytes32_t dst);
```

Decrypts the private key from a json keystore file using PBKDF2 or SCRYPT (if enabled)

arguments:

*d_token_t **	**key_data**
`char *`	**password**
bytes32_t	**dst**

returns: *`in3_ret_t`* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

to_checksum

```
in3_ret_t to_checksum(address_t adr, chain_id_t chain_id, char out[43]);
```

converts the given address to a checksum address.

If chain_id is passed, it will use the EIP1191 to include it as well.

arguments:

address_t	**adr**
chain_id_t	**chain_id**
`char`	**out**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

api_set_error_fn

```
void api_set_error_fn(set_error_fn fn);
```

arguments:

set_error_fn	**fn**

api_get_error_fn

```
void api_get_error_fn(get_error_fn fn);
```

arguments:

get_error_fn	**fn**

api_last_error

```
char* api_last_error();
```

returns current error or null if all is ok

returns: `char *`

9.8 Module core

9.8.1 client.h

this file defines the incubed configuration struct and it registration.

File: c/src/core/client/client.h

IN3_PROTO_VER

the protocol version used when sending requests from the this client

```
#define IN3_PROTO_VER "2.1.0"
```

CHAIN_ID_MULTICHAIN

chain_id working with all known chains

```
#define CHAIN_ID_MULTICHAIN 0x0
```

CHAIN_ID_MAINNET

chain_id for mainnet

```
#define CHAIN_ID_MAINNET 0x01
```

CHAIN_ID_KOVAN

chain_id for kovan

```
#define CHAIN_ID_KOVAN 0x2a
```

CHAIN_ID_TOBALABA

chain_id for tobalaba

```
#define CHAIN_ID_TOBALABA 0x44d
```

CHAIN_ID_GOERLI

chain_id for goerlii

```
#define CHAIN_ID_GOERLI 0x5
```

CHAIN_ID_EVAN

chain_id for evan

```
#define CHAIN_ID_EVAN 0x4b1
```

CHAIN_ID_EWC

chain_id for ewc

```
#define CHAIN_ID_EWC 0xf6
```

CHAIN_ID_IPFS

chain_id for ipfs

```
#define CHAIN_ID_IPFS 0x7d0
```

CHAIN_ID_BTC

chain_id for btc

```
#define CHAIN_ID_BTC 0x99
```

CHAIN_ID_LOCAL

chain_id for local chain

```
#define CHAIN_ID_LOCAL 0x11
```

DEF_REPL_LATEST_BLK

default replace_latest_block

```
#define DEF_REPL_LATEST_BLK 6
```

in3_node_props_init (np)

Initializer for in3_node_props_t.

```
#define in3_node_props_init (np) *(np) = 0
```

PLGN_ACT_TRANSPORT

```
#define PLGN_ACT_TRANSPORT (PLGN_ACT_TRANSPORT_SEND | PLGN_ACT_TRANSPORT_RECEIVE |␣
↪PLGN_ACT_TRANSPORT_CLEAN)
```

PLGN_ACT_CACHE

```
#define PLGN_ACT_CACHE (PLGN_ACT_CACHE_SET | PLGN_ACT_CACHE_GET | PLGN_ACT_CACHE_
↪CLEAR)
```

in3_for_chain (chain_id)

creates a new Incubes configuration for a specified chain and returns the pointer.

when creating the client only the one chain will be configured. (saves memory). but if you pass `CHAIN_ID_MULTICHAIN` as argument all known chains will be configured allowing you to switch between chains within the same client or configuring your own chain.

you need to free this instance with `in3_free` after use!

Before using the client you still need to set the tramsport and optional the storage handlers:

- example of initialization: , ** This Method is depricated. you should use `in3_for_chain` instead.**

```
// register verifiers
in3_register_eth_full();

// create new client
in3_t* client = in3_for_chain(CHAIN_ID_MAINNET);

// configure transport
client->transport    = send_curl;

// configure storage
in3_set_storage_handler(c, storage_get_item, storage_set_item, storage_clear, NULL);

// ready to use ...
```

```
#define in3_for_chain (chain_id) in3_for_chain_default(chain_id)
```

assert_in3 (c)

```
#define assert_in3 (c) assert(c);                                     \
  assert(c->chain_id);                         \
  assert(c->plugins);                          \
  assert(c->chains);                           \
  assert(c->request_count > 0);                \
  assert(c->chains_length > 0);                \
  assert(c->chains_length < 10);               \
  assert(c->max_attempts > 0);                 \
  assert(c->proof >= 0 && c->proof <= PROOF_FULL); \
  assert(c->proof >= 0 && c->proof <= PROOF_FULL);
```

in3_chain_type_t

the type of the chain.

for incubed a chain can be any distributed network or database with incubed support. Depending on this chain-type the previously registered verifyer will be choosen and used.

The enum type contains the following values:

CHAIN_ETH	0	Ethereum chain.
CHAIN_SUBSTRATE	1	substrate chain
CHAIN_IPFS	2	ipfs verifiaction
CHAIN_BTC	3	Bitcoin chain.
CHAIN_EOS	4	EOS chain.
CHAIN_IOTA	5	IOTA chain.
CHAIN_GENERIC	6	other chains

in3_proof_t

the type of proof.

Depending on the proof-type different levels of proof will be requested from the node.

The enum type contains the following values:

PROOF_NONE	0	No Verification.
PROOF_STANDARD	1	Standard Verification of the important properties.
PROOF_FULL	2	All field will be validated including uncles.

in3_node_props_type_t

The enum type contains the following values:

NODE_PROP_PROOF	0x1	filter out nodes which are providing no proof
NODE_PROP_MULTICHAIN	0x2	filter out nodes other then which have capability of the same RPC endpoint may also accept requests for different chains
NODE_PROP_ARCHIVE	0x4	filter out non-archive supporting nodes
NODE_PROP_HTTP	0x8	filter out non-http nodes
NODE_PROP_BINARY	0x10	filter out nodes that don't support binary encoding
NODE_PROP_ONION	0x20	filter out non-onion nodes
NODE_PROP_SIGNER	0x40	filter out non-signer nodes
NODE_PROP_DATA	0x80	filter out non-data provider nodes
NODE_PROP_STATS	0x100	filter out nodes that do not provide stats
NODE_PROP_MIN_BLOCK_HEIGHT	0x400	filter out nodes that will sign blocks with lower min block height than specified

in3_flags_type_t

a list of flags definiing the behavior of the incubed client.

They should be used as bitmask for the flags-property.

The enum type contains the following values:

FLAGS_KEEP_IN3	0x1	the in3-section with the proof will also returned
FLAGS_AUTO_UPDATE_LIST	0x2	the nodelist will be automaticly updated if the last_block is newer
FLAGS_INCLUDE_CODE	0x4	the code is included when sending eth_call-requests
FLAGS_BINARY	0x8	the client will use binary format
FLAGS_HTTP	0x10	the client will try to use http instead of https
FLAGS_STATS	0x20	nodes will keep track of the stats (default=true)
FLAGS_NODE_LIST_NO_SIG	0x40	nodelist update request will not automatically ask for signatures and proof
FLAGS_BOOT_WEIGHTS	0x80	if true the client will initialize the first weights from the nodelist given by the nodelist.

in3_node_attr_type_t

a list of node attributes (mostly used internally)

The enum type contains the following values:

ATTR_WHITELISTED	1	indicates if node exists in whiteList
ATTR_BOOT_NODE	2	used to avoid filtering manually added nodes before first nodeList update

in3_filter_type_t

Filter type used internally when managing filters.

The enum type contains the following values:

FILTER_EVENT	0	Event filter.
FILTER_BLOCK	1	Block filter.
FILTER_PENDING	2	Pending filter (Unsupported)

in3_plugin_act_t

plugin action list

The enum type contains the following values:

PLGN_ACT_INIT	0x1	initialize plugin - use for allocating/setting-up internal resources
PLGN_ACT_TERM	0x2	terminate plugin - use for releasing internal resources and cleanup.
PLGN_ACT_TRANSPORT_SEND	0x4	sends out a request - the transport plugin will receive a request_t as plgn_ctx, it may set a cptr which will be passed back when fetching more resonses.
PLGN_ACT_TRANSPORT_RECEIVE	0x8	fetch next response - the transport plugin will receive a request_t as plgn_ctx, which contains a cptr if set previously
PLGN_ACT_TRANSPORT_CLEAN	0x10	freeup transport resources - the transport plugin will receive a request_t as plgn_ctx if the cptr was set.
PLGN_ACT_SIGN_ACCOUNT	0x20	returns the default account of the signer
PLGN_ACT_SIGN_PREPARE	0x40	allowes a wallet to manipulate the payload before signing - the plgn_ctx will be in3_sign_ctx_t. This way a tx can be send through a multisig
PLGN_ACT_SIGN	0x80	signs the payload - the plgn_ctx will be in3_sign_ctx_t.
PLGN_ACT_RPC_HANDLE	0x100	a plugin may respond to a rpc-request directly (without sending it to the node).
PLGN_ACT_RPC_VERIFY	0x200	verifies the response. the plgn_ctx will be a in3_vctx_t holding all data
PLGN_ACT_CACHE_SET	0x400	stores data to be reused later - the plgn_ctx will be a in3_cache_ctx_t containing the data
PLGN_ACT_CACHE_GET	0x800	reads data to be previously stored - the plgn_ctx will be a in3_cache_ctx_t containing the key. if the data was found the data-property needs to be set.
PLGN_ACT_CACHE_CLEAR	0x1000	clears alls stored data - plgn_ctx will be NULL
PLGN_ACT_CONFIG_SET	0x2000	gets a config-token and reads data from it
PLGN_ACT_CONFIG_GET	0x4000	gets a stringbuilder and adds all config to it.
PLGN_ACT_PAY_PREPARE	0x8000	prerpares a payment
PLGN_ACT_PAY_FOLLOWUP	0x10000	called after a requeest to update stats.
PLGN_ACT_PAY_HANDLE	0x20000	handles the payment
PLGN_ACT_PAY_SIGN_REQ	0x40000	signs a request
PLGN_ACT_NL_PICK_DATA	0x80000	picks the data nodes
PLGN_ACT_NL_PICK_SIGNER	0x100000	picks the signer nodes
PLGN_ACT_NL_PICK_FOLLOWUP	0x200000	called after receiving a response in order to decide whether a update is needed.
PLGN_ACT_LOG_ERROR	0x400000	report an error

chain_id_t

type for a chain_id.

```
typedef uint32_t chain_id_t
```

in3_node_props_t

Node capabilities.

```
typedef uint64_t in3_node_props_t
```

in3_node_attr_t

```
typedef uint8_t in3_node_attr_t
```

in3_node_t

incubed node-configuration.

These information are read from the Registry contract and stored in this struct representing a server or node.

The stuct contains following fields:

address_t	**address**	address of the server
`uint64_t`	**deposit**	the deposit stored in the registry contract, which this would lose if it sends a wrong blockhash
`uint_fast16_t`	**index**	index within the nodelist, also used in the contract as key
`uint_fast16_t`	**capacity**	the maximal capacity able to handle
in3_node_props_t	**props**	used to identify the capabilities of the node. See in3_node_props_type_t in nodelist.h
`char *`	**url**	the url of the node
`uint_fast8_t`	**attrs**	bitmask of internal attributes

in3_node_weight_t

Weight or reputation of a node.

Based on the past performance of the node a weight is calculated given faster nodes a higher weight and chance when selecting the next node from the nodelist. These weights will also be stored in the cache (if available)

The stuct contains following fields:

`uint32_t`	**response_count**	counter for responses
`uint32_t`	**total_response_time**	total of all response times
`uint64_t`	**blacklisted_until**	if >0 this node is blacklisted until k. k is a unix timestamp

in3_whitelist_t

defines a whitelist structure used for the nodelist.

The stuct contains following fields:

`bool`	**needs_update**	if true the nodelist should be updated and will trigger a *in3_nodeList*-request before the next request is send.
`uint64_t`	**last_block**	last blocknumber the whiteList was updated, which is used to detect changed in the whitelist
address_t	**contract**	address of whiteList contract. If specified, whiteList is always auto-updated and manual whiteList is overridden
bytes_t	**addresses**	serialized list of node addresses that constitute the whiteList

in3_verified_hash_t

represents a blockhash which was previously verified

The stuct contains following fields:

`uint64_t`	**block_number**	the number of the block
bytes32_t	**hash**	the blockhash

in3_chain_t

Chain definition inside incubed.

for incubed a chain can be any distributed network or database with incubed support.

The stuct contains following fields:

bool	**dirty**	indicates whether the nodelist has been modified after last read from cache
uint8_t	**version**	version of the chain
unsigned int	**nodelist_length**	number of nodes in the nodeList
uint16_t	**avg_block_time**	average block time (seconds) for this chain (calculated internally)
chain_id_t	**chain_id**	chain_id, which could be a free or based on the public ethereum networkId
in3_chain_type_t	**type**	chaintype
uint64_t	**last_block**	last blocknumber the nodeList was updated, which is used to detect changed in the nodelist
in3_node_t *	**nodelist**	array of nodes
in3_node_weight_t *	**weights**	stats and weights recorded for each node
bytes_t **	**init_addresses**	array of addresses of nodes that should always part of the nodeList
bytes_t *	**contract**	the address of the registry contract
bytes32_t	**registry_id**	the identifier of the registry
in3_verified_hash_t *	**verified_hashes**	contains the list of already verified blockhashes
in3_whitelist_t *	**whitelist**	if set the whitelist of the addresses.
uint64_t	**exp_last_block**	the last_block when the nodelist last changed reported by this node
uint64_t	**timestamp**	approx. time when nodelist must be updated (i.e. when reported last_block will be considered final)
address_t	**node**	node that reported the last_block which necessitated a nodeList update
struct in3_chain::@7 *	**nodelist_upd8_params**	

in3_pay_prepare

payment prepearation function.

allows the payment to handle things before the request will be send.

```
typedef in3_ret_t(* in3_pay_prepare) (struct in3_ctx *ctx, void *cptr)
```

returns: `in3_ret_t (*` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`*)*

in3_pay_follow_up

called after receiving a parseable response with a in3-section.

```
typedef in3_ret_t(* in3_pay_follow_up) (struct in3_ctx *ctx, void *node, d_token_t *in3, d_token_t *error, void *cptr)
```

returns: `in3_ret_t (*` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`*)*

in3_pay_free

free function for the custom pointer.

```
typedef void(* in3_pay_free) (void *cptr)
```

in3_pay_handle_request

handles the request.

this function is called when the in3-section of payload of the request is built and allows the handler to add properties.

```
typedef in3_ret_t(* in3_pay_handle_request) (struct in3_ctx *ctx, sb_t *sb, void
↪*cptr)
```

returns: `in3_ret_t (*` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_pay_t

the payment handler.

if a payment handler is set it will be used when generating the request.

The stuct contains following fields:

in3_pay_prepare	**prepare**	payment prepearation function.
in3_pay_follow_up	**follow_up**	payment function to be called after the request.
in3_pay_handle_request	**handle_request**	this function is called when the in3-section of payload of the request is built and allows the handler to add properties.
in3_pay_free	**free**	frees the custom pointer (cptr).
`void *`	**cptr**	custom object whill will be passed to functions

in3_t

Incubed Configuration.

This struct holds the configuration and also point to internal resources such as filters or chain configs.

The stuct contains following fields:

`uint8_t`	**request_count**	the number of request send when getting a first answer
`uint8_t`	**signature_count**	the number of signatures used to proof the blockhash.
`uint8_t`	**replace_latest_block**	if specified, the blocknumber *latest* will be replaced by blockNumber- specified value
`uint_fast8_t`	**flags**	a bit mask with flags defining the behavior of the incubed client. See the FLAG…-defines
`uint16_t`	**node_limit**	the limit of nodes to store in the client.
`uint16_t`	**finality**	the number of signatures in percent required for the request
`uint16_t`	**chains_length**	number of configured chains
`uint_fast16_t`	**max_attempts**	the max number of attempts before giving up
`uint_fast16_t`	**max_verified_hashes**	max number of verified hashes to cache (actual number may temporarily exceed this value due to pending requests)
`uint_fast16_t`	**alloc_verified_hashes**	number of currently allocated verified hashes
`uint_fast16_t`	**pending**	number of pending requests created with this instance
`uint32_t`	**cache_timeout**	number of seconds requests can be cached.
`uint32_t`	**timeout**	specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection.
chain_id_t	**chain_id**	servers to filter for the given chain. The chain-id based on EIP-155.
`in3_plugin_supp`	**plugin_acts**	bitmask of supported actions of all plugins registered with this client
in3_proof_t	**proof**	the type of proof used
`uint64_t`	**min_deposit**	min stake of the server. Only nodes owning at least this amount will be chosen.
in3_node_props_t	**node_props**	used to identify the capabilities of the node.
*in3_chain_t **	**chains**	chain spec and nodeList definitions
*in3_filter_handler_t **	**filters**	filter handler
*in3_plugin_t **	**plugins**	list of registered plugins
`uint32_t`	**id_count**	counter for use as JSON RPC id - incremented for every request
`void *`	**internal**	pointer to internal data

in3_filter_t

The stuct contains following fields:

`bool`	**is_first_usage**	if true the filter was not used previously
in3_filter_type_t	**type**	filter type: (event, block or pending)
`uint64_t`	**last_block**	block no. when filter was created OR eth_getFilterChanges was called
`char *`	**options**	associated filter options
`void(*`	**release**	method to release owned resources

in3_plugin_t

plugin interface definition

The stuct contains following fields:

`in3_plugin_supp_acts_t`	**acts**	bitmask of supported actions this plugin can handle
`void *`	**data**	opaque pointer to plugin data
in3_plugin_act_fn	**action_fn**	plugin action handler
*in3_plugin_t **	**next**	pointer to next plugin in list

in3_plugin_act_fn

plugin action handler

Implementations of this function must strictly follow the below pattern for return values -

- IN3_OK - successfully handled specified action
- IN3_WAITING - handling specified action, but waiting for more information
- IN3_EIGNORE - could handle specified action, but chose to ignore it so maybe another handler could handle it
- Other errors - handled but failed

```
typedef in3_ret_t(* in3_plugin_act_fn) (void *plugin_data, in3_plugin_act_t action,␣
↪void *plugin_ctx)
```

returns: `in3_ret_t(*` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_plugin_supp_acts_t

```
typedef uint32_t in3_plugin_supp_acts_t
```

in3_filter_handler_t

Handler which is added to client config in order to handle filter.

The stuct contains following fields:

*in3_filter_t ***	**array**	
`size_t`	**count**	array of filters

plgn_register

a register-function for a plugion.

```
typedef in3_ret_t(* plgn_register) (in3_t *c)
```

returns: `in3_ret_t(*` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_node_props_set

```
NONULL void in3_node_props_set(in3_node_props_t *node_props, in3_node_props_type_t␣
↪type, uint8_t value);
```

setter method for interacting with in3_node_props_t.

arguments:

*in3_node_props_t **	**node_props**	pointer to the properties to change
in3_node_props_type_t	**type**	key or type of the property
`uint8_t`	**value**	value to set

returns: `NONULL void`

in3_node_props_get

```
static uint32_t in3_node_props_get(in3_node_props_t np, in3_node_props_type_t t);
```

returns the value of the specified propertytype.

< the value to extract

arguments:

in3_node_props_t	**np**	property to read from
in3_node_props_type_t	**t**	

returns: `uint32_t` : value as a number

in3_node_props_matches

```
static bool in3_node_props_matches(in3_node_props_t np, in3_node_props_type_t t);
```

checkes if the given type is set in the properties

< the value to extract

arguments:

in3_node_props_t	**np**	property to read from
in3_node_props_type_t	**t**	

returns: `bool` : true if set

in3_new

```
in3_t* in3_new() __attribute__((deprecated("use in3_for_chain(CHAIN_ID_MULTICHAIN)
↪")));
```

creates a new Incubes configuration and returns the pointer.

This Method is depricated. you should use `in3_for_chain(CHAIN_ID_MULTICHAIN)` instead.

you need to free this instance with `in3_free` after use!

Before using the client you still need to set the tramsport and optional the storage handlers:

- example of initialization:

```
// register verifiers
in3_register_eth_full();

// create new client
in3_t* client = in3_new();

// configure transport
client->transport    = send_curl;

// configure storage
in3_set_storage_handler(c, storage_get_item, storage_set_item, storage_clear, NULL);

// ready to use ...
```

returns: *in3_t* * : the incubed instance.

in3_for_chain_default

```
in3_t* in3_for_chain_default(chain_id_t chain_id);
```

arguments:

chain_id_t	**chain_id**	the chain_id (see **CHAIN_ID_**. . . constants).

returns: *in3_t* *

in3_client_rpc

```
NONULL in3_ret_t in3_client_rpc(in3_t *c, const char *method, const char *params,␣
↪char **result, char **error);
```

sends a request and stores the result in the provided buffer

arguments:

in3_t *	**c**	the pointer to the incubed client config.
`const char *`	**method**	the name of the rpc-funcgtion to call.
`const char *`	**params**	docs for input parameter v.
`char **`	**result**	pointer to string which will be set if the request was successfull. This will hold the result as json-rpc-string. (make sure you free this after use!)
`char **`	**error**	pointer to a string containg the error-message. (make sure you free it after use!)

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_rpc_raw

```
NONULL in3_ret_t in3_client_rpc_raw(in3_t *c, const char *request, char **result,␣
↪char **error);
```

sends a request and stores the result in the provided buffer, this method will always return the first, so bulk-requests are not saupported.

arguments:

*in3_t **	**c**	the pointer to the incubed client config.
`const char *`	**request**	the rpc request including method and params.
`char **`	**result**	pointer to string which will be set if the request was successfull. This will hold the result as json-rpc-string. (make sure you free this after use!)
`char **`	**error**	pointer to a string containg the error-message. (make sure you free it after use!)

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_client_exec_req

```
NONULL char* in3_client_exec_req(in3_t *c, char *req);
```

executes a request and returns result as string.

in case of an error, the error-property of the result will be set. This fuinction also supports sending bulk-requests, but you can not mix internal and external calls, since bulk means all requests will be send to picked nodes. The resulting string must be free by the the caller of this function!

arguments:

*in3_t **	**c**	the pointer to the incubed client config.
`char *`	**req**	the request as rpc.

returns: `NONULL char *`

in3_client_register_chain

```
in3_ret_t in3_client_register_chain(in3_t *client, chain_id_t chain_id, in3_chain_
↪type_t type, address_t contract, bytes32_t registry_id, uint8_t version, address_t␣
↪wl_contract);
```

registers a new chain or replaces a existing (but keeps the nodelist)

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
chain_id_t	**chain_id**	the chain id.
in3_chain_type_t	**type**	the verification type of the chain.
address_t	**contract**	contract of the registry.
bytes32_t	**registry_id**	the identifier of the registry.
uint8_t	**version**	the chain version.
address_t	**wl_contract**	contract of whiteList.

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_add_node

```
NONULL in3_ret_t in3_client_add_node(in3_t *client, chain_id_t chain_id, char *url,␣
↪in3_node_props_t props, address_t address);
```

adds a node to a chain ore updates a existing node

[in] public address of the signer.

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
chain_id_t	**chain_id**	the chain id.
char *	**url**	url of the nodes.
in3_node_props_t	**props**	properties of the node.
address_t	**address**	

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_remove_node

```
NONULL in3_ret_t in3_client_remove_node(in3_t *client, chain_id_t chain_id, address_t␣
↪address);
```

removes a node from a nodelist

[in] public address of the signer.

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
chain_id_t	**chain_id**	the chain id.
address_t	**address**	

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_clear_nodes

```
NONULL in3_ret_t in3_client_clear_nodes(in3_t *client, chain_id_t chain_id);
```

removes all nodes from the nodelist

[in] the chain id.

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
chain_id_t	**chain_id**	

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_free

```
NONULL void in3_free(in3_t *a);
```

frees the references of the client

arguments:

*in3_t **	**a**	the pointer to the incubed client config to free.

returns: `NONULL void`

in3_cache_init

```
NONULL in3_ret_t in3_cache_init(in3_t *c);
```

inits the cache.

this will try to read the nodelist from cache.

inits the cache.

arguments:

*in3_t **	**c**	the incubed client

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_get_chain

```
NONULL in3_chain_t* in3_get_chain(const in3_t *c);
```

returns the chain-config for the current chain_id.

arguments:

*in3_tconst , **	**c**	the incubed client

returns: `in3_chain_tNONULL , *`

in3_find_chain

```
NONULL in3_chain_t* in3_find_chain(const in3_t *c, chain_id_t chain_id);
```

finds the chain-config for the given chain_id.

My return NULL if not found.

arguments:

*in3_tconst , **	**c**	the incubed client
chain_id_t	**chain_id**	chain_id

returns: `in3_chain_tNONULL , *`

in3_configure

```
NONULL char* in3_configure(in3_t *c, const char *config);
```

configures the clent based on a json-config.

For details about the structure of ther config see https://in3.readthedocs.io/en/develop/api-ts.html#type-in3config Returns NULL on success, and error string on failure (to be freed by caller) - in which case the client state is undefined

arguments:

*in3_t **	**c**	the incubed client
`const char *`	**config**	JSON-string with the configuration to set.

returns: `NONULL char *`

in3_get_config

```
NONULL char* in3_get_config(in3_t *c);
```

gets the current config as json.

For details about the structure of ther config see https://in3.readthedocs.io/en/develop/api-ts.html#type-in3config

arguments:

*in3_t **	**c**	the incubed client

returns: `NONULL char *`

9.8.2 context.h

Request Context. This is used for each request holding request and response-pointers but also controls the execution process.

File: c/src/core/client/context.h

ctx_type

type of the request context,

The enum type contains the following values:

CT_RPC	0	a json-rpc request, which needs to be send to a incubed node
CT_SIGN	1	a sign request

state

The current state of the context.

you can check this state after each execute-call.

The enum type contains the following values:

CTX_SUCCESS	0	The ctx has a verified result.
CTX_WAITING_TO_SEND	1	the request has not been sent yet
CTX_WAITING_FOR_RESPONSE	2	the request is sent but not all of the response are set ()
CTX_ERROR	-1	the request has a error

ctx_type_t

type of the request context,

The enum type contains the following values:

CT_RPC	0	a json-rpc request, which needs to be send to a incubed node
CT_SIGN	1	a sign request

node_match_t

the weight of a certain node as linked list.

This will be used when picking the nodes to send the request to. A linked list of these structs desribe the result.

The stuct contains following fields:

`unsigned int`	**index**	index of the node in the nodelist
`bool`	**blocked**	if true this node has been blocked for sending wrong responses
`uint32_t`	**s**	The starting value.
`uint32_t`	**w**	weight value
weightstruct , *	**next**	next in the linkedlist or NULL if this is the last element

in3_response_t

response-object.

if the error has a length>0 the response will be rejected

The stuct contains following fields:

`uint32_t`	**time**	measured time (in ms) which will be used for ajusting the weights
in3_ret_t	**state**	the state of the response
sb_t	**data**	a stringbuilder to add the result

in3_ctx_t

The Request config.

This is generated for each request and represents the current state. it holds the state until the request is finished and must be freed afterwards.

The stuct contains following fields:

`uint_fast8_t`	**signers_length**	number or addresses
`uint16_t`	**len**	the number of requests
`uint_fast16_t`	**attempt**	the number of attempts
ctx_type_t	**type**	the type of the request
in3_ret_t	**verification_state**	state of the verification
`char *`	**error**	in case of an error this will hold the message, if not it points to *NULL*
json_ctx_t *	**request_context**	the result of the json-parser for the request.
json_ctx_t *	**response_context**	the result of the json-parser for the response.
d_token_t **	**requests**	references to the tokens representring the requests
d_token_t **	**responses**	references to the tokens representring the parsed responses
in3_response_t *	**raw_response**	the raw response-data, which should be verified.
`uint8_t *`	**signers**	the addresses of servers requested to sign the blockhash
node_match_t *	**nodes**	selected nodes to process the request, which are stored as linked list.
cache_entry_t *	**cache**	optional cache-entries. These entries will be freed when cleaning up the context.
in3_ctxstruct, *	**required**	pointer to the next required context. if not NULL the data from this context need get finished first, before being able to resume this context.
in3_t *	**client**	reference to the client
`uint32_t`	**id**	JSON RPC id of request at index 0.

in3_ctx_state_t

The current state of the context.

you can check this state after each execute-call.

The enum type contains the following values:

CTX_SUCCESS	0	The ctx has a verified result.
CTX_WAITING_TO_SEND	1	the request has not been sent yet
CTX_WAITING_FOR_RESPONSE	2	the request is sent but not all of the response are set ()
CTX_ERROR	-1	the request has a error

ctx_new

```
NONULL in3_ctx_t* ctx_new(in3_t *client, const char *req_data);
```

creates a new context.

the request data will be parsed and represented in the context. calling this function will only parse the request data, but not send anything yet.

Important: the req_data will not be cloned but used during the execution. The caller of the this function is also responsible for freeing this string afterwards.

arguments:

*in3_t **	**client**	the client-config.
`const char *`	**req_data**	the rpc-request as json string.

returns: *in3_ctx_tNONULL , **

in3_send_ctx

```
NONULL in3_ret_t in3_send_ctx(in3_ctx_t *ctx);
```

sends a previously created context to nodes and verifies it.

The execution happens within the same thread, thich mean it will be blocked until the response ha beedn received and verified. In order to handle calls asynchronously, you need to call the `in3_ctx_execute` function and provide the data as needed.

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_ctx_last_waiting

```
NONULL in3_ctx_t* in3_ctx_last_waiting(in3_ctx_t *ctx);
```

finds the last waiting request-context.

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *in3_ctx_tNONULL , **

in3_ctx_exec_state

```
NONULL in3_ctx_state_t in3_ctx_exec_state(in3_ctx_t *ctx);
```

executes the context and returns its state.

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *in3_ctx_state_tNONULL*

in3_ctx_execute

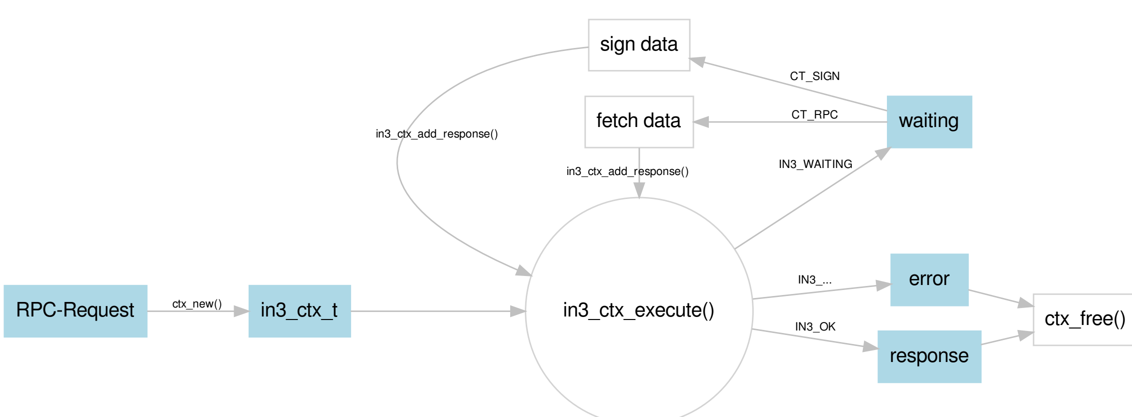
```
NONULL in3_ret_t in3_ctx_execute(in3_ctx_t *ctx);
```

execute the context, but stops whenever data are required.

This function should be used in order to call data in a asyncronous way, since this function will not use the transport-function to actually send it.

The caller is responsible for delivering the required responses. After calling you need to check the return-value:

- IN3_WAITING : provide the required data and then call in3_ctx_execute again.
- IN3_OK : success, we have a result.
- any other status = error



Here is a example how to use this function:

```
in3_ret_t in3_send_ctx(in3_ctx_t* ctx) {
 in3_ret_t ret;
 // execute the context and store the return value.
 // if the return value is 0 == IN3_OK, it was successful and we return,
 // if not, we keep on executing
 while ((ret = in3_ctx_execute(ctx))) {
   // error we stop here, because this means we got an error
   if (ret != IN3_WAITING) return ret;

   // handle subcontexts first, if they have not been finished
   while (ctx->required && in3_ctx_state(ctx->required) != CTX_SUCCESS) {
     // exxecute them, and return the status if still waiting or error
     if ((ret = in3_send_ctx(ctx->required))) return ret;

     // recheck in order to prepare the request.
     // if it is not waiting, then it we cannot do much, becaus it will an error or␣
↪successfull.
     if ((ret = in3_ctx_execute(ctx)) != IN3_WAITING) return ret;
   }

   // only if there is no response yet...
   if (!ctx->raw_response) {

     // what kind of request do we need to provide?
     switch (ctx->type) {

       // RPC-request to send to the nodes
       case CT_RPC: {

           // build the request
           in3_request_t* request = in3_create_request(ctx);

           // here we use the transport, but you can also try to fetch the data in␣
↪any other way.
           ctx->client->transport(request);

           // clean up
           request_free(request);
           break;
       }

       // this is a request to sign a transaction
       case CT_SIGN: {
           // read the data to sign from the request
           d_token_t* params = d_get(ctx->requests[0], K_PARAMS);
           // the data to sign
           bytes_t    data    = d_to_bytes(d_get_at(params, 0));
           // the account to sign with
           bytes_t    from    = d_to_bytes(d_get_at(params, 1));

           // prepare the response
           ctx->raw_response = _malloc(sizeof(in3_response_t));
           sb_init(&ctx->raw_response[0].error);
           sb_init(&ctx->raw_response[0].result);

           // data for the signature
```

(continues on next page)

(continued from previous page)

```
            uint8_t sig[65];
            // use the signer to create the signature
            ret = ctx->client->signer->sign(ctx, SIGN_EC_HASH, data, from, sig);
            // if it fails we report this as error
            if (ret < 0) return ctx_set_error(ctx, ctx->raw_response->error.data,␣
↪ret);
            // otherwise we simply add the raw 65 bytes to the response.
            sb_add_range(&ctx->raw_response->result, (char*) sig, 0, 65);
        }
      }
    }
  }
  // done...
  return ret;
}
```

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_ctx_state

```
NONULL in3_ctx_state_t in3_ctx_state(in3_ctx_t *ctx);
```

returns the current state of the context.

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *in3_ctx_state_tNONULL*

ctx_get_error_data

```
char* ctx_get_error_data(in3_ctx_t *ctx);
```

returns the error of the context.

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: `char *`

ctx_get_response_data

```
char* ctx_get_response_data(in3_ctx_t *ctx);
```

returns json response for that context

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: `char *`

ctx_get_type

```
NONULL ctx_type_t ctx_get_type(in3_ctx_t *ctx);
```

returns the type of the request

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: *ctx_type_tNONULL*

ctx_free

```
NONULL void ctx_free(in3_ctx_t *ctx);
```

frees all resources allocated during the request.

But this will not free the request string passed when creating the context!

arguments:

*in3_ctx_t **	**ctx**	the request context.

returns: `NONULL void`

ctx_add_required

```
NONULL in3_ret_t ctx_add_required(in3_ctx_t *parent, in3_ctx_t *ctx);
```

adds a new context as a requirment.

Whenever a verifier needs more data and wants to send a request, we should create the request and add it as dependency and stop.

If the function is called again, we need to search and see if the required status is now useable.

Here is an example of how to use it:

```
in3_ret_t get_from_nodes(in3_ctx_t* parent, char* method, char* params, bytes_t* dst)
↪{
  // check if the method is already existing
  in3_ctx_t* ctx = ctx_find_required(parent, method);
  if (ctx) {
    // found one - so we check if it is useable.
    switch (in3_ctx_state(ctx)) {
      // in case of an error, we report it back to the parent context
      case CTX_ERROR:
        return ctx_set_error(parent, ctx->error, IN3_EUNKNOWN);
      // if we are still waiting, we stop here and report it.
      case CTX_WAITING_FOR_REQUIRED_CTX:
      case CTX_WAITING_FOR_RESPONSE:
        return IN3_WAITING;

      // if it is useable, we can now handle the result.
      case CTX_SUCCESS: {
        d_token_t* r = d_get(ctx->responses[0], K_RESULT);
        if (r) {
          // we have a result, so write it back to the dst
          *dst = d_to_bytes(r);
          return IN3_OK;
        } else
          // or check the error and report it
          return ctx_check_response_error(parent, 0);
      }
    }
  }

  // no required context found yet, so we create one:

  // since this is a subrequest it will be freed when the parent is freed.
  // allocate memory for the request-string
  char* req = _malloc(strlen(method) + strlen(params) + 200);
  // create it
  sprintf(req, "{\"method\":\"%s\",\"jsonrpc\":\"2.0\",\"id\":1,\"params\":%s}",
↪method, params);
  // and add the request context to the parent.
  return ctx_add_required(parent, ctx_new(parent->client, req));
}
```

arguments:

*in3_ctx_t **	**parent**	the current request context.
*in3_ctx_t **	**ctx**	the new request context to add.

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

ctx_find_required

```
NONULL in3_ctx_t* ctx_find_required(const in3_ctx_t *parent, const char *method);
```

searches within the required request contextes for one with the given method.

This method is used internaly to find a previously added context.

arguments:

*in3_ctx_tconst , **	**parent**	the current request context.
`const char *`	**method**	the method of the rpc-request.

returns: *in3_ctx_tNONULL , **

ctx_remove_required

```
NONULL in3_ret_t ctx_remove_required(in3_ctx_t *parent, in3_ctx_t *ctx, bool rec);
```

removes a required context after usage.

removing will also call free_ctx to free resources.

arguments:

*in3_ctx_t **	**parent**	the current request context.
*in3_ctx_t **	**ctx**	the request context to remove.
`bool`	**rec**	if true all sub contexts will aösp be removed

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

ctx_check_response_error

```
NONULL in3_ret_t ctx_check_response_error(in3_ctx_t *c, int i);
```

check if the response contains a error-property and reports this as error in the context.

arguments:

*in3_ctx_t **	**c**	the current request context.
`int`	**i**	the index of the request to check (if this is a batch-request, otherwise 0).

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

ctx_get_error

```
NONULL in3_ret_t ctx_get_error(in3_ctx_t *ctx, int id);
```

determins the errorcode for the given request.

arguments:

*in3_ctx_t **	**ctx**	the current request context.
`int`	**id**	the index of the request to check (if this is a batch-request, otherwise 0).

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_rpc_ctx_raw

```
NONULL in3_ctx_t* in3_client_rpc_ctx_raw(in3_t *c, const char *request);
```

sends a request and returns a context used to access the result or errors.

This context *MUST* be freed with ctx_free(ctx) after usage to release the resources.

arguments:

*in3_t **	**c**	the client config.
const char *	**request**	rpc request.

returns: *in3_ctx_tNONULL , **

in3_client_rpc_ctx

```
NONULL in3_ctx_t* in3_client_rpc_ctx(in3_t *c, const char *method, const char␣
↪*params);
```

sends a request and returns a context used to access the result or errors.

This context *MUST* be freed with ctx_free(ctx) after usage to release the resources.

arguments:

*in3_t **	**c**	the clientt config.
const char *	**method**	rpc method.
const char *	**params**	params as string.

returns: *in3_ctx_tNONULL , **

in3_ctx_get_proof

```
NONULL in3_proof_t in3_ctx_get_proof(in3_ctx_t *ctx, int i);
```

determines the proof as set in the request.

arguments:

*in3_ctx_t **	**ctx**	the current request.
int	**i**	the index within the request.

returns: *in3_proof_tNONULL*

ctx_get_node

```
static NONULL in3_node_t* ctx_get_node(const in3_chain_t *chain, const node_match_t␣
↪*node);
```

arguments:

*in3_chain_tconst , **	**chain**
*node_match_tconst , **	**node**

returns: *in3_node_tNONULL , **

ctx_get_node_weight

```
static NONULL in3_node_weight_t* ctx_get_node_weight(const in3_chain_t *chain, const␣
↪node_match_t *node);
```

arguments:

*in3_chain_tconst , **	**chain**
*node_match_tconst , **	**node**

returns: *in3_node_weight_tNONULL , **

9.8.3 plugin.h

this file defines the plugin-contexts

File: c/src/core/client/plugin.h

in3_plugin_is_registered (client,action)

checks if a plugin for specified action is registered with the client

```
#define in3_plugin_is_registered (client,action) ((client)->plugin_acts & (action))
```

plugin_register (c,acts,action_fn,data,replace_ex)

registers a plugin and uses the function name as plugin name

```
#define plugin_register (c,acts,action_fn,data,replace_ex) in3_plugin_register(
↪#action_fn, c, acts, action_fn, data, replace_ex)
```

vc_err (vc,msg)

```
#define vc_err (vc,msg) vc_set_error(vc, NULL)
```

in3_signer_type_t

defines the type of signer used

The enum type contains the following values:

SIGNER_ECDSA	1
SIGNER_EIP1271	2

d_signature_type_t

type of the requested signature

The enum type contains the following values:

SIGN_EC_RAW	0	sign the data directly
SIGN_EC_HASH	1	hash and sign the data

in3_request_t

request-object.

represents a RPC-request

The stuct contains following fields:

`char *`	**payload**	the payload to send
`char **`	**urls**	array of urls
`uint_fast16_t`	**urls_len**	number of urls
in3_ctxstruct , *	**ctx**	the current context
`void *`	**cptr**	a custom ptr to hold information during
`uint32_t`	**wait**	time in ms to wait before sending out the request

in3_transport_legacy

```
typedef in3_ret_t(* in3_transport_legacy) (in3_request_t *request)
```

returns: `in3_ret_t (*` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_sign_account_ctx_t

action context when retrieving the account of a signer.

The stuct contains following fields:

in3_ctxstruct , *	**ctx**	the context of the request in order report errors
address_t	**account**	the account to use for the signature
in3_signer_type_t	**signer_type**	the type of the signer used for this account.

in3_sign_prepare_ctx_t

action context when retrieving the account of a signer.

The stuct contains following fields:

*in3_ctxstruct , **	**ctx**	the context of the request in order report errors
address_t	**account**	the account to use for the signature
bytes_t	**old_tx**	
bytes_t	**new_tx**	

in3_sign_ctx_t

signing context.

This Context is passed to the signer-function.

The stuct contains following fields:

bytes_t	**signature**	the resulting signature
d_signature_type_t	**type**	the type of signature
*in3_ctxstruct , **	**ctx**	the context of the request in order report errors
bytes_t	**message**	the message to sign
bytes_t	**account**	the account to use for the signature

in3_configure_ctx_t

context used during configure

The stuct contains following fields:

*in3_t **	**client**	the client to configure
*d_token_t **	**token**	the token not handled yet
`char *`	**error_msg**	message in case of an incorrect config

in3_get_config_ctx_t

context used during get config

The stuct contains following fields:

*in3_t **	**client**	the client to configure
*sb_t **	**sb**	stringbuilder to add json-config

in3_storage_get_item

storage handler function for reading from cache.

```
typedef bytes_t*(* in3_storage_get_item) (void *cptr, const char *key)
```

returns: `bytes_t *(*` : the found result. if the key is found this function should return the values as bytes otherwise `NULL`.

in3_storage_set_item

storage handler function for writing to the cache.

```
typedef void(* in3_storage_set_item) (void *cptr, const char *key, bytes_t *value)
```

in3_storage_clear

storage handler function for clearing the cache.

```
typedef void(* in3_storage_clear) (void *cptr)
```

in3_cache_ctx_t

context used during get config

The stuct contains following fields:

*in3_ctx_t **	**ctx**	the request context
`char *`	**key**	the key to fetch
*bytes_t **	**content**	the content to set

in3_plugin_register

```
in3_ret_t in3_plugin_register(const char *name, in3_t *c, in3_plugin_supp_acts_t acts,
↪ in3_plugin_act_fn action_fn, void *data, bool replace_ex);
```

registers a plugin with the client

arguments:

`const char *`	**name**	the name of the plugin (optional), which is ignored if LOGGIN is not defined
*in3_t **	**c**	the client
`in3_plugin_supp_acts`	**acts**	the actions to register for combined with OR
in3_plugin_act_fn	**action_fn**	the plugin action function
`void *`	**data**	an optional data or config struct which will be passed to the action function when executed
`bool`	**replace_ex**	if this is true and an plugin with the same action is already registered, it will replace it

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_register_default

```
void in3_register_default(plgn_register reg_fn);
```

adds a plugin rregister function to the default.

All defaults functions will automaticly called and registered for every new in3_t instance.

arguments:

plgn_register	**reg_fn**

in3_plugin_execute_all

```
in3_ret_t in3_plugin_execute_all(in3_t *c, in3_plugin_act_t action, void *plugin_ctx);
```

executes all plugins irrespective of their return values, returns first error (if any)

arguments:

*in3_t **	**c**
in3_plugin_act_t	**action**
`void *`	**plugin_ctx**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_plugin_execute_first

```
in3_ret_t in3_plugin_execute_first(in3_ctx_t *ctx, in3_plugin_act_t action, void␣
↪*plugin_ctx);
```

executes all plugin actions one-by-one, stops when a plugin returns anything other than IN3_EIGNORE.

returns IN3_EPLGN_NONE if no plugin was able to handle specified action, otherwise returns IN3_OK plugin errors are reported via the in3_ctx_t

arguments:

*in3_ctx_t **	**ctx**
in3_plugin_act_t	**action**
`void *`	**plugin_ctx**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_plugin_execute_first_or_none

```
in3_ret_t in3_plugin_execute_first_or_none(in3_ctx_t *ctx, in3_plugin_act_t action,␣
↪void *plugin_ctx);
```

same as in3_plugin_execute_first(), but returns IN3_OK even if no plugin could handle specified action

arguments:

*in3_ctx_t **	**ctx**
in3_plugin_act_t	**action**
void *	**plugin_ctx**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_rpc_handle_start

```
NONULL sb_t* in3_rpc_handle_start(in3_rpc_handle_ctx_t *hctx);
```

creates a response and returns a stringbuilder to add the result-data.

arguments:

*in3_rpc_handle_ctx_t **	**hctx**

returns: `sb_tNONULL , *`

in3_rpc_handle_finish

```
NONULL in3_ret_t in3_rpc_handle_finish(in3_rpc_handle_ctx_t *hctx);
```

finish the response.

arguments:

*in3_rpc_handle_ctx_t **	**hctx**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_rpc_handle_with_bytes

```
NONULL in3_ret_t in3_rpc_handle_with_bytes(in3_rpc_handle_ctx_t *hctx, bytes_t data);
```

creates a response with bytes.

arguments:

*in3_rpc_handle_ctx_t **	**hctx**
bytes_t	**data**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_rpc_handle_with_string

```
NONULL in3_ret_t in3_rpc_handle_with_string(in3_rpc_handle_ctx_t *hctx, char *data);
```

creates a response with string.

arguments:

*in3_rpc_handle_ctx_t **	**hctx**
`char *`	**data**

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_rpc_handle_with_int

```
NONULL in3_ret_t in3_rpc_handle_with_int(in3_rpc_handle_ctx_t *hctx, uint64_t value);
```

creates a response with a value which is added as hex-string.

arguments:

*in3_rpc_handle_ctx_t **	**hctx**
`uint64_t`	**value**

returns: *in3_ret_tNONULL* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_get_request_payload

```
char* in3_get_request_payload(in3_request_t *request);
```

getter to retrieve the payload from a in3_request_t struct

arguments:

*in3_request_t **	**request**	request struct

returns: `char *`

in3_get_request_urls

```
char** in3_get_request_urls(in3_request_t *request);
```

getter to retrieve the urls list from a in3_request_t struct

arguments:

*in3_request_t **	**request**	request struct

returns: `char **`

in3_get_request_urls_len

```
int in3_get_request_urls_len(in3_request_t *request);
```

getter to retrieve the urls list length from a in3_request_t struct

arguments:

*in3_request_t **	**request**	request struct

returns: `int`

in3_get_request_timeout

```
uint32_t in3_get_request_timeout(in3_request_t *request);
```

getter to retrieve the urls list length from a in3_request_t struct

arguments:

*in3_request_t **	**request**	request struct

returns: `uint32_t`

in3_req_add_response

```
NONULL void in3_req_add_response(in3_request_t *req, int index, bool is_error, const␣
↪char *data, int data_len, uint32_t time);
```

adds a response for a request-object.

This function should be used in the transport-function to set the response.

arguments:

*in3_request_t **	**req**	the the request
`int`	**index**	the index of the url, since this request could go out to many urls
`bool`	**is_error**	if true this will be reported as error. the message should then be the error-message
`const char *`	**data**	the data or the the string
`int`	**data_len**	the length of the data or the the string (use -1 if data is a null terminated string)
`uint32_t`	**time**	the time this request took in ms or 0 if not possible (it will be used to calculate the weights)

returns: `NONULL void`

in3_ctx_add_response

```
NONULL void in3_ctx_add_response(in3_ctx_t *ctx, int index, bool is_error, const char␣
↪*data, int data_len, uint32_t time);
```

adds a response to a context.

This function should be used in the transport-function to set the response.

arguments:

*in3_ctx_t **	**ctx**	the current context
`int`	**index**	the index of the url, since this request could go out to many urls
`bool`	**is_error**	if true this will be reported as error. the message should then be the error-message
`const char *`	**data**	the data or the the string
`int`	**data_len**	the length of the data or the the string (use -1 if data is a null terminated string)
`uint32_t`	**time**	the time this request took in ms or 0 if not possible (it will be used to calculate the weights)

returns: `NONULL void`

in3_set_default_legacy_transport

```
void in3_set_default_legacy_transport(in3_transport_legacy transport);
```

defines a default transport which is used when creating a new client.

arguments:

`in3_transport_legacy`	**transport**	the default transport-function.

in3_sign_ctx_get_message

```
bytes_t in3_sign_ctx_get_message(in3_sign_ctx_t *ctx);
```

helper function to retrieve and message from a in3_sign_ctx_t

helper function to retrieve and message from a in3_sign_ctx_t

arguments:

*in3_sign_ctx_t **	**ctx**	the signer context

returns: *bytes_t*

in3_sign_ctx_get_account

```
bytes_t in3_sign_ctx_get_account(in3_sign_ctx_t *ctx);
```

helper function to retrieve and account from a in3_sign_ctx_t

helper function to retrieve and account from a in3_sign_ctx_t

arguments:

*in3_sign_ctx_t **	**ctx**	the signer context

returns: *bytes_t*

in3_sign_ctx_set_signature_hex

```
void in3_sign_ctx_set_signature_hex(in3_sign_ctx_t *ctx, const char *signature);
```

helper function to retrieve the signature from a in3_sign_ctx_t

arguments:

*in3_sign_ctx_t **	**ctx**	the signer context
`const char *`	**signature**	the signature in hex

create_sign_ctx

```
NONULL in3_sign_ctx_t* create_sign_ctx(in3_ctx_t *ctx);
```

creates a signer ctx to be used for async signing.

arguments:

*in3_ctx_t **	**ctx**	the rpc context

returns: *in3_sign_ctx_tNONULL , **

in3_set_storage_handler

```
void in3_set_storage_handler(in3_t *c, in3_storage_get_item get_item, in3_storage_set_
↪item set_item, in3_storage_clear clear, void *cptr);
```

create a new storage handler-object to be set on the client.

the caller will need to free this pointer after usage.

arguments:

*in3_t **	**c**	the incubed client
in3_storage_get_item	**get_item**	function pointer returning a stored value for the given key.
in3_storage_set_item	**set_item**	function pointer setting a stored value for the given key.
in3_storage_clear	**clear**	function pointer clearing all contents of cache.
`void *`	**cptr**	custom pointer which will will be passed to functions

vc_set_error

```
in3_ret_t vc_set_error(in3_vctx_t *vc, char *msg);
```

arguments:

*in3_vctx_t **	**vc**	the verification context.
`char *`	**msg**	the error message.

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

9.8.4 bytes.h

util helper on byte arrays.

File: c/src/core/util/bytes.h

bb_new ()

creates a new bytes_builder with a initial size of 32 bytes

```
#define bb_new () bb_newl(32)
```

bb_read (*bb*,*i*,*vptr*)

```
#define bb_read (_bb_,_i_,_vptr_) bb_readl((_bb_), (_i_), (_vptr_), sizeof(*_vptr_))
```

bb_read_next (*bb*,*iptr*,*vptr*)

```
#define bb_read_next (_bb_,_iptr_,_vptr_) do {
↪   \
    size_t _l_ = sizeof(*_vptr_);              \
    bb_readl((_bb_), *(_iptr_), (_vptr_), _l_); \
    *(_iptr_) += _l_;                          \
  } while (0)
```

bb_readl (*bb*,*i*,*vptr*,*l*)

```
#define bb_readl (_bb_,_i_,_vptr_,_l_) memcpy((_vptr_), (_bb_)->b.data + (_i_), _l_)
```

b_read (*b*,*i*,*vptr*)

```
#define b_read (_b_,_i_,_vptr_) b_readl((_b_), (_i_), _vptr_, sizeof(*_vptr_))
```

b_readl (*b*,*i*,*vptr*,*l*)

```
#define b_readl (_b_,_i_,_vptr_,_l_) memcpy(_vptr_, (_b_)->data + (_i_), (_l_))
```

address_t

pointer to a 20byte address

```
typedef uint8_t address_t[20]
```

bytes32_t

pointer to a 32byte word

```
typedef uint8_t bytes32_t[32]
```

wlen_t

number of bytes within a word (min 1byte but usually a uint)

```
typedef uint_fast8_t wlen_t
```

bytes_t

a byte array

The stuct contains following fields:

`uint8_t *`	**data**	the byte-data
`uint32_t`	**len**	the length of the array ion bytes

b_new

```
RETURNS_NONULL bytes_t* b_new(const uint8_t *data, uint32_t len);
```

allocates a new byte array with 0 filled

arguments:

`const uint8_t *`	**data**
`uint32_t`	**len**

returns: *bytes_tRETURNS_NONULL , **

b_get_data

```
NONULL uint8_t* b_get_data(const bytes_t *b);
```

gets the data field from an input byte array

arguments:

*bytes_tconst , **	**b**

returns: `NONULL uint8_t *`

b_get_len

```
NONULL uint32_t b_get_len(const bytes_t *b);
```

gets the len field from an input byte array

arguments:

*bytes_tconst , **	**b**

returns: `NONULL uint32_t`

b_print

```
NONULL void b_print(const bytes_t *a);
```

prints a the bytes as hex to stdout

arguments:

*bytes_tconst , **	**a**

returns: `NONULL void`

ba_print

```
NONULL void ba_print(const uint8_t *a, size_t l);
```

prints a the bytes as hex to stdout

arguments:

`const uint8_t *`	**a**
`size_t`	**l**

returns: `NONULL void`

b_cmp

```
NONULL int b_cmp(const bytes_t *a, const bytes_t *b);
```

compares 2 byte arrays and returns 1 for equal and 0 for not equal

arguments:

*bytes_tconst , **	**a**
*bytes_tconst , **	**b**

returns: `NONULL int`

bytes_cmp

```
int bytes_cmp(const bytes_t a, const bytes_t b);
```

compares 2 byte arrays and returns 1 for equal and 0 for not equal

arguments:

bytes_tconst	**a**
bytes_tconst	**b**

returns: `int`

b_free

```
void b_free(bytes_t *a);
```

frees the data

arguments:

bytes_t *	**a**

b_concat

```
bytes_t b_concat(int cnt,...);
```

duplicates the content of bytes

arguments:

`int`	**cnt**
`...`	

returns: *bytes_t*

b_dup

```
NONULL bytes_t* b_dup(const bytes_t *a);
```

clones a byte array

arguments:

bytes_tconst , *	**a**

returns: *bytes_tNONULL , **

b_read_byte

```
NONULL uint8_t b_read_byte(bytes_t *b, size_t *pos);
```

reads a byte on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
`size_t *`	**pos**

returns: `NONULL uint8_t`

b_read_int

```
NONULL uint32_t b_read_int(bytes_t *b, size_t *pos);
```

reads a integer on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
`size_t *`	**pos**

returns: `NONULL uint32_t`

b_read_long

```
NONULL uint64_t b_read_long(bytes_t *b, size_t *pos);
```

reads a long on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
`size_t *`	**pos**

returns: `NONULL uint64_t`

b_new_chars

```
NONULL char* b_new_chars(bytes_t *b, size_t *pos);
```

creates a new string (needs to be freed) on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
`size_t *`	**pos**

returns: `NONULL char *`

b_new_fixed_bytes

```
NONULL bytes_t* b_new_fixed_bytes(bytes_t *b, size_t *pos, int len);
```

reads bytes with a fixed length on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
`size_t *`	**pos**
`int`	**len**

returns: `bytes_tNONULL , *`

bb_newl

```
bytes_builder_t* bb_newl(size_t l);
```

creates a new bytes_builder

arguments:

`size_t`	**l**

returns: `bytes_builder_t *`

bb_free

```
NONULL void bb_free(bytes_builder_t *bb);
```

frees a bytebuilder and its content.

arguments:

*bytes_builder_t **	**bb**

returns: `NONULL void`

bb_check_size

```
NONULL int bb_check_size(bytes_builder_t *bb, size_t len);
```

internal helper to increase the buffer if needed

arguments:

*bytes_builder_t **	**bb**
`size_t`	**len**

returns: `NONULL int`

bb_write_chars

```
NONULL void bb_write_chars(bytes_builder_t *bb, char *c, int len);
```

writes a string to the builder.

arguments:

*bytes_builder_t **	**bb**
`char *`	**c**
`int`	**len**

returns: `NONULL void`

bb_write_dyn_bytes

```
NONULL void bb_write_dyn_bytes(bytes_builder_t *bb, const bytes_t *src);
```

writes bytes to the builder with a prefixed length.

arguments:

*bytes_builder_t **	**bb**
*bytes_tconst , **	**src**

returns: `NONULL void`

bb_write_fixed_bytes

```
NONULL void bb_write_fixed_bytes(bytes_builder_t *bb, const bytes_t *src);
```

writes fixed bytes to the builder.

arguments:

*bytes_builder_t **	**bb**
*bytes_tconst , **	**src**

returns: `NONULL void`

bb_write_int

```
NONULL void bb_write_int(bytes_builder_t *bb, uint32_t val);
```

writes a ineteger to the builder.

arguments:

*bytes_builder_t **	**bb**
`uint32_t`	**val**

returns: `NONULL void`

bb_write_long

```
NONULL void bb_write_long(bytes_builder_t *bb, uint64_t val);
```

writes s long to the builder.

arguments:

*bytes_builder_t **	**bb**
`uint64_t`	**val**

returns: `NONULL void`

bb_write_long_be

```
NONULL void bb_write_long_be(bytes_builder_t *bb, uint64_t val, int len);
```

writes any integer value with the given length of bytes

arguments:

*bytes_builder_t **	**bb**
`uint64_t`	**val**
`int`	**len**

returns: `NONULL void`

bb_write_byte

```
NONULL void bb_write_byte(bytes_builder_t *bb, uint8_t val);
```

writes a single byte to the builder.

arguments:

*bytes_builder_t **	**bb**
`uint8_t`	**val**

returns: `NONULL void`

bb_write_raw_bytes

```
NONULL void bb_write_raw_bytes(bytes_builder_t *bb, void *ptr, size_t len);
```

writes the bytes to the builder.

arguments:

*bytes_builder_t **	**bb**
`void *`	**ptr**
`size_t`	**len**

returns: NONULL void

bb_clear

```
NONULL void bb_clear(bytes_builder_t *bb);
```

resets the content of the builder.

arguments:

*bytes_builder_t **	**bb**

returns: NONULL void

bb_replace

```
NONULL void bb_replace(bytes_builder_t *bb, int offset, int delete_len, uint8_t *data,
↪ int data_len);
```

replaces or deletes a part of the content.

arguments:

*bytes_builder_t **	**bb**
int	**offset**
int	**delete_len**
uint8_t *	**data**
int	**data_len**

returns: NONULL void

bb_move_to_bytes

```
RETURNS_NONULL NONULL bytes_t* bb_move_to_bytes(bytes_builder_t *bb);
```

frees the builder and moves the content in a newly created bytes struct (which needs to be freed later).

arguments:

*bytes_builder_t **	**bb**

returns: *bytes_tRETURNS_NONULL NONULL , **

bb_read_long

```
NONULL uint64_t bb_read_long(bytes_builder_t *bb, size_t *i);
```

reads a long from the builder

arguments:

*bytes_builder_t **	**bb**
`size_t *`	**i**

returns: `NONULL uint64_t`

bb_read_int

```
NONULL uint32_t bb_read_int(bytes_builder_t *bb, size_t *i);
```

reads a int from the builder

arguments:

*bytes_builder_t **	**bb**
`size_t *`	**i**

returns: `NONULL uint32_t`

bytes

```
static bytes_t bytes(uint8_t *a, uint32_t len);
```

converts the given bytes to a bytes struct

arguments:

`uint8_t *`	**a**
`uint32_t`	**len**

returns: *bytes_t*

cloned_bytes

```
bytes_t cloned_bytes(bytes_t data);
```

cloned the passed data

arguments:

bytes_t	**data**

returns: *bytes_t*

b_optimize_len

```
static NONULL void b_optimize_len(bytes_t *b);
```

< changed the data and len to remove leading 0-bytes

arguments:

*bytes_t **	**b**

returns: `NONULL void`

9.8.5 data.h

json-parser.

The parser can read from :

- json
- bin

When reading from json all '0x'... values will be stored as bytes_t. If the value is lower than 0xFFFFFFF, it is converted as integer.

File: c/src/core/util/data.h

DATA_DEPTH_MAX

the max DEPTH of the JSON-data allowed.

It will throw an error if reached.

```
#define DATA_DEPTH_MAX 11
```

printX

```
#define printX printf
```

fprintX

```
#define fprintX fprintf
```

snprintX

```
#define snprintX snprintf
```

vprintX

```
#define vprintX vprintf
```

d_type_t

type of a token.

The enum type contains the following values:

T_BYTES	0	content is stored as data ptr.
T_STRING	1	content is stored a c-str
T_ARRAY	2	the node is an array with the length stored in length
T_OBJECT	3	the node is an object with properties
T_BOOLEAN	4	boolean with the value stored in len
T_INTEGER	5	a integer with the value stored
T_NULL	6	a NULL-value

d_key_t

```
typedef uint16_t d_key_t
```

d_token_t

a token holding any kind of value.

use d_type, d_len or the cast-function to get the value.

The stuct contains following fields:

`uint8_t *`	**data**	the byte or string-data
`uint32_t`	**len**	the length of the content (or number of properties) depending + type.
`d_key_t`	**key**	the key of the property.

str_range_t

internal type used to represent the a range within a string.

The stuct contains following fields:

`char *`	**data**	pointer to the start of the string
`size_t`	**len**	len of the characters

json_ctx_t

parser for json or binary-data.

it needs to freed after usage.

The stuct contains following fields:

*d_token_t **	**result**	the list of all tokens. the first token is the main-token as returned by the parser.
`char *`	**c**	pointer to the src-data
`size_t`	**allocated**	amount of tokens allocated result
`size_t`	**len**	number of tokens in result
`size_t`	**depth**	max depth of tokens in result
`uint8_t *`	**keys**	
`size_t`	**keys_last**	

d_iterator_t

iterator over elements of a array opf object.

usage:

```
for (d_iterator_t iter = d_iter( parent ); iter.left ; d_iter_next(&iter)) {
  uint32_t val = d_int(iter.token);
}
```

The stuct contains following fields:

*d_token_t **	**token**	current token
`int`	**left**	number of result left

d_to_bytes

```
bytes_t d_to_bytes(d_token_t *item);
```

returns the byte-representation of token.

In case of a number it is returned as bigendian. booleans as 0x01 or 0x00 and NULL as 0x. Objects or arrays will return 0x.

arguments:

*d_token_t **	**item**

returns: *bytes_t*

d_bytes_to

```
int d_bytes_to(d_token_t *item, uint8_t *dst, const int max);
```

writes the byte-representation to the dst.

details see d_to_bytes.

arguments:

*d_token_t **	**item**
`uint8_t *`	**dst**
`const int`	**max**

returns: `int`

d_bytes

```
bytes_t* d_bytes(const d_token_t *item);
```

returns the value as bytes (Carefully, make sure that the token is a bytes-type!)

arguments:

*d_token_tconst , **	**item**

returns: *bytes_t **

d_bytesl

```
bytes_t* d_bytesl(d_token_t *item, size_t l);
```

returns the value as bytes with length l (may reallocates)

arguments:

*d_token_t **	**item**
`size_t`	**l**

returns: *bytes_t **

d_string

```
char* d_string(const d_token_t *item);
```

converts the value as string.

Make sure the type is string!

arguments:

*d_token_tconst , **	**item**

returns: `char *`

d_int

```
int32_t d_int(const d_token_t *item);
```

returns the value as integer.

only if type is integer

arguments:

*d_token_tconst , **	**item**

returns: `int32_t`

d_intd

```
int32_t d_intd(const d_token_t *item, const uint32_t def_val);
```

returns the value as integer or if NULL the default.

only if type is integer

arguments:

*d_token_tconst , **	**item**
`const uint32_t`	**def_val**

returns: `int32_t`

d_long

```
uint64_t d_long(const d_token_t *item);
```

returns the value as long.

only if type is integer or bytes, but short enough

arguments:

*d_token_tconst , **	**item**

returns: `uint64_t`

d_longd

```
uint64_t d_longd(const d_token_t *item, const uint64_t def_val);
```

returns the value as long or if NULL the default.

only if type is integer or bytes, but short enough

arguments:

*d_token_tconst , **	**item**
`const uint64_t`	**def_val**

returns: `uint64_t`

d_create_bytes_vec

```
bytes_t** d_create_bytes_vec(const d_token_t *arr);
```

arguments:

*d_token_tconst , **	**arr**

returns: *bytes_t ***

d_type

```
static d_type_t d_type(const d_token_t *item);
```

creates a array of bytes from JOSN-array

type of the token

arguments:

*d_token_tconst , **	**item**

returns: *d_type_t*

d_len

```
static int d_len(const d_token_t *item);
```

< number of elements in the token (only for object or array, other will return 0)

arguments:

*d_token_tconst , **	**item**

returns: int

d_eq

```
bool d_eq(const d_token_t *a, const d_token_t *b);
```

compares 2 token and if the value is equal

arguments:

*d_token_tconst , **	**a**
*d_token_tconst , **	**b**

returns: bool

keyn

```
NONULL d_key_t keyn(const char *c, const size_t len);
```

generates the keyhash for the given stringrange as defined by len

arguments:

const char *	**c**
const size_t	**len**

returns: NONULL d_key_t

ikey

```
d_key_t ikey(json_ctx_t *ctx, const char *name);
```

returnes the indexed key for the given name.

arguments:

*json_ctx_t **	**ctx**
const char *	**name**

returns: d_key_t

d_get

```
d_token_t* d_get(d_token_t *item, const uint16_t key);
```

returns the token with the given propertyname (only if item is a object)

arguments:

*d_token_t **	**item**
const uint16_t	**key**

returns: *d_token_t **

d_get_or

```
d_token_t* d_get_or(d_token_t *item, const uint16_t key1, const uint16_t key2);
```

returns the token with the given propertyname or if not found, tries the other.

(only if item is a object)

arguments:

*d_token_t **	**item**
const uint16_t	**key1**
const uint16_t	**key2**

returns: *d_token_t **

d_get_at

```
d_token_t* d_get_at(d_token_t *item, const uint32_t index);
```

returns the token of an array with the given index

arguments:

*d_token_t **	**item**
const uint32_t	**index**

returns: `d_token_t *`

d_next

```
d_token_t* d_next(d_token_t *item);
```

returns the next sibling of an array or object

arguments:

*d_token_t **	**item**

returns: `d_token_t *`

d_serialize_binary

```
NONULL void d_serialize_binary(bytes_builder_t *bb, d_token_t *t);
```

write the token as binary data into the builder

arguments:

*bytes_builder_t **	**bb**
*d_token_t **	**t**

returns: `NONULL void`

parse_binary

```
NONULL json_ctx_t* parse_binary(const bytes_t *data);
```

parses the data and returns the context with the token, which needs to be freed after usage!

arguments:

*bytes_tconst , **	**data**

returns: `json_ctx_tNONULL , *`

parse_binary_str

```
NONULL json_ctx_t* parse_binary_str(const char *data, int len);
```

parses the data and returns the context with the token, which needs to be freed after usage!

arguments:

`const char *`	**data**
`int`	**len**

returns: *json_ctx_tNONULL , **

parse_json

```
NONULL json_ctx_t* parse_json(const char *js);
```

parses json-data, which needs to be freed after usage!

arguments:

`const char *`	**js**

returns: *json_ctx_tNONULL , **

parse_json_indexed

```
NONULL json_ctx_t* parse_json_indexed(const char *js);
```

parses json-data, which needs to be freed after usage!

arguments:

`const char *`	**js**

returns: *json_ctx_tNONULL , **

json_free

```
NONULL void json_free(json_ctx_t *parser_ctx);
```

frees the parse-context after usage

arguments:

*json_ctx_t **	**parser_ctx**

returns: `NONULL void`

d_to_json

```
NONULL str_range_t d_to_json(const d_token_t *item);
```

returns the string for a object or array.

This only works for json as string. For binary it will not work!

arguments:

*d_token_tconst , **	**item**

returns: *str_range_tNONULL*

d_create_json

```
char* d_create_json(json_ctx_t *ctx, d_token_t *item);
```

creates a json-string.

It does not work for objects if the parsed data were binary!

arguments:

*json_ctx_t **	**ctx**
*d_token_t **	**item**

returns: `char *`

json_create

```
json_ctx_t* json_create();
```

returns: *json_ctx_t **

json_create_null

```
NONULL d_token_t* json_create_null(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_tNONULL , **

json_create_bool

```
NONULL d_token_t* json_create_bool(json_ctx_t *jp, bool value);
```

arguments:

*json_ctx_t **	**jp**
`bool`	**value**

returns: *d_token_tNONULL , **

json_create_int

```
NONULL d_token_t* json_create_int(json_ctx_t *jp, uint64_t value);
```

arguments:

*json_ctx_t **	**jp**
`uint64_t`	**value**

returns: *d_token_tNONULL , **

json_create_string

```
NONULL d_token_t* json_create_string(json_ctx_t *jp, char *value, int len);
```

arguments:

*json_ctx_t **	**jp**
`char *`	**value**
`int`	**len**

returns: *d_token_tNONULL , **

json_create_bytes

```
NONULL d_token_t* json_create_bytes(json_ctx_t *jp, bytes_t value);
```

arguments:

*json_ctx_t **	**jp**
bytes_t	**value**

returns: *d_token_tNONULL , **

json_create_object

```
NONULL d_token_t* json_create_object(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_tNONULL , **

json_create_array

```
NONULL d_token_t* json_create_array(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_tNONULL , **

json_object_add_prop

```
NONULL d_token_t* json_object_add_prop(d_token_t *object, d_key_t key, d_token_t
↪*value);
```

arguments:

*d_token_t **	**object**
d_key_t	**key**
*d_token_t **	**value**

returns: *d_token_tNONULL , **

json_array_add_value

```
NONULL d_token_t* json_array_add_value(d_token_t *object, d_token_t *value);
```

arguments:

*d_token_t **	**object**
*d_token_t **	**value**

returns: *d_token_tNONULL , **

d_get_keystr

```
char* d_get_keystr(json_ctx_t *json, d_key_t k);
```

returns the string for a key.

This only works for index keys or known keys!

arguments:

*json_ctx_t **	**json**
d_key_t	**k**

returns: `char *`

key

```
static NONULL d_key_t key(const char *c);
```

arguments:

const char *	**c**

returns: `NONULL d_key_t`

d_get_stringk

```
static char* d_get_stringk(d_token_t *r, d_key_t k);
```

reads token of a property as string.

arguments:

*d_token_t **	**r**
d_key_t	**k**

returns: `char *`

d_get_string

```
static char* d_get_string(d_token_t *r, char *k);
```

reads token of a property as string.

arguments:

*d_token_t **	**r**
char *	**k**

returns: `char *`

d_get_string_at

```
static char* d_get_string_at(d_token_t *r, uint32_t pos);
```

reads string at given pos of an array.

arguments:

*d_token_t **	**r**
`uint32_t`	**pos**

returns: `char *`

d_get_intk

```
static int32_t d_get_intk(d_token_t *r, d_key_t k);
```

reads token of a property as int.

arguments:

*d_token_t **	**r**
`d_key_t`	**k**

returns: `int32_t`

d_get_intkd

```
static int32_t d_get_intkd(d_token_t *r, d_key_t k, uint32_t d);
```

reads token of a property as int.

arguments:

*d_token_t **	**r**
`d_key_t`	**k**
`uint32_t`	**d**

returns: `int32_t`

d_get_int

```
static int32_t d_get_int(d_token_t *r, char *k);
```

reads token of a property as int.

arguments:

*d_token_t **	**r**
`char *`	**k**

returns: `int32_t`

d_get_int_at

```
static int32_t d_get_int_at(d_token_t *r, uint32_t pos);
```

reads a int at given pos of an array.

arguments:

*d_token_t **	**r**
`uint32_t`	**pos**

returns: `int32_t`

d_get_longk

```
static uint64_t d_get_longk(d_token_t *r, d_key_t k);
```

reads token of a property as long.

arguments:

*d_token_t **	**r**
`d_key_t`	**k**

returns: `uint64_t`

d_get_longkd

```
static uint64_t d_get_longkd(d_token_t *r, d_key_t k, uint64_t d);
```

reads token of a property as long.

arguments:

*d_token_t **	**r**
`d_key_t`	**k**
`uint64_t`	**d**

returns: `uint64_t`

d_get_long

```
static uint64_t d_get_long(d_token_t *r, char *k);
```

reads token of a property as long.

arguments:

*d_token_t **	**r**
`char *`	**k**

returns: `uint64_t`

d_get_long_at

```
static uint64_t d_get_long_at(d_token_t *r, uint32_t pos);
```

reads long at given pos of an array.

arguments:

*d_token_t **	**r**
`uint32_t`	**pos**

returns: `uint64_t`

d_get_bytesk

```
static bytes_t* d_get_bytesk(d_token_t *r, d_key_t k);
```

reads token of a property as bytes.

arguments:

*d_token_t **	**r**
`d_key_t`	**k**

returns: *bytes_t **

d_get_bytes

```
static bytes_t* d_get_bytes(d_token_t *r, char *k);
```

reads token of a property as bytes.

arguments:

*d_token_t **	**r**
`char *`	**k**

returns: *bytes_t **

d_get_bytes_at

```
static bytes_t* d_get_bytes_at(d_token_t *r, uint32_t pos);
```

reads bytes at given pos of an array.

arguments:

*d_token_t **	**r**
`uint32_t`	**pos**

returns: *bytes_t **

d_is_binary_ctx

```
static bool d_is_binary_ctx(json_ctx_t *ctx);
```

check if the parser context was created from binary data.

arguments:

*json_ctx_t **	**ctx**

returns: `bool`

d_get_byteskl

```
bytes_t* d_get_byteskl(d_token_t *r, d_key_t k, uint32_t minl);
```

arguments:

*d_token_t **	**r**
`d_key_t`	**k**
`uint32_t`	**minl**

returns: *bytes_t **

d_getl

```
d_token_t* d_getl(d_token_t *item, uint16_t k, uint32_t minl);
```

arguments:

*d_token_t **	**item**
`uint16_t`	**k**
`uint32_t`	**minl**

returns: *d_token_t **

d_iter

```
d_iterator_t d_iter(d_token_t *parent);
```

creates a iterator for a object or array

arguments:

*d_token_t **	**parent**

returns: *d_iterator_t*

d_iter_next

```
static bool d_iter_next(d_iterator_t *const iter);
```

fetched the next token an returns a boolean indicating whther there is a next or not.

arguments:

*d_iterator_t *const*	**iter**

returns: `bool`

9.8.6 debug.h

logs debug data only if the DEBUG-flag is set.

File: c/src/core/util/debug.h

dbg_log (msg,. . .)

logs a debug-message including file and linenumber

dbg_log_raw (msg,. . .)

logs a debug-message without the filename

msg_dump

```
void msg_dump(const char *s, const unsigned char *data, unsigned len);
```

dumps the given data as hex coded bytes to stdout

arguments:

`const char *`	**s**
`const unsigned char *`	**data**
`unsigned`	**len**

9.8.7 error.h

defines the return-values of a function call.

File: c/src/core/util/error.h

DEPRECATED

depreacted-attribute

```
#define DEPRECATED __attribute__((deprecated))
```

OPTIONAL_T (t)

Optional type similar to C++ std::optional Optional types must be defined prior to usage (e.g.

DEFINE_OPTIONAL_T(int)) Use OPTIONAL_T_UNDEFINED(t) & OPTIONAL_T_VALUE(t, v) for easy initialization (rvalues) Note: Defining optional types for pointers is ill-formed by definition. This is because redundant

```
#define OPTIONAL_T (t) opt_##t
```

DEFINE_OPTIONAL_T (t)

Optional types must be defined prior to usage (e.g.

DEFINE_OPTIONAL_T(int)) Use OPTIONAL_T_UNDEFINED(t) & OPTIONAL_T_VALUE(t, v) for easy initialization (rvalues)

```
#define DEFINE_OPTIONAL_T (t) typedef struct {            \
    t    value;              \
    bool defined;            \
  } OPTIONAL_T(t)
```

OPTIONAL_T_UNDEFINED (t)

marks a used value as undefined.

```
#define OPTIONAL_T_UNDEFINED (t)  ((OPTIONAL_T(t)){.defined = false})
```

OPTIONAL_T_VALUE (t,v)

sets the value of an optional type.

```
#define OPTIONAL_T_VALUE (t,v)  ((OPTIONAL_T(t)){.value = v, .defined = true})
```

in3_ret_t

ERROR types used as return values.

All values (except IN3_OK) indicate an error. IN3_WAITING may be treated like an error, since we have stop executing until the response has arrived, but it is a valid return value.

The enum type contains the following values:

IN3_OK	0	Success.
IN3_EUNKNOWN	-1	Unknown error - usually accompanied with specific error msg.
IN3_ENOMEM	-2	No memory.
IN3_ENOTSUP	-3	Not supported.
IN3_EINVAL	-4	Invalid value.
IN3_EFIND	-5	Not found.
IN3_ECONFIG	-6	Invalid config.
IN3_ELIMIT	-7	Limit reached.
IN3_EVERS	-8	Version mismatch.
IN3_EINVALDT	-9	Data invalid, eg. invalid/incomplete JSON
IN3_EPASS	-10	Wrong password.
IN3_ERPC	-11	RPC error (i.e. in3_ctx_t::error set)
IN3_ERPCNRES	-12	RPC no response.
IN3_EUSNURL	-13	USN URL parse error.
IN3_ETRANS	-14	Transport error.
IN3_ERANGE	-15	Not in range.
IN3_WAITING	-16	the process can not be finished since we are waiting for responses
IN3_EIGNORE	-17	Ignorable error.
IN3_EPAYMENT_REQUIRED	-18	payment required
IN3_ENODEVICE	-19	harware wallet device not connected
IN3_EAPDU	-20	error in hardware wallet communication
IN3_EPLGN_NONE	-21	no plugin could handle specified action

in3_errmsg

```
char* in3_errmsg(in3_ret_t err);
```

converts a error code into a string.

These strings are constants and do not need to be freed.

arguments:

in3_ret_t	**err**	the error code

returns: `char *`

9.8.8 scache.h

util helper on byte arrays.

File: c/src/core/util/scache.h

cache_props

The enum type contains the following values:

CACHE_PROP_MUST_FREE	0x1	indicates the content must be freed
CACHE_PROP_SRC_REQ	0x2	the value holds the src-request
CACHE_PROP_ONLY_EXTERNAL	0x4	should only be freed if the context is external

cache_props_t

The enum type contains the following values:

CACHE_PROP_MUST_FREE	0x1	indicates the content must be freed
CACHE_PROP_SRC_REQ	0x2	the value holds the src-request
CACHE_PROP_ONLY_EXTERNAL	0x4	should only be freed if the context is external

cache_entry_t

represents a single cache entry in a linked list.

These are used within a request context to cache values and automaticly free them.

The stuct contains following fields:

bytes_t	**key**	an optional key of the entry
bytes_t	**value**	the value
`uint8_t`	**buffer**	the buffer is used to store extra data, which will be cleaned when freed.
`cache_props_t`	**props**	if true, the cache-entry will be freed when the request context is cleaned up.
cache_entrystruct , *	**next**	pointer to the next entry.

in3_cache_get_entry

```
bytes_t* in3_cache_get_entry(cache_entry_t *cache, bytes_t *key);
```

get the entry for a given key.

arguments:

*cache_entry_t **	**cache**	the root entry of the linked list.
*bytes_t **	**key**	the key to compare with

returns: `bytes_t *`

in3_cache_add_entry

```
cache_entry_t* in3_cache_add_entry(cache_entry_t **cache, bytes_t key, bytes_t value);
```

adds an entry to the linked list.

arguments:

*cache_entry_t ***	**cache**	the root entry of the linked list.
bytes_t	**key**	an optional key
bytes_t	**value**	the value of the entry

returns: *cache_entry_t **

in3_cache_free

```
void in3_cache_free(cache_entry_t *cache, bool is_external);
```

clears all entries in the linked list.

arguments:

*cache_entry_t **	**cache**	the root entry of the linked list.
bool	**is_external**	true if this is the root context or an external.

in3_cache_add_ptr

```
static NONULL cache_entry_t* in3_cache_add_ptr(cache_entry_t **cache, void *ptr);
```

adds a pointer, which should be freed when the context is freed.

arguments:

*cache_entry_t ***	**cache**	the root entry of the linked list.
void *	**ptr**	pointer to memory which shold be freed.

returns: *cache_entry_tNONULL , **

9.8.9 stringbuilder.h

simple string buffer used to dynamicly add content.

File: c/src/core/util/stringbuilder.h

sb_add_hexuint (sb,i)

shortcut macro for adding a uint to the stringbuilder using sizeof(i) to automaticly determine the size

```
#define sb_add_hexuint (sb,i) sb_add_hexuint_l(sb, i, sizeof(i))
```

sb_t

string build struct, which is able to hold and modify a growing string.

The stuct contains following fields:

char *	**data**	the current string (null terminated)
size_t	**allocted**	number of bytes currently allocated
size_t	**len**	the current length of the string

sb_stack

```
static NONULL sb_t sb_stack(char *p);
```

creates a stringbuilder which is allocating any new memory, but uses an existing string and is used directly on the stack.

Since it will not grow the memory you need to pass a char* which allocated enough memory.

arguments:

`char *`	**p**

returns: *sb_tNONULL*

sb_new

```
sb_t* sb_new(const char *chars);
```

creates a new stringbuilder and copies the inital characters into it.

arguments:

`const char *`	**chars**

returns: *sb_t **

sb_init

```
NONULL sb_t* sb_init(sb_t *sb);
```

initializes a stringbuilder by allocating memory.

arguments:

*sb_t **	**sb**

returns: *sb_tNONULL , **

sb_free

```
NONULL void sb_free(sb_t *sb);
```

frees all resources of the stringbuilder

arguments:

*sb_t **	**sb**

returns: `NONULL void`

sb_add_char

```
NONULL sb_t* sb_add_char(sb_t *sb, char c);
```

add a single character

arguments:

*sb_t **	**sb**
char	**c**

returns: *sb_tNONULL , **

sb_add_chars

```
NONULL sb_t* sb_add_chars(sb_t *sb, const char *chars);
```

adds a string

arguments:

*sb_t **	**sb**
const char *	**chars**

returns: *sb_tNONULL , **

sb_add_range

```
NONULL sb_t* sb_add_range(sb_t *sb, const char *chars, int start, int len);
```

add a string range

arguments:

*sb_t **	**sb**
const char *	**chars**
int	**start**
int	**len**

returns: *sb_tNONULL , **

sb_add_key_value

```
NONULL sb_t* sb_add_key_value(sb_t *sb, const char *key, const char *value, int value_
↪len, bool as_string);
```

adds a value with an optional key.

if as_string is true the value will be quoted.

arguments:

*sb_t **	**sb**
`const char *`	**key**
`const char *`	**value**
`int`	**value_len**
`bool`	**as_string**

returns: *sb_tNONULL , **

sb_add_bytes

```
sb_t* sb_add_bytes(sb_t *sb, const char *prefix, const bytes_t *bytes, int len, bool␣
↪as_array);
```

add bytes as 0x-prefixed hexcoded string (including an optional prefix), if len>1 is passed bytes maybe an array (if as_array==true)

arguments:

*sb_t **	**sb**
`const char *`	**prefix**
*bytes_tconst , **	**bytes**
`int`	**len**
`bool`	**as_array**

returns: *sb_t **

sb_add_hexuint_l

```
NONULL sb_t* sb_add_hexuint_l(sb_t *sb, uintmax_t uint, size_t l);
```

add a integer value as hexcoded, 0x-prefixed string

Other types not supported

arguments:

*sb_t **	**sb**
`uintmax_t`	**uint**
`size_t`	**l**

returns: *sb_tNONULL , **

sb_add_escaped_chars

```
NONULL sb_t* sb_add_escaped_chars(sb_t *sb, const char *chars);
```

add chars but escapes all quotes

arguments:

*sb_t **	**sb**
`const char *`	**chars**

returns: `sb_tNONULL , *`

sb_add_int

```
NONULL sb_t* sb_add_int(sb_t *sb, uint64_t val);
```

adds a numeric value to the stringbuilder

arguments:

*sb_t **	**sb**
`uint64_t`	**val**

returns: `sb_tNONULL , *`

format_json

```
NONULL char* format_json(const char *json);
```

format a json string and returns a new string, which needs to be freed

arguments:

`const char *`	**json**

returns: `NONULL char *`

sb_add_rawbytes

```
sb_t* sb_add_rawbytes(sb_t *sb, char *prefix, bytes_t b, unsigned int fix_size);
```

arguments:

*sb_t **	**sb**
`char *`	**prefix**
bytes_t	**b**
`unsigned int`	**fix_size**

returns: `sb_t *`

sb_print

```
sb_t* sb_print(sb_t *sb, const char *fmt,...);
```

arguments:

sb_t *	**sb**
`const char *`	**fmt**
`...`	

returns: `sb_t *`

sb_vprint

```
sb_t* sb_vprint(sb_t *sb, const char *fmt, va_list args);
```

arguments:

sb_t *	**sb**
`const char *`	**fmt**
`va_list`	**args**

returns: `sb_t *`

9.8.10 utils.h

utility functions.

File: c/src/core/util/utils.h

_strtoull (str,endptr,base)

```
#define _strtoull (str,endptr,base) strtoull(str, endptr, base)
```

SWAP (a,b)

simple swap macro for integral types

```
#define SWAP (a,b) {                \
    void* p = a;    \
    a       = b;    \
    b       = p;    \
  }
```

min (a,b)

simple min macro for interagl types

```
#define min (a,b) ((a) < (b) ? (a) : (b))
```

max (a,b)

simple max macro for interagl types

```
#define max (a,b) ((a) > (b) ? (a) : (b))
```

IS_APPROX (n1,n2,err)

Check if n1 & n2 are at max err apart Expects n1 & n2 to be integral types.

```
#define IS_APPROX (n1,n2,err) ((n1 > n2) ? ((n1 - n2) <= err) : ((n2 - n1) <= err))
```

STR_IMPL_ (x)

simple macro to stringify other macro defs eg.

usage - to concatenate a const with a string at compile time -> define SOME_CONST_UINT 10U printf("Using default value of " STR(SOME_CONST_UINT));

```
#define STR_IMPL_ (x) #x
```

STR (x)

```
#define STR (x) STR_IMPL_(x)
```

optimize_len (a,l)

changes to pointer (a) and it length (l) to remove leading 0 bytes.

```
#define optimize_len (a,l) while (l > 1 && *a == 0) { \
    l--;                    \
    a++;                    \
  }
```

TRY (exp)

executes the expression and expects the return value to be a int indicating the error.

if the return value is negative it will stop and return this value otherwise continue.

```
#define TRY (exp) {                           \
    int _r = (exp);        \
    if (_r < 0) return _r; \
  }
```

TRY_FINAL (exp,final)

executes the expression and expects the return value to be a int indicating the error.

if the return value is negative it will stop and return this value otherwise continue.

```
#define TRY_FINAL (exp,final) {                            \
    int _r = (exp);            \
    final;                     \
    if (_r < 0) return _r;     \
  }
```

EXPECT_EQ (exp,val)

executes the expression and expects value to equal val.

if not it will return IN3_EINVAL

```
#define EXPECT_EQ (exp,val) if ((exp) != val) return IN3_EINVAL;
```

TRY_SET (var,exp)

executes the expression and expects the return value to be a int indicating the error.

the return value will be set to a existing variable (var). if the return value is negative it will stop and return this value otherwise continue.

```
#define TRY_SET (var,exp) {                            \
    var = (exp);            \
    if (var < 0) return var; \
  }
```

TRY_GOTO (exp)

executes the expression and expects the return value to be a int indicating the error.

if the return value is negative it will stop and jump (goto) to a marked position "clean". it also expects a previously declared variable "in3_ret_t res".

```
#define TRY_GOTO (exp) {                            \
    res = (exp);            \
    if (res < 0) goto clean; \
  }
```

time_func

Pluggable functions: Mechanism to replace library functions with custom alternatives.

This is particularly useful for embedded systems which have their own time or rand functions.

eg. // define function with specified signature uint64_t my_time(void* t) { // … }

// then call in3_set_func_*() int main() { in3_set_func_time(my_time); // Henceforth, all library calls will use my_time() instead of the platform default time function } time function defaults to k_uptime_get() for zeohyr and time(NULL) for other platforms expected to return a u64 value representative of time (from epoch/start)

```
typedef uint64_t(* time_func) (void *t)
```

returns: `uint64_t(*`

rand_func

rand function defaults to k_uptime_get() for zeohyr and rand() for other platforms expected to return a random number

```
typedef int(* rand_func) (void *s)
```

returns: `int(*`

srand_func

srand function defaults to NOOP for zephyr and srand() for other platforms expected to set the seed for a new sequence of random numbers to be returned by in3_rand()

```
typedef void(* srand_func) (unsigned int s)
```

bytes_to_long

```
uint64_t bytes_to_long(const uint8_t *data, int len);
```

converts the bytes to a unsigned long (at least the last max len bytes)

arguments:

`const uint8_t *`	**data**
`int`	**len**

returns: `uint64_t`

bytes_to_int

```
static uint32_t bytes_to_int(const uint8_t *data, int len);
```

converts the bytes to a unsigned int (at least the last max len bytes)

arguments:

`const uint8_t *`	**data**
`int`	**len**

returns: `uint32_t`

char_to_long

```
uint64_t char_to_long(const char *a, int l);
```

converts a character into a uint64_t

arguments:

`const char *`	**a**
`int`	**l**

returns: `uint64_t`

hexchar_to_int

```
uint8_t hexchar_to_int(char c);
```

converts a hexchar to byte (4bit)

arguments:

`char`	**c**

returns: `uint8_t`

u64_to_str

```
const char* u64_to_str(uint64_t value, char *pBuf, int szBuf);
```

converts a uint64_t to string (char*); buffer-size min.

21 bytes

arguments:

`uint64_t`	**value**
`char *`	**pBuf**
`int`	**szBuf**

returns: `const char *`

hex_to_bytes

```
int hex_to_bytes(const char *hexdata, int hexlen, uint8_t *out, int outlen);
```

convert a c hex string to a byte array storing it into an existing buffer.

arguments:

`const char *`	**hexdata**
`int`	**hexlen**
`uint8_t *`	**out**
`int`	**outlen**

returns: `int`

hex_to_new_bytes

```
bytes_t* hex_to_new_bytes(const char *buf, int len);
```

convert a c string to a byte array creating a new buffer

arguments:

const char *	**buf**
int	**len**

returns: *bytes_t **

bytes_to_hex

```
int bytes_to_hex(const uint8_t *buffer, int len, char *out);
```

convefrts a bytes into hex

arguments:

const uint8_t *	**buffer**
int	**len**
char *	**out**

returns: int

sha3

```
bytes_t* sha3(const bytes_t *data);
```

hashes the bytes and creates a new bytes_t

arguments:

*bytes_tconst , **	**data**

returns: *bytes_t **

keccak

```
int keccak(bytes_t data, void *dst);
```

writes 32 bytes to the pointer.

arguments:

bytes_t	**data**
void *	**dst**

returns: int

long_to_bytes

```
void long_to_bytes(uint64_t val, uint8_t *dst);
```

converts a long to 8 bytes

arguments:

uint64_t	**val**
uint8_t *	**dst**

int_to_bytes

```
void int_to_bytes(uint32_t val, uint8_t *dst);
```

converts a int to 4 bytes

arguments:

uint32_t	**val**
uint8_t *	**dst**

_strdupn

```
char* _strdupn(const char *src, int len);
```

duplicate the string

arguments:

const char *	**src**
int	**len**

returns: `char *`

min_bytes_len

```
int min_bytes_len(uint64_t val);
```

calculate the min number of byte to represents the len

arguments:

uint64_t	**val**

returns: `int`

uint256_set

```
void uint256_set(const uint8_t *src, wlen_t src_len, bytes32_t dst);
```

sets a variable value to 32byte word.

arguments:

`const uint8_t *`	**src**
wlen_t	**src_len**
bytes32_t	**dst**

str_replace

```
char* str_replace(char *orig, const char *rep, const char *with);
```

replaces a string and returns a copy.

arguments:

`char *`	**orig**
`const char *`	**rep**
`const char *`	**with**

returns: `char *`

str_replace_pos

```
char* str_replace_pos(char *orig, size_t pos, size_t len, const char *rep);
```

replaces a string at the given position.

arguments:

`char *`	**orig**
`size_t`	**pos**
`size_t`	**len**
`const char *`	**rep**

returns: `char *`

str_find

```
char* str_find(char *haystack, const char *needle);
```

lightweight strstr() replacements

arguments:

`char *`	**haystack**
`const char *`	**needle**

returns: `char *`

str_remove_html

```
char* str_remove_html(char *data);
```

remove all html-tags in the text.

arguments:

`char *`	**data**

returns: `char *`

current_ms

```
uint64_t current_ms();
```

current timestamp in ms.

returns: `uint64_t`

memiszero

```
static bool memiszero(uint8_t *ptr, size_t l);
```

arguments:

`uint8_t *`	**ptr**
`size_t`	**l**

returns: `bool`

in3_set_func_time

```
void in3_set_func_time(time_func fn);
```

arguments:

time_func	**fn**

in3_time

```
uint64_t in3_time(void *t);
```

arguments:

`void *`	**t**

returns: `uint64_t`

in3_set_func_rand

```
void in3_set_func_rand(rand_func fn);
```

arguments:

rand_func	**fn**

in3_rand

```
int in3_rand(void *s);
```

arguments:

`void *`	**s**

returns: `int`

in3_set_func_srand

```
void in3_set_func_srand(srand_func fn);
```

arguments:

srand_func	**fn**

in3_srand

```
void in3_srand(unsigned int s);
```

arguments:

`unsigned int`	**s**

in3_sleep

```
void in3_sleep(uint32_t ms);
```

arguments:

`uint32_t`	**ms**

parse_float_val

```
int64_t parse_float_val(const char *data, int32_t expo);
```

parses a float-string and returns the value as int

arguments:

const char *	**data**	the data string
int32_t	**expo**	the exponent

returns: `int64_t`

9.9 Module pay

9.9.1 pay_eth.h

USN API.

This header-file defines easy to use function, which are verifying USN-Messages.

File: c/src/pay/eth/pay_eth.h

in3_pay_eth_config_t

The stuct contains following fields:

uint64_t	**bulk_size**
uint64_t	**max_price**
uint64_t	**nonce**
uint64_t	**gas_price**

in3_register_pay_eth

```
void in3_register_pay_eth();
```

pay_eth_configure

```
char* pay_eth_configure(in3_t *c, d_token_t *cconfig);
```

arguments:

*in3_t **	**c**
*d_token_t **	**cconfig**

returns: `char *`

9.9.2 zksync.h

ZKSync API.

This header-file registers zksync api functions.

File: c/src/pay/zksync/zksync.h

zk_msg_type

The enum type contains the following values:

ZK_TRANSFER	5
ZK_WITHDRAW	3

zk_msg_type_t

```
typedef enum zk_msg_type  zk_msg_type_t
```

in3_register_zksync

```
in3_ret_t in3_register_zksync(in3_t *c);
```

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

zksync_sign_transfer

```
in3_ret_t zksync_sign_transfer(sb_t *sb, zksync_tx_data_t *data, in3_ctx_t *ctx,
↪uint8_t *sync_key);
```

arguments:

*sb_t **	**sb**
*zksync_tx_data_t **	**data**
*in3_ctx_t **	**ctx**
`uint8_t *`	**sync_key**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

zksync_sign_change_pub_key

```
in3_ret_t zksync_sign_change_pub_key(sb_t *sb, in3_ctx_t *ctx, uint8_t *sync_pub_key,
↪uint32_t nonce, uint8_t *account, uint32_t account_id);
```

arguments:

*sb_t **	**sb**
*in3_ctx_t **	**ctx**
uint8_t *	**sync_pub_key**
uint32_t	**nonce**
uint8_t *	**account**
uint32_t	**account_id**

returns: in3_ret_t the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.10 Module signer

9.10.1 ethereum_apdu_client.h

this file defines the incubed configuration struct and it registration.

File: c/src/signer/ledger-nano/signer/ethereum_apdu_client.h

eth_ledger_set_signer_txn

```
in3_ret_t eth_ledger_set_signer_txn(in3_t *in3, uint8_t *bip_path);
```

attaches ledger nano hardware wallet signer with incubed .

bip32 path to be given to point the specific public/private key in HD tree for Ethereum!

arguments:

*in3_t **	**in3**
uint8_t *	**bip_path**

returns: in3_ret_t the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_ledger_get_public_addr

```
in3_ret_t eth_ledger_get_public_addr(uint8_t *i_bip_path, uint8_t *o_public_key);
```

returns public key at the bip_path .

returns IN3_ENODEVICE error if ledger nano device is not connected

arguments:

uint8_t *	**i_bip_path**
uint8_t *	**o_public_key**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.10.2 ledger_signer.h

this file defines the incubed configuration struct and it registration.

File: c/src/signer/ledger-nano/signer/ledger_signer.h

eth_ledger_set_signer

```
in3_ret_t eth_ledger_set_signer(in3_t *in3, uint8_t *bip_path);
```

attaches ledger nano hardware wallet signer with incubed .

bip32 path to be given to point the specific public/private key in HD tree for Ethereum!

arguments:

*in3_t **	**in3**
uint8_t *	**bip_path**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_ledger_get_public_key

```
in3_ret_t eth_ledger_get_public_key(uint8_t *bip_path, uint8_t *public_key);
```

returns public key at the bip_path .

returns IN3_ENODEVICE error if ledger nano device is not connected

arguments:

uint8_t *	**bip_path**
uint8_t *	**public_key**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.10.3 signer.h

Ethereum Nano verification.

File: c/src/signer/pk-signer/signer.h

hasher_t

The enum type contains the following values:

hasher_sha2	0
hasher_sha2d	1
hasher_sha2_ripemd	2
hasher_sha3	3
hasher_sha3k	4
hasher_blake	5
hasher_blaked	6
hasher_blake_ripemd	7
hasher_groestld_trunc	8
hasher_overwinter_prevouts	9
hasher_overwinter_sequence	10
hasher_overwinter_outputs	11
hasher_overwinter_preimage	12
hasher_sapling_preimage	13

eth_set_pk_signer

```
in3_ret_t eth_set_pk_signer(in3_t *in3, bytes32_t pk);
```

simply signer with one private key.

since the pk pointting to the 32 byte private key is not cloned, please make sure, you manage memory allocation correctly!

simply signer with one private key.

arguments:

*in3_t **	**in3**
bytes32_t	**pk**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_register_pk_signer

```
in3_ret_t eth_register_pk_signer(in3_t *in3);
```

registers pk signer as plugin so you can use config or in3_addKeys as rpc

arguments:

*in3_t **	**in3**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_set_request_signer

```
in3_ret_t eth_set_request_signer(in3_t *in3, bytes32_t pk);
```

sets the signer and a pk to the client

arguments:

*in3_t **	**in3**
bytes32_t	**pk**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_set_pk_signer_hex

```
void eth_set_pk_signer_hex(in3_t *in3, char *key);
```

simply signer with one private key as hex.

simply signer with one private key as hex.

arguments:

*in3_t **	**in3**
`char *`	**key**

ec_sign_pk_hash

```
in3_ret_t ec_sign_pk_hash(uint8_t *message, size_t len, uint8_t *pk, hasher_t hasher,␣
↪uint8_t *dst);
```

Signs message after hashing it with hasher function given in 'hasher_t', with the given private key.

Signs message after hashing it with hasher function given in 'hasher_t', with the given private key.

arguments:

`uint8_t *`	**message**
`size_t`	**len**
`uint8_t *`	**pk**
`hasher_t`	**hasher**
`uint8_t *`	**dst**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

ec_sign_pk_raw

```
in3_ret_t ec_sign_pk_raw(uint8_t *message, uint8_t *pk, uint8_t *dst);
```

Signs message raw with the given private key.

Signs message raw with the given private key.

arguments:

uint8_t *	**message**
uint8_t *	**pk**
uint8_t *	**dst**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.11 Module transport

9.11.1 in3_curl.h

transport-handler using libcurl.

File: c/src/transport/curl/in3_curl.h

send_curl

```
in3_ret_t send_curl(void *plugin_data, in3_plugin_act_t action, void *plugin_ctx);
```

a transport function using curl.

arguments:

void *	**plugin_data**
in3_plugin_act_t	**action**
void *	**plugin_ctx**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_register_curl

```
in3_ret_t in3_register_curl(in3_t *c);
```

registers curl as a default transport.

arguments:

*in3_t **	**c**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.11.2 in3_http.h

transport-handler using simple http.

File: c/src/transport/http/in3_http.h

send_http

```
in3_ret_t send_http(void *plugin_data, in3_plugin_act_t action, void *plugin_ctx);
```

a very simple transport function, which allows to send http-requests without a dependency to curl.

Here each request will be transformed to http instead of https.

You can use it by setting the transport-function-pointer in the in3_t->transport to this function:

```
#include <in3/in3_http.h>
...
c->transport = send_http;
```

arguments:

`void *`	**plugin_data**
in3_plugin_act_t	**action**
`void *`	**plugin_ctx**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_register_http

```
in3_ret_t in3_register_http(in3_t *c);
```

registers http as a default transport.

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.11.3 in3_winhttp.h

transport-handler using simple http.

File: c/src/transport/winhttp/in3_winhttp.h

send_winhttp

```
in3_ret_t send_winhttp(void *plugin_data, in3_plugin_act_t action, void *plugin_ctx);
```

arguments:

`void *`	**plugin_data**
in3_plugin_act_t	**action**
`void *`	**plugin_ctx**

returns: *`in3_ret_t`* the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

in3_register_winhttp

```
in3_ret_t in3_register_winhttp(in3_t *c);
```

registers http as a default transport.

arguments:

*in3_t **	**c**

returns: *`in3_ret_t`* the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

9.12 Module verifier

9.12.1 btc.h

Bitcoin verification.

File: c/src/verifier/btc/btc.h

in3_register_btc

```
in3_ret_t in3_register_btc(in3_t *c);
```

this function should only be called once and will register the bitcoin verifier.

arguments:

*in3_t **	**c**

returns: *`in3_ret_t`* the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

9.12.2 eth_basic.h

Ethereum Nanon verification.

File: c/src/verifier/eth1/basic/eth_basic.h

eth_verify_tx_values

```
in3_ret_t eth_verify_tx_values(in3_vctx_t *vc, d_token_t *tx, bytes_t *raw);
```

verifies internal tx-values.

arguments:

*in3_vctx_t **	**vc**
*d_token_t **	**tx**
*bytes_t **	**raw**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_eth_getTransaction

```
in3_ret_t eth_verify_eth_getTransaction(in3_vctx_t *vc, bytes_t *tx_hash);
```

verifies a transaction.

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**tx_hash**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_eth_getTransactionByBlock

```
in3_ret_t eth_verify_eth_getTransactionByBlock(in3_vctx_t *vc, d_token_t *blk, uint32_
↪t tx_idx);
```

verifies a transaction by block hash/number and id.

arguments:

*in3_vctx_t **	**vc**
*d_token_t **	**blk**
`uint32_t`	**tx_idx**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_account_proof

```
in3_ret_t eth_verify_account_proof(in3_vctx_t *vc);
```

verify account-proofs

arguments:

*in3_vctx_t **	**vc**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_eth_getBlock

```
in3_ret_t eth_verify_eth_getBlock(in3_vctx_t *vc, bytes_t *block_hash, uint64_t␣
↪blockNumber);
```

verifies a block

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**block_hash**
`uint64_t`	**blockNumber**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_eth_getBlockTransactionCount

```
in3_ret_t eth_verify_eth_getBlockTransactionCount(in3_vctx_t *vc, bytes_t *block_hash,
↪ uint64_t blockNumber);
```

verifies block transaction count by number or hash

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**block_hash**
`uint64_t`	**blockNumber**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_register_eth_basic

```
in3_ret_t in3_register_eth_basic(in3_t *c);
```

this function should only be called once and will register the eth-nano verifier.

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_verify_eth_getLog

```
in3_ret_t eth_verify_eth_getLog(in3_vctx_t *vc, int l_logs);
```

verify logs

arguments:

*in3_vctx_t **	**vc**
`int`	**l_logs**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_prepare_unsigned_tx

```
in3_ret_t eth_prepare_unsigned_tx(d_token_t *tx, in3_ctx_t *ctx, bytes_t *dst);
```

prepares a transaction and writes the data to the dst-bytes.

In case of success, you MUST free only the data-pointer of the dst.

arguments:

*d_token_t **	**tx**	a json-token desribing the transaction
*in3_ctx_t **	**ctx**	the current context
*bytes_t **	**dst**	the bytes to write the result to.

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

eth_sign_raw_tx

```
in3_ret_t eth_sign_raw_tx(bytes_t raw_tx, in3_ctx_t *ctx, address_t from, bytes_t␣
↪*dst);
```

signs a unsigned raw transaction and writes the raw data to the dst-bytes.

In case of success, you MUST free only the data-pointer of the dst.

arguments:

bytes_t	**raw_tx**	the unsigned raw transaction to sign
*in3_ctx_t **	**ctx**	the current context
address_t	**from**	the address of the account to sign with
*bytes_t **	**dst**	the bytes to write the result to.

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

handle_eth_sendTransaction

```
in3_ret_t handle_eth_sendTransaction(in3_ctx_t *ctx, d_token_t *req);
```

expects a req-object for a transaction and converts it into a sendRawTransaction after signing.

expects a req-object for a transaction and converts it into a sendRawTransaction after signing.

arguments:

*in3_ctx_t **	**ctx**	the current context
*d_token_t **	**req**	the request

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

eth_wallet_sign

```
RETURNS_NONULL NONULL char* eth_wallet_sign(const char *key, const char *data);
```

minimum signer for the wallet, returns the signed message which needs to be freed

arguments:

`const char *`	**key**
`const char *`	**data**

returns: `RETURNS_NONULL NONULL char *`

9.12.3 trie.h

Patricia Merkle Tree Imnpl

File: c/src/verifier/eth1/basic/trie.h

trie_node_type_t

Node types.

The enum type contains the following values:

NODE_EMPTY	0	empty node
NODE_BRANCH	1	a Branch
NODE_LEAF	2	a leaf containing the value.
NODE_EXT	3	a extension

in3_hasher_t

hash-function

```
typedef void(* in3_hasher_t) (bytes_t *src, uint8_t *dst)
```

in3_codec_add_t

codec to organize the encoding of the nodes

```
typedef void(* in3_codec_add_t) (bytes_builder_t *bb, bytes_t *val)
```

in3_codec_finish_t

```
typedef void(* in3_codec_finish_t) (bytes_builder_t *bb, bytes_t *dst)
```

in3_codec_decode_size_t

```
typedef int(* in3_codec_decode_size_t) (bytes_t *src)
```

returns: `int(*`

in3_codec_decode_index_t

```
typedef int(* in3_codec_decode_index_t) (bytes_t *src, int index, bytes_t *dst)
```

returns: `int(*`

trie_node_t

single node in the merkle trie.

The stuct contains following fields:

`uint8_t`	**hash**	the hash of the node
bytes_t	**data**	the raw data
bytes_t	**items**	the data as list
`uint8_t`	**own_memory**	if true this is a embedded node with own memory
trie_node_type_t	**type**	type of the node
trie_nodestruct , *	**next**	used as linked list

trie_codec_t

the codec used to encode nodes.

The stuct contains following fields:

in3_codec_add_t	**encode_add**
`in3_codec_finish_t`	**encode_finish**
`in3_codec_decode_size_t`	**decode_size**
`in3_codec_decode_index_t`	**decode_item**

trie_t

a merkle trie implementation.

This is a Patricia Merkle Tree.

The stuct contains following fields:

in3_hasher_t	**hasher**	hash-function.
*trie_codec_t **	**codec**	encoding of the nocds.
bytes32_t	**root**	The root-hash.
*trie_node_t **	**nodes**	linked list of containes nodes

trie_new

```
trie_t* trie_new();
```

creates a new Merkle Trie.

returns: `trie_t *`

trie_free

```
void trie_free(trie_t *val);
```

frees all resources of the trie.

arguments:

*trie_t **	**val**

trie_set_value

```
void trie_set_value(trie_t *t, bytes_t *key, bytes_t *value);
```

sets a value in the trie.

The root-hash will be updated automaticly.

arguments:

*trie_t **	**t**
*bytes_t **	**key**
*bytes_t **	**value**

9.12.4 big.h

Ethereum Nanon verification.

File: c/src/verifier/eth1/evm/big.h

big_is_zero

```
uint8_t big_is_zero(uint8_t *data, wlen_t l);
```

arguments:

uint8_t *	**data**
wlen_t	**l**

returns: uint8_t

big_shift_left

```
void big_shift_left(uint8_t *a, wlen_t len, int bits);
```

arguments:

uint8_t *	**a**
wlen_t	**len**
int	**bits**

big_shift_right

```
void big_shift_right(uint8_t *a, wlen_t len, int bits);
```

arguments:

uint8_t *	**a**
wlen_t	**len**
int	**bits**

big_cmp

```
int big_cmp(const uint8_t *a, const wlen_t len_a, const uint8_t *b, const wlen_t len_
↪b);
```

arguments:

`const uint8_t *`	**a**
wlen_tconst	**len_a**
`const uint8_t *`	**b**
wlen_tconst	**len_b**

returns: `int`

big_signed

```
int big_signed(uint8_t *val, wlen_t len, uint8_t *dst);
```

returns 0 if the value is positive or 1 if negavtive.

in this case the absolute value is copied to dst.

arguments:

`uint8_t *`	**val**
wlen_t	**len**
`uint8_t *`	**dst**

returns: `int`

big_int

```
int32_t big_int(uint8_t *val, wlen_t len);
```

arguments:

`uint8_t *`	**val**
wlen_t	**len**

returns: `int32_t`

big_add

```
int big_add(uint8_t *a, wlen_t len_a, uint8_t *b, wlen_t len_b, uint8_t *out, wlen_t␣
↪max);
```

arguments:

`uint8_t *`	**a**
wlen_t	**len_a**
`uint8_t *`	**b**
wlen_t	**len_b**
`uint8_t *`	**out**
wlen_t	**max**

returns: `int`

big_sub

```
int big_sub(uint8_t *a, wlen_t len_a, uint8_t *b, wlen_t len_b, uint8_t *out);
```

arguments:

uint8_t *	**a**
wlen_t	**len_a**
uint8_t *	**b**
wlen_t	**len_b**
uint8_t *	**out**

returns: `int`

big_mul

```
int big_mul(uint8_t *a, wlen_t la, uint8_t *b, wlen_t lb, uint8_t *res, wlen_t max);
```

arguments:

uint8_t *	**a**
wlen_t	**la**
uint8_t *	**b**
wlen_t	**lb**
uint8_t *	**res**
wlen_t	**max**

returns: `int`

big_div

```
int big_div(uint8_t *a, wlen_t la, uint8_t *b, wlen_t lb, wlen_t sig, uint8_t *res);
```

arguments:

uint8_t *	**a**
wlen_t	**la**
uint8_t *	**b**
wlen_t	**lb**
wlen_t	**sig**
uint8_t *	**res**

returns: `int`

big_mod

```
int big_mod(uint8_t *a, wlen_t la, uint8_t *b, wlen_t lb, wlen_t sig, uint8_t *res);
```

arguments:

uint8_t *	**a**
wlen_t	**la**
uint8_t *	**b**
wlen_t	**lb**
wlen_t	**sig**
uint8_t *	**res**

returns: int

big_exp

```
int big_exp(uint8_t *a, wlen_t la, uint8_t *b, wlen_t lb, uint8_t *res);
```

arguments:

uint8_t *	**a**
wlen_t	**la**
uint8_t *	**b**
wlen_t	**lb**
uint8_t *	**res**

returns: int

big_log256

```
int big_log256(uint8_t *a, wlen_t len);
```

arguments:

uint8_t *	**a**
wlen_t	**len**

returns: int

9.12.5 code.h

code cache.

File: c/src/verifier/eth1/evm/code.h

in3_get_code

```
in3_ret_t in3_get_code(in3_vctx_t *vc, address_t address, cache_entry_t **target);
```

fetches the code and adds it to the context-cache as cache_entry.

So calling this function a second time will take the result from cache.

arguments:

*in3_vctx_t **	**vc**
address_t	**address**
*cache_entry_t ***	**target**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

9.12.6 evm.h

main evm-file.

File: c/src/verifier/eth1/evm/evm.h

gas_options

EVM_ERROR_EMPTY_STACK

the no more elements on the stack

```
#define EVM_ERROR_EMPTY_STACK -20
```

EVM_ERROR_INVALID_OPCODE

the opcode is not supported

```
#define EVM_ERROR_INVALID_OPCODE -21
```

EVM_ERROR_BUFFER_TOO_SMALL

reading data from a position, which is not initialized

```
#define EVM_ERROR_BUFFER_TOO_SMALL -22
```

EVM_ERROR_ILLEGAL_MEMORY_ACCESS

the memory-offset does not exist

```
#define EVM_ERROR_ILLEGAL_MEMORY_ACCESS -23
```

EVM_ERROR_INVALID_JUMPDEST

the jump destination is not marked as valid destination

```
#define EVM_ERROR_INVALID_JUMPDEST -24
```

EVM_ERROR_INVALID_PUSH

the push data is empy

```
#define EVM_ERROR_INVALID_PUSH -25
```

EVM_ERROR_UNSUPPORTED_CALL_OPCODE

error handling the call, usually because static-calls are not allowed to change state

```
#define EVM_ERROR_UNSUPPORTED_CALL_OPCODE -26
```

EVM_ERROR_TIMEOUT

the evm ran into a loop

```
#define EVM_ERROR_TIMEOUT -27
```

EVM_ERROR_INVALID_ENV

the enviroment could not deliver the data

```
#define EVM_ERROR_INVALID_ENV -28
```

EVM_ERROR_OUT_OF_GAS

not enough gas to exewcute the opcode

```
#define EVM_ERROR_OUT_OF_GAS -29
```

EVM_ERROR_BALANCE_TOO_LOW

not enough funds to transfer the requested value.

```
#define EVM_ERROR_BALANCE_TOO_LOW -30
```

EVM_ERROR_STACK_LIMIT

stack limit reached

```
#define EVM_ERROR_STACK_LIMIT -31
```

EVM_ERROR_SUCCESS_CONSUME_GAS

write success but consume all gas

```
#define EVM_ERROR_SUCCESS_CONSUME_GAS -32
```

EVM_PROP_FRONTIER

```
#define EVM_PROP_FRONTIER 1
```

EVM_PROP_EIP150

```
#define EVM_PROP_EIP150 2
```

EVM_PROP_EIP158

```
#define EVM_PROP_EIP158 4
```

EVM_PROP_CONSTANTINOPL

```
#define EVM_PROP_CONSTANTINOPL 16
```

EVM_PROP_ISTANBUL

```
#define EVM_PROP_ISTANBUL 32
```

EVM_PROP_NO_FINALIZE

```
#define EVM_PROP_NO_FINALIZE 32768
```

EVM_PROP_STATIC

```
#define EVM_PROP_STATIC 256
```

EVM_ENV_BALANCE

```
#define EVM_ENV_BALANCE 1
```

EVM_ENV_CODE_SIZE

```
#define EVM_ENV_CODE_SIZE 2
```

EVM_ENV_CODE_COPY

```
#define EVM_ENV_CODE_COPY 3
```

EVM_ENV_BLOCKHASH

```
#define EVM_ENV_BLOCKHASH 4
```

EVM_ENV_STORAGE

```
#define EVM_ENV_STORAGE 5
```

EVM_ENV_BLOCKHEADER

```
#define EVM_ENV_BLOCKHEADER 6
```

EVM_ENV_CODE_HASH

```
#define EVM_ENV_CODE_HASH 7
```

EVM_ENV_NONCE

```
#define EVM_ENV_NONCE 8
```

MATH_ADD

```
#define MATH_ADD 1
```

MATH_SUB

```
#define MATH_SUB 2
```

MATH_MUL

```
#define MATH_MUL 3
```

MATH_DIV

```
#define MATH_DIV 4
```

MATH_SDIV

```
#define MATH_SDIV 5
```

MATH_MOD

```
#define MATH_MOD 6
```

MATH_SMOD

```
#define MATH_SMOD 7
```

MATH_EXP

```
#define MATH_EXP 8
```

MATH_SIGNEXP

```
#define MATH_SIGNEXP 9
```

CALL_CALL

```
#define CALL_CALL 0
```

CALL_CODE

```
#define CALL_CODE 1
```

CALL_DELEGATE

```
#define CALL_DELEGATE 2
```

CALL_STATIC

```
#define CALL_STATIC 3
```

OP_AND

```
#define OP_AND 0
```

OP_OR

```
#define OP_OR 1
```

OP_XOR

```
#define OP_XOR 2
```

EVM_DEBUG_BLOCK (...)

OP_LOG (...)

```
#define OP_LOG (...) EVM_ERROR_UNSUPPORTED_CALL_OPCODE
```

OP_SLOAD_GAS (...)

OP_CREATE (...)

```
#define OP_CREATE (...) EVM_ERROR_UNSUPPORTED_CALL_OPCODE
```

OP_ACCOUNT_GAS (...)

```
#define OP_ACCOUNT_GAS (...) exit_zero()
```

OP_SELFDESTRUCT (...)

```
#define OP_SELFDESTRUCT (...) EVM_ERROR_UNSUPPORTED_CALL_OPCODE
```

OP_EXTCODECOPY_GAS (evm)

OP_SSTORE (...)

```
#define OP_SSTORE (...) EVM_ERROR_UNSUPPORTED_CALL_OPCODE
```

EVM_CALL_MODE_STATIC

```
#define EVM_CALL_MODE_STATIC 1
```

EVM_CALL_MODE_DELEGATE

```
#define EVM_CALL_MODE_DELEGATE 2
```

EVM_CALL_MODE_CALLCODE

```
#define EVM_CALL_MODE_CALLCODE 3
```

EVM_CALL_MODE_CALL

```
#define EVM_CALL_MODE_CALL 4
```

evm_state

the current state of the evm

The enum type contains the following values:

EVM_STATE_INIT	0	just initialised, but not yet started
EVM_STATE_RUNNING	1	started and still running
EVM_STATE_STOPPED	2	successfully stopped
EVM_STATE_REVERTED	3	stopped, but results must be reverted

evm_state_t

the current state of the evm

The enum type contains the following values:

EVM_STATE_INIT	0	just initialised, but not yet started
EVM_STATE_RUNNING	1	started and still running
EVM_STATE_STOPPED	2	successfully stopped
EVM_STATE_REVERTED	3	stopped, but results must be reverted

evm_get_env

This function provides data from the enviroment.

depending on the key the function will set the out_data-pointer to the result. This means the enviroment is responsible for memory management and also to clean up resources afterwards.

```
typedef int(* evm_get_env) (void *evm, uint16_t evm_key, uint8_t *in_data, int in_len,
↪ uint8_t **out_data, int offset, int len)
```

returns: `int(*`

storage_t

The stuct contains following fields:

bytes32_t	**key**
bytes32_t	**value**
*account_storagestruct , **	**next**

logs_t

The stuct contains following fields:

bytes_t	**topics**
bytes_t	**data**
*logsstruct , **	**next**

account_t

The stuct contains following fields:

address_t	**address**
bytes32_t	**balance**
bytes32_t	**nonce**
bytes_t	**code**
*storage_t **	**storage**
*accountstruct , **	**next**

evm_t

The stuct contains following fields:

bytes_builder_t	**stack**	
bytes_builder_t	**memory**	
int	**stack_size**	
bytes_t	**code**	
uint32_t	**pos**	
evm_state_t	**state**	
bytes_t	**last_returned**	
bytes_t	**return_data**	
uint32_t *	**invalid_jumpdest**	
uint32_t	**properties**	
evm_get_env	**env**	
void *	**env_ptr**	
uint64_t	**chain_id**	the chain_id as returned by the opcode
uint8_t *	**address**	the address of the current storage
uint8_t *	**account**	the address of the code
uint8_t *	**origin**	the address of original sender of the root-transaction
uint8_t *	**caller**	the address of the parent sender
bytes_t	**call_value**	value send
bytes_t	**call_data**	data send in the tx
bytes_t	**gas_price**	current gasprice
uint64_t	**gas**	
	gas_options	

exit_zero

```
int exit_zero(void);
```

arguments:

void

returns: int

evm_stack_push

```
int evm_stack_push(evm_t *evm, uint8_t *data, uint8_t len);
```

arguments:

evm_t *	**evm**
uint8_t *	**data**
uint8_t	**len**

returns: int

evm_stack_push_ref

```
int evm_stack_push_ref(evm_t *evm, uint8_t **dst, uint8_t len);
```

arguments:

*evm_t **	**evm**
uint8_t **	**dst**
uint8_t	**len**

returns: int

evm_stack_push_int

```
int evm_stack_push_int(evm_t *evm, uint32_t val);
```

arguments:

*evm_t **	**evm**
uint32_t	**val**

returns: int

evm_stack_push_long

```
int evm_stack_push_long(evm_t *evm, uint64_t val);
```

arguments:

*evm_t **	**evm**
uint64_t	**val**

returns: int

evm_stack_get_ref

```
int evm_stack_get_ref(evm_t *evm, uint8_t pos, uint8_t **dst);
```

arguments:

*evm_t **	**evm**
uint8_t	**pos**
uint8_t **	**dst**

returns: int

evm_stack_pop

```
int evm_stack_pop(evm_t *evm, uint8_t *dst, uint8_t len);
```

arguments:

evm_t *	**evm**
uint8_t *	**dst**
uint8_t	**len**

returns: int

evm_stack_pop_ref

```
int evm_stack_pop_ref(evm_t *evm, uint8_t **dst);
```

arguments:

evm_t *	**evm**
uint8_t **	**dst**

returns: int

evm_stack_pop_byte

```
int evm_stack_pop_byte(evm_t *evm, uint8_t *dst);
```

arguments:

evm_t *	**evm**
uint8_t *	**dst**

returns: int

evm_stack_pop_int

```
int32_t evm_stack_pop_int(evm_t *evm);
```

arguments:

evm_t *	**evm**

returns: int32_t

evm_run

```
int evm_run(evm_t *evm, address_t code_address);
```

arguments:

*evm_t **	**evm**
address_t	**code_address**

returns: `int`

evm_sub_call

```
int evm_sub_call(evm_t *parent, uint8_t address[20], uint8_t account[20], uint8_t␣
↪*value, wlen_t l_value, uint8_t *data, uint32_t l_data, uint8_t caller[20], uint8_t␣
↪origin[20], uint64_t gas, wlen_t mode, uint32_t out_offset, uint32_t out_len);
```

handle internal calls.

arguments:

*evm_t **	**parent**
`uint8_t`	**address**
`uint8_t`	**account**
`uint8_t *`	**value**
wlen_t	**l_value**
`uint8_t *`	**data**
`uint32_t`	**l_data**
`uint8_t`	**caller**
`uint8_t`	**origin**
`uint64_t`	**gas**
wlen_t	**mode**
`uint32_t`	**out_offset**
`uint32_t`	**out_len**

returns: `int`

evm_ensure_memory

```
int evm_ensure_memory(evm_t *evm, uint32_t max_pos);
```

arguments:

*evm_t **	**evm**
`uint32_t`	**max_pos**

returns: `int`

in3_get_env

```
int in3_get_env(void *evm_ptr, uint16_t evm_key, uint8_t *in_data, int in_len, uint8_
↪t **out_data, int offset, int len);
```

arguments:

`void *`	**evm_ptr**
`uint16_t`	**evm_key**
`uint8_t *`	**in_data**
`int`	**in_len**
`uint8_t **`	**out_data**
`int`	**offset**
`int`	**len**

returns: `int`

evm_call

```
int evm_call(void *vc, uint8_t address[20], uint8_t *value, wlen_t l_value, uint8_t␣
↪*data, uint32_t l_data, uint8_t caller[20], uint64_t gas, uint64_t chain_id, bytes_
↪t **result);
```

run a evm-call

arguments:

`void *`	**vc**
`uint8_t`	**address**
`uint8_t *`	**value**
wlen_t	**l_value**
`uint8_t *`	**data**
`uint32_t`	**l_data**
`uint8_t`	**caller**
`uint64_t`	**gas**
`uint64_t`	**chain_id**
*bytes_t ***	**result**

returns: `int`

evm_print_stack

```
void evm_print_stack(evm_t *evm, uint64_t last_gas, uint32_t pos);
```

arguments:

*evm_t **	**evm**
`uint64_t`	**last_gas**
`uint32_t`	**pos**

evm_free

```
void evm_free(evm_t *evm);
```

arguments:

*evm_t **	**evm**

evm_execute

```
int evm_execute(evm_t *evm);
```

arguments:

*evm_t **	**evm**

returns: `int`

9.12.7 gas.h

evm gas defines.

File: c/src/verifier/eth1/evm/gas.h

op_exec (m,gas)

```
#define op_exec (m,gas) return m;
```

subgas (g)

GAS_CC_NET_SSTORE_NOOP_GAS

Once per SSTORE operation if the value doesn't change.

```
#define GAS_CC_NET_SSTORE_NOOP_GAS 200
```

GAS_CC_NET_SSTORE_INIT_GAS

Once per SSTORE operation from clean zero.

```
#define GAS_CC_NET_SSTORE_INIT_GAS 20000
```

GAS_CC_NET_SSTORE_CLEAN_GAS

Once per SSTORE operation from clean non-zero.

```
#define GAS_CC_NET_SSTORE_CLEAN_GAS 5000
```

GAS_CC_NET_SSTORE_DIRTY_GAS

Once per SSTORE operation from dirty.

```
#define GAS_CC_NET_SSTORE_DIRTY_GAS 200
```

GAS_CC_NET_SSTORE_CLEAR_REFUND

Once per SSTORE operation for clearing an originally existing storage slot.

```
#define GAS_CC_NET_SSTORE_CLEAR_REFUND 15000
```

GAS_CC_NET_SSTORE_RESET_REFUND

Once per SSTORE operation for resetting to the original non-zero value.

```
#define GAS_CC_NET_SSTORE_RESET_REFUND 4800
```

GAS_CC_NET_SSTORE_RESET_CLEAR_REFUND

Once per SSTORE operation for resetting to the original zero valuev.

```
#define GAS_CC_NET_SSTORE_RESET_CLEAR_REFUND 19800
```

G_ZERO

Nothing is paid for operations of the set Wzero.

```
#define G_ZERO 0
```

G_JUMPDEST

JUMP DEST.

```
#define G_JUMPDEST 1
```

G_BASE

This is the amount of gas to pay for operations of the set Wbase.

```
#define G_BASE 2
```

G_VERY_LOW

This is the amount of gas to pay for operations of the set Wverylow.

```
#define G_VERY_LOW 3
```

G_LOW

This is the amount of gas to pay for operations of the set Wlow.

```
#define G_LOW 5
```

G_MID

This is the amount of gas to pay for operations of the set Wmid.

```
#define G_MID 8
```

G_HIGH

This is the amount of gas to pay for operations of the set Whigh.

```
#define G_HIGH 10
```

G_EXTCODE

This is the amount of gas to pay for operations of the set Wextcode.

```
#define G_EXTCODE 700
```

G_BALANCE

This is the amount of gas to pay for a BALANCE operation.

```
#define G_BALANCE 400
```

G_SLOAD

This is paid for an SLOAD operation.

```
#define G_SLOAD 200
```

G_SSET

This is paid for an SSTORE operation when the storage value is set to non-zero from zero.

```
#define G_SSET 20000
```

G_SRESET

This is the amount for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.

```
#define G_SRESET 5000
```

R_SCLEAR

This is the refund given (added into the refund counter) when the storage value is set to zero from non-zero.

```
#define R_SCLEAR 15000
```

R_SELFDESTRUCT

This is the refund given (added into the refund counter) for self-destructing an account.

```
#define R_SELFDESTRUCT 24000
```

G_SELFDESTRUCT

This is the amount of gas to pay for a SELFDESTRUCT operation.

```
#define G_SELFDESTRUCT 5000
```

G_CREATE

This is paid for a CREATE operation.

```
#define G_CREATE 32000
```

G_CODEDEPOSIT

This is paid per byte for a CREATE operation to succeed in placing code into the state.

```
#define G_CODEDEPOSIT 200
```

G_CALL

This is paid for a CALL operation.

```
#define G_CALL 700
```

G_CALLVALUE

This is paid for a non-zero value transfer as part of the CALL operation.

```
#define G_CALLVALUE 9000
```

G_CALLSTIPEND

This is a stipend for the called contract subtracted from Gcallvalue for a non-zero value transfer.

```
#define G_CALLSTIPEND 2300
```

G_NEWACCOUNT

This is paid for a CALL or for a SELFDESTRUCT operation which creates an account.

```
#define G_NEWACCOUNT 25000
```

G_EXP

This is a partial payment for an EXP operation.

```
#define G_EXP 10
```

G_EXPBYTE

This is a partial payment when multiplied by dlog256(exponent)e for the EXP operation.

```
#define G_EXPBYTE 50
```

G_MEMORY

This is paid for every additional word when expanding memory.

```
#define G_MEMORY 3
```

G_TXCREATE

This is paid by all contract-creating transactions after the Homestead transition.

```
#define G_TXCREATE 32000
```

G_TXDATA_ZERO

This is paid for every zero byte of data or code for a transaction.

```
#define G_TXDATA_ZERO 4
```

G_TXDATA_NONZERO

This is paid for every non-zero byte of data or code for a transaction.

```
#define G_TXDATA_NONZERO 68
```

G_TRANSACTION

This is paid for every transaction.

```
#define G_TRANSACTION 21000
```

G_LOG

This is a partial payment for a LOG operation.

```
#define G_LOG 375
```

G_LOGDATA

This is paid for each byte in a LOG operation's data.

```
#define G_LOGDATA 8
```

G_LOGTOPIC

This is paid for each topic of a LOG operation.

```
#define G_LOGTOPIC 375
```

G_SHA3

This is paid for each SHA3 operation.

```
#define G_SHA3 30
```

G_SHA3WORD

This is paid for each word (rounded up) for input data to a SHA3 operation.

```
#define G_SHA3WORD 6
```

G_COPY

This is a partial payment for *COPY operations, multiplied by the number of words copied, rounded up.

```
#define G_COPY 3
```

G_BLOCKHASH

This is a payment for a BLOCKHASH operation.

```
#define G_BLOCKHASH 20
```

G_PRE_EC_RECOVER

Precompile EC RECOVER.

```
#define G_PRE_EC_RECOVER 3000
```

G_PRE_SHA256

Precompile SHA256.

```
#define G_PRE_SHA256 60
```

G_PRE_SHA256_WORD

Precompile SHA256 per word.

```
#define G_PRE_SHA256_WORD 12
```

G_PRE_RIPEMD160

Precompile RIPEMD160.

```
#define G_PRE_RIPEMD160 600
```

G_PRE_RIPEMD160_WORD

Precompile RIPEMD160 per word.

```
#define G_PRE_RIPEMD160_WORD 120
```

G_PRE_IDENTITY

Precompile IDENTIY (copyies data)

```
#define G_PRE_IDENTITY 15
```

G_PRE_IDENTITY_WORD

Precompile IDENTIY per word.

```
#define G_PRE_IDENTITY_WORD 3
```

G_PRE_MODEXP_GQUAD_DIVISOR

Gquaddivisor from modexp precompile for gas calculation.

```
#define G_PRE_MODEXP_GQUAD_DIVISOR 20
```

G_PRE_ECADD

Gas costs for curve addition precompile.

```
#define G_PRE_ECADD 500
```

G_PRE_ECMUL

Gas costs for curve multiplication precompile.

```
#define G_PRE_ECMUL 40000
```

G_PRE_ECPAIRING

Base gas costs for curve pairing precompile.

```
#define G_PRE_ECPAIRING 100000
```

G_PRE_ECPAIRING_WORD

Gas costs regarding curve pairing precompile input length.

```
#define G_PRE_ECPAIRING_WORD 80000
```

EVM_STACK_LIMIT

max elements of the stack

```
#define EVM_STACK_LIMIT 1024
```

EVM_MAX_CODE_SIZE

max size of the code

```
#define EVM_MAX_CODE_SIZE 24576
```

FRONTIER_G_EXPBYTE

fork values

This is a partial payment when multiplied by dlog256(exponent)e for the EXP operation.

```
#define FRONTIER_G_EXPBYTE 10
```

FRONTIER_G_SLOAD

This is a partial payment when multiplied by dlog256(exponent)e for the EXP operation.

```
#define FRONTIER_G_SLOAD 50
```

FREE_EVM (. . .)

INIT_EVM (. . .)

INIT_GAS (. . .)

SUBGAS (. . .)

FINALIZE_SUBCALL_GAS (. . .)

UPDATE_SUBCALL_GAS (. . .)

FINALIZE_AND_REFUND_GAS (. . .)

KEEP_TRACK_GAS (evm)

```
#define KEEP_TRACK_GAS (evm) 0
```

UPDATE_ACCOUNT_CODE (. . .)

9.12.8 eth_full.h

Ethereum Nanon verification.

File: c/src/verifier/eth1/full/eth_full.h

in3_register_eth_full

```
in3_ret_t in3_register_eth_full(in3_t *c);
```

this function should only be called once and will register the eth-full verifier.

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

9.12.9 chainspec.h

Ethereum chain specification

File: c/src/verifier/eth1/nano/chainspec.h

BLOCK_LATEST

```
#define BLOCK_LATEST 0xFFFFFFFFFFFFFFFF
```

eth_consensus_type_t

the consensus type.

The enum type contains the following values:

ETH_POW	0	Pro of Work (Ethash)
ETH_POA_AURA	1	Proof of Authority using Aura.
ETH_POA_CLIQUE	2	Proof of Authority using clique.

eip_transition_t

The stuct contains following fields:

`uint64_t`	**transition_block**
`eip_t`	**eips**

consensus_transition_t

The stuct contains following fields:

`uint64_t`	**transition_block**
eth_consensus_type_t	**type**
bytes_t	**validators**
`uint8_t *`	**contract**

chainspec_t

The stuct contains following fields:

`uint64_t`	**network_id**
`uint64_t`	**account_start_nonce**
`uint32_t`	**eip_transitions_len**
*eip_transition_t **	**eip_transitions**
`uint32_t`	**consensus_transitions_len**
*consensus_transition_t **	**consensus_transitions**

attribute

```
struct __attribute__((__packed__)) eip_;
```

defines the flags for the current activated EIPs.

Since it does not make sense to support a evm defined before Homestead, homestead EIP is always turned on!

< REVERT instruction

< Bitwise shifting instructions in EVM

< Gas cost changes for IO-heavy operations

< Simple replay attack protection

< EXP cost increase

< Contract code size limit

< Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128

< Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128

< Big integer modular exponentiation

< New opcodes: RETURNDATASIZE and RETURNDATACOPY

< New opcode STATICCALL

< Embedding transaction status code in receipts

< Skinny CREATE2

< EXTCODEHASH opcode

< Net gas metering for SSTORE without dirty maps

arguments:

(__packed__)

returns: `struct`

chainspec_create_from_json

```
chainspec_t* chainspec_create_from_json(json_ctx_t *data);
```

arguments:

*json_ctx_t **	**data**

returns: *`chainspec_t *`*

chainspec_get_eip

```
eip_t chainspec_get_eip(chainspec_t *spec, uint64_t block_number);
```

arguments:

*chainspec_t **	**spec**
`uint64_t`	**block_number**

returns: eip_t

chainspec_get_consensus

```
consensus_transition_t* chainspec_get_consensus(chainspec_t *spec, uint64_t block_
↪number);
```

arguments:

*chainspec_t **	**spec**
uint64_t	**block_number**

returns: *consensus_transition_t **

chainspec_to_bin

```
in3_ret_t chainspec_to_bin(chainspec_t *spec, bytes_builder_t *bb);
```

arguments:

*chainspec_t **	**spec**
*bytes_builder_t **	**bb**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

chainspec_from_bin

```
chainspec_t* chainspec_from_bin(void *raw);
```

arguments:

void *	**raw**

returns: *chainspec_t **

chainspec_get

```
chainspec_t* chainspec_get(chain_id_t chain_id);
```

arguments:

chain_id_t	**chain_id**

returns: *chainspec_t **

chainspec_put

```
void chainspec_put(chain_id_t chain_id, chainspec_t *spec);
```

arguments:

chain_id_t	**chain_id**
*chainspec_t **	**spec**

9.12.10 eth_nano.h

Ethereum Nanon verification.

File: c/src/verifier/eth1/nano/eth_nano.h

in3_verify_eth_nano

```
NONULL in3_ret_t in3_verify_eth_nano(void *p_data, in3_plugin_act_t action, void␣
↪*pctx);
```

entry-function to execute the verification context.

arguments:

`void *`	**p_data**
in3_plugin_act_t	**action**
`void *`	**pctx**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_blockheader

```
in3_ret_t eth_verify_blockheader(in3_vctx_t *vc, bytes_t *header, bytes_t *expected_
↪blockhash);
```

verifies a blockheader.

verifies a blockheader.

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**header**
*bytes_t **	**expected_blockhash**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_signature

```
NONULL int eth_verify_signature(in3_vctx_t *vc, bytes_t *msg_hash, d_token_t *sig);
```

verifies a single signature blockheader.

This function will return a positive integer with a bitmask holding the bit set according to the address that signed it. This is based on the signatiures in the request-config.

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**msg_hash**
*d_token_t **	**sig**

returns: `NONULL int`

ecrecover_signature

```
NONULL bytes_t* ecrecover_signature(bytes_t *msg_hash, d_token_t *sig);
```

returns the address of the signature if the msg_hash is correct

arguments:

*bytes_t **	**msg_hash**
*d_token_t **	**sig**

returns: `bytes_tNONULL , *`

eth_verify_eth_getTransactionReceipt

```
NONULL in3_ret_t eth_verify_eth_getTransactionReceipt(in3_vctx_t *vc, bytes_t *tx_
↪hash);
```

verifies a transaction receipt.

arguments:

*in3_vctx_t **	**vc**
*bytes_t **	**tx_hash**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

eth_verify_in3_nodelist

```
in3_ret_t eth_verify_in3_nodelist(in3_vctx_t *vc, uint32_t node_limit, bytes_t *seed,␣
↪d_token_t *required_addresses);
```

verifies the nodelist.

arguments:

*in3_vctx_t **	**vc**
`uint32_t`	**node_limit**
*bytes_t **	**seed**
*d_token_t **	**required_addresses**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

eth_verify_in3_whitelist

```
NONULL in3_ret_t eth_verify_in3_whitelist(in3_vctx_t *vc);
```

verifies the nodelist.

arguments:

*in3_vctx_t **	**vc**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_register_eth_nano

```
NONULL in3_ret_t in3_register_eth_nano(in3_t *c);
```

this function should only be called once and will register the eth-nano verifier.

arguments:

*in3_t **	**c**

returns: `in3_ret_tNONULL` the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

create_tx_path

```
bytes_t* create_tx_path(uint32_t index);
```

helper function to rlp-encode the transaction_index.

The result must be freed after use!

arguments:

`uint32_t`	**index**

returns: `bytes_t *`

9.12.11 merkle.h

Merkle Proof Verification.

File: c/src/verifier/eth1/nano/merkle.h

MERKLE_DEPTH_MAX

```
#define MERKLE_DEPTH_MAX 64
```

trie_verify_proof

```
int trie_verify_proof(bytes_t *rootHash, bytes_t *path, bytes_t **proof, bytes_t␣
↪*expectedValue);
```

verifies a merkle proof.

expectedValue == NULL : value must not exist expectedValue.data ==NULL : please copy the data I want to evaluate it afterwards. expectedValue.data !=NULL : the value must match the data.

arguments:

*bytes_t **	**rootHash**
*bytes_t **	**path**
*bytes_t ***	**proof**
*bytes_t **	**expectedValue**

returns: `int`

trie_path_to_nibbles

```
NONULL uint8_t* trie_path_to_nibbles(bytes_t path, int use_prefix);
```

helper function split a path into 4-bit nibbles.

The result must be freed after use!

arguments:

bytes_t	**path**
`int`	**use_prefix**

returns: `NONULL uint8_t *` : the resulting bytes represent a 4bit-number each and are terminated with a 0xFF.

trie_matching_nibbles

```
NONULL int trie_matching_nibbles(uint8_t *a, uint8_t *b);
```

helper function to find the number of nibbles matching both paths.

arguments:

uint8_t *	**a**
uint8_t *	**b**

returns: NONULL int

9.12.12 rlp.h

RLP-En/Decoding as described in the Ethereum RLP-Spec.

This decoding works without allocating new memory.

File: c/src/verifier/eth1/nano/rlp.h

rlp_decode

```
int rlp_decode(bytes_t *b, int index, bytes_t *dst);
```

this function decodes the given bytes and returns the element with the given index by updating the reference of dst.

the bytes will only hold references and do **not** need to be freed!

```
bytes_t* tx_raw = serialize_tx(tx);

bytes_t item;

// decodes the tx_raw by letting the item point to range of the first element, which␣
↪should be the body of a list.
if (rlp_decode(tx_raw, 0, &item) !=2) return -1 ;

// now decode the 4th element (which is the value) and let item point to that range.
if (rlp_decode(&item, 4, &item) !=1) return -1 ;
```

arguments:

bytes_t *	**b**
int	**index**
bytes_t *	**dst**

returns: int : - 0 : means item out of range

- 1 : item found
- 2 : list found (you can then decode the same bytes again)

rlp_decode_in_list

```
int rlp_decode_in_list(bytes_t *b, int index, bytes_t *dst);
```

this function expects a list item (like the blockheader as first item and will then find the item within this list).

It is a shortcut for

```
// decode the list
if (rlp_decode(b,0,dst)!=2) return 0;
// and the decode the item
return rlp_decode(dst,index,dst);
```

arguments:

*bytes_t **	**b**
`int`	**index**
*bytes_t **	**dst**

returns: `int` : - 0 : means item out of range

- 1 : item found
- 2 : list found (you can then decode the same bytes again)

rlp_decode_len

```
int rlp_decode_len(bytes_t *b);
```

returns the number of elements found in the data.

arguments:

*bytes_t **	**b**

returns: `int`

rlp_encode_item

```
void rlp_encode_item(bytes_builder_t *bb, bytes_t *val);
```

encode a item as single string and add it to the bytes_builder.

arguments:

*bytes_builder_t **	**bb**
*bytes_t **	**val**

rlp_encode_list

```
void rlp_encode_list(bytes_builder_t *bb, bytes_t *val);
```

encode a the value as list of already encoded items.

arguments:

*bytes_builder_t **	**bb**
*bytes_t **	**val**

rlp_encode_to_list

```
bytes_builder_t* rlp_encode_to_list(bytes_builder_t *bb);
```

converts the data in the builder to a list.

This function is optimized to not increase the memory more than needed and is fastet than creating a second builder to encode the data.

arguments:

*bytes_builder_t **	**bb**

returns: `bytes_builder_t *` : the same builder.

rlp_encode_to_item

```
bytes_builder_t* rlp_encode_to_item(bytes_builder_t *bb);
```

converts the data in the builder to a rlp-encoded item.

This function is optimized to not increase the memory more than needed and is fastet than creating a second builder to encode the data.

arguments:

*bytes_builder_t **	**bb**

returns: `bytes_builder_t *` : the same builder.

rlp_add_length

```
void rlp_add_length(bytes_builder_t *bb, uint32_t len, uint8_t offset);
```

helper to encode the prefix for a value

arguments:

*bytes_builder_t **	**bb**
`uint32_t`	**len**
`uint8_t`	**offset**

9.12.13 serialize.h

serialization of ETH-Objects.

This incoming tokens will represent their values as properties based on JSON-RPC.

File: c/src/verifier/eth1/nano/serialize.h

BLOCKHEADER_PARENT_HASH

```
#define BLOCKHEADER_PARENT_HASH 0
```

BLOCKHEADER_SHA3_UNCLES

```
#define BLOCKHEADER_SHA3_UNCLES 1
```

BLOCKHEADER_MINER

```
#define BLOCKHEADER_MINER 2
```

BLOCKHEADER_STATE_ROOT

```
#define BLOCKHEADER_STATE_ROOT 3
```

BLOCKHEADER_TRANSACTIONS_ROOT

```
#define BLOCKHEADER_TRANSACTIONS_ROOT 4
```

BLOCKHEADER_RECEIPT_ROOT

```
#define BLOCKHEADER_RECEIPT_ROOT 5
```

BLOCKHEADER_LOGS_BLOOM

```
#define BLOCKHEADER_LOGS_BLOOM 6
```

BLOCKHEADER_DIFFICULTY

```
#define BLOCKHEADER_DIFFICULTY 7
```

BLOCKHEADER_NUMBER

```
#define BLOCKHEADER_NUMBER 8
```

BLOCKHEADER_GAS_LIMIT

```
#define BLOCKHEADER_GAS_LIMIT 9
```

BLOCKHEADER_GAS_USED

```
#define BLOCKHEADER_GAS_USED 10
```

BLOCKHEADER_TIMESTAMP

```
#define BLOCKHEADER_TIMESTAMP 11
```

BLOCKHEADER_EXTRA_DATA

```
#define BLOCKHEADER_EXTRA_DATA 12
```

BLOCKHEADER_SEALED_FIELD1

```
#define BLOCKHEADER_SEALED_FIELD1 13
```

BLOCKHEADER_SEALED_FIELD2

```
#define BLOCKHEADER_SEALED_FIELD2 14
```

BLOCKHEADER_SEALED_FIELD3

```
#define BLOCKHEADER_SEALED_FIELD3 15
```

serialize_tx_receipt

```
bytes_t* serialize_tx_receipt(d_token_t *receipt);
```

creates rlp-encoded raw bytes for a receipt.

The bytes must be freed with b_free after use!

arguments:

*d_token_t **	**receipt**

returns: `bytes_t *`

serialize_tx

```
bytes_t* serialize_tx(d_token_t *tx);
```

creates rlp-encoded raw bytes for a transaction.

The bytes must be freed with b_free after use!

arguments:

*d_token_t **	**tx**

returns: `bytes_t *`

serialize_tx_raw

```
bytes_t* serialize_tx_raw(bytes_t nonce, bytes_t gas_price, bytes_t gas_limit, bytes_
↪t to, bytes_t value, bytes_t data, uint64_t v, bytes_t r, bytes_t s);
```

creates rlp-encoded raw bytes for a transaction from direct values.

The bytes must be freed with b_free after use!

arguments:

bytes_t	**nonce**
bytes_t	**gas_price**
bytes_t	**gas_limit**
bytes_t	**to**
bytes_t	**value**
bytes_t	**data**
`uint64_t`	**v**
bytes_t	**r**
bytes_t	**s**

returns: `bytes_t *`

serialize_account

```
bytes_t* serialize_account(d_token_t *a);
```

creates rlp-encoded raw bytes for a account.

The bytes must be freed with b_free after use!

arguments:

*d_token_t **	**a**

returns: `bytes_t *`

serialize_block_header

```
bytes_t* serialize_block_header(d_token_t *block);
```

creates rlp-encoded raw bytes for a blockheader.

The bytes must be freed with b_free after use!

arguments:

*d_token_t **	**block**

returns: `bytes_t *`

rlp_add

```
int rlp_add(bytes_builder_t *rlp, d_token_t *t, int ml);
```

adds the value represented by the token rlp-encoded to the byte_builder.

arguments:

*bytes_builder_t **	**rlp**
*d_token_t **	**t**
`int`	**ml**

returns: `int` : 0 if added -1 if the value could not be handled.

9.12.14 in3_init.h

IN3 init module for auto initializing verifiers and transport based on build config.

File: c/src/verifier/in3_init.h

in3_for_chain (chain_id)

```
#define in3_for_chain (chain_id) in3_for_chain_auto_init(chain_id)
```

in3_init

```
void in3_init();
```

Global initialization for the in3 lib.

Note: This function is not MT-safe and is expected to be called early during during program startup (i.e. in main()) before other threads are spawned.

in3_for_chain_auto_init

```
in3_t* in3_for_chain_auto_init(chain_id_t chain_id);
```

Auto-init fallback for easy client initialization meant for single-threaded apps.

This function automatically calls `in3_init()` before calling `in3_for_chain_default()`. To enable this feature, make sure you include this header file (i.e. `in3_init.h`) before `client.h`. Doing so will replace the call to `in3_for_chain()` with this function.

arguments:

chain_id_t	**chain_id**

returns: `in3_t *`

9.12.15 ipfs.h

IPFS verification.

File: c/src/verifier/ipfs/ipfs.h

ipfs_verify_hash

```
in3_ret_t ipfs_verify_hash(const char *content, const char *encoding, const char␣
↪*requsted_hash);
```

verifies an IPFS hash.

Supported encoding schemes - hex, utf8 and base64

arguments:

`const char *`	**content**
`const char *`	**encoding**
`const char *`	**requsted_hash**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

in3_register_ipfs

```
in3_ret_t in3_register_ipfs(in3_t *c);
```

this function should only be called once and will register the IPFS verifier.

arguments:

*in3_t **	**c**

returns: `in3_ret_t` the *result-status* of the function.

Please make sure you check if it was successfull (==`IN3_OK`)

CHAPTER 10

API Reference TS

This page contains a list of all Datastructures and Classes used within the TypeScript IN3 Client.

10.1 Examples

This is a collection of different incubed-examples.

10.1.1 using Web3

Since incubed works with on a JSON-RPC-Level it can easily be used as Provider for Web3:

```
// import in3-Module
import In3Client from 'in3'
import * as web3 from 'web3'

// use the In3Client as Http-Provider
const web3 = new Web3(new In3Client({
    proof         : 'standard',
    signatureCount: 1,
    requestCount  : 2,
    chainId       : 'mainnet'
}).createWeb3Provider())

// use the web3
const block = await web.eth.getBlockByNumber('latest')
...
```

10.1.2 using Incubed API

Incubed includes a light API, allowinng not only to use all RPC-Methods in a typesafe way, but also to sign transactions and call funnctions of a contract without the web3-library.

For more details see the API-Doc

```
// import in3-Module
import In3Client from 'in3'

// use the In3Client
const in3 = new In3Client({
    proof         : 'standard',
    signatureCount: 1,
    requestCount  : 2,
    chainId       : 'mainnet'
})

// use the api to call a funnction..
const myBalance = await in3.eth.callFn(myTokenContract, 'balanceOf(address):uint',␣
↪myAccount)

// ot to send a transaction..
const receipt = await in3.eth.sendTransaction({
  to           : myTokenContract,
  method       : 'transfer(address,uint256)',
  args         : [target,amount],
  confirmations: 2,
  pk           : myKey
})

...
```

10.1.3 Reading event with incubed

```
// import in3-Module
import In3Client from 'in3'

// use the In3Client
const in3 = new In3Client({
    proof         : 'standard',
    signatureCount: 1,
    requestCount  : 2,
    chainId       : 'mainnet'
})

// use the ABI-String of the smart contract
abi = [{"anonymous":false,"inputs":[{"indexed":false,"name":"name","type":"string"},{
↪"indexed":true,"name":"label","type":"bytes32"},{"indexed":true,"name":"owner","type
↪":"address"},{"indexed":false,"name":"cost","type":"uint256"},{"indexed":false,"name
↪":"expires","type":"uint256"}],"name":"NameRegistered","type":"event"}]

// create a contract-object for a given address
const contract = in3.eth.contractAt(abi, '0xF0AD5cAd05e10572EfcEB849f6Ff0c68f9700455
↪') // ENS contract.

// read all events starting from a specified block until the latest
const logs = await c.events.NameRegistered.getLogs({fromBlock:8022948}))

// print out the properties of the event.
for (const ev of logs)
```

(continues on next page)

(continued from previous page)

```
  console.log(`${ev.owner} registered ${ev.name} for ${ev.cost} wei until ${new␣
↪Date(ev.expires.toNumber()*1000).toString()}`)

...
```

10.2 Main Module

Importing incubed is as easy as

```
import Client,{util} from "in3"
```

While the In3Client-class is the default import, the following imports can be used:

Type	*ABI*	the ABI
Interface	*AccountProof*	the AccountProof
Interface	*AuraValidatoryProof*	the AuraValidatoryProof
Type	*BlockData*	the BlockData
Type	*BlockType*	the BlockType
Interface	*ChainSpec*	the ChainSpec
Class	*IN3Client*	the IN3Client
Interface	*IN3Config*	the IN3Config
Interface	*IN3NodeConfig*	the IN3NodeConfig
Interface	*IN3NodeWeight*	the IN3NodeWeight
Interface	*IN3RPCConfig*	the IN3RPCConfig
Interface	*IN3RPCHandlerConfig*	the IN3RPCHandlerConfig

Continued on next page

Table 1 – continued from previous page

Interface	*IN3RPCRequestConfig*	the IN3RPCRequestConfig
Interface	*IN3ResponseConfig*	the IN3ResponseConfig
Type	*Log*	the Log
Type	*LogData*	the LogData
Interface	*LogProof*	the LogProof
Interface	*Proof*	the Proof
Interface	*RPCRequest*	the RPCRequest
Interface	*RPCResponse*	the RPCResponse
Type	*ReceiptData*	the ReceiptData
Interface	*ServerList*	the ServerList
Interface	*Signature*	the Signature
Type	*Transaction*	the Transaction
Type	*TransactionData*	the TransactionData
Type	*TransactionReceipt*	the TransactionReceipt
Type	*Transport*	the Transport
`any`	AxiosTransport	the AxiosTransport value= `transport.AxiosTransport`

Continued on next page

Table 1 – continued from previous page

EthAPI	EthAPI	the EthAPI value= `_ethapi.default`
`any`	cbor	the cbor value= `_cbor`
	chainAliases	the chainAliases value= `aliases`
chainData	chainData	the chainData value= `_chainData`
`number` []	createRandomIndexes (len:`number`, limit:`number`, seed:*Buffer* , result:`number` [])	helper function creating deterministic random indexes used for limited nodelists
header	header	the header value= `_header`
serialize	serialize	the serialize value= `_serialize`
`any`	storage	the storage value= `_storage`
`any`	transport	the transport value= `_transport`
	typeDefs	the typeDefs value= `types.validationDef`
`any`	util	the util value= `_util`

Continued on next page

Table 1 – continued from previous page

`any`	validate	the validate value= `validateOb.validate`

10.3 Package client

10.3.1 Type Client

Source: client/Client.ts

Client for N3.

`number`	defaultMaxListeners	the defaultMaxListeners
`number`	listenerCount (emitter:*EventEmitter* , event:`string` | `symbol`)	listener count
Client	constructor (config:*Partial<IN3Config>* , transport:*Transport*)	creates a new Client.
IN3Config	defConfig	the defConfig
EthAPI	eth	the eth
IpfsAPI	ipfs	the ipfs
IN3Config	config	config
`this`	addListener (event:`string` | `symbol`, listener:)	add listener

Continued on next page

Table 2 – continued from previous page

`Promise<any>`	call (method:`string`, params:`any`, chain:`string`, config:*Partial<IN3Config>*)	sends a simply RPC-Request
`void`	clearStats ()	clears all stats and weights, like blocklisted nodes
`any`	createWeb3Provider ()	create web3 provider
`boolean`	emit (event:`string` | `symbol`, args:`any` [])	emit
Array<>	eventNames ()	event names
ChainContext	getChainContext (chainId:`string`)	Context for a specific chain including cache and chainSpecs.
`number`	getMaxListeners ()	get max listeners
`number`	listenerCount (type:`string` | `symbol`)	listener count
Function []	listeners (event:`string` | `symbol`)	listeners
`this`	off (event:`string` | `symbol`, listener:)	off

Continued on next page

Table 2 – continued from previous page

`this`	on (event:`string` | `symbol`, listener:)	on
`this`	once (event:`string` | `symbol`, listener:)	once
`this`	prependListener (event:`string` | `symbol`, listener:)	prepend listener
`this`	prependOnceListener (event:`string` | `symbol`, listener:)	prepend once listener
Function []	rawListeners (event:`string` | `symbol`)	raw listeners
`this`	removeAllListeners (event:`string` | `symbol`)	remove all listeners
`this`	removeListener (event:`string` | `symbol`, listener:)	remove listener

Continued on next page

Table 2 – continued from previous page

`Promise<>`	send (request:*RPCRequest* [] | *RPCRequest* , callback:, config:*Partial<IN3Config>*)	sends one or a multiple requests. if the request is a array the response will be a array as well. If the callback is given it will be called with the response, if not a Promise will be returned. This function supports callback so it can be used as a Provider for the web3.
Promise<RPCResponse>	sendRPC (method:`string`, params:`any` [], chain:`string`, config:*Partial<IN3Config>*)	sends a simply RPC-Request
`this`	setMaxListeners (n:`number`)	set max listeners
`Promise<void>`	updateNodeList (chainId:`string`, conf:*Partial<IN3Config>* , retryCount:`number`)	fetches the nodeList from the servers.
`Promise<void>`	updateWhiteListNodes (config:*IN3Config*)	update white list nodes

Continued on next page

Table 2 – continued from previous page

`Promise<boolean>`	verifyResponse (request:*RPCRequest* , response:*RPCResponse* , chain:`string`, config:*Partial<IN3Config>*)	Verify the response of a request without any effect on the state of the client. Note: The node-list will not be updated. The method will either return *true* if its inputs could be verified. Or else, it will throw an exception with a helpful message.

10.3.2 Type ChainContext

Source: client/ChainContext.ts

Context for a specific chain including cache and chainSpecs.

ChainContext	constructor (client:*Client* , chainId:string, chainSpec:*ChainSpec* [])	Context for a specific chain including cache and chainSpecs.
string	chainId	the chainId
ChainSpec []	chainSpec	the chainSpec
Client	client	the client
	genericCache	the genericCache
number	lastValidatorChange	the lastValidatorChange
Module	module	the module
string	registryId	the registryId *(optional)*
void	clearCache (prefix:string)	clear cache
ChainSpec	getChainSpec (block:number)	returns the chainspec for th given block number
string	getFromCache (key:string)	get from cache
Promise<RPCResponse>	handleIntern (request:*RPCRequest*)	this function is calleds before the server is asked. If it returns a promise than the request is handled internally otherwise the server will handle the response. this function should be overriden by modules that want to handle calls internally

10.3.3 Type Module

Source: client/modules.ts

`string`	name	the name
ChainContext	createChainContext (client:*Client* , chainId:`string`, spec:*ChainSpec* [])	Context for a specific chain including cache and chainSpecs.
`Promise<boolean>`	verifyProof (request:*RPCRequest* , response:*RPCResponse* , allowWithoutProof:`boolean`, ctx:*ChainContext*)	general verification-function which handles it according to its given type.

10.4 Package index.ts

10.4.1 Type AccountProof

Source: index.ts

the Proof-for a single Account the Proof-for a single Account

string []	accountProof	the serialized merle-noodes beginning with the root-node
string	address	the address of this account
string	balance	the balance of this account as hex
string	code	the code of this account as hex (if required) *(optional)*
string	codeHash	the codeHash of this account as hex
string	nonce	the nonce of this account as hex
string	storageHash	the storageHash of this account as hex
[]	storageProof	proof for requested storage-data

10.4.2 Type AuraValidatoryProof

Source: index.ts

a Object holding proofs for validator logs. The key is the blockNumber as hex a Object holding proofs for validator logs. The key is the blockNumber as hex

`string`	block	the serialized blockheader example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6
`any` []	finalityBlocks	the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6 *(optional)*
`number`	logIndex	the transaction log index
`string` []	proof	the merkleProof
`number`	txIndex	the transactionIndex within the block

10.4.3 Type ChainSpec

Source: index.ts

describes the chainspecific consensus params describes the chainspecific consensus params

`number`	block	the blocknumnber when this configuration should apply *(optional)*
`number`	bypassFinality	Bypass finality check for transition to contract based Aura Engines example: bypassFinality = 10960502 -> will skip the finality check and add the list at block 10960502 *(optional)*
`string`	contract	The validator contract at the block *(optional)*
`'ethHash' |'authorityRound' |'clique'`	engine	the engine type (like Ethhash, authorityRound, . . .) *(optional)*
`string` []	list	The list of validators at the particular block *(optional)*
`boolean`	requiresFinality	indicates whether the transition requires a finality check example: true *(optional)*

10.4.4 Type IN3Client

Source: index.ts

Client for N3. Client for N3.

`number`	defaultMaxListeners	the defaultMaxListeners

Continued on next page

Table 3 – continued from previous page

`number`	listenerCount (emitter:*EventEmitter* , event:`string` | `symbol`)	listener count
Client	constructor (config:*Partial<IN3Config>* , transport:*Transport*)	creates a new Client.
IN3Config	defConfig	the defConfig
EthAPI	eth	the eth
IpfsAPI	ipfs	the ipfs
IN3Config	config	config
`this`	addListener (event:`string` | `symbol`, listener:)	add listener
`Promise<any>`	call (method:`string`, params:`any`, chain:`string`, config:*Partial<IN3Config>*)	sends a simply RPC-Request
`void`	clearStats ()	clears all stats and weights, like blocklisted nodes
`any`	createWeb3Provider ()	create web3 provider

Continued on next page

Table 3 – continued from previous page

`boolean`	emit (event:`string` | `symbol`, args:`any` [])	emit
Array<>	eventNames ()	event names
ChainContext	getChainContext (chainId:`string`)	Context for a specific chain including cache and chainSpecs.
`number`	getMaxListeners ()	get max listeners
`number`	listenerCount (type:`string` | `symbol`)	listener count
Function []	listeners (event:`string` | `symbol`)	listeners
`this`	off (event:`string` | `symbol`, listener:)	off
`this`	on (event:`string` | `symbol`, listener:)	on
`this`	once (event:`string` | `symbol`, listener:)	once

Continued on next page

Table 3 – continued from previous page

`this`	prependListener (event:`string` | `symbol`, listener:)	prepend listener
`this`	prependOnceListener (event:`string` | `symbol`, listener:)	prepend once listener
Function []	rawListeners (event:`string` | `symbol`)	raw listeners
`this`	removeAllListeners (event:`string` | `symbol`)	remove all listeners
`this`	removeListener (event:`string` | `symbol`, listener:)	remove listener
`Promise<>`	send (request:*RPCRequest* [] | *RPCRequest* , callback:, config:*Partial<IN3Config>*)	sends one or a multiple requests. if the request is a array the response will be a array as well. If the callback is given it will be called with the response, if not a Promise will be returned. This function supports callback so it can be used as a Provider for the web3.
Promise<RPCResponse>	sendRPC (method:`string`, params:`any` [], chain:`string`, config:*Partial<IN3Config>*)	sends a simply RPC-Request

Continued on next page

Table 3 – continued from previous page

`this`	setMaxListeners (n:`number`)	set max listeners
`Promise<void>`	updateNodeList (chainId:`string`, conf:*Partial<IN3Config>* , retryCount:`number`)	fetches the nodeList from the servers.
`Promise<void>`	updateWhiteListNodes (config:*IN3Config*)	update white list nodes
`Promise<boolean>`	verifyResponse (request:*RPCRequest* , response:*RPCResponse* , chain:`string`, config:*Partial<IN3Config>*)	Verify the response of a request without any effect on the state of the client. Note: The node-list will not be updated. The method will either return *true* if its inputs could be verified. Or else, it will throw an exception with a helpful message.

10.4.5 Type IN3Config

Source: index.ts

the iguration of the IN3-Client. This can be paritally overriden for every request. the iguration of the IN3-Client. This can be paritally overriden for every request.

`boolean`	archiveNodes	if true the in3 client will filter out non archive supporting nodes example: true *(optional)*
`boolean`	autoConfig	if true the config will be adjusted depending on the request *(optional)*

Continued on next page

Table 4 – continued from previous page

`boolean`	autoUpdateList	if true the nodelist will be automaticly updated if the lastBlock is newer example: true *(optional)*
`boolean`	binaryNodes	if true the in3 client will only include nodes that support binary encording example: true *(optional)*
`any`	cacheStorage	a cache handler offering 2 functions (setItem(string,string), getItem(string)) *(optional)*
`number`	cacheTimeout	number of seconds requests can be cached. *(optional)*
`string`	chainId	servers to filter for the given chain. The chain-id based on EIP-155. example: 0x1
`string`	chainRegistry	main chain-registry contract example: 0xe36179e2286ef405e929C90ad3E70E649B22a945 *(optional)*
`number`	depositTimeout	timeout after which the owner is allowed to receive its stored deposit. This information is also important for the client example: 3000 *(optional)*
`number`	finality	the number in percent needed in order reach finality (% of signature of the validators) example: 50 *(optional)*

Continued on next page

Table 4 – continued from previous page

`'json'|'jsonRef'|'cbor'`	format	the format for sending the data to the client. Default is json, but using cbor means using only 30-40% of the payload since it is using binary encoding example: json *(optional)*
`boolean`	httpNodes	if true the in3 client will include http nodes example: true *(optional)*
`boolean`	includeCode	if true, the request should include the codes of all accounts. otherwise only the the codeHash is returned. In this case the client may ask by calling eth_getCode() afterwards example: true *(optional)*
`boolean`	keepIn3	if true, the in3-section of thr response will be kept. Otherwise it will be removed after validating the data. This is useful for debugging or if the proof should be used afterwards. *(optional)*
`any`	key	the client key to sign requests example: 0x387a8233c96e1fc0ad5e284353276177af2186e7afa8529 *(optional)*
`string`	loggerUrl	a url of RES-Endpoint, the client will log all errors to. The client will post to this endpoint JSON like { id?, level, message, meta? } *(optional)*
`string`	mainChain	main chain-id, where the chain registry is running. example: 0x1 *(optional)*

Continued on next page

Table 4 – continued from previous page

`number`	maxAttempts	max number of attempts in case a response is rejected example: 10 *(optional)*
`number`	maxBlockCache	number of number of blocks cached in memory example: 100 *(optional)*
`number`	maxCodeCache	number of max bytes used to cache the code in memory example: 100000 *(optional)*
`number`	minDeposit	min stake of the server. Only nodes owning at least this amount will be chosen.
`boolean`	multichainNodes	if true the in3 client will filter out nodes other then which have capability of the same RPC endpoint may also accept requests for different chains example: true *(optional)*
`number`	nodeLimit	the limit of nodes to store in the client. example: 150 *(optional)*
`'none'|'standard'|'full'`	proof	if true the nodes should send a proof of the response example: true *(optional)*
`boolean`	proofNodes	if true the in3 client will filter out nodes which are providing no proof example: true *(optional)*
`number`	replaceLatestBlock	if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6 *(optional)*

Continued on next page

Table 4 – continued from previous page

`number`	requestCount	the number of request send when getting a first answer example: 3
`boolean`	retryWithoutProof	if true the the request may be handled without proof in case of an error. (use with care!) *(optional)*
`string`	rpc	url of one or more rpc-endpoints to use. (list can be comma seperated) *(optional)*
	servers	the nodelist per chain *(optional)*
`number`	signatureCount	number of signatures requested example: 2 *(optional)*
`number`	timeout	specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection. example: 3000 *(optional)*
`boolean`	torNodes	if true the in3 client will filter out non tor nodes example: true *(optional)*
`string` []	verifiedHashes	if the client sends a array of blockhashes the server will not deliver any signatures or blockheaders for these blocks, but only return a string with a number. This is automaticly updated by the cache, but can be overriden per request. *(optional)*

Continued on next page

Table 4 – continued from previous page

`string` []	whiteList	a list of in3 server addresses which are whitelisted manually by client example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,0x6d *(optional)*
`string`	whiteListContract	White list contract address *(optional)*

10.4.6 Type IN3NodeConfig

Source: index.ts

a configuration of a in3-server. a configuration of a in3-server.

`string`	address	the address of the node, which is the public address it iis signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
`number`	capacity	the capacity of the node. example: 100 *(optional)*
`string` []	chainIds	the list of supported chains example: 0x1
`number`	deposit	the deposit of the node in wei example: 12350000
`number`	index	the index within the contract example: 13 *(optional)*
`number`	props	the properties of the node. example: 3 *(optional)*
`number`	registerTime	the UNIX-timestamp when the node was registered example: 1563279168 *(optional)*
`number`	timeout	the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600 *(optional)*
`number`	unregisterTime	the UNIX-timestamp when the node is allowed to be deregister example: 1563279168 *(optional)*
`string`	url	the endpoint to post to example: https://in3.slock.it

10.4.7 Type IN3NodeWeight

Source: index.ts

a local weight of a n3-node. (This is used internally to weight the requests) a local weight of a n3-node. (This is used internally to weight the requests)

`number`	avgResponseTime	average time of a response in ms example: 240 *(optional)*
`number`	blacklistedUntil	blacklisted because of failed requests until the timestamp example: 1529074639623 *(optional)*
`number`	lastRequest	timestamp of the last request in ms example: 1529074632623 *(optional)*
`number`	pricePerRequest	last price *(optional)*
`number`	responseCount	number of uses. example: 147 *(optional)*
`number`	weight	factor the weight this noe (default 1.0) example: 0.5 *(optional)*

10.4.8 Type IN3RPCConfig

Source: index.ts

the configuration for the rpc-handler the configuration for the rpc-handler

	chains	a definition of the Handler per chain *(optional)*
	db	the db *(optional)*
`string`	defaultChain	the default chainId in case the request does not contain one. *(optional)*
`string`	id	a identifier used in logfiles as also for reading the config from the database *(optional)*
	logging	logger config *(optional)*
`number`	port	the listeneing port for the server *(optional)*
	profile	the profile *(optional)*

10.4.9 Type IN3RPCHandlerConfig

Source: index.ts

the configuration for the rpc-handler the configuration for the rpc-handler

	autoRegistry	the autoRegistry *(optional)*
`string`	clientKeys	a comma sepearted list of client keys to use for simulating clients for the watchdog *(optional)*
`number`	freeScore	the score for requests without a valid signature *(optional)*
`'eth'|'ipfs'|'btc'`	handler	the impl used to handle the calls *(optional)*
`string`	ipfsUrl	the url of the ipfs-client *(optional)*
`number`	maxThreads	the maximal number of threads ofr running parallel processes *(optional)*
`number`	minBlockHeight	the minimal blockheight in order to sign *(optional)*
`string`	persistentFile	the filename of the file keeping track of the last handled blocknumber *(optional)*
`string`	privateKey	the private key used to sign blockhashes. this can be either a 0x-prefixed string with the raw private key or the path to a key-file.
`string`	privateKeyPassphrase	the password used to decrpyt the private key *(optional)*
`string`	registry	the address of the server registry used in order to update the nodeList

10.4.10 Type IN3RPCRequestConfig

Source: index.ts

additional config for a IN3 RPC-Request additional config for a IN3 RPC-Request

`string`	chainId	the requested chainId example: 0x1
`any`	clientSignature	the signature of the client *(optional)*
`number`	finality	if given the server will deliver the blockheaders of the following blocks until at least the number in percent of the validators is reached. *(optional)*
`boolean`	includeCode	if true, the request should include the codes of all accounts. otherwise only the the codeHash is returned. In this case the client may ask by calling eth_getCode() afterwards example: true *(optional)*
`number`	latestBlock	if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6 *(optional)*
`string` []	signatures	a list of addresses requested to sign the blockhash example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679 *(optional)*
`boolean`	useBinary	if true binary-data will be used. *(optional)*
`boolean`	useFullProof	if true all data in the response will be proven, which leads to a higher payload. *(optional)*
`boolean`	useRef	if true binary-data (starting with 0x) will be refered if occuring again. *(optional)*

10.4.11 Type IN3ResponseConfig

Source: index.ts

additional data returned from a IN3 Server additional data returned from a IN3 Server

`number`	currentBlock	the current blocknumber. example: 320126478 *(optional)*
`number`	lastNodeList	the blocknumber for the last block updating the nodelist. If the client has a smaller blocknumber he should update the nodeList. example: 326478 *(optional)*
`number`	lastValidatorChange	the blocknumber of the last change of the validatorList *(optional)*
`number`	lastWhiteList	The blocknumber of the last white list event *(optional)*
Proof	proof	the Proof-data *(optional)*
`string`	version	IN3 protocol version example: 1.0.0 *(optional)*

10.4.12 Type LogProof

Source: index.ts

a Object holding proofs for event logs. The key is the blockNumber as hex a Object holding proofs for event logs. The key is the blockNumber as hex

10.4.13 Type Proof

Source: index.ts

the Proof-data as part of the in3-section the Proof-data as part of the in3-section

	accounts	a map of addresses and their AccountProof *(optional)*
`string`	block	the serialized blockheader as hex, required in most proofs example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6 *(optional)*
`any` []	finalityBlocks	the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6 *(optional)*
LogProof	logProof	the Log Proof in case of a Log-Request *(optional)*
`string` []	merkleProof	the serialized merle-noodes beginning with the root-node *(optional)*
`string` []	merkleProofPrev	the serialized merkle-noodes beginning with the root-node of the previous entry (only for full proof of receipts) *(optional)*
Signature []	signatures	requested signatures *(optional)*
`any` []	transactions	the list of transactions of the block example: *(optional)*
`number`	txIndex	the transactionIndex within the block example: 4 *(optional)*
`string` []	txProof	the serialized merkle-nodes beginning with the root-node in order to prrof the transactionIndex *(optional)*

10.4.14 Type RPCRequest

Source: index.ts

a JSONRPC-Request with N3-Extension a JSONRPC-Request with N3-Extension

`number|string`	id	the identifier of the request example: 2 *(optional)*
IN3RPCRequestConfig	in3	the IN3-Config *(optional)*
`'2.0'`	jsonrpc	the version
`string`	method	the method to call example: eth_getBalance
`any []`	params	the params example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,lates *(optional)*

10.4.15 Type RPCResponse

Source: index.ts

a JSONRPC-Responset with N3-Extension a JSONRPC-Responset with N3-Extension

`string`	error	in case of an error this needs to be set *(optional)*
`string|number`	id	the id matching the request example: 2
IN3ResponseConfig	in3	the IN3-Result *(optional)*
IN3NodeConfig	in3Node	the node handling this response (internal only) *(optional)*
`'2.0'`	jsonrpc	the version
`any`	result	the params example: 0xa35bc *(optional)*

10.4.16 Type ServerList

Source: index.ts

a List of nodes a List of nodes

`string`	contract	IN3 Registry *(optional)*
`number`	lastBlockNumber	last Block number *(optional)*
IN3NodeConfig []	nodes	the list of nodes
Proof	proof	the proof *(optional)*
`string`	registryId	registry id of the contract *(optional)*
`number`	totalServers	number of servers *(optional)*

10.4.17 Type Signature

Source: index.ts

Verified ECDSA Signature. Signatures are a pair (r, s). Where r is computed as the X coordinate of a point R, modulo the curve order n. Verified ECDSA Signature. Signatures are a pair (r, s). Where r is computed as the X coordinate of a point R, modulo the curve order n.

`string`	address	the address of the signing node example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679 *(optional)*
`number`	block	the blocknumber example: 3123874
`string`	blockHash	the hash of the block example: 0x6C1a01C2aB554930A937B0a212346037E8105fB4794
`string`	msgHash	hash of the message example: 0x9C1a01C2aB554930A937B0a212346037E8105fB4794
`string`	r	Positive non-zero Integer signature.r example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6
`string`	s	Positive non-zero Integer signature.s example: 0x6d17b34aeaf95fee98c0437b4ac839d8a2ece1b18166da
`number`	v	Calculated curve point, or identity element O. example: 28

10.4.18 Type Transport

Source: index.ts

= *_transporttype*

10.5 Package modules/eth

10.5.1 Type EthAPI

Source: modules/eth/api.ts

EthAPI	constructor (client:*Client*)	constructor
Client	client	the client
Signer	signer	the signer *(optional)*
`Promise<number>`	blockNumber ()	Returns the number of most recent block. (as number)
`Promise<string>`	call (tx:*Transaction* , block:*BlockType*)	Executes a new message call immediately without creating a transaction on the block chain.
`Promise<any>`	callFn (to:*Address* , method:`string`, args:`any` [])	Executes a function of a contract, by passing a [method-signature](https://github.com/ethereumjs/ethereumjs-abi/blob/master/README.md#simple-encoding-and-decoding) and the arguments, which will then be ABI-encoded and send as eth_call.
`Promise<string>`	chainId ()	Returns the EIP155 chain ID used for transaction signing at the current best block. Null is returned if not available.
	contractAt (abi:*ABI* [], address:*Address*)	contract at

Continued on next page

Table 5 – continued from previous page

any	decodeEventData (log:*Log* , d:*ABI*)	decode event data
Promise<number>	estimateGas (tx:*Transaction*)	Makes a call or transaction, which won't be added to the blockchain and returns the used gas, which can be used for estimating the used gas.
Promise<number>	gasPrice ()	Returns the current price per gas in wei. (as number)
Promise<BN>	getBalance (address:*Address* , block:*BlockType*)	Returns the balance of the account of given address in wei (as hex).
Promise<Block>	getBlockByHash (hash:*Hash* , includeTransactions:boolean)	Returns information about a block by hash.
Promise<Block>	getBlockByNumber (block:*BlockType* , includeTransactions:boolean)	Returns information about a block by block number.
Promise<number>	getBlockTransactionCountByHash (block:*Hash*)	Returns the number of transactions in a block from a block matching the given block hash.
Promise<number>	getBlockTransactionCountByNumber (block:*Hash*)	Returns the number of transactions in a block from a block matching the given block number.
Promise<string>	getCode (address:*Address* , block:*BlockType*)	Returns code at a given address.

Continued on next page

Table 5 – continued from previous page

`Promise<>`	getFilterChanges (id:*Quantity*)	Polling method for a filter, which returns an array of logs which occurred since last poll.
`Promise<>`	getFilterLogs (id:*Quantity*)	Returns an array of all logs matching filter with given id.
`Promise<>`	getLogs (filter:*LogFilter*)	Returns an array of all logs matching a given filter object.
`Promise<string>`	getStorageAt (address:*Address* , pos:*Quantity* , block:*BlockType*)	Returns the value from a storage position at a given address.
Promise<TransactionDetail>	getTransactionByBlockHashAndIndex (hash:*Hash* , pos:*Quantity*)	Returns information about a transaction by block hash and transaction index position.
Promise<TransactionDetail>	getTransactionByBlockNumberAndIndex (block:*BlockType* , pos:*Quantity*)	Returns information about a transaction by block number and transaction index position.
Promise<TransactionDetail>	getTransactionByHash (hash:*Hash*)	Returns the information about a transaction requested by transaction hash.
`Promise<number>`	getTransactionCount (address:*Address* , block:*BlockType*)	Returns the number of transactions sent from an address. (as number)
Promise<TransactionReceipt>	getTransactionReceipt (hash:*Hash*)	Returns the receipt of a transaction by transaction hash. Note That the receipt is available even for pending transactions.

Continued on next page

Table 5 – continued from previous page

Promise<Block>	getUncleByBlockHashAndIndex (hash:*Hash* , pos:*Quantity*)	Returns information about a uncle of a block by hash and uncle index position. Note: An uncle doesn't contain individual transactions.
Promise<Block>	getUncleByBlockNumberAndIndex (block:*BlockType* , pos:*Quantity*)	Returns information about a uncle of a block number and uncle index position. Note: An uncle doesn't contain individual transactions.
`Promise<number>`	getUncleCountByBlockHash (hash:*Hash*)	Returns the number of uncles in a block from a block matching the given block hash.
`Promise<number>`	getUncleCountByBlockNumber (block:*BlockType*)	Returns the number of uncles in a block from a block matching the given block hash.
Buffer	hashMessage (data:*Data* | *Buffer*)	hash message
`Promise<string>`	newBlockFilter ()	Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.
`Promise<string>`	newFilter (filter:*LogFilter*)	Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call eth_getFilterChanges.
`Promise<string>`	newPendingTransactionFilter ()	Creates a filter in the node, to notify when new pending transactions arrive.

Continued on next page

Table 5 – continued from previous page

`Promise<string>`	protocolVersion ()	Returns the current ethereum protocol version.
`Promise<string>`	sendRawTransaction (data:*Data*)	Creates new message call transaction or a contract creation for signed transactions.
`Promise<>`	sendTransaction (args:*TxRequest*)	sends a Transaction
Promise<Signature>	sign (account:*Address* , data:*Data*)	signs any kind of message using the *x19Ethereum Signed Message:n*-prefix
`Promise<>`	syncing ()	Returns the current ethereum protocol version.
Promise<Quantity>	uninstallFilter (id:*Quantity*)	Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with eth_getFilterChanges for a period of time.

10.5.2 Type chainData

Source: modules/eth/chainData.ts

`Promise<any>`	callContract (client:*Client* , contract:`string`, chainId:`string`, signature:`string`, args:`any` [], config:*IN3Config*)	call contract
`Promise<>`	getChainData (client:*Client* , chainId:`string`, config:*IN3Config*)	get chain data

10.5.3 Type header

Source: modules/eth/header.ts

Interface	*AuthSpec*	Authority specification for proof of authority chains
Interface	*HistoryEntry*	the HistoryEntry
`Promise<void>`	addAuraValidators (history:*DeltaHistory<string>* , ctx:*ChainContext* , states:*HistoryEntry* [], contract:`string`)	add aura validators
`void`	addCliqueValidators (history:*DeltaHistory<string>* , ctx:*ChainContext* , states:*HistoryEntry* [])	add clique validators
`Promise<number>`	checkBlockSignatures (blockHeaders:`any` [], getChainSpec:)	verify a Blockheader and returns the percentage of finality
`void`	checkForFinality (stateBlockNumber:`number`, proof:*AuraValidatoryProof* , current:*Buffer* [], _finality:`number`)	check for finality
`Promise<void>`	checkForValidators (ctx:*ChainContext* , validators:*DeltaHistory<string>*)	check for validators
Promise<AuthSpec>	getChainSpec (b:*Block* , ctx:*ChainContext*)	get chain spec

10.5.4 Type Signer

Source: modules/eth/api.ts

Promise<Transaction>	prepareTransaction (client:*Client* , tx:*Transaction*)	optiional method which allows to change the transaction-data before sending it. This can be used for redirecting it through a multisig.
Promise<Signature>	sign (data:*Buffer* , account:*Address*)	signing of any data.
`Promise<boolean>`	hasAccount (account:*Address*)	returns true if the account is supported (or unlocked)

10.5.5 Type Transaction

Source: modules/eth/api.ts

`any`	chainId	optional chain id *(optional)*
`string`	data	4 byte hash of the method signature followed by encoded parameters. For details see Ethereum Contract ABI.
Address	from	20 Bytes - The address the transaction is send from.
Quantity	gas	Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
Quantity	gasPrice	Integer of the gas price used for each paid gas.
Quantity	nonce	nonce
Address	to	(optional when creating new contract) 20 Bytes - The address the transaction is directed to.
Quantity	value	Integer of the value sent with this transaction.

10.5.6 Type BlockType

Source: modules/eth/api.ts

= `number` | `'latest'` | `'earliest'` | `'pending'`

10.5.7 Type Address

Source: modules/eth/api.ts

= `string`

10.5.8 Type ABI

Source: modules/eth/api.ts

`boolean`	anonymous	the anonymous *(optional)*
`boolean`	constant	the constant *(optional)*
ABIField []	inputs	the inputs *(optional)*
`string`	name	the name *(optional)*
ABIField []	outputs	the outputs *(optional)*
`boolean`	payable	the payable *(optional)*
`'nonpayable'` | `'payable'` | `'view'` | `'pure'`	stateMutability	the stateMutability *(optional)*
`'event'` | `'function'` | `'constructor'` | `'fallback'`	type	the type

10.5.9 Type Log

Source: modules/eth/api.ts

Address	address	20 Bytes - address from which this log originated.
Hash	blockHash	Hash, 32 Bytes - hash of the block where this log was in. null when its pending. null when its pending log.
Quantity	blockNumber	the block number where this log was in. null when its pending. null when its pending log.
Data	data	contains the non-indexed arguments of the log.
Quantity	logIndex	integer of the log index position in the block. null when its pending log.
`boolean`	removed	true when the log was removed, due to a chain reorganization. false if its a valid log.
Data []	topics	- Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier.)
Hash	transactionHash	Hash, 32 Bytes - hash of the transactions this log was created from. null when its pending log.
Quantity	transactionIndex	integer of the transactions index position log was created from. null when its pending log.

10.5.10 Type Block

Source: modules/eth/api.ts

Address	author	20 Bytes - the address of the author of the block (the beneficiary to whom the mining rewards were given)
Quantity	difficulty	integer of the difficulty for this block
Data	extraData	the ‘extra data’ field of this block
Quantity	gasLimit	the maximum gas allowed in this block
Quantity	gasUsed	the total used gas by all transactions in this block
Hash	hash	hash of the block. null when its pending block
Data	logsBloom	256 Bytes - the bloom filter for the logs of the block. null when its pending block
Address	miner	20 Bytes - alias of ‘author’
Data	nonce	8 bytes hash of the generated proof-of-work. null when its pending block. Missing in case of PoA.
Quantity	number	The block number. null when its pending block
Hash	parentHash	hash of the parent block
Data	receiptsRoot	32 Bytes - the root of the receipts trie of the block

10.5.11 Type Hash

Source: modules/eth/api.ts

= `string`

10.5.12 Type Quantity

Source: modules/eth/api.ts

= `number` | *Hex*

10.5.13 Type LogFilter

Source: modules/eth/api.ts

Address	address	(optional) 20 Bytes - Contract address or a list of addresses from which logs should originate.
BlockType	fromBlock	Quantity or Tag - (optional) (default: latest) Integer block number, or 'latest' for the last mined block or 'pending', 'earliest' for not yet mined transactions.
Quantity	limit	å(optional) The maximum number of entries to retrieve (latest first).
BlockType	toBlock	Quantity or Tag - (optional) (default: latest) Integer block number, or 'latest' for the last mined block or 'pending', 'earliest' for not yet mined transactions.
`string` | `string` [] []	topics	(optional) Array of 32 Bytes Data topics. Topics are order-dependent. It's possible to pass in null to match any topic, or a subarray of multiple topics of which one should be matching.

10.5.14 Type TransactionDetail

Source: modules/eth/api.ts

Hash	blockHash	32 Bytes - hash of the block where this transaction was in. null when its pending.
BlockType	blockNumber	block number where this transaction was in. null when its pending.
Quantity	chainId	the chain id of the transaction, if any.
`any`	condition	(optional) conditional submission, Block number in block or timestamp in time or null. (parity-feature)
Address	creates	creates contract address
Address	from	20 Bytes - address of the sender.
Quantity	gas	gas provided by the sender.
Quantity	gasPrice	gas price provided by the sender in Wei.
Hash	hash	32 Bytes - hash of the transaction.
Data	input	the data send along with the transaction.
Quantity	nonce	the number of transactions made by the sender prior to this one.
`any`	pk	optional private key for signing *(optional)*

10.5.15 Type TransactionReceipt

Source: modules/eth/api.ts

Hash	blockHash	32 Bytes - hash of the block where this transaction was in.
BlockType	blockNumber	block number where this transaction was in.
Address	contractAddress	20 Bytes - The contract address created, if the transaction was a contract creation, otherwise null.
Quantity	cumulativeGasUsed	The total amount of gas used when this transaction was executed in the block.
Address	from	20 Bytes - The address of the sender.
Quantity	gasUsed	The amount of gas used by this specific transaction alone.
Log []	logs	Array of log objects, which this transaction generated.
Data	logsBloom	256 Bytes - A bloom filter of logs/events generated by contracts during transaction execution. Used to efficiently rule out transactions without expected logs.
Hash	root	32 Bytes - Merkle root of the state trie after the transaction has been executed (optional after Byzantium hard fork EIP609)
Quantity	status	0x0 indicates transaction failure , 0x1 indicates transaction success. Set for blocks mined after Byzantium hard fork EIP609, null before.

10.5.16 Type Data

Source: modules/eth/api.ts

= `string`

10.5.17 Type TxRequest

Source: modules/eth/api.ts

`any` []	args	the argument to pass to the method *(optional)*
`number`	confirmations	number of block to wait before confirming *(optional)*
Data	data	the data to send *(optional)*
Address	from	address of the account to use *(optional)*
`number`	gas	the gas needed *(optional)*
`number`	gasPrice	the gasPrice used *(optional)*
`string`	method	the ABI of the method to be used *(optional)*
`number`	nonce	the nonce *(optional)*
Hash	pk	raw private key in order to sign *(optional)*
Address	to	contract *(optional)*
Quantity	value	the value in wei *(optional)*

10.5.18 Type AuthSpec

Source: modules/eth/header.ts

Authority specification for proof of authority chains

Buffer []	authorities	List of validator addresses storead as an buffer array
Buffer	proposer	proposer of the block this authspec belongs
ChainSpec	spec	chain specification

10.5.19 Type HistoryEntry

Source: modules/eth/header.ts

`number`	block	the block
AuraValidatoryProof | `string` []	proof	the proof
`string` []	validators	the validators

10.5.20 Type ABIField

Source: modules/eth/api.ts

`boolean`	indexed	the indexed *(optional)*
`string`	name	the name
`string`	type	the type

10.5.21 Type Hex

Source: modules/eth/api.ts

= `string`

10.6 Package modules/ipfs

10.6.1 Type IpfsAPI

Source: modules/ipfs/api.ts

simple API for IPFS

IpfsAPI	constructor (_client:*Client*)	simple API for IPFS
Client	client	the client
Promise<Buffer>	get (hash:`string`, resultEncoding:`string`)	retrieves the conent for a hash from IPFS.
`Promise<string>`	put (data:*Buffer* , dataEncoding:`string`)	stores the data on ipfs and returns the IPFS-Hash.

10.7 Package util

a collection of util classes inside incubed. They can be get directly through `require('in3/js/srrc/util/util')`

10.7.1 Type DeltaHistory

Source: util/DeltaHistory.ts

DeltaHistory	constructor (init:*T* [], deltaStrings:`boolean`)	constructor
Delta<T> []	data	the data
`void`	addState (start:`number`, data:*T* [])	add state
T []	getData (index:`number`)	get data
`number`	getLastIndex ()	get last index
`void`	loadDeltaStrings (deltas:`string` [])	load delta strings
`string` []	toDeltaStrings ()	to delta strings

10.7.2 Type Delta

Source: util/DeltaHistory.ts

This file is part of the Incubed project. Sources: https://github.com/slockit/in3

`number`	block	the block
T []	data	the data
`number`	len	the len
`number`	start	the start

10.8 Common Module

The common module (in3-common) contains all the typedefs used in the node and server.

Interface	*BlockData*	the BlockData
Interface	*LogData*	the LogData
Type	*Receipt*	the Receipt
Interface	*ReceiptData*	the ReceiptData
Type	*Transaction*	the Transaction
Interface	*TransactionData*	the TransactionData
Interface	*Transport*	the Transport
AxiosTransport	AxiosTransport	the AxiosTransport value= `_transport.AxiosTransport`

Continued on next page

Table 6 – continued from previous page

Block	Block	the Block value= `_serialize.Block`
`any`	address (val:`any`)	converts it to a Buffer with 20 bytes length
Block	blockFromHex (hex:`string`)	converts a hexstring to a block-object
`any`	bytes (val:`any`)	converts it to a Buffer
`any`	bytes32 (val:`any`)	converts it to a Buffer with 32 bytes length
`any`	bytes8 (val:`any`)	converts it to a Buffer with 8 bytes length
cbor	cbor	the cbor value= `_cbor`
	chainAliases	the chainAliases value= `_util.aliases`
`number` []	createRandomIndexes (len:`number`, limit:`number`, seed:*Buffer* , result:`number` [])	create random indexes
`any`	createTx (transaction:`any`)	creates a Transaction-object from the rpc-transaction-data
Buffer	getSigner (data:*Block*)	get signer

Continued on next page

Table 6 – continued from previous page

Buffer	hash (val:*Block* | *Transaction* | *Receipt* | *Account* | *Buffer*)	returns the hash of the object
index	rlp	the rlp value= `_serialize.rlp`
serialize	serialize	the serialize value= `_serialize`
storage	storage	the storage value= `_storage`
Buffer []	toAccount (account:*AccountData*)	to account
Buffer []	toBlockHeader (block:*BlockData*)	create a Buffer[] from RPC-Response
`Object`	toReceipt (r:*ReceiptData*)	create a Buffer[] from RPC-Response
Buffer []	toTransaction (tx:*TransactionData*)	create a Buffer[] from RPC-Response
transport	transport	the transport value= `_transport`
`any`	uint (val:`any`)	converts it to a Buffer with a variable length. 0 = length 0
`any`	uint128 (val:`any`)	uint128

Continued on next page

Table 6 – continued from previous page

any	uint64 (val:any)	uint64
util	util	the util value= _util
validate	validate	the validate value= _validate

10.9 Package index.ts

10.9.1 Type BlockData

Source: index.ts

Block as returned by eth_getBlockByNumber Block as returned by eth_getBlockByNumber

`string`	coinbase	the coinbase *(optional)*
`string|number`	difficulty	the difficulty
`string`	extraData	the extraData
`string|number`	gasLimit	the gasLimit
`string|number`	gasUsed	the gasUsed
`string`	hash	the hash
`string`	logsBloom	the logsBloom
`string`	miner	the miner
`string`	mixHash	the mixHash *(optional)*
`string|number`	nonce	the nonce *(optional)*
`string|number`	number	the number
`string`	parentHash	the parentHash
`string`	receiptRoot	the receiptRoot *(optional)*
`string`	receiptsRoot	the receiptsRoot
`string[]`	sealFields	the sealFields *(optional)*
`string`	sha3Uncles	the sha3Uncles

10.9.2 Type LogData

Source: index.ts

LogData as part of the TransactionReceipt LogData as part of the TransactionReceipt

`string`	address	the address
`string`	blockHash	the blockHash
`string`	blockNumber	the blockNumber
`string`	data	the data
`string`	logIndex	the logIndex
`boolean`	removed	the removed
`string` []	topics	the topics
`string`	transactionHash	the transactionHash
`string`	transactionIndex	the transactionIndex
`string`	transactionLogIndex	the transactionLogIndex

10.9.3 Type ReceiptData

Source: index.ts

TransactionReceipt as returned by eth_getTransactionReceipt TransactionReceipt as returned by eth_getTransactionReceipt

string	blockHash	the blockHash *(optional)*
string|number	blockNumber	the blockNumber *(optional)*
string|number	cumulativeGasUsed	the cumulativeGasUsed *(optional)*
string|number	gasUsed	the gasUsed *(optional)*
LogData []	logs	the logs
string	logsBloom	the logsBloom *(optional)*
string	root	the root *(optional)*
string|boolean	status	the status *(optional)*
string	transactionHash	the transactionHash *(optional)*
number	transactionIndex	the transactionIndex *(optional)*

10.9.4 Type TransactionData

Source: index.ts

Transaction as returned by eth_getTransactionByHash Transaction as returned by eth_getTransactionByHash

`string`	blockHash	the blockHash *(optional)*
`number | string`	blockNumber	the blockNumber *(optional)*
`number | string`	chainId	the chainId *(optional)*
`string`	condition	the condition *(optional)*
`string`	creates	the creates *(optional)*
`string`	data	the data *(optional)*
`string`	from	the from *(optional)*
`number | string`	gas	the gas *(optional)*
`number | string`	gasLimit	the gasLimit *(optional)*
`number | string`	gasPrice	the gasPrice *(optional)*
`string`	hash	the hash
`string`	input	the input
`number | string`	nonce	the nonce
`string`	publicKey	the publicKey *(optional)*
`string`	r	the r *(optional)*
`string`	raw	the raw *(optional)*
`string`	s	the s *(optional)*

10.9.5 Type Transport

Source: index.ts

A Transport-object responsible to transport the message to the handler. A Transport-object responsible to transport the message to the handler.

`Promise<>`	handle (url:`string`, data:*RPCRequest* | *RPCRequest* [], timeout:`number`)	handles a request by passing the data to the handler
`Promise<boolean>`	isOnline ()	check whether the handler is onlne.
`number` []	random (count:`number`)	generates random numbers (between 0-1)

10.10 Package modules/eth

10.10.1 Type Block

Source: modules/eth/serialize.ts

encodes and decodes the blockheader

Block	constructor (data:*Buffer* | `string` | *BlockData*)	creates a Block-Onject from either the block-data as returned from rpc, a buffer or a hex-string of the encoded blockheader
BlockHeader	raw	the raw Buffer fields of the BlockHeader
Tx []	transactions	the transaction-Object (if given)
Buffer	bloom	bloom
Buffer	coinbase	coinbase
Buffer	difficulty	difficulty
Buffer	extra	extra
Buffer	gasLimit	gas limit
Buffer	gasUsed	gas used
Buffer	number	number
Buffer	parentHash	parent hash
Buffer	receiptTrie	receipt trie
Buffer []	sealedFields	sealed fields
Buffer	stateRoot	state root
Buffer	timestamp	timestamp

10.10.2 Type Transaction

Source: modules/eth/serialize.ts

Buffer[] of the transaction = *Buffer* []

10.10.3 Type Receipt

Source: modules/eth/serialize.ts

Buffer[] of the Receipt = [*Buffer* ,*Buffer* ,*Buffer* ,*Buffer* , *Buffer* [] , *Buffer* []]

10.10.4 Type Account

Source: modules/eth/serialize.ts

Buffer[] of the Account = *Buffer* []

10.10.5 Type serialize

Source: modules/eth/serialize.ts

Class	*Block*	encodes and decodes the blockheader
Interface	*AccountData*	Account-Object
Interface	*BlockData*	Block as returned by eth_getBlockByNumber
Interface	*LogData*	LogData as part of the TransactionReceipt
Interface	*ReceiptData*	TransactionReceipt as returned by eth_getTransactionReceipt
Interface	*TransactionData*	Transaction as returned by eth_getTransactionByHash
Type	*Account*	Buffer[] of the Account
Type	*BlockHeader*	Buffer[] of the header
Type	*Receipt*	Buffer[] of the Receipt
Type	*Transaction*	Buffer[] of the transaction
index	rlp	RLP-functions value= `ethUtil.rlp`
`any`	address (val:`any`)	converts it to a Buffer with 20 bytes length
Block	blockFromHex (hex:`string`)	converts a hexstring to a block-object
`string`	blockToHex (block:`any`)	converts blockdata to a hexstring

10.10.6 Type storage

Source: modules/eth/storage.ts

<table>
<tr><td>any</td><td>getStorageArrayKey (
pos:number,
arrayIndex:number,
structSize:number,
structPos:number)</td><td>calc the storrage array key</td></tr>
<tr><td>any</td><td>getStorageMapKey (
pos:number,
key:string,
structPos:number)</td><td>calcs the storage Map key.</td></tr>
<tr><td>Promise<></td><td>getStorageValue (
rpc:string,
contract:string,
pos:number,
type:'address'
| 'bytes32'
| 'bytes16'
| 'bytes4'
| 'int'
| 'string',
keyOrIndex:number
| string,
structSize:number,
structPos:number)</td><td>get a storage value from the server</td></tr>
<tr><td>string |</td><td>getStringValue (
data:Buffer ,
storageKey:Buffer)</td><td>creates a string from storage.</td></tr>
<tr><td>string</td><td>getStringValueFromList (
values:Buffer [],
len:number)</td><td>concats the storage values to a string.</td></tr>
<tr><td>BN</td><td>toBN (
val:any)</td><td>converts any value to BN</td></tr>
</table>

10.10.7 Type AccountData

Source: modules/eth/serialize.ts

Account-Object

`string`	balance	the balance
`string`	code	the code *(optional)*
`string`	codeHash	the codeHash
`string`	nonce	the nonce
`string`	storageHash	the storageHash

10.10.8 Type BlockHeader

Source: modules/eth/serialize.ts

Buffer[] of the header = *Buffer* []

10.11 Package types

10.11.1 Type RPCRequest

Source: types/types.ts

a JSONRPC-Request with N3-Extension

`number|string`	id	the identifier of the request example: 2 *(optional)*
IN3RPCRequestConfig	in3	the IN3-Config *(optional)*
`'2.0'`	jsonrpc	the version
`string`	method	the method to call example: eth_getBalance
`any` []	params	the params example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,lates *(optional)*

10.11.2 Type RPCResponse

Source: types/types.ts

a JSONRPC-Responset with N3-Extension

`string`	error	in case of an error this needs to be set *(optional)*
`string|number`	id	the id matching the request example: 2
IN3ResponseConfig	in3	the IN3-Result *(optional)*
IN3NodeConfig	in3Node	the node handling this response (internal only) *(optional)*
`'2.0'`	jsonrpc	the version
`any`	result	the params example: 0xa35bc *(optional)*

10.11.3 Type IN3RPCRequestConfig

Source: types/types.ts

additional config for a IN3 RPC-Request

`string`	chainId	the requested chainId example: 0x1
`any`	clientSignature	the signature of the client *(optional)*
`number`	finality	if given the server will deliver the blockheaders of the following blocks until at least the number in percent of the validators is reached. *(optional)*
`boolean`	includeCode	if true, the request should include the codes of all accounts. otherwise only the the codeHash is returned. In this case the client may ask by calling eth_getCode() afterwards example: true *(optional)*
`number`	latestBlock	if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6 *(optional)*
`string` []	signatures	a list of addresses requested to sign the blockhash example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679 *(optional)*
`boolean`	useBinary	if true binary-data will be used. *(optional)*
`boolean`	useFullProof	if true all data in the response will be proven, which leads to a higher payload. *(optional)*
`boolean`	useRef	if true binary-data (starting with a 0x) will be refered if occuring again. *(optional)*

10.11.4 Type IN3ResponseConfig

Source: types/types.ts

additional data returned from a IN3 Server

`number`	currentBlock	the current blocknumber. example: 320126478 *(optional)*
`number`	lastNodeList	the blocknumber for the last block updating the nodelist. If the client has a smaller blocknumber he should update the nodeList. example: 326478 *(optional)*
`number`	lastValidatorChange	the blocknumber of gthe last change of the validatorList *(optional)*
Proof	proof	the Proof-data *(optional)*
`string`	version	the in3 protocol version. example: 1.0.0 *(optional)*

10.11.5 Type IN3NodeConfig

Source: types/types.ts

a configuration of a in3-server.

`string`	address	the address of the node, which is the public address it iis signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
`number`	capacity	the capacity of the node. example: 100 *(optional)*
`string` []	chainIds	the list of supported chains example: 0x1
`number`	deposit	the deposit of the node in wei example: 12350000
`number`	index	the index within the contract example: 13 *(optional)*
`number`	props	the properties of the node. example: 3 *(optional)*
`number`	registerTime	the UNIX-timestamp when the node was registered example: 1563279168 *(optional)*
`number`	timeout	the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600 *(optional)*
`number`	unregisterTime	the UNIX-timestamp when the node is allowed to be deregister example: 1563279168 *(optional)*
`string`	url	the endpoint to post to example: https://in3.slock.it

10.11.6 Type Proof

Source: types/types.ts

the Proof-data as part of the in3-section

	accounts	a map of addresses and their AccountProof *(optional)*
`string`	block	the serialized blockheader as hex, required in most proofs example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6 *(optional)*
`any` []	finalityBlocks	the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6 *(optional)*
LogProof	logProof	the Log Proof in case of a Log-Request *(optional)*
`string` []	merkleProof	the serialized merle-noodes beginning with the root-node *(optional)*
`string` []	merkleProofPrev	the serialized merkle-noodes beginning with the root-node of the previous entry (only for full proof of receipts) *(optional)*
Signature []	signatures	requested signatures *(optional)*
`any` []	transactions	the list of transactions of the block example: *(optional)*
`number`	txIndex	the transactionIndex within the block example: 4 *(optional)*
`string` []	txProof	the serialized merkle-nodes beginning with the root-node in order to prrof the transactionIndex *(optional)*

10.11.7 Type LogProof

Source: types/types.ts

a Object holding proofs for event logs. The key is the blockNumber as hex

10.11.8 Type Signature

Source: types/types.ts

Verified ECDSA Signature. Signatures are a pair (r, s). Where r is computed as the X coordinate of a point R, modulo the curve order n.

`string`	address	the address of the signing node example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679 *(optional)*
`number`	block	the blocknumber example: 3123874
`string`	blockHash	the hash of the block example: 0x6C1a01C2aB554930A937B0a212346037E8105fB4794
`string`	msgHash	hash of the message example: 0x9C1a01C2aB554930A937B0a212346037E8105fB4794
`string`	r	Positive non-zero Integer signature.r example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6
`string`	s	Positive non-zero Integer signature.s example: 0x6d17b34aeaf95fee98c0437b4ac839d8a2ece1b18166da
`number`	v	Calculated curve point, or identity element O. example: 28

10.12 Package util

10.12.1 Type AxiosTransport

Source: util/transport.ts

Default Transport impl sending http-requests.

AxiosTransport	constructor (format:`'json'` | `'cbor'` | `'jsonRef'`)	Default Transport impl sending http-requests.
`'json'` | `'cbor'` | `'jsonRef'`	format	the format
`Promise<>`	handle (url:`string`, data:*RPCRequest* | *RPCRequest* [], timeout:`number`)	handle
`Promise<boolean>`	isOnline ()	is online
`number` []	random (count:`number`)	random

10.12.2 Type cbor

Source: util/cbor.ts

`any`	convertToBuffer (val:`any`)	convert to buffer
`any`	convertToHex (val:`any`)	convert to hex
T	createRefs (val:*T* , cache:`string` [])	create refs
RPCRequest []	decodeRequests (request:*Buffer*)	decode requests
RPCResponse []	decodeResponses (responses:*Buffer*)	decode responses
Buffer	encodeRequests (requests:*RPCRequest* [])	turn
Buffer	encodeResponses (responses:*RPCResponse* [])	encode responses
T	resolveRefs (val:*T* , cache:`string` [])	resolve refs

10.12.3 Type transport

Source: util/transport.ts

Class	*AxiosTransport*	Default Transport impl sending http-requests.
Interface	*Transport*	A Transport-object responsible to transport the message to the handler.

10.12.4 Type util

Source: util/util.ts

BN	BN	the BN value= `ethUtil.BN`
`any`	Buffer	This file is part of the Incubed project. Sources: https://github.com/slockit/in3-common value= `require('buffer').Buffer`
T	checkForError (res:*T*)	check a RPC-Response for errors and rejects the promise if found
`number` []	createRandomIndexes (len:`number`, limit:`number`, seed:*Buffer* , result:`number` [])	create random indexes
`string`	fixLength (hex:`string`)	fix length
`string`	getAddress (pk:`string`)	returns a address from a private key
Buffer	getSigner (data:*Block*)	get signer
`string`	padEnd (val:`string`, minLength:`number`, fill:`string`)	padEnd for legacy
`string`	padStart (val:`string`, minLength:`number`, fill:`string`)	padStart for legacy
`Promise<any>`	promisify (	simple promisy-function

10.12.5 Type validate

Source: util/validate.ts

Ajv	ajv	the ajv instance with custom formatters and keywords value= `new Ajv()`
`void`	validate (ob:`any`, def:`any`)	validate
`void`	validateAndThrow (fn:*Ajv.ValidateFunction* , ob:`any`)	validates the data and throws an error in case they are not valid.

CHAPTER 11

API Reference WASM

Even though the incubed client is written in C, we are using emscripten to build wasm. Together with some binding-code incubed runs in any Javascript-Runtime. Using WASM gives us 3 important features:

1. Performance. Since WASM runs at almost native speed it is very fast
2. Security Since the WASM-Module has no dependencies it reduces the risk of using a malicious dependency, which would be able to manipulate Prototypes. Also, since the real work is happening inside the wasm, trying to change Prototype would not work.
3. Size The current wasm-file is about 200kb. This is smaller then most other libraries and can easily be used in any app or website.

11.1 Installing

This client uses the in3-core sources compiled to wasm. The wasm is included into the js-file wich makes it easier to include the data. This module has **no** dependencies! All it needs is included inta a wasm of about 300kB.

Installing incubed is as easy as installing any other module:

```
npm install --save in3-wasm
```

11.1.1 WASM-support

Even though most browsers and javascript enviroment such as nodejs, have full support for wasm, there are ocasions, where WASM is fully supported. In case you want to run incubed within a react native app, you might face such issues. In this case you can use in3-asmjs, which has the same API, but runs on pure javascript (a bit slower and bigger, but full support everywhere).

11.2 Building from Source

11.2.1 install emscripten

In order to build the wasm or asmjs from source you need to install emscripten first. In case you have not done it yet:

```
# Get the emsdk repo
git clone https://github.com/emscripten-core/emsdk.git

# Enter that directory
cd emsdk

# install the latest-upstream sdk and activate it
./emsdk install latest-upstream && ./emsdk activate latest-upstream
```

```
# Please make sure you add this line to your .bash_profile or .zshrc
source <PATH_TO_EMSDK>/emsdk_env.sh > /dev/null
```

11.2.2 CMake

With emscripten set up, you can now configure the wasm and build it (in the in3-c directory):

```
# create a build directory
mkdir -p build
cd build

# configure CMake
emcmake cmake -DWASM=true -DCMAKE_BUILD_TYPE=MINSIZEREL ..

# and build it
make -j8 in3_wasm

# optionally you can also run the tests
make test
```

Per default the generated wasm embedded the wasm-data as base64 and resulted in the build/module. If you want to build asmjs, use the `-DASMJS=true` as an additional option. If you don't want to embedd the wasm, add `-DWASM_EMBED=false`. If you want to set the `-DCMAKE_BUILD_TYPE=DEBUG` your filesize increases but all function names are kept (resulting in readable stacktraces) and emscriptten will add a lot of checks and assertions.

For more options please see the CMake Options.

11.3 Examples

11.3.1 get_block_rpc

source : in3-c/wasm/examples/get_block_rpc.js

read block as rpc

```
/// read block as rpc
```

(continues on next page)

(continued from previous page)

```
const IN3 = require('in3-wasm')

async function showLatestBlock() {
    // create new incubed instance
    var c = new IN3()

    await c.setConfig({
        chainId: 0x5 // use goerli
    })

    // send raw RPC-Request (this would throw if the response contains an error)
    const lastBlockResponse = await c.sendRPC('eth_getBlockByNumber', ['latest', false])

    console.log("latest Block: ", JSON.stringify(lastBlockResponse, null, 2))

    // clean up
    c.free()

}

showLatestBlock().catch(console.error)
```

11.3.2 get_block_api

source : in3-c/wasm/examples/get_block_api.ts

read block with API

```
/// read block with API

import { IN3 } from 'in3-wasm'

async function showLatestBlock() {
  // create new incubed instance
  const client = new IN3({
    chainId: 'goerli'
  })

  // send raw RPC-Request
  const lastBlock = await client.eth.getBlockByNumber()

  console.log("latest Block: ", JSON.stringify(lastBlock, null, 2))

  // clean up
  client.free()

}

showLatestBlock().catch(console.error)
```

11.3.3 register_pugin

source : in3-c/wasm/examples/register_pugin.ts register a custom plugin

```
/// register a custom plugin

import { IN3, RPCRequest } from 'in3-wasm'
import * as crypto from 'crypto'

class Sha256Plugin {

  // this function will register for handling rpc-methods
  // only if we return something other then `undefined`, it will be taken as the␣
↪result of the rpc.
  // if we don't return, the request will be forwarded to the incubed nodes
  handleRPC(c: IN3, request: RPCRequest): any {
    if (request.method === 'sha256') {
      // assert params
      if (request.params.length != 1 || typeof (request.params[0]) != 'string')
        throw new Error('Only one parameter with as string is expected!')

      // create hash
      const hash = crypto.createHash('sha256').update(Buffer.from(request.params[0],
↪'utf8')).digest()

      // return the result
      return '0x' + hash.toString('hex')
    }
  }

}



async function registerPlugin() {
  // create new incubed instance
  const client = new IN3()

  // register the plugin
  client.registerPlugin(new Sha256Plugin())

  // exeucte a rpc-call
  const result = await client.sendRPC("sha256", ["testdata"])

  console.log(" sha256: ", result)

  // clean up
  client.free()

}

registerPlugin().catch(console.error)
```

11.3.4 use_web3

source : in3-c/wasm/examples/use_web3.ts

use incubed as Web3Provider in web3js

```
/// use incubed as Web3Provider in web3js

// import in3-Module
import { IN3 } from 'in3-wasm'
const Web3 = require('web3')

const in3 = new IN3({
    proof: 'standard',
    signatureCount: 1,
    requestCount: 1,
    chainId: 'mainnet',
    replaceLatestBlock: 10
})

// use the In3Client as Http-Provider
const web3 = new Web3(in3.createWeb3Provider());

(async () => {

    // use the web3
    const block = await web3.eth.getBlock('latest')
    console.log("Block : ", block)

})().catch(console.error);
```

11.3.5 in3_in_browser

source : in3-c/wasm/examples/in3_in_browser.html

use incubed directly in html

```
<!-- use incubed directly in html -->
<html>

<head>
    <script src="node_modules/in3-wasm/index.js"></script>
</head>

<body>
    IN3-Demo
    <div>
        result:
        <pre id="result"> ...waiting... </pre>
    </div>
    <script>
        var in3 = new IN3({ chainId: 0x1, replaceLatestBlock: 6, requestCount: 3 });
        in3.eth.getBlockByNumber('latest', false)
            .then(block => document.getElementById('result').innerHTML = JSON.
↪stringify(block, null, 2))
            .catch(alert)
            .finally(() => in3.free())
    </script>
</body>

</html>
```

11.3.6 Building

In order to run those examples, you need to install in3-wasm and typescript first. The build.sh will do this and the run the tsc-compiler

```
./build.sh
```

In order to run a example use

```
node build/get_block_api.ts
```

11.4 Incubed Module

This page contains a list of all Datastructures and Classes used within the IN3 WASM-Client

Importing incubed is as easy as

```
import {IN3} from "in3-wasm"
```

11.4.1 BufferType and BigIntType

The WASM-Module comes with no dependencies. This means per default it uses the standard classes provided as part of the EMCAScript-Standard.

If you work with a library which expects different types, you can change the generic-type and giving a converter:

Type BigIntType

Per default we use `bigint`. This is used whenever we work with number too big to be stored as a `number`-type.

If you want to change this type, use *setConverBigInt()* function.

Type Buffer

Per default we use `UInt8Array`. This is used whenever we work with raw bytes.

If you want to change this type, use *setConverBuffer()* function.

Generics

```
import {IN3Generic} from 'in3-wasm'
import BN from 'bn.js'

// create a new client by setting the Generic Types
const c = new IN3Generic<BN,Buffer>()

// set the converter-functions
IN3Generic.setConverBuffer(val => Buffer.from(val))
IN3Generic.setConverBigInt(val => new BN(val))
```

11.4.2 Package

While the In3Client-class is also the default import, the following imports can be used:

IN3	Class	default Incubed client with bigint for big numbers Uint8Array for bytes
IN3Generic	Class	the IN3Generic
SimpleSigner	Class	the SimpleSigner
AccountAPI	Interface	The Account API
BTCBlock	Interface	a full Block including the transactions
BTCBlockHeader	Interface	a Block header
BlockInfo	Interface	the BlockInfo
BtcAPI	Interface	API for handling BitCoin data
BtcTransaction	Interface	a BitCoin Transaction.
BtcTransactionInput	Interface	a Input of a Bitcoin Transaction
BtcTransactionOutput	Interface	a Input of a Bitcoin Transaction
DepositResponse	Interface	the DepositResponse
ETHOpInfoResp	Interface	the ETHOpInfoResp
EthAPI	Interface	The API for ethereum operations.

Continued on next page

Table 1 – continued from previous page

Fee	Interface	the Fee
IN3Config	Interface	the configuration of the IN3-Client. This can be changed at any time. All properties are optional and will be verified when sending the next request.
IN3NodeConfig	Interface	a configuration of a in3-server.
IN3NodeWeight	Interface	a local weight of a n3-node. (This is used internally to weight the requests)
IN3Plugin	Interface	a Incubed plugin. Depending on the methods this will register for those actions.
IpfsAPI	Interface	API for storing and retrieving IPFS-data.
RPCRequest	Interface	a JSONRPC-Request with N3-Extension
RPCResponse	Interface	a JSONRPC-Responset with N3-Extension
Signer	Interface	the Signer
Token	Interface	the Token
Tokens	Interface	the Tokens
TxInfo	Interface	the TxInfo

Continued on next page

Table 1 – continued from previous page

TxType	Interface	the TxType
Utils	Interface	Collection of different util-functions.
Web3Contract	Interface	the Web3Contract
Web3Event	Interface	the Web3Event
Web3TransactionObject	Interface	the Web3TransactionObject
ZKAccountInfo	Interface	the ZKAccountInfo
ZksyncAPI	Interface	API for zksync.
ABI	Type literal	the ABI
ABIField	Type literal	the ABIField
Address	Type alias	a 20 byte Address encoded as Hex (starting with 0x)
Block	Type literal	the Block
BlockType	Type	BlockNumber or predefined Block
Data	Type alias	data encoded as Hex (starting with 0x)
Hash	Type alias	a 32 byte Hash encoded as Hex (starting with 0x)
Hex	Type	a Hexcoded String (starting with 0x)

Continued on next page

Table 1 – continued from previous page

Log	Type literal	the Log
LogFilter	Type literal	the LogFilter
Quantity	Type	a BigInteger encoded as hex.
Signature	Type literal	Signature
Transaction	Type literal	the Transaction
TransactionDetail	Type literal	the TransactionDetail
TransactionReceipt	Type literal	the TransactionReceipt
TxRequest	Type literal	the TxRequest
btc_config	Interface	bitcoin configuration.
zksync_config	Interface	zksync configuration.

11.5 Package index

11.5.1 Type IN3

Source: index.d.ts

default Incubed client with bigint for big numbers Uint8Array for bytes

default	*IN3Generic*	supporting both ES6 and UMD usage
util	*Utils<any>*	collection of util-functions.
btc	*BtcAPI<Uint8Array>*	btc API
config	*IN3Config*	IN3 config
eth	*EthAPI<bigint,Uint8Array>*	eth1 API.
ipfs	*IpfsAPI<Uint8Array>*	ipfs API
signer	*Signer<bigint,Uint8Array>*	the signer, if specified this interface will be used to sign transactions, if not, sending transaction will not be possible.
util	*Utils<Uint8Array>*	collection of util-functions.
zksync	*ZksyncAPI<Uint8Array>*	zksync API

freeAll()

frees all Incubed instances.

static `void` freeAll ()

onInit()

registers a function to be called as soon as the wasm is ready. If it is already initialized it will call it right away.

static *Promise<T>* onInit (fn:() => *T*)

Parameters:

fn	() => *T*	the function to call

Returns:

static *Promise<T>*

setConvertBigInt()

set convert big int

static any setConvertBigInt (convert:(`any`) => `any`)

Parameters:

convert	(`any`) => `any`	convert

Returns:

static `any`

setConvertBuffer()

set convert buffer

static any setConvertBuffer (convert:(`any`) => `any`)

Parameters:

convert	(`any`) => `any`	convert

Returns:

static `any`

setStorage()

changes the storage handler, which is called to read and write to the cache.

static void setStorage (handler:)

Parameters:

handler		handler

setTransport()

changes the default transport-function.

static `void` setTransport (fn:(`string`, `string`, `number`) => `Promise<string>`)

Parameters:

fn	(`string`, `string`, `number`) => `Promise<string>`	the function to fetch the response for the given url

constructor()

creates a new client.

IN3 **constructor** (config:*Partial<IN3Config>*)

Parameters:

config	*Partial<IN3Config>*	a optional config

Returns:

IN3

createWeb3Provider()

returns a Object, which can be used as Web3Provider.

```
const web3 = new Web3(new IN3().createWeb3Provider())
```

`any` createWeb3Provider ()

Returns:

`any`

free()

disposes the Client. This must be called in order to free allocated memory!

`any` free ()

Returns:

`any`

registerPlugin()

rregisters a plugin. The plugin may define methods which will be called by the client.

void registerPlugin (plugin:*IN3Plugin<bigint,Uint8Array>*)

Parameters:

plugin	*IN3Plugin<bigint,Uint8Array>*	the plugin-object to register

send()

sends a raw request. if the request is a array the response will be a array as well. If the callback is given it will be called with the response, if not a Promise will be returned. This function supports callback so it can be used as a Provider for the web3.

***Promise<RPCResponse>* send** (request:*RPCRequest* , callback:(*Error* , *RPCResponse*) => `void`)

Parameters:

request	*RPCRequest*	a JSONRPC-Request with N3-Extension
callback	(*Error* , *RPCResponse*) => `void`	callback

Returns:

Promise<RPCResponse>

sendRPC()

sends a RPC-Requests specified by name and params.

if the response contains an error, this will be thrown. if not the result will be returned.

Promise<any> sendRPC (method:`string`, params:`any` [])

Parameters:

method	`string`	the method to call.
params	`any` []	params

Returns:

`Promise<any>`

sendSyncRPC()

sends a RPC-Requests specified by name and params as a sync call. This is only alowed if the request is handled internally, like web3_sha3,

if the response contains an error, this will be thrown. if not the result will be returned.

any sendSyncRPC (method:`string`, params:`any` [])

Parameters:

method	`string`	the method to call.
params	`any` []	params

Returns:

`any`

setConfig()

sets configuration properties. You can pass a partial object specifieing any of defined properties.

void setConfig (config:*Partial<IN3Config>*)

Parameters:

config	*Partial<IN3Config>*	config

11.5.2 Type IN3Generic

Source: index.d.ts

default	*IN3Generic*	supporting both ES6 and UMD usage
util	*Utils<any>*	collection of util-functions.
btc	*BtcAPI<BufferType>*	btc API
config	*IN3Config*	IN3 config
eth	*EthAPI<BigIntType,BufferType>*	eth1 API.
ipfs	*IpfsAPI<BufferType>*	ipfs API
signer	*Signer<BigIntType,BufferType>*	the signer, if specified this interface will be used to sign transactions, if not, sending transaction will not be possible.
util	*Utils<BufferType>*	collection of util-functions.
zksync	*ZksyncAPI<BufferType>*	zksync API

freeAll()

frees all Incubed instances.

static `void` freeAll ()

onInit()

registers a function to be called as soon as the wasm is ready. If it is already initialized it will call it right away.

static *Promise<T>* onInit (fn:() => *T* **)**

Parameters:

fn	() => *T*	the function to call

Returns:

static *Promise<T>*

setConvertBigInt()

set convert big int

static any setConvertBigInt (convert:(any) => any)

Parameters:

convert	(any) => any	convert

Returns:

static any

setConvertBuffer()

set convert buffer

static any setConvertBuffer (convert:(any) => any)

Parameters:

convert	(any) => any	convert

Returns:

static any

setStorage()

changes the storage handler, which is called to read and write to the cache.

static void setStorage (handler:)

Parameters:

handler		handler

setTransport()

changes the default transport-function.

static void setTransport (fn:(string , string , number) => Promise<string>)

Parameters:

fn	(string , string , number) => Promise<string>	the function to fetch the response for the given url

constructor()

creates a new client.

IN3Generic **constructor** (config:*Partial<IN3Config>*)

Parameters:

config	*Partial<IN3Config>*	a optional config

Returns:

IN3Generic

createWeb3Provider()

returns a Object, which can be used as Web3Provider.

```
const web3 = new Web3(new IN3().createWeb3Provider())
```

any createWeb3Provider ()

Returns:

any

free()

disposes the Client. This must be called in order to free allocated memory!

any free ()

Returns:

any

registerPlugin()

rregisters a plugin. The plugin may define methods which will be called by the client.

void registerPlugin (plugin:*IN3Plugin<BigIntType,BufferType>*)

Parameters:

plugin	*IN3Plugin<BigIntType,BufferType>*	the plugin-object to register

send()

sends a raw request. if the request is a array the response will be a array as well. If the callback is given it will be called with the response, if not a Promise will be returned. This function supports callback so it can be used as a Provider for the web3.

Promise<RPCResponse> **send** (request:*RPCRequest* , callback:(*Error* , *RPCResponse*) => `void`)

Parameters:

request	*RPCRequest*	a JSONRPC-Request with N3-Extension
callback	(*Error* , *RPCResponse*) => `void`	callback

Returns:

Promise<RPCResponse>

sendRPC()

sends a RPC-Requests specified by name and params.

if the response contains an error, this will be thrown. if not the result will be returned.

Promise<any> sendRPC (method:`string`, params:`any` [])

Parameters:

method	`string`	the method to call.
params	`any` []	params

Returns:

`Promise<any>`

sendSyncRPC()

sends a RPC-Requests specified by name and params as a sync call. This is only alowed if the request is handled internally, like web3_sha3,

if the response contains an error, this will be thrown. if not the result will be returned.

any sendSyncRPC (method:`string`, params:`any` [])

Parameters:

method	`string`	the method to call.
params	`any` []	params

Returns:

`any`

setConfig()

sets configuration properties. You can pass a partial object specifieing any of defined properties.

void setConfig (config:*Partial<IN3Config>*)

Parameters:

config	*Partial<IN3Config>*	config

11.5.3 Type SimpleSigner

Source: index.d.ts

accounts		the accounts

constructor()

constructor

***SimpleSigner* constructor** (pks:`string` | *BufferType* [])

Parameters:

pks	`string` | *BufferType* []	pks

Returns:

SimpleSigner

prepareTransaction()

optiional method which allows to change the transaction-data before sending it. This can be used for redirecting it through a multisig.

***Promise<Transaction>* prepareTransaction** (client:*IN3Generic<BigIntType,BufferType>* , tx:*Transaction*)

Parameters:

client	*IN3Generic<BigIntType,BufferType>*	client
tx	*Transaction*	tx

Returns:

Promise<Transaction>

sign()

signing of any data. if hashFirst is true the data should be hashed first, otherwise the data is the hash.

***Promise<BufferType>* sign** (data:*Hex* , account:*Address* , hashFirst:`boolean`, ethV:`boolean`)

Parameters:

data	*Hex*	a Hexcoded String (starting with 0x)
account	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
hashFirst	`boolean`	hash first
ethV	`boolean`	eth v

Returns:

Promise<BufferType>

addAccount()

add account

string addAccount (pk:*Hash*)

Parameters:

pk	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

`string`

canSign()

returns true if the account is supported (or unlocked)

Promise<boolean> canSign (address:*Address*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)

Returns:

`Promise<boolean>`

11.5.4 Type AccountAPI

Source: index.d.ts

The Account API

add()

adds a private key to sign with. This method returns address of the pk

`Promise<string>` **add** (pk:`string` | *BufferType*)

Parameters:

pk	`string` | *BufferType*	

Returns:

`Promise<string>`

11.5.5 Type BTCBlock

Source: index.d.ts

a full Block including the transactions

bits	`string`	bits (target) for the block as hex
chainwork	`string`	total amount of work since genesis
confirmations	`number`	number of confirmations or blocks mined on top of the containing block
difficulty	`number`	difficulty of the block
hash	`string`	the hash of the blockheader
height	`number`	block number
mediantime	`string`	unix timestamp in seconds since 1970
merkleroot	`string`	merkle root of the trie of all transactions in the block
nTx	`number`	number of transactions in the block
nextblockhash	`string`	hash of the next blockheader
nonce	`number`	nonce-field of the block
previousblockhash	`string`	hash of the parent blockheader
time	`string`	unix timestamp in seconds since 1970
tx	*T* []	the transactions
version	`number`	used version

11.5.6 Type BTCBlockHeader

Source: index.d.ts

a Block header

bits	`string`	bits (target) for the block as hex
chainwork	`string`	total amount of work since genesis
confirmations	`number`	number of confirmations or blocks mined on top of the containing block
difficulty	`number`	difficulty of the block
hash	`string`	the hash of the blockheader
height	`number`	block number
mediantime	`string`	unix timestamp in seconds since 1970
merkleroot	`string`	merkle root of the trie of all transactions in the block
nTx	`number`	number of transactions in the block
nextblockhash	`string`	hash of the next blockheader
nonce	`number`	nonce-field of the block
previousblockhash	`string`	hash of the parent blockheader
time	`string`	unix timestamp in seconds since 1970
version	`number`	used version
versionHex	`string`	version as hex

11.5.7 Type BlockInfo

Source: index.d.ts

blockNumber	`number`	the blockNumber
committed	`boolean`	the committed
verified	`boolean`	the verified

11.5.8 Type BtcAPI

Source: index.d.ts

API for handling BitCoin data

getBlockBytes()

retrieves the serialized block (bytes) including all transactions

***Promise<BufferType>* getBlockBytes** (blockHash:*Hash*)

Parameters:

blockHash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BufferType>

getBlockHeader()

retrieves the blockheader and returns the data as json.

***Promise<BTCBlockHeader>* getBlockHeader** (blockHash:*Hash*)

Parameters:

blockHash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BTCBlockHeader>

getBlockHeaderBytes()

retrieves the serialized blockheader (bytes)

Promise<BufferType> **getBlockHeaderBytes** (blockHash:*Hash*)

Parameters:

blockHash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BufferType>

getBlockWithTxData()

retrieves the block including all tx data as json.

Promise<BTCBlock> **getBlockWithTxData** (blockHash:*Hash*)

Parameters:

blockHash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BTCBlock>

getBlockWithTxIds()

retrieves the block including all tx ids as json.

Promise<BTCBlock> **getBlockWithTxIds** (blockHash:*Hash*)

Parameters:

blockHash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BTCBlock>

getTransaction()

retrieves the transaction and returns the data as json.

***Promise<BtcTransaction>* getTransaction** (txid:*Hash*)

Parameters:

txid	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BtcTransaction>

getTransactionBytes()

retrieves the serialized transaction (bytes)

***Promise<BufferType>* getTransactionBytes** (txid:*Hash*)

Parameters:

txid	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<BufferType>

11.5.9 Type BtcTransaction

Source: index.d.ts

a BitCoin Transaction.

blockhash	*Hash*	the block hash of the block containing this transaction.
blocktime	number	The block time in seconds since epoch (Jan 1 1970 GMT)
confirmations	number	The confirmations.
hash	*Hash*	The transaction hash (differs from txid for witness transactions)
hex	*Data*	the hex representation of raw data
in_active_chain	boolean	true if this transaction is part of the longest chain
locktime	number	The locktime
size	number	The serialized transaction size
time	number	The transaction time in seconds since epoch (Jan 1 1970 GMT)
txid	*Hash*	The requested transaction id.
version	number	The version
vin	*BtcTransactionInput* []	the transaction inputs
vout	*BtcTransactionOutput* []	the transaction outputs
vsize	number	The virtual transaction size (differs from size for witness transactions)

11.5.10 Type BtcTransactionInput

Source: index.d.ts

a Input of a Bitcoin Transaction

scriptSig		the script
sequence	`number`	The script sequence number
txid	*Hash*	the transaction id
txinwitness	*Data* []	hex-encoded witness data (if any)
vout	`number`	the index of the transactionoutput

11.5.11 Type BtcTransactionOutput

Source: index.d.ts

a Input of a Bitcoin Transaction

n	`number`	the index
scriptPubKey		the script
value	`number`	the value in BTC
vout	`number`	the index of the transactionoutput

11.5.12 Type DepositResponse

Source: index.d.ts

receipt	*TransactionReceipt*	the receipt

11.5.13 Type ETHOpInfoResp

Source: index.d.ts

block	*BlockInfo*	the block
executed	`boolean`	the executed

11.5.14 Type EthAPI

Source: index.d.ts

The API for ethereum operations.

accounts	*AccountAPI<BufferType>*	accounts-API
client	*IN3Generic<BigIntType,BufferType>*	the client used.
signer	*Signer<BigIntType,BufferType>*	a custom signer *(optional)*

blockNumber()

Returns the number of most recent block. (as number)

`Promise<number>` blockNumber ()

Returns:

`Promise<number>`

call()

Executes a new message call immediately without creating a transaction on the block chain.

`Promise<string>` **call** (tx:*Transaction* , block:*BlockType*)

Parameters:

tx	*Transaction*	tx
block	*BlockType*	BlockNumber or predefined Block

Returns:

`Promise<string>`

callFn()

Executes a function of a contract, by passing a method-signature and the arguments, which will then be ABI-encoded and send as eth_call.

`Promise<any>` **callFn** (to:*Address* , method:`string`, args:`any` [])

Parameters:

to	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
method	`string`	method
args	`any` []	args

Returns:

`Promise<any>`

chainId()

Returns the EIP155 chain ID used for transaction signing at the current best block. Null is returned if not available.

`Promise<string>` chainId ()

Returns:

`Promise<string>`

clientVersion()

Returns the clientVersion. This may differ in case of an network, depending on the node it communicates with.

`Promise<string>` clientVersion ()

Returns:

`Promise<string>`

constructor()

constructor

any constructor (client:*IN3Generic<BigIntType,BufferType>*)

Parameters:

client	*IN3Generic<BigIntType,BufferType>*	client

Returns:

`any`

contractAt()

contract at

contractAt (abi:*ABI* [], address:*Address*)

Parameters:

abi	*ABI* []	abi
address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)

decodeEventData()

decode event data

any decodeEventData (log:*Log* , d:*ABI*)

Parameters:

log	*Log*	log
d	*ABI*	d

Returns:

`any`

estimateGas()

Makes a call or transaction, which won't be added to the blockchain and returns the used gas, which can be used for estimating the used gas.

Promise<number> **estimateGas** (tx:*Transaction*)

Parameters:

tx	*Transaction*	tx

Returns:

`Promise<number>`

gasPrice()

Returns the current price per gas in wei. (as number)

`Promise<number>` gasPrice ()

Returns:

`Promise<number>`

getBalance()

Returns the balance of the account of given address in wei (as hex).

Promise<BigIntType> **getBalance** (address:*Address* , block:*BlockType*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
block	*BlockType*	BlockNumber or predefined Block

Returns:

Promise<BigIntType>

getBlockByHash()

Returns information about a block by hash.

***Promise<Block>* getBlockByHash** (hash:*Hash* , includeTransactions:`boolean`)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)
includeTransactions	`boolean`	include transactions

Returns:

Promise<Block>

getBlockByNumber()

Returns information about a block by block number.

***Promise<Block>* getBlockByNumber** (block:*BlockType* , includeTransactions:`boolean`)

Parameters:

block	*BlockType*	BlockNumber or predefined Block
includeTransactions	`boolean`	include transactions

Returns:

Promise<Block>

getBlockTransactionCountByHash()

Returns the number of transactions in a block from a block matching the given block hash.

`Promise<number>` **getBlockTransactionCountByHash** (block:*Hash*)

Parameters:

block	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

`Promise<number>`

getBlockTransactionCountByNumber()

Returns the number of transactions in a block from a block matching the given block number.

Promise<number> getBlockTransactionCountByNumber (block:*Hash*)

Parameters:

block	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

`Promise<number>`

getCode()

Returns code at a given address.

Promise<string> getCode (address:*Address* , block:*BlockType*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
block	*BlockType*	BlockNumber or predefined Block

Returns:

`Promise<string>`

getFilterChanges()

Polling method for a filter, which returns an array of logs which occurred since last poll.

Promise<> getFilterChanges (id:*Quantity*)

Parameters:

id	*Quantity*	a BigInteger encoded as hex.

Returns:

`Promise<>`

getFilterLogs()

Returns an array of all logs matching filter with given id.

`Promise<>` getFilterLogs (id:*Quantity*)

Parameters:

id	*Quantity*	a BigInteger encoded as hex.

Returns:

`Promise<>`

getLogs()

Returns an array of all logs matching a given filter object.

`Promise<>` getLogs (filter:*LogFilter*)

Parameters:

filter	*LogFilter*	filter

Returns:

`Promise<>`

getStorageAt()

Returns the value from a storage position at a given address.

`Promise<string>` getStorageAt (address:*Address* , pos:*Quantity* , block:*BlockType*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
pos	*Quantity*	a BigInteger encoded as hex.
block	*BlockType*	BlockNumber or predefined Block

Returns:

`Promise<string>`

getTransactionByBlockHashAndIndex()

Returns information about a transaction by block hash and transaction index position.

Promise<TransactionDetail> **getTransactionByBlockHashAndIndex** (hash:*Hash* , pos:*Quantity*)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)
pos	*Quantity*	a BigInteger encoded as hex.

Returns:

Promise<TransactionDetail>

getTransactionByBlockNumberAndIndex()

Returns information about a transaction by block number and transaction index position.

Promise<TransactionDetail> **getTransactionByBlockNumberAndIndex** (block:*BlockType* , pos:*Quantity*)

Parameters:

block	*BlockType*	BlockNumber or predefined Block
pos	*Quantity*	a BigInteger encoded as hex.

Returns:

Promise<TransactionDetail>

getTransactionByHash()

Returns the information about a transaction requested by transaction hash.

Promise<TransactionDetail> **getTransactionByHash** (hash:*Hash*)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<TransactionDetail>

getTransactionCount()

Returns the number of transactions sent from an address. (as number)

`Promise<number>` **getTransactionCount** (address:*Address* , block:*BlockType*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
block	*BlockType*	BlockNumber or predefined Block

Returns:

`Promise<number>`

getTransactionReceipt()

Returns the receipt of a transaction by transaction hash. Note That the receipt is available even for pending transactions.

***Promise<TransactionReceipt>* getTransactionReceipt** (hash:*Hash*)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

Promise<TransactionReceipt>

getUncleByBlockHashAndIndex()

Returns information about a uncle of a block by hash and uncle index position. Note: An uncle doesn't contain individual transactions.

***Promise<Block>* getUncleByBlockHashAndIndex** (hash:*Hash* , pos:*Quantity*)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)
pos	*Quantity*	a BigInteger encoded as hex.

Returns:

Promise<Block>

getUncleByBlockNumberAndIndex()

Returns information about a uncle of a block number and uncle index position. Note: An uncle doesn't contain individual transactions.

***Promise<Block>* getUncleByBlockNumberAndIndex** (block:*BlockType* , pos:*Quantity*)

Parameters:

block	*BlockType*	BlockNumber or predefined Block
pos	*Quantity*	a BigInteger encoded as hex.

Returns:

Promise<Block>

getUncleCountByBlockHash()

Returns the number of uncles in a block from a block matching the given block hash.

Promise<number> getUncleCountByBlockHash (hash:*Hash*)

Parameters:

hash	*Hash*	a 32 byte Hash encoded as Hex (starting with 0x)

Returns:

`Promise<number>`

getUncleCountByBlockNumber()

Returns the number of uncles in a block from a block matching the given block hash.

Promise<number> getUncleCountByBlockNumber (block:*BlockType*)

Parameters:

block	*BlockType*	BlockNumber or predefined Block

Returns:

`Promise<number>`

hashMessage()

a Hexcoded String (starting with 0x)

***Hex* hashMessage** (data:*Data*)

Parameters:

data	*Data*	data encoded as Hex (starting with 0x)

Returns:

Hex

newBlockFilter()

Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.

`Promise<string>` newBlockFilter ()

Returns:

`Promise<string>`

newFilter()

Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call eth_getFilterChanges.

A note on specifying topic filters: Topics are order-dependent. A transaction with a log with topics [A, B] will be matched by the following topic filters:

[] "anything" [A] "A in first position (and anything after)" [null, B] "anything in first position AND B in second position (and anything after)" [A, B] "A in first position AND B in second position (and anything after)" [[A, B], [A, B]] "(A OR B) in first position AND (A OR B) in second position (and anything after)"

`Promise<string>` newFilter (filter:*LogFilter*)

Parameters:

filter	*LogFilter*	filter

Returns:

`Promise<string>`

newPendingTransactionFilter()

Creates a filter in the node, to notify when new pending transactions arrive.

To check if the state has changed, call eth_getFilterChanges.

`Promise<string>` newPendingTransactionFilter ()

Returns:

`Promise<string>`

protocolVersion()

Returns the current ethereum protocol version.

`Promise<string>` protocolVersion ()

Returns:

`Promise<string>`

resolveENS()

resolves a name as an ENS-Domain.

***Promise<Address>* resolveENS** (name:`string`, type:*Address* , registry:`string`)

Parameters:

name	`string`	the domain name
type	*Address*	the type (currently only addr is supported)
registry	`string`	optionally the address of the registry (default is the mainnet ens registry)

Returns:

Promise<Address>

sendRawTransaction()

Creates new message call transaction or a contract creation for signed transactions.

`Promise<string>` sendRawTransaction (data:*Data*)

Parameters:

data	*Data*	data encoded as Hex (starting with 0x)

Returns:

`Promise<string>`

sendTransaction()

sends a Transaction

Promise<> **sendTransaction** (args:*TxRequest*)

Parameters:

args	*TxRequest*	args

Returns:

`Promise<>`

sign()

signs any kind of message using the `\x19Ethereum Signed Message:\n`-prefix

Promise<BufferType> **sign** (account:*Address* , data:*Data*)

Parameters:

account	*Address*	the address to sign the message with (if this is a 32-bytes hex-string it will be used as private key)
data	*Data*	the data to sign (Buffer, hexstring or utf8-string)

Returns:

Promise<BufferType>

syncing()

Returns the state of the underlying node.

`Promise<>` syncing ()

Returns:

`Promise<>`

toWei()

Returns the value in wei as hexstring.

`string` **toWei** (value:`string`, unit:`string`)

Parameters:

value	`string`	value
unit	`string`	unit

Returns:

`string`

uninstallFilter()

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with eth_getFilterChanges for a period of time.

Promise<Quantity> **uninstallFilter** (id:*Quantity*)

Parameters:

id	*Quantity*	a BigInteger encoded as hex.

Returns:

Promise<Quantity>

web3ContractAt()

web3 contract at

Web3Contract **web3ContractAt** (abi:*ABI* [], address:*Address* , options:)

Parameters:

abi	*ABI* []	abi
address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
options		options

Returns:

Web3Contract

11.5.15 Type Fee

Source: index.d.ts

feeType	*TxType*	the feeType
gasFee	`number`	the gasFee
gasPrice	`number`	the gasPrice
totalFee	`number`	the totalFee
totalGas	`number`	the totalGas
zkpFee	`number`	the zkpFee

11.5.16 Type IN3Config

Source: index.d.ts

the configuration of the IN3-Client. This can be changed at any time. All properties are optional and will be verified when sending the next request.

autoUpdateList	`boolean`	if true the nodelist will be automaticly updated if the lastBlock is newer. default: true *(optional)*
bootWeights	`boolean`	if true, the first request (updating the nodelist) will also fetch the current health status and use it for blacklisting unhealthy nodes. This is used only if no nodelist is availabkle from cache. default: false *(optional)*
btc	*btc_config*	config for btc *(optional)*
chainId	`string`	The chain-id based on EIP-155. or the name of the supported chain. Currently we support 'mainnet', 'goerli', 'kovan', 'ipfs' and 'local' While most of the chains use preconfigured chain settings, 'local' actually uses the local running client turning of proof. example: '0x1' or 'mainnet' or 'goerli' default: 'mainnet'
chainRegistry	`string`	main chain-registry contract example: 0xe36179e2286ef405e929C90ad3E70E649B22a945 *(optional)*
finality	`number`	the number in percent needed in order reach finality if you run on a POA-Chain.

transport()

sets the transport-function.

Promise<string> **transport** (url:`string`, payload:`string`, timeout:`number`)

Parameters:

url	`string`	url
payload	`string`	payload
timeout	`number`	timeout

Returns:

`Promise<string>`

11.5.17 Type IN3NodeConfig

Source: index.d.ts

a configuration of a in3-server.

address	`string`	the address of the node, which is the public address it iis signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
capacity	`number`	the capacity of the node. example: 100 *(optional)*
chainIds	`string` []	the list of supported chains example: 0x1
deposit	`number`	the deposit of the node in wei example: 12350000
index	`number`	the index within the contract example: 13 *(optional)*
props	`number`	the properties of the node. example: 3 *(optional)*
registerTime	`number`	the UNIX-timestamp when the node was registered example: 1563279168 *(optional)*
timeout	`number`	the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600 *(optional)*
unregisterTime	`number`	the UNIX-timestamp when the node is allowed to be deregister example: 1563279168 *(optional)*
url	`string`	the endpoint to post to example: https://in3.slock.it

11.5.18 Type IN3NodeWeight

Source: index.d.ts

a local weight of a n3-node. (This is used internally to weight the requests)

avgResponseTime	`number`	average time of a response in ms example: 240 *(optional)*
blacklistedUntil	`number`	blacklisted because of failed requests until the timestamp example: 1529074639623 *(optional)*
lastRequest	`number`	timestamp of the last request in ms example: 1529074632623 *(optional)*
pricePerRequest	`number`	last price *(optional)*
responseCount	`number`	number of uses. example: 147 *(optional)*
weight	`number`	factor the weight this noe (default 1.0) example: 0.5 *(optional)*

11.5.19 Type IN3Plugin

Source: index.d.ts

11.5.20 Type IpfsAPI

Source: index.d.ts

API for storing and retrieving IPFS-data.

get()

retrieves the content for a hash from IPFS.

***Promise<BufferType>* get** (multihash:`string`)

Parameters:

multihash	`string`	the IPFS-hash to fetch

Returns:

Promise<BufferType>

put()

stores the data on ipfs and returns the IPFS-Hash.

`Promise<string>` put (content:*BufferType*)

Parameters:

content	*BufferType*	puts a IPFS content

Returns:

`Promise<string>`

11.5.21 Type RPCRequest

Source: index.d.ts

a JSONRPC-Request with N3-Extension

id	`number | string`	the identifier of the request example: 2 *(optional)*
jsonrpc	`'2.0'`	the version
method	`string`	the method to call example: eth_getBalance
params	`any []`	the params example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,lates *(optional)*

11.5.22 Type RPCResponse

Source: index.d.ts

a JSONRPC-Responset with N3-Extension

error	`string`	in case of an error this needs to be set *(optional)*
id	`string | number`	the id matching the request example: 2
jsonrpc	`'2.0'`	the version
result	`any`	the params example: 0xa35bc *(optional)*

11.5.23 Type Signer

Source: index.d.ts

prepareTransaction()

optiional method which allows to change the transaction-data before sending it. This can be used for redirecting it through a multisig.

***Promise<Transaction>* prepareTransaction** (client:*IN3Generic<BigIntType,BufferType>* , tx:*Transaction*)

Parameters:

client	*IN3Generic<BigIntType,BufferType>*	client
tx	*Transaction*	tx

Returns:

Promise<Transaction>

sign()

signing of any data. if hashFirst is true the data should be hashed first, otherwise the data is the hash.

***Promise<BufferType>* sign** (data:*Hex* , account:*Address* , hashFirst:`boolean`, ethV:`boolean`)

Parameters:

data	*Hex*	a Hexcoded String (starting with 0x)
account	*Address*	a 20 byte Address encoded as Hex (starting with 0x)
hashFirst	`boolean`	hash first
ethV	`boolean`	eth v

Returns:

Promise<BufferType>

canSign()

returns true if the account is supported (or unlocked)

Promise<boolean> canSign (address:*Address*)

Parameters:

address	*Address*	a 20 byte Address encoded as Hex (starting with 0x)

Returns:

`Promise<boolean>`

11.5.24 Type Token

Source: index.d.ts

address	*String*	the address
decimals	`number`	the decimals
id	`number`	the id
symbol	*String*	the symbol

11.5.25 Type Tokens

Source: index.d.ts

11.5.26 Type TxInfo

Source: index.d.ts

block	*BlockInfo*	the block
executed	`boolean`	the executed
failReason	`string`	the failReason
success	`boolean`	the success

11.5.27 Type TxType

Source: index.d.ts

type	`'Withdraw'`
	 `|'Transfer'`
	 `|'TransferToNew'`
		the type

11.5.28 Type Utils

Source: index.d.ts

Collection of different util-functions.

abiDecode()

decodes the given data as ABI-encoded (without the methodHash)

`any` **[] abiDecode (** signature:`string`, data:*Data* **)**

Parameters:

signature	`string`	the method signature, which must contain a return description
data	*Data*	the data to decode

Returns:

`any` []

abiEncode()

encodes the given arguments as ABI-encoded (including the methodHash)

Hex **abiEncode** (signature:`string`, args:`any` [])

Parameters:

signature	`string`	the method signature
args	`any` []	the arguments

Returns:

Hex

checkAddressChecksum()

checks whether the given address is a correct checksumAddress If the chainId is passed, it will be included accord to EIP 1191

`boolean` **checkAddressChecksum** (address:*Address* , chainId:`number`)

Parameters:

address	*Address*	the address (as hex)
chainId	`number`	the chainId (if supported)

Returns:

`boolean`

createSignatureHash()

a Hexcoded String (starting with 0x)

Hex **createSignatureHash** (def:*ABI*)

Parameters:

def	*ABI*	def

Returns:

Hex

decodeEvent()

decode event

any **decodeEvent** (log:*Log* , d:*ABI*)

Parameters:

log	*Log*	log
d	*ABI*	d

Returns:

`any`

ecSign()

create a signature (65 bytes) for the given message and kexy

BufferType **ecSign** (pk:*Hex* | *BufferType* , msg:*Hex* | *BufferType* , hashFirst:`boolean`, adjustV:`boolean`)

Parameters:

pk	*Hex* | *BufferType*	the private key
msg	*Hex* | *BufferType*	the message
hashFirst	`boolean`	if true the message will be hashed first (default:true), if not the message is the hash.
adjustV	`boolean`	if true (default) the v value will be adjusted by adding 27

Returns:

BufferType

getVersion()

returns the incubed version.

`string` getVersion ()

Returns:

`string`

isAddress()

checks whether the given address is a valid hex string with 0x-prefix and 20 bytes

`boolean` **isAddress** (address:*Address*)

Parameters:

address	*Address*	the address (as hex)

Returns:

`boolean`

keccak()

calculates the keccack hash for the given data.

BufferType **keccak** (data:*BufferType* | *Data*)

Parameters:

data	*BufferType* | *Data*	the data as Uint8Array or hex data.

Returns:

BufferType

private2address()

generates the public address from the private key.

Address **private2address** (pk:*Hex* | *BufferType*)

Parameters:

pk	*Hex* | *BufferType*	the private key.

Returns:

Address

randomBytes()

returns a Buffer with strong random bytes. Thsi will use the browsers crypto-module or in case of nodejs use the crypto-module there.

BufferType **randomBytes** (len:`number`)

Parameters:

len	`number`	the number of bytes to generate.

Returns:

BufferType

soliditySha3()

solidity sha3

`string` **soliditySha3** (args:`any` [])

Parameters:

args	`any` []	args

Returns:

`string`

splitSignature()

takes raw signature (65 bytes) and splits it into a signature object.

Signature **splitSignature** (signature:*Hex* | *BufferType* , message:*BufferType* | *Hex* , hashFirst:`boolean`)

Parameters:

signature	*Hex* | *BufferType*	the 65 byte-signature
message	*BufferType* | *Hex*	the message
hashFirst	`boolean`	if true (default) this will be taken as raw-data and will be hashed first.

Returns:

Signature

toBuffer()

converts any value to a Buffer. optionally the target length can be specified (in bytes)

BufferType **toBuffer** (data:*Hex* | *BufferType* | `number` | `bigint`, len:`number`)

Parameters:

data	*Hex* | *BufferType* | `number` | `bigint`	data
len	`number`	len

Returns:

BufferType

toChecksumAddress()

generates a checksum Address for the given address. If the chainId is passed, it will be included accord to EIP 1191

***Address* toChecksumAddress** (address:*Address* , chainId:`number`)

Parameters:

address	*Address*	the address (as hex)
chainId	`number`	the chainId (if supported)

Returns:

Address

toHex()

converts any value to a hex string (with prefix 0x). optionally the target length can be specified (in bytes)

***Hex* toHex** (data:*Hex* | *BufferType* | `number` | `bigint`, len:`number`)

Parameters:

data	*Hex* | *BufferType* | `number` | `bigint`	data
len	`number`	len

Returns:

Hex

toMinHex()

removes all leading 0 in the hexstring

`string` **toMinHex** (key:`string` | *BufferType* | `number`)

Parameters:

key	`string` | *BufferType* | `number`	key

Returns:

`string`

toNumber()

converts any value to a hex string (with prefix 0x). optionally the target length can be specified (in bytes)

`number` **toNumber** (data:`string` | *BufferType* | `number` | `bigint`)

Parameters:

data	`string` | *BufferType* | `number` | `bigint`	data

Returns:

`number`

toUint8Array()

converts any value to a Uint8Array. optionally the target length can be specified (in bytes)

BufferType **toUint8Array** (data:*Hex* | *BufferType* | `number` | `bigint`, len:`number`)

Parameters:

data	*Hex* | *BufferType* | `number` | `bigint`	data
len	`number`	len

Returns:

BufferType

toUtf8()

convert to String

`string` **toUtf8** (val:`any`)

Parameters:

val	`any`	val

Returns:

`string`

11.5.29 Type Web3Contract

Source: index.d.ts

events		the events
methods		the methods
options		the options

deploy()

deploy

Web3TransactionObject **deploy** (args:)

Parameters:

args		args

Returns:

Web3TransactionObject

once()

once

void **once** (eventName:`string`, options:, handler:(*Error* , *Web3Event*) => `void`)

Parameters:

eventName	`string`	event name
options		options
handler	(*Error* , *Web3Event*) => `void`	handler

getPastEvents()

get past events

Promise<> **getPastEvents** (evName:`string`, options:)

Parameters:

evName	`string`	ev name
options		options

Returns:

`Promise<>`

11.5.30 Type Web3Event

Source: index.d.ts

address	*Address*	the address
blockHash	*Hash*	the blockHash
blockNumber	`number`	the blockNumber
event	`string`	the event
logIndex	`number`	the logIndex
raw		the raw
returnValues		the returnValues
signature	`string`	the signature
transactionHash	*Hash*	the transactionHash
transactionIndex	`number`	the transactionIndex

11.5.31 Type Web3TransactionObject

Source: index.d.ts

call()

call

`Promise<any>` call (options:)

Parameters:

options		options

Returns:

`Promise<any>`

encodeABI()

a Hexcoded String (starting with 0x)

Hex encodeABI ()

Returns:

Hex

estimateGas()

estimate gas

`Promise<number>` estimateGas (options:)

Parameters:

options		options

Returns:

`Promise<number>`

send()

send

`Promise<any>` send (options:)

Parameters:

options		options

Returns:

`Promise<any>`

11.5.32 Type ZKAccountInfo

Source: index.d.ts

address	`string`	the address
committed		the committed
depositing		the depositing
id	`number`	the id
verified		the verified

11.5.33 Type ZksyncAPI

Source: index.d.ts

API for zksync.

deposit()

deposits the declared amount into the rollup

Promise<DepositResponse> **deposit** (amount:`number`, token:`string`, approveDepositAmountForERC20:`boolean`, account:`string`)

Parameters:

amount	`number`	amount in wei to deposit
token	`string`	the token identifier e.g. ETH
approveDepositAmountForERC20	`boolean`	bool that is set to true if it is a erc20 token that needs approval
account	`string`	address of the account that wants to deposit (if left empty it will be taken from current signer)

Returns:

Promise<DepositResponse>

emergencyWithdraw()

executes an emergency withdrawel onchain

Promise<String> **emergencyWithdraw** (token:`string`)

Parameters:

token	`string`	the token identifier e.g. ETH

Returns:

Promise<String>

getAccountInfo()

gets current account Infoa and balances.

Promise<ZKAccountInfo> **getAccountInfo** (account:`string`)

Parameters:

account	`string`	the address of the account . if not specified, the first signer is used.

Returns:

Promise<ZKAccountInfo>

getContractAddress()

gets the contract address of the zksync contract

Promise<String> getContractAddress ()

Returns:

Promise<String>

getEthopInfo()

returns the state of receipt of the PriorityOperation

Promise<ETHOpInfoResp> **getEthopInfo** (opId:`number`)

Parameters:

opId	`number`	the id of the PriorityOperation

Returns:

Promise<ETHOpInfoResp>

getSyncKey()

returns private key used for signing zksync transactions

String getSyncKey ()

Returns:

String

getTokenPrice()

returns the current token price

Promise<Number> **getTokenPrice** (tokenSymbol:`string`)

Parameters:

tokenSymbol	`string`	the address of the token

Returns:

Promise<Number>

getTokens()

returns an object containing Token objects with its short name as key

Promise<Tokens> getTokens ()

Returns:

Promise<Tokens>

getTxFee()

returns the transaction fee

Promise<Fee> **getTxFee** (txType:*TxType* , receipient:`string`, token:`string`)

Parameters:

txType	*TxType*	either Withdraw or Transfer
receipient	`string`	the address the transaction is send to
token	`string`	the token identifier e.g. ETH

Returns:

Promise<Fee>

getTxInfo()

get transaction info

Promise<TxInfo> **getTxInfo** (txHash:`string`)

Parameters:

txHash	`string`	the has of the tx you want the info about

Returns:

Promise<TxInfo>

setKey()

set the signer key based on the current pk

Promise<String> setKey ()

Returns:

Promise<String>

transfer()

transfers the specified amount to another address within the zksync rollup

***Promise<String>* transfer** (to:`string`, amount:`number`, token:`string`, account:`string`)

Parameters:

to	`string`	address of the receipient
amount	`number`	amount to send in wei
token	`string`	the token indentifier e.g. ETH
account	`string`	address of the account that wants to transfer (if left empty it will be taken from current signer)

Returns:

Promise<String>

withdraw()

withdraws the specified amount from the rollup to a specific address

***Promise<String>* withdraw** (ethAddress:`string`, amount:`number`, token:`string`, account:`string`)

Parameters:

ethAddress	`string`	the receipient address
amount	`number`	amount to withdraw in wei
token	`string`	the token identifier e.g. ETH
account	`string`	address of the account that wants to withdraw (if left empty it will be taken from current signer)

Returns:

Promise<String>

11.5.34 Type ABI

Source: index.d.ts

anonymous	`boolean`	the anonymous *(optional)*
components	*ABIField* []	the components *(optional)*
constant	`boolean`	the constant *(optional)*
inputs	*ABIField* []	the inputs *(optional)*
internalType	`string`	the internalType *(optional)*
name	`string`	the name *(optional)*
outputs	*ABIField* [] | `any` []	the outputs *(optional)*
payable	`boolean`	the payable *(optional)*
stateMutability	`'pure'` | `'view'` | `'nonpayable'` | `'payable'` | `string`	the stateMutability *(optional)*
type	`'function'` | `'constructor'` | `'event'` | `'fallback'` | `string`	the type

11.5.35 Type ABIField

Source: index.d.ts

indexed	`boolean`	the indexed *(optional)*
internalType	`string`	the internalType *(optional)*
name	`string`	the name
type	`string`	the type

11.5.36 Type Address

Source: index.d.ts

a 20 byte Address encoded as Hex (starting with 0x) a Hexcoded String (starting with 0x) = `string`

11.5.37 Type Block

Source: index.d.ts

author	*Address*	20 Bytes - the address of the author of the block (the beneficiary to whom the mining rewards were given)
difficulty	*Quantity*	integer of the difficulty for this block
extraData	*Data*	the 'extra data' field of this block
gasLimit	*Quantity*	the maximum gas allowed in this block
gasUsed	*Quantity*	the total used gas by all transactions in this block
hash	*Hash*	hash of the block. null when its pending block
logsBloom	*Data*	256 Bytes - the bloom filter for the logs of the block. null when its pending block
miner	*Address*	20 Bytes - alias of 'author'
nonce	*Data*	8 bytes hash of the generated proof-of-work. null when its pending block. Missing in case of PoA.
number	*Quantity*	The block number. null when its pending block
parentHash	*Hash*	hash of the parent block
receiptsRoot	*Data*	32 Bytes - the root of the receipts trie of the block

11.5.38 Type Data

Source: index.d.ts

data encoded as Hex (starting with 0x) a Hexcoded String (starting with 0x) = `string`

11.5.39 Type Hash

Source: index.d.ts

a 32 byte Hash encoded as Hex (starting with 0x) a Hexcoded String (starting with 0x) = `string`

11.5.40 Type Log

Source: index.d.ts

address	*Address*	20 Bytes - address from which this log originated.
blockHash	*Hash*	Hash, 32 Bytes - hash of the block where this log was in. null when its pending. null when its pending log.
blockNumber	*Quantity*	the block number where this log was in. null when its pending. null when its pending log.
data	*Data*	contains the non-indexed arguments of the log.
logIndex	*Quantity*	integer of the log index position in the block. null when its pending log.
removed	`boolean`	true when the log was removed, due to a chain reorganization. false if its a valid log.
topics	*Data* []	- Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier.)
transactionHash	*Hash*	Hash, 32 Bytes - hash of the transactions this log was created from. null when its pending log.
transactionIndex	*Quantity*	integer of the transactions index position log was created from. null when its pending log.

11.5.41 Type LogFilter

Source: index.d.ts

address	*Address*	(optional) 20 Bytes - Contract address or a list of addresses from which logs should originate.
fromBlock	*BlockType*	Quantity or Tag - (optional) (default: latest) Integer block number, or 'latest' for the last mined block or 'pending', 'earliest' for not yet mined transactions.
limit	*Quantity*	å(optional) The maximum number of entries to retrieve (latest first).
toBlock	*BlockType*	Quantity or Tag - (optional) (default: latest) Integer block number, or 'latest' for the last mined block or 'pending', 'earliest' for not yet mined transactions.
topics	`string` | `string` [] []	(optional) Array of 32 Bytes Data topics. Topics are order-dependent. It's possible to pass in null to match any topic, or a subarray of multiple topics of which one should be matching.

11.5.42 Type Signature

Source: index.d.ts

Signature

message	*Data*	the message
messageHash	*Hash*	the messageHash
r	*Hash*	the r
s	*Hash*	the s
signature	*Data*	the signature *(optional)*
v	*Hex*	the v

11.5.43 Type Transaction

Source: index.d.ts

chainId	`any`	optional chain id *(optional)*
data	`string`	4 byte hash of the method signature followed by encoded parameters. For details see Ethereum Contract ABI.
from	*Address*	20 Bytes - The address the transaction is send from.
gas	*Quantity*	Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
gasPrice	*Quantity*	Integer of the gas price used for each paid gas.
nonce	*Quantity*	nonce
to	*Address*	(optional when creating new contract) 20 Bytes - The address the transaction is directed to.
value	*Quantity*	Integer of the value sent with this transaction.

11.5.44 Type TransactionDetail

Source: index.d.ts

blockHash	*Hash*	32 Bytes - hash of the block where this transaction was in. null when its pending.
blockNumber	*BlockType*	block number where this transaction was in. null when its pending.
chainId	*Quantity*	the chain id of the transaction, if any.
condition	`any`	(optional) conditional submission, Block number in block or timestamp in time or null. (parity-feature)
creates	*Address*	creates contract address
from	*Address*	20 Bytes - address of the sender.
gas	*Quantity*	gas provided by the sender.
gasPrice	*Quantity*	gas price provided by the sender in Wei.
hash	*Hash*	32 Bytes - hash of the transaction.
input	*Data*	the data send along with the transaction.
nonce	*Quantity*	the number of transactions made by the sender prior to this one.
pk	`any`	optional: the private key to use for signing *(optional)*

11.5.45 Type TransactionReceipt

Source: index.d.ts

blockHash	*Hash*	32 Bytes - hash of the block where this transaction was in.
blockNumber	*BlockType*	block number where this transaction was in.
contractAddress	*Address*	20 Bytes - The contract address created, if the transaction was a contract creation, otherwise null.
cumulativeGasUsed	*Quantity*	The total amount of gas used when this transaction was executed in the block.
events		event objects, which are only added in the web3Contract *(optional)*
from	*Address*	20 Bytes - The address of the sender.
gasUsed	*Quantity*	The amount of gas used by this specific transaction alone.
logs	*Log* []	Array of log objects, which this transaction generated.
logsBloom	*Data*	256 Bytes - A bloom filter of logs/events generated by contracts during transaction execution. Used to efficiently rule out transactions without expected logs.
root	*Hash*	32 Bytes - Merkle root of the state trie after the transaction has been executed (optional after Byzantium hard fork EIP609)
status	*Quantity*	0x0 indicates transaction

11.5.46 Type TxRequest

Source: index.d.ts

args	`any []`	the argument to pass to the method *(optional)*
confirmations	`number`	number of block to wait before confirming *(optional)*
data	*Data*	the data to send *(optional)*
from	*Address*	address of the account to use *(optional)*
gas	`number`	the gas needed *(optional)*
gasPrice	`number`	the gasPrice used *(optional)*
method	`string`	the ABI of the method to be used *(optional)*
nonce	`number`	the nonce *(optional)*
pk	*Hash*	raw private key in order to sign *(optional)*
timeout	`number`	number of seconds to wait for confirmations before giving up. Default: 10 *(optional)*
to	*Address*	contract *(optional)*
value	*Quantity*	the value in wei *(optional)*

11.5.47 Type btc_config

Source: index.d.ts

bitcoin configuration.

maxDAP	`number`	max number of DAPs (Difficulty Adjustment Periods) allowed when accepting new targets. *(optional)*
maxDiff	`number`	max increase (in percent) of the difference between targets when accepting new targets. *(optional)*

11.5.48 Type zksync_config

Source: index.d.ts

zksync configuration.

account	`string`	the account to be used. if not specified, the first signer will be used. *(optional)*
provider_url	`string`	url of the zksync-server *(optional)*

11.5.49 Type Hex

Source: index.d.ts

a Hexcoded String (starting with 0x) = `string`

11.5.50 Type BlockType

Source: index.d.ts

BlockNumber or predefined Block = `number | 'latest' | 'earliest' | 'pending'`

11.5.51 Type Quantity

Source: index.d.ts

a BigInteger encoded as hex. = `number` | *Hex*

CHAPTER 12

API Reference Python

12.1 Python Incubed client

This library is based on the C version of Incubed, which limits the compatibility for Cython, so please contribute by compiling it to your own platform and sending us a pull-request!

Go to our readthedocs page for more.

12.1.1 Quickstart

Install with pip

```
pip install in3
```

In3 Client API

```
import in3

in3_client = in3.Client()
# Sends a request to the Incubed Network, that in turn will collect proofs from the
↪Ethereum client,
# attest and sign the response, then send back to the client, that will verify
↪signatures and proofs.
block_number = in3_client.eth.block_number()
print(block_number) # Mainnet's block number

in3_client  # incubed network api
in3_client.eth  # ethereum api
in3_client.account  # ethereum account api
in3_client.contract  # ethereum smart-contract api
```

Developing & Tests

Install dev dependencies, IDEs should automatically recognize interpreter if done like this.

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Compile local libraries and run tests. Make sure you have cmake installed.

```
./buidl_libs.sh
```

Index

Explanation of this source code architecture and how it is organized. For more on design-patterns see here or on Martin Fowler's Catalog of Patterns of Enterprise Application Architecture.

- **in3.init.py**: Library entry point, imports organization. Standard for any pipy package.
- **in3.client**: Incubed Client and API.
- **in3.model**: MVC Model classes for the Incubed client module domain.
- **in3.transport**: HTTP Transport function and error handling.
- **in3.wallet**: WiP - Wallet API.
- **in3.exception**: Custom exceptions.
- **in3.eth**: Ethereum module.
- **in3.eth.api**: Ethereum API.
- **in3.eth.account**: Ethereum accounts.
- **in3.eth.contract**: Ethereum smart-contracts API.
- **in3.eth.model**: MVC Model classes for the Ethereum client module domain. Manages serialization.
- **in3.eth.factory**: Ethereum Object Factory. Manages deserialization.
- **in3.libin3**: Module for the libin3 runtime. Libin3 is written in C and can be found here.
- **in3.libin3.shared**: Native shared libraries for multiple operating systems and platforms.
- **in3.libin3.enum**: Enumerations mapping C definitions to python.
- **in3.libin3.lib_loader**: Bindings using Ctypes.
- **in3.libin3.runtime**: Runtime object, bridging the remote procedure calls to the libin3 instances.

12.2 Examples

12.2.1 connect_to_ethereum

source : in3-c/python/examples/connect_to_ethereum.py

```
"""
Connects to Ethereum and fetches attested information from each chain.
"""
import in3


print('\nEthereum Main Network')
client = in3.Client()
latest_block = client.eth.block_number()
gas_price = client.eth.gas_price()
print('Latest BN: {}\nGas Price: {} Wei'.format(latest_block, gas_price))

print('\nEthereum Kovan Test Network')
client = in3.Client('kovan')
latest_block = client.eth.block_number()
gas_price = client.eth.gas_price()
print('Latest BN: {}\nGas Price: {} Wei'.format(latest_block, gas_price))

print('\nEthereum Goerli Test Network')
client = in3.Client('goerli')
latest_block = client.eth.block_number()
gas_price = client.eth.gas_price()
print('Latest BN: {}\nGas Price: {} Wei'.format(latest_block, gas_price))

# Produces
"""
Ethereum Main Network
Latest BN: 9801135
Gas Price: 2000000000 Wei

Ethereum Kovan Test Network
Latest BN: 17713464
Gas Price: 6000000000 Wei

Ethereum Goerli Test Network
Latest BN: 2460853
Gas Price: 4610612736 Wei
"""
```

12.2.2 incubed_network

source : in3-c/python/examples/incubed_network.py

```
"""
Shows Incubed Network Nodes Stats
"""
import in3

print('\nEthereum Goerli Test Network')
client = in3.Client('goerli')
node_list = client.refresh_node_list()
print('\nIncubed Registry:')
print('\ttotal servers:', node_list.totalServers)
print('\tlast updated in block:', node_list.lastBlockNumber)
print('\tregistry ID:', node_list.registryId)
print('\tcontract address:', node_list.contract)
```

(continues on next page)

(continued from previous page)

```
print('\nNodes Registered:\n')
for node in node_list.nodes:
    print('\turl:', node.url)
    print('\tdeposit:', node.deposit)
    print('\tweight:', node.weight)
    print('\tregistered in block:', node.registerTime)
    print('\n')

# Produces
"""
Ethereum Goerli Test Network

Incubed Registry:
    total servers: 7
    last updated in block: 2320627
    registry ID: 0x67c02e5e272f9d6b4a33716614061dd298283f86351079ef903bf0d4410a44ea
    contract address: 0x5f51e413581dd76759e9eed51e63d14c8d1379c8

Nodes Registered:

    url: https://in3-v2.slock.it/goerli/nd-1
    deposit: 10000000000000000
    weight: 2000
    registered in block: 1576227711


    url: https://in3-v2.slock.it/goerli/nd-2
    deposit: 10000000000000000
    weight: 2000
    registered in block: 1576227741


    url: https://in3-v2.slock.it/goerli/nd-3
    deposit: 10000000000000000
    weight: 2000
    registered in block: 1576227801


    url: https://in3-v2.slock.it/goerli/nd-4
    deposit: 10000000000000000
    weight: 2000
    registered in block: 1576227831


    url: https://in3-v2.slock.it/goerli/nd-5
    deposit: 10000000000000000
    weight: 2000
    registered in block: 1576227876


    url: https://tincubeth.komputing.org/
    deposit: 10000000000000000
    weight: 1
    registered in block: 1578947320


    url: https://h5l45fkzz7oc3gmb.onion/
```

(continues on next page)

(continued from previous page)

```
    deposit: 10000000000000000
    weight: 1
    registered in block: 1578954071
"""
```

12.2.3 resolve_eth_names

source : in3-c/python/examples/resolve_eth_names.py

```
"""
Resolves ENS domains to Ethereum addresses
ENS is a smart-contract system that registers and resolves `.eth` domains.
"""
import in3


def _print():
    print('\nAddress for {} @ {}: {}'.format(domain, chain, address))
    print('Owner for {} @ {}: {}'.format(domain, chain, owner))


# Find ENS for the desired chain or the address of your own ENS resolver. https://
↪docs.ens.domains/ens-deployments
domain = 'depraz.eth'

print('\nEthereum Name Service')

# Instantiate In3 Client for Goerli
chain = 'goerli'
client = in3.Client(chain, cache_enabled=False)
address = client.ens_address(domain)
# owner = client.ens_owner(domain)
# _print()

# Instantiate In3 Client for Mainnet
chain = 'mainnet'
client = in3.Client(chain, cache_enabled=False)
address = client.ens_address(domain)
owner = client.ens_owner(domain)
_print()

# Instantiate In3 Client for Kovan
chain = 'kovan'
client = in3.Client(chain, cache_enabled=True)
try:
    address = client.ens_address(domain)
    owner = client.ens_owner(domain)
    _print()
except in3.ClientException:
    print('\nENS is not available on Kovan.')


# Produces
"""
Ethereum Name Service
```

(continues on next page)

(continued from previous page)

```
Address for depraz.eth @ mainnet: 0x0b56ae81586d2728ceaf7c00a6020c5d63f02308
Owner for depraz.eth @ mainnet: 0x6fa33809667a99a805b610c49ee2042863b1bb83

ENS is not available on Kovan.
"""
```

12.2.4 send_transaction

source : in3-c/python/examples/send_transaction.py

```
"""
Sends Ethereum transactions using Incubed.
Incubed send Transaction does all necessary automation to make sending a transaction␣
↪a breeze.
Works with included `data` field for smart-contract calls.
"""
import json
import in3
import time


# On Metamask, be sure to be connected to the correct chain, click on the `...` icon␣
↪on the right corner of
# your Account name, select `Account Details`. There, click `Export Private Key`,␣
↪copy the value to use as secret.
# By reading the terminal input, this value will stay in memory only. Don't forget to␣
↪cls or clear terminal after ;)
sender_secret = input("Sender secret: ")
receiver = input("Receiver address: ")
#     1000000000000000000 == 1 ETH
#              1000000000 == 1 Gwei Check https://etherscan.io/gasTracker.
value_in_wei = 1463926659
# None for Eth mainnet
chain = 'goerli'
client = in3.Client(chain if chain else 'mainnet')
# A transaction is only final if a certain number of blocks are mined on top of it.
# This number varies with the chain's consensus algorithm. Time can be calculated␣
↪over using:
# wait_time = blocks_for_consensus * avg_block_time_in_secs
# For mainnet and paying low gas, it might take 10 minutes.
confirmation_wait_time_in_seconds = 30
etherscan_link_mask = 'https://{}{}etherscan.io/tx/{}'

print('-= Ethereum Transaction using Incubed =- \n')
try:
    sender = client.eth.account.recover(sender_secret)
    tx = in3.eth.NewTransaction(to=receiver, value=value_in_wei)
    print('[.] Sending {} Wei from {} to {}. Please wait.\n'.format(tx.value, sender.
↪address, tx.to))
    tx_hash = client.eth.account.send_transaction(sender, tx)
    print('[.] Transaction accepted with hash {}.'.format(tx_hash))
    add_dot_if_chain = '.' if chain else ''
    print(etherscan_link_mask.format(chain, add_dot_if_chain, tx_hash))
    while True:
```

(continues on next page)

(continued from previous page)

```
        try:
            print('\n[.] Waiting {} seconds for confirmation.\n'.format(confirmation_
↪wait_time_in_seconds))
            time.sleep(confirmation_wait_time_in_seconds)
            receipt: in3.eth.TransactionReceipt = client.eth.transaction_receipt(tx_
↪hash)
            print('[.] Transaction was sent successfully!\n')
            print(json.dumps(receipt.to_dict(), indent=4, sort_keys=True))
            print('[.] Mined on block {} used {} GWei.'.format(receipt.blockNumber,␣
↪receipt.gasUsed))
            break
        except Exception:
            print('[!] Transaction not mined yet, check https://etherscan.io/
↪gasTracker.')
            print('[!] Just wait some minutes longer than the average for the price␣
↪paid!')
except in3.PrivateKeyNotFoundException as e:
    print(str(e))
except in3.ClientException as e:
    print('Client returned error: ', str(e))
    print('Please try again.')

# Response
"""
Ethereum Transaction using Incubed

Sending 1463926659 Wei from 0x0b56Ae81586D2728Ceaf7C00A6020C5D63f02308 to␣
↪0x6fa33809667a99a805b610c49ee2042863b1bb83.

Transaction accepted with hash␣
↪0xbeebda39e31e42d2a26476830fdcdc2d21e9df090af203e7601d76a43074d8d3.
https://goerli.etherscan.io/tx/
↪0xbeebda39e31e42d2a26476830fdcdc2d21e9df090af203e7601d76a43074d8d3

Waiting 25 seconds for confirmation.

Transaction was sent successfully!
{
    "From": "0x0b56Ae81586D2728Ceaf7C00A6020C5D63f02308",
    "blockHash": "0x9693714c9d7dbd31f36c04fbd262532e68301701b1da1a4ee8fc04e0386d868b",
    "blockNumber": 2615346,
    "cumulativeGasUsed": 21000,
    "gasUsed": 21000,
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "status": 1,
    "to": "0x6FA33809667A99A805b610C49EE2042863b1bb83",
    "transactionHash":
↪"0xbeebda39e31e42d2a26476830fdcdc2d21e9df090af203e7601d76a43074d8d3",
    "transactionIndex": 0
}

Mined on block 2615346 used 21000 GWei.
"""
```

12.2.5 smart_contract

source : in3-c/python/examples/smart_contract.py

```
"""
Manually calling ENS smart-contract
![UML Sequence Diagram of how Ethereum Name Service ENS resolves a name.](https://lh5.
↪googleusercontent.com/_
↪OPPzaxTxKggx9HuxloeWtK8ggEfIIBKRCEA6BKMwZdzAfUpIY6cz7NK5CFmiuw7TwknbhFNVRCJsswHLqkxUEJ5KdRzpeNbyg8_
↪H9d2RZdG28kgipT64JyPZUP--bAizozaDcxCq34)
"""
import in3


client = in3.Client('goerli')
domain_name = client.ens_namehash('depraz.eth')
ens_registry_addr = '0x00000000000c2e074ec69a0dfb2997ba6c7d2e1e'
ens_resolver_abi = 'resolver(bytes32):address'

# Find resolver contract for ens name
resolver_tx = {
    "to": ens_registry_addr,
    "data": client.eth.contract.encode(ens_resolver_abi, domain_name)
}
tx = in3.eth.NewTransaction(**resolver_tx)
encoded_resolver_addr = client.eth.contract.call(tx)
resolver_address = client.eth.contract.decode(ens_resolver_abi, encoded_resolver_addr)

# Resolve name
ens_addr_abi = 'addr(bytes32):address'
name_tx = {
    "to": resolver_address,
    "data": client.eth.contract.encode(ens_addr_abi, domain_name)
}
encoded_domain_address = client.eth.contract.call(in3.eth.NewTransaction(**name_tx))
domain_address = client.eth.contract.decode(ens_addr_abi, encoded_domain_address)

print('END domain:\n{}\nResolved by:\n{}\nTo address:\n{}'.format(domain_name,␣
↪resolver_address, domain_address))

# Produces
"""
END domain:
0x4a17491df266270a8801cee362535e520a5d95896a719e4a7d869fb22a93162e
Resolved by:
0x4b1488b7a6b320d2d721406204abc3eeaa9ad329
To address:
0x0b56ae81586d2728ceaf7c00a6020c5d63f02308
"""
```

12.2.6 Running the examples

To run an example, you need to install in3 first:

```
pip install in3
```

This will install the library system-wide. Please consider using `virtualenv` or `pipenv` for a project-wide install.

Then copy one of the examples and paste into a file, i.e. `example.py`:

MacOS

```
pbpaste > example.py
```

Execute the example with python:

```
python example.py
```

12.3 Incubed Modules

12.3.1 Client

```
Client(self,
chain: str = 'mainnet',
in3_config: ClientConfig = None,
cache_enabled: bool = True,
transport=<function https_transport at 0x1016b7f80>)
```

Incubed network client. Connect to the blockchain via a list of bootnodes, then gets the latest list of nodes in the network and ask a certain number of the to sign the block header of given list, putting their deposit at stake. Once with the latest list at hand, the client can request any other on-chain information using the same scheme.

Arguments:

- `chain` *str* - Ethereum chain to connect to. Defaults to mainnet. Options: 'mainnet', 'kovan', 'goerli', 'ewc'.
- `in3_config` *ClientConfig or str* - (optional) Configuration for the client. If not provided, default is loaded.
- `cache_enabled` *bool* - False will disable local storage caching.
- `transport` *function* - Transport function for custom request routing. Defaults to https.

refresh_node_list

```
Client.refresh_node_list()
```

Gets the list of Incubed nodes registered in the selected chain registry contract.

Returns:

- `node_list` *NodeList* - List of registered in3 nodes and metadata.

config

```
Client.config()
```

Client configuration dictionary.

Returns:

- `config` *dict* - Client configuration keys and values.

ens_namehash

```
Client.ens_namehash(domain_name: str)
```

Name format based on EIP-137

Arguments:

- `domain_name` - ENS supported domain. mydomain.ens, mydomain.xyz, etc

Returns:

- `node` *str* - Formatted string referred as `node` in ENS documentation

ens_address

```
Client.ens_address(domain_name: str, registry: str = None)
```

Resolves ENS domain name to what account that domain points to.

Arguments:

- `domain_name` - ENS supported domain. mydomain.ens, mydomain.xyz, etc
- `registry` - ENS registry contract address. i.e. 0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e

Returns:

- `address` *str* - Ethereum address corresponding to what account that domain points to.

ens_owner

```
Client.ens_owner(domain_name: str, registry: str = None)
```

Resolves ENS domain name to Ethereum address of domain owner.

Arguments:

- `domain_name` - ENS supported domain. i.e mydomain.eth
- `registry` - ENS registry contract address. i.e. 0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e

Returns:

- `owner_address` *str* - Ethereum address corresponding to domain owner.

ens_resolver

```
Client.ens_resolver(domain_name: str, registry: str = None)
```

Resolves ENS domain name to Smart-contract address of the resolver registered for that domain.

Arguments:

- `domain_name` - ENS supported domain. i.e mydomain.eth
- `registry` - ENS registry contract address. i.e. 0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e

Returns:

- `resolver_contract_address` *str* - Smart-contract address of the resolver registered for that domain.

12.3.2 ClientConfig

```
ClientConfig(self,
chain_finality_threshold: int = None,
account_secret: str = None,
latest_block_stall: int = None,
node_signatures: int = None,
node_signature_consensus: int = None,
node_min_deposit: int = None,
node_list_auto_update: bool = None,
node_limit: int = None,
request_timeout: int = None,
request_retries: int = None,
response_proof_level: str = None,
response_includes_code: bool = None,
response_keep_proof: bool = None,
transport_binary_format: bool = None,
transport_ignore_tls: bool = None,
boot_weights: bool = None,
in3_registry: dict = None)
```

Determines the behavior of the in3 client, which chain to connect to and how to manage information security policies.

Considering integrity is guaranteed and confidentiality is not available on public blockchains, these settings will provide a balance between availability, and financial stake in case of repudiation.

The newer the block is, higher are the chances it gets repudiated by a fork in the chain. In3 nodes will decide individually to sign on repudiable information, reducing the availability. If the application needs the very latest block, consider using a calculated value in `node_signature_consensus` and set `node_signatures` to zero. This setting is as secure as a light-client.

The older the block gets, the highest is its availability because of the close-to-zero repudiation risk, but blocks older than circa one year are stored in Archive Nodes, expensive computers, so, despite of the low risk, there are not many nodes available with such information, and they must search for the requested block in its database, lowering the availability as well. If the application needs access to *old* blocks, consider setting `request_timeout` and `request_retries` to accomodate the time the archive nodes take to fetch the inforamtion.

The verification policy enforces an extra step of security, adding a financial stake in case of repudiation or false/broken proof. For high security application, consider setting a calculated value in `node_min_deposit` and request as much signatures as necessary in `node_signatures`. Setting `chain_finality_threshold` high will guarantee non-repudiability.

All args are Optional. Defaults connect to Ethereum main network with regular security levels.

Arguments:

- `chain_finality_threshold` *int* - Behavior depends on the chain consensus algorithm: POA - percent of signers needed in order reach finality (% of the validators) i.e.: 60 %. POW - mined blocks on top of the requested, i.e. 8 blocks. Defaults are defined in enum.Chain.
- `latest_block_stall` *int* - Distance considered safe, consensus wise, from the very latest block. Higher values exponentially increases state finality, and therefore data security, as well guaranteeded responses from in3 nodes. example: 10 - will ask for the state from (latestBlock-10).
- `account_secret` *str* - Account SK to sign all in3 requests. (Experimental use `set_account_sk`) example: 0x387a8233c96e1fc0ad5e284353276177af2186e7afa85296f106336e376669f7
- `node_signatures` *int* - Node signatures attesting the response to your request. Will send a separate request for each. example: 3 nodes will have to sign the response.

- `node_signature_consensus` *int* - Useful when node_signatures <= 1. The client will check for consensus in responses. example: 10 - will ask for 10 different nodes and compare results looking for a consensus in the responses.
- `node_min_deposit` *int* - Only nodes owning at least this amount will be chosen to sign responses to your requests. i.e. 1000000000000000000 Wei
- `node_list_auto_update` *bool* - If true the nodelist will be automatically updated. False may compromise data security.
- `node_limit` *int* - Limit nodes stored in the client. example: 150 nodes
- `request_timeout` *int* - Milliseconds before a request times out. example: 100000 ms
- `request_retries` *int* - Maximum times the client will retry to contact a certain node. example: 10 retries
- `response_proof_level` *str* - 'none'|'standard'|'full' Full gets the whole block Patricia-Merkle-Tree, Standard only verifies the specific tree branch concerning the request, None only verifies the root hashes, like a light-client does.
- `response_includes_code` *bool* - If true, every request with the address field will include the data, if existent, that is stored in that wallet/smart-contract. If false, only the code digest is included.
- `response_keep_proof` *bool* - If true, proof data will be kept in every rpc response. False will remove this data after using it to verify the responses. Useful for debugging and manually verifying the proofs.
- `transport_binary_format` - If true, the client will communicate with the server using a binary payload instead of json.
- `transport_ignore_tls` - The client usually verify https tls certificates. To communicate over insecure http, turn this on.
- `boot_weights` *bool* - if true, the first request (updating the nodelist) will also fetch the current health status and use it for blacklisting unhealthy nodes. This is used only if no nodelist is availabkle from cache.
- `in3_registry` *dict* - In3 Registry Smart Contract configuration data

12.3.3 In3Node

```
In3Node(self, url: str, address: Account, index: int, deposit: int,
props: int, timeout: int, registerTime: int, weight: int)
```

Registered remote verifier that attest, by signing the block hash, that the requested block and transaction were indeed mined are in the correct chain fork.

Arguments:

- `url` *str* - Endpoint to post to example: https://in3.slock.it
- `index` *int* - Index within the contract example: 13
- `address` *in3.Account* - Address of the node, which is the public address it is signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- `deposit` *int* - Deposit of the node in wei example: 12350000
- `props` *int* - Properties of the node. example: 3
- `timeout` *int* - Time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600
- `registerTime` *int* - When the node was registered in (unixtime?)

- `weight` *int* - Score based on qualitative metadata to base which nodes to ask signatures from.

12.3.4 NodeList

```
NodeList(self, nodes: [<class 'in3.model.In3Node'>], contract: Account,
registryId: str, lastBlockNumber: int, totalServers: int)
```

List of incubed nodes and its metadata, in3 registry contract from which the list was taken, network/registry id, and last block number in the selected chain.

Arguments:

- `nodes` *[In3Node]* - list of incubed nodes
- `contract` *Account* - incubed registry contract from which the list was taken
- `registryId` *str* - uuid of this incubed network. one chain could contain more than one incubed networks.
- `lastBlockNumber` *int* - last block signed by the network
- `totalServers` *int* - Total servers number (for integrity?)

12.3.5 EthAccountApi

```
EthAccountApi(self, runtime: In3Runtime, factory: EthObjectFactory)
```

Manages accounts and smart-contracts

create

```
EthAccountApi.create(qrng=False)
```

Creates a new Ethereum account and saves it in the wallet.

Arguments:

- `qrng` *bool* - True uses a Quantum Random Number Generator api for generating the private key.

Returns:

- `account` *Account* - Newly created Ethereum account.

recover

```
EthAccountApi.recover(secret: str)
```

Recovers an account from a secret.

Arguments:

- `secret` *str* - Account private key in hexadecimal string

Returns:

- `account` *Account* - Recovered Ethereum account.

parse_mnemonics

```
EthAccountApi.parse_mnemonics(mnemonics: str)
```

Recovers an account secret from mnemonics phrase

Arguments:

- `mnemonics` *str* - BIP39 mnemonics phrase.

Returns:

- `secret` *str* - Account secret. Use `recover_account` to create a new account with this secret.

sign

```
EthAccountApi.sign(private_key: str, message: str)
```

Use ECDSA to sign a message.

Arguments:

- `private_key` *str* - Must be either an address(20 byte) or an raw private key (32 byte)"}}'
- `message` *str* - Data to be hashed and signed. Dont input hashed data unless you know what you are doing.

Returns:

- `signed_message` *str* - ECDSA calculated r, s, and parity v, concatenated. v = 27 + (r % 2)

balance

```
EthAccountApi.balance(address: str, at_block: int = 'latest')
```

Returns the balance of the account of given address.

Arguments:

- `address` *str* - address to check for balance
- `at_block` *int or str* - block number IN3BlockNumber or EnumBlockStatus

Returns:

- `balance` *int* - integer of the current balance in wei.

send_transaction

```
EthAccountApi.send_transaction(sender: Account,
transaction: NewTransaction)
```

Signs and sends the assigned transaction. Requires `account.secret` value set. Transactions change the state of an account, just the balance, or additionally, the storage and the code. Every transaction has a cost, gas, paid in Wei. The transaction gas is calculated over estimated gas times the gas cost, plus an additional miner fee, if the sender wants to be sure that the transaction will be mined in the latest block.

Arguments:

- `sender` *Account* - Sender Ethereum account. Senders generally pay the gas costs, so they must have enough balance to pay gas + amount sent, if any.
- `transaction` *NewTransaction* - All information needed to perform a transaction. Minimum is to and value. Client will add the other required fields, gas and chaindId.

Returns:

- `tx_hash` *hex* - Transaction hash, used to get the receipt and check if the transaction was mined.

send_raw_transaction

```
EthAccountApi.send_raw_transaction(signed_transaction: str)
```

Sends a signed and encoded transaction.

Arguments:

- `signed_transaction` - Signed keccak hash of the serialized transaction Client will add the other required fields, gas and chaindId.

Returns:

- `tx_hash` *hex* - Transaction hash, used to get the receipt and check if the transaction was mined.

estimate_gas

```
EthAccountApi.estimate_gas(transaction: NewTransaction)
```

Gas estimation for transaction. Used to fill transaction.gas field. Check RawTransaction docs for more on gas.

Arguments:

- `transaction` - Unsent transaction to be estimated. Important that the fields data or/and value are filled in.

Returns:

- `gas` *int* - Calculated gas in Wei.

transaction_count

```
EthAccountApi.transaction_count(address: str, at_block: int = 'latest')
```

Number of transactions mined from this address. Used to set transaction nonce. Nonce is a value that will make a transaction fail in case it is different from (transaction count + 1). It exists to mitigate replay attacks.

Arguments:

- `address` *str* - Ethereum account address
- `at_block` *int* - Block number

Returns:

- `tx_count` *int* - Number of transactions mined from this address.

checksum_address

```
EthAccountApi.checksum_address(address: str, add_chain_id: bool = True)
```

Will convert an upper or lowercase Ethereum address to a checksum address, that uses case to encode values. See EIP55.

Arguments:

- `address` - Ethereum address string or object.
- `add_chain_id` *bool* - Will append the chain id of the address, for multi-chain support, canonical for Eth.

Returns:

- `checksum_address` - EIP-55 compliant, mixed-case address object.

12.3.6 EthContractApi

```
EthContractApi(self, runtime: In3Runtime, factory: EthObjectFactory)
```

Manages smart-contract data and transactions

call

```
EthContractApi.call(transaction: NewTransaction,
block_number: int = 'latest')
```

Calls a smart-contract method. Will be executed locally by Incubed's EVM or signed and sent over to save the state changes. Check https://ethereum.stackexchange.com/questions/3514/how-to-call-a-contract-method-using-the-eth-call-json-rpc-api for more.

Arguments:

transaction (NewTransaction):

- `block_number` *int or str* - Desired block number integer or 'latest', 'earliest', 'pending'.

Returns:

- `method_returned_value` - A hexadecimal. For decoding use in3.abi_decode.

storage_at

```
EthContractApi.storage_at(address: str,
position: int = 0,
at_block: int = 'latest')
```

Stored value in designed position at a given address. Storage can be used to store a smart contract state, constructor or just any data. Each contract consists of a EVM bytecode handling the execution and a storage to save the state of the contract. The storage is essentially a key/value store. Use get_code to get the smart-contract code.

Arguments:

- `address` *str* - Ethereum account address
- `position` *int* - Position index, 0x0 up to 0x64

- `at_block` *int or str* - Block number

Returns:

- `storage_at` *str* - Stored value in designed position. Use decode('hex') to see ascii format of the hex data.

code

```
EthContractApi.code(address: str, at_block: int = 'latest')
```

Smart-Contract bytecode in hexadecimal. If the account is a simple wallet the function will return '0x'.

Arguments:

- `address` *str* - Ethereum account address
- `at_block` *int or str* - Block number

Returns:

- `bytecode` *str* - Smart-Contract bytecode in hexadecimal.

encode

```
EthContractApi.encode(fn_signature: str, *fn_args)
```

Smart-contract ABI encoder. Used to serialize a rpc to the EVM. Based on the Solidity specification. Note: Parameters refers to the list of variables in a method declaration. Arguments are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Arguments:

- `fn_signature` *str* - Function name, with parameters. i.e. `getBalance(uint256):uint256`, can contain the return types but will be ignored.
- `fn_args` *tuple* - Function parameters, in the same order as in passed on to method_name.

Returns:

- `encoded_fn_call` *str* - i.e. "0xf8b2cb4f000000000000000000000000001234567890123456789012345678901234567890"

decode

```
EthContractApi.decode(fn_signature: str, encoded_value: str)
```

Smart-contract ABI decoder. Used to parse rpc responses from the EVM. Based on the Solidity specification.

Arguments:

- `fn_signature` - Function signature. e.g. `(address,string,uint256)` or `getBalance(address):uint256`. In case of the latter, the function signature will be ignored and only the return types will be parsed.
- `encoded_value` - Abi encoded values. Usually the string returned from a rpc to the EVM.

Returns:

- `decoded_return_values` *tuple* - "0x1234567890123456789012345678901234567890", "0x05"

12.3.7 EthereumApi

```
EthereumApi(self, runtime: In3Runtime)
```

Module based on Ethereum's api and web3.js

keccak256

```
EthereumApi.keccak256(message: str)
```

Keccak-256 digest of the given data. Compatible with Ethereum but not with SHA3-256.

Arguments:

- `message` *str* - Message to be hashed.

Returns:

- `digest` *str* - The message digest.

gas_price

```
EthereumApi.gas_price()
```

The current gas price in Wei (1 ETH equals 1000000000000000000 Wei).

Returns:

- `price` *int* - minimum gas value for the transaction to be mined

block_number

```
EthereumApi.block_number()
```

Returns the number of the most recent block the in3 network can collect signatures to verify. Can be changed by Client.Config.replaceLatestBlock. If you need the very latest block, change Client.Config.signatureCount to zero.

Returns:

block_number (int) : Number of the most recent block

block_by_hash

```
EthereumApi.block_by_hash(block_hash: str, get_full_block: bool = False)
```

Blocks can be identified by root hash of the block merkle tree (this), or sequential number in which it was mined (get_block_by_number).

Arguments:

- `block_hash` *str* - Desired block hash
- `get_full_block` *bool* - If true, returns the full transaction objects, otherwise only its hashes.

Returns:

- `block` *Block* - Desired block, if exists.

block_by_number

```
EthereumApi.block_by_number(block_number: [<class 'int'>],
get_full_block: bool = False)
```

Blocks can be identified by sequential number in which it was mined, or root hash of the block merkle tree (this) (get_block_by_hash).

Arguments:

- `block_number` *int or str* - Desired block number integer or 'latest', 'earliest', 'pending'.
- `get_full_block` *bool* - If true, returns the full transaction objects, otherwise only its hashes.

Returns:

- `block` *Block* - Desired block, if exists.

transaction_by_hash

```
EthereumApi.transaction_by_hash(tx_hash: str)
```

Transactions can be identified by root hash of the transaction merkle tree (this) or by its position in the block transactions merkle tree. Every transaction hash is unique for the whole chain. Collision could in theory happen, chances are 67148E-63%.

Arguments:

- `tx_hash` - Transaction hash.

Returns:

- `transaction` - Desired transaction, if exists.

transaction_receipt

```
EthereumApi.transaction_receipt(tx_hash: str)
```

After a transaction is received the by the client, it returns the transaction hash. With it, it is possible to gather the receipt, once a miner has mined and it is part of an acknowledged block. Because how it is possible, in distributed systems, that data is asymmetric in different parts of the system, the transaction is only "final" once a certain number of blocks was mined after it, and still it can be possible that the transaction is discarded after some time. But, in general terms, it is accepted that after 6 to 8 blocks from latest, that it is very likely that the transaction will stay in the chain.

Arguments:

- `tx_hash` - Transaction hash.

Returns:

- `tx_receipt` - The mined Transaction data including event logs.

12.3.8 Ethereum Objects

DataTransferObject

```
DataTransferObject()
```

Maps marshalling objects transferred to, and from a remote facade, in this case, libin3 rpc api. For more on design-patterns see Martin Fowler's Catalog of Patterns of Enterprise Application Architecture.

Transaction

```
Transaction(self, From: str, to: str, gas: int, gasPrice: int, hash: str,
nonce: int, transactionIndex: int, blockHash: str,
value: int, input: str, publicKey: str, standardV: int,
raw: str, creates: str, chainId: int, r: int, s: int,
v: int)
```

Arguments:

- `From` *hex str* - Address of the sender account.
- `to` *hex str* - Address of the receiver account. Left undefined for a contract creation transaction.
- `gas` *int* - Gas for the transaction miners and execution in wei. Will get multiplied by `gasPrice`. Use in3.eth.account.estimate_gas to get a calculated value. Set too low and the transaction will run out of gas.
- `value` *int* - Value transferred in wei. The endowment for a contract creation transaction.
- `data` *hex str* - Either a ABI byte string containing the data of the function call on a contract, or in the case of a contract-creation transaction the initialisation code.
- `gasPrice` *int* - Price of gas in wei, defaults to in3.eth.gasPrice. Also know as `tx fee price`. Set your gas price too low and your transaction may get stuck. Set too high on your own loss. gasLimit (int); Maximum gas paid for this transaction. Set by the client using this rationale if left empty: gasLimit = G(transaction) + G(txdatanonzero) × dataByteLength. Minimum is 21000.
- `nonce` *int* - Number of transactions mined from this address. Nonce is a value that will make a transaction fail in case it is different from (transaction count + 1). It exists to mitigate replay attacks. This allows to overwrite your own pending transactions by sending a new one with the same nonce. Use in3.eth.account.get_transaction_count to get the latest value.
- `hash` *hex str* - Keccak of the transaction bytes, not part of the transaction. Also known as receipt, because this field is filled after the transaction is sent, by eth_sendTransaction
- `blockHash` *hex str* - Block hash that this transaction was mined in. null when its pending.
- `blockHash` *int* - Block number that this transaction was mined in. null when its pending.
- `transactionIndex` *int* - Integer of the transactions index position in the block. null when its pending.
- `signature` *hex str* - ECDSA of transaction.data, calculated r, s and v concatenated. V is parity set by v = 27 + (r % 2).

NewTransaction

```
NewTransaction(self,
From: str = None,
to: str = None,
nonce: int = None,
value: int = None,
```

(continues on next page)

(continued from previous page)

```
data: str = None,
gasPrice: int = None,
gasLimit: int = None,
hash: str = None,
signature: str = None)
```

Unsent transaction. Use to send a new transaction.

Arguments:

- `From` *hex str* - Address of the sender account.
- `to` *hex str* - Address of the receiver account. Left undefined for a contract creation transaction.
- `value` *int* - (optional) Value transferred in wei. The endowment for a contract creation transaction.
- `data` *hex str* - (optional) Either a ABI byte string containing the data of the function call on a contract, or in the case of a contract-creation transaction the initialisation code.
- `gasPrice` *int* - (optional) Price of gas in wei, defaults to in3.eth.gasPrice. Also know as `tx fee price`. Set your gas price too low and your transaction may get stuck. Set too high on your own loss. gasLimit (int); (optional) Maximum gas paid for this transaction. Set by the client using this rationale if left empty: gasLimit = G(transaction) + G(txdatanonzero) × dataByteLength. Minimum is 21000.
- `nonce` *int* - (optional) Number of transactions mined from this address. Nonce is a value that will make a transaction fail in case it is different from (transaction count + 1). It exists to mitigate replay attacks. This allows to overwrite your own pending transactions by sending a new one with the same nonce. Use in3.eth.account.get_transaction_count to get the latest value.
- `hash` *hex str* - (optional) Keccak of the transaction bytes, not part of the transaction. Also known as receipt, because this field is filled after the transaction is sent.
- `signature` *hex str* - (optional) ECDSA of transaction, r, s and v concatenated. V is parity set by v = 27 + (r % 2).

Filter

```
Filter(self, fromBlock: int, toBlock: int, address: str, topics: list,
blockhash: str)
```

Filters are event catchers running on the Ethereum Client. Incubed has a client-side implementation. An event will be stored in case it is within to and from blocks, or in the block of blockhash, contains a transaction to the designed address, and has a word listed on topics.

Log

```
Log(self, address: <built-in function hex>,
blockHash: <built-in function hex>, blockNumber: int,
data: <built-in function hex>, logIndex: int, removed: bool,
topics: [<built-in function hex>],
transactionHash: <built-in function hex>, transactionIndex: int,
transactionLogIndex: int, Type: str)
```

Transaction Log for events and data returned from smart-contract method calls.

TransactionReceipt

```
TransactionReceipt(self,
blockHash: <built-in function hex>,
blockNumber: int,
cumulativeGasUsed: int,
From: str,
gasUsed: int,
logsBloom: <built-in function hex>,
status: int,
transactionHash: <built-in function hex>,
transactionIndex: int,
logs: [<class 'in3.eth.model.Log'>] = None,
to: str = None,
contractAddress: str = None)
```

Receipt from a mined transaction.

Arguments:

blockHash: blockNumber:

- `cumulativeGasUsed` - total amount of gas used by block From:
- `gasUsed` - amount of gas used by this specific transaction logs: logsBloom:
- `status` - 1 if transaction succeeded, 0 otherwise. transactionHash: transactionIndex:
- `to` - Account to which this transaction was sent. If the transaction was a contract creation this value is set to None.
- `contractAddress` - Contract Account address created, f the transaction was a contract creation, or None otherwise.

Account

```
Account(self,
address: str,
chain_id: int,
secret: int = None,
domain: str = None)
```

An Ethereum account.

Arguments:

- `address` - Account address. Derived from public key.
- `chain_id` - ID of the chain the account is used in.
- `secret` - Account private key. A 256 bit number.
- `domain` - ENS Domain name. ie. niceguy.eth

12.4 Library Runtime

Shared Library Runtime module

Loads `libin3` according to host hardware architecture and OS. Maps symbols, methods and types from the library. Encapsulates low-level rpc calls into a comprehensive runtime.

12.4.1 In3Runtime

```
In3Runtime(self, chain_id: int, cache_enabled: bool, transport_fn)
```

Instantiate libin3 and frees it when garbage collected.

Arguments:

- `chain_id` *int* - Chain-id based on EIP-155. Default is 0x1 for Ethereum mainNet.
- `cache_enabled` *bool* - False will disable local storage cache.
- `transport_fn` - Transport function to handle the HTTP Incubed Network requests.

12.4.2 in3.libin3.rpc_api

Load libin3 shared library for the current system, map function ABI, sets in3 network transport functions.

libin3_new

```
libin3_new(chain_id: int, cache_enabled: bool,
transport_fn: <function CFUNCTYPE at 0x1019e3320>)
```

Instantiate new In3 Client instance.

Arguments:

- `chain_id` *int* - Chain id as integer
- `cache_enabled` *bool* - False will disable local storage cache.
- `transport_fn` - Transport function for the in3 network requests
- `storage_fn` - Cache Storage function for node list and requests caching

Returns:

- `instance` *int* - Memory address of the client instance, return value from libin3_new

libin3_free

```
libin3_free(instance: int)
```

Free In3 Client objects from memory.

Arguments:

- `instance` *int* - Memory address of the client instance, return value from libin3_new

libin3_call

```
libin3_call(instance: int, fn_name: bytes, fn_args: bytes)
```

Make Remote Procedure Call to an arbitrary method of a libin3 instance

Arguments:

- `instance` *int* - Memory address of the client instance, return value from libin3_new
- `fn_name` *bytes* - Name of function that will be called in the client rpc.
- `fn_args` - (bytes) Serialized list of arguments, matching the parameters order of this function. i.e. ['0x123']

Returns:

- `result` *int* - Function execution status.

libin3_set_pk

```
libin3_set_pk(instance: int, private_key: bytes)
```

Register the signer module in the In3 Client instance, with selected private key loaded in memory.

Arguments:

- `instance` *int* - Memory address of the client instance, return value from libin3_new
- `private_key` - 256 bit number.

libin3_in3_req_add_response

```
libin3_in3_req_add_response(*args)
```

Transport function that registers a response to a request.

Arguments:

*args:

libin3_new_bytes_t

```
libin3_new_bytes_t(value: bytes, length: int)
```

C Bytes struct

Arguments:

- `length` - byte array length
- `value` - byte array

Returns:

- `ptr_addr` - address of the instance of this struct

CHAPTER 13

API Reference Java

13.1 Installing

The Incubed Java client uses JNI in order to call native functions. But all the native-libraries are bundled inside the jar-file. This jar file ha **no** dependencies and can even be used standalone:

like

```
java -cp in3.jar in3.IN3 eth_getBlockByNumber latest false
```

13.1.1 Downloading

The jar file can be downloaded from the latest release. here.

Alternatively, If you wish to download Incubed using the maven package manager, add this to your pom.xml

```
<dependency>
  <groupId>it.slock</groupId>
  <artifactId>in3</artifactId>
  <version>2.21</version>
</dependency>
```

After which, install in3 with `mvn install`.

13.1.2 Building

For building the shared library you need to enable java by using the `-DJAVA=true` flag:

```
git clone git@github.com:slockit/in3-c.git
mkdir -p in3-c/build
cd in3-c/build
cmake -DJAVA=true .. && make
```

You will find the `in3.jar` in the build/lib - folder.

13.1.3 Android

In order to use Incubed in android simply follow these steps:

Step 1: Create a top-level CMakeLists.txt in android project inside app folder and link this to gradle. Follow the steps using this guide on howto link.

The Content of the `CMakeLists.txt` should look like this:

```
cmake_minimum_required(VERSION 3.4.1)

# turn off FAST_MATH in the evm.
ADD_DEFINITIONS(-DIN3_MATH_LITE)

# loop through the required module and cretae the build-folders
foreach(module
  c/src/core
  c/src/verifier/eth1/nano
  c/src/verifier/eth1/evm
  c/src/verifier/eth1/basic
  c/src/verifier/eth1/full
  java/src
  c/src/third-party/crypto
  c/src/third-party/tommath
  c/src/api/eth1)
        file(MAKE_DIRECTORY in3-c/${module}/outputs)
        add_subdirectory( in3-c/${module} in3-c/${module}/outputs )
endforeach()
```

Step 2: clone in3-c into the `app`-folder or use this script to clone and update in3:

```
#!/usr/bin/env sh

#github-url for in3-c
IN3_SRC=https://github.com/slockit/in3-c.git

cd app

# if it exists we only call git pull
if [ -d in3-c ]; then
    cd in3-c
    git pull
    cd ..
else
# if not we clone it
    git clone $IN3_SRC
fi


# copy the java-sources to the main java path
cp -r in3-c/java/src/in3 src/main/java/
```

Step 3: Use methods available in app/src/main/java/in3/IN3.java from android activity to access IN3 functions.

Here is example how to use it:

https://github.com/slockit/in3-example-android

13.2 Examples

13.2.1 CallFunction

source : in3-c/java/examples/CallFunction.java

Calling Functions of Contracts

```
// This Example shows how to call functions and use the decoded results. Here we get␣
↪the struct from the registry.

import in3.*;
import in3.eth1.*;

public class CallFunction {
  //
  public static void main(String[] args) {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also dthe␣
↪default)

    // call a contract, which uses eth_call to get the result.
    Object[] result = (Object[]) in3.getEth1API().call(                   // call␣
↪a function of a contract
        "0x2736D225f85740f42D17987100dc8d58e9e16252",                       //␣
↪address of the contract
        "servers(uint256):(string,address,uint256,uint256,uint256,address)", //␣
↪function signature
        1);                                                                  // first␣
↪argument, which is the index of the node we are looking for.

    System.out.println("url     : " + result[0]);
    System.out.println("owner   : " + result[1]);
    System.out.println("deposit : " + result[2]);
    System.out.println("props   : " + result[3]);
  }
}
```

13.2.2 Configure

source : in3-c/java/examples/Configure.java

Changing the default configuration

```
// In order to change the default configuration, just use the classes inside in3.
↪config package.

package in3;

import in3.*;
import in3.config.*;
import in3.eth1.Block;

public class Configure {
  //
  public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
    // create incubed client
    IN3 in3 = IN3.forChain(Chain.GOERLI); // set it to goerli

    // Setup a Configuration object for the client
    ClientConfiguration clientConfig = in3.getConfig();
    clientConfig.setReplaceLatestBlock(6); // define that latest will be -6
    clientConfig.setAutoUpdateList(false); // prevents node automatic update
    clientConfig.setMaxAttempts(1);         // sets max attempts to 1 before giving up
    clientConfig.setProof(Proof.none);      // does not require proof (not recommended)

    // Setup the ChainConfiguration object for the nodes on a certain chain
    ChainConfiguration chainConfiguration = new ChainConfiguration(Chain.GOERLI,␣
↪clientConfig);
    chainConfiguration.setNeedsUpdate(false);
    chainConfiguration.setContract("0xac1b824795e1eb1f6e609fe0da9b9af8beaab60f");
    chainConfiguration.setRegistryId(
↪"0x23d5345c5c13180a8080bd5ddbe7cde64683755dcce6e734d95b7b573845facb");

    in3.setConfig(clientConfig);

    Block block = in3.getEth1API().getBlockByNumber(Block.LATEST, true);
    System.out.println(block.getHash());
  }
}
```

13.2.3 GetBalance

source : in3-c/java/examples/GetBalance.java

getting the Balance with or without API

```
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;
import java.util.*;

public class GetBalance {

  static String AC_ADDR = "0xc94770007dda54cF92009BFF0dE90c06F603a09f";

  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also dthe␣
↪default)

    System.out.println("Balance API" + getBalanceAPI(in3).longValue());

    System.out.println("Balance RPC " + getBalanceRPC(in3));
  }

  static BigInteger getBalanceAPI(IN3 in3) {
    return in3.getEth1API().getBalance(AC_ADDR, Block.LATEST);
  }

  static String getBalanceRPC(IN3 in3) {
```

(continues on next page)

(continued from previous page)

```
    return in3.sendRPC("eth_getBalance", new Object[] {AC_ADDR, "latest"});
  }
}
```

13.2.4 GetBlockAPI

source : in3-c/java/examples/GetBlockAPI.java

getting a block with API

```
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;
import java.util.*;

public class GetBlockAPI {
  //
  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also dthe␣
↪default)

    // read the latest Block including all Transactions.
    Block latestBlock = in3.getEth1API().getBlockByNumber(Block.LATEST, true);

    // Use the getters to retrieve all containing data
    System.out.println("current BlockNumber : " + latestBlock.getNumber());
    System.out.println("minded at : " + new Date(latestBlock.getTimeStamp()) + " by "␣
↪+ latestBlock.getAuthor());

    // get all Transaction of the Block
    Transaction[] transactions = latestBlock.getTransactions();

    BigInteger sum = BigInteger.valueOf(0);
    for (int i = 0; i < transactions.length; i++)
      sum = sum.add(transactions[i].getValue());

    System.out.println("total Value transfered in all Transactions : " + sum + " wei
↪");
  }
}
```

13.2.5 GetBlockRPC

source : in3-c/java/examples/GetBlockRPC.java

getting a block without API

```
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;
import java.util.*;

public class GetBlockRPC {
```

(continues on next page)

(continued from previous page)

```
  //
  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also the␣
↪default)

    // read the latest Block without the Transactions.
    String result = in3.sendRPC("eth_getBlockByNumber", new Object[] {"latest", false}
↪);

    // print the json-data
    System.out.println("current Block : " + result);
  }
}
```

13.2.6 GetTransaction

source : in3-c/java/examples/GetTransaction.java

getting a Transaction with or without API

```
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;
import java.util.*;

public class GetTransaction {

  static String TXN_HASH =
↪"0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e";

  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also dthe␣
↪default)

    Transaction txn = getTransactionAPI(in3);
    System.out.println("Transaction API #blockNumber: " + txn.getBlockNumber());

    System.out.println("Transaction RPC :" + getTransactionRPC(in3));
  }

  static Transaction getTransactionAPI(IN3 in3) {
    return in3.getEth1API().getTransactionByHash(TXN_HASH);
  }

  static String getTransactionRPC(IN3 in3) {
    return in3.sendRPC("eth_getTransactionByHash", new Object[] {TXN_HASH});
  }
}
```

13.2.7 GetTransactionReceipt

source : in3-c/java/examples/GetTransactionReceipt.java getting a TransactionReceipt with or without API

```
import in3.*;
import in3.eth1.*;
import java.math.BigInteger;
import java.util.*;

public class GetTransactionReceipt {
  static String TRANSACTION_HASH =
↪"0xdd80249a0631cf0f1593c7a9c9f9b8545e6c88ab5252287c34bc5d12457eab0e";

  //
  public static void main(String[] args) throws Exception {
    // create incubed
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also the␣
↪default)

    TransactionReceipt txn = getTransactionReceiptAPI(in3);
    System.out.println("TransactionRerceipt API : for txIndex " + txn.
↪getTransactionIndex() + " Block num " + txn.getBlockNumber() + " Gas used " + txn.
↪getGasUsed() + " status " + txn.getStatus());

    System.out.println("TransactionReceipt RPC : " + getTransactionReceiptRPC(in3));
  }

  static TransactionReceipt getTransactionReceiptAPI(IN3 in3) {
    return in3.getEth1API().getTransactionReceipt(TRANSACTION_HASH);
  }

  static String getTransactionReceiptRPC(IN3 in3) {
    return in3.sendRPC("eth_getTransactionReceipt", new Object[] {TRANSACTION_HASH});
  }
}
```

13.2.8 SendTransaction

source : in3-c/java/examples/SendTransaction.java

Sending Transactions

```
// In order to send, you need a Signer. The SimpleWallet class is a basic␣
↪implementation which can be used.

package in3;

import in3.*;
import in3.eth1.*;
import java.io.IOException;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class SendTransaction {
  //
  public static void main(String[] args) throws IOException {
    // create incubed
```

(continues on next page)

(continued from previous page)

```
    IN3 in3 = IN3.forChain(Chain.MAINNET); // set it to mainnet (which is also dthe␣
↪default)

    // create a wallet managing the private keys
    SimpleWallet wallet = new SimpleWallet();

    // add accounts by adding the private keys
    String keyFile      = "myKey.json";
    String myPassphrase = "<secrect>";

    // read the keyfile and decoded the private key
    String account = wallet.addKeyStore(
        Files.readString(Paths.get(keyFile)),
        myPassphrase);

    // use the wallet as signer
    in3.setSigner(wallet);

    String     receipient = "0x1234567890123456789012345678901234567890";
    BigInteger value      = BigInteger.valueOf(100000);

    // create a Transaction
    TransactionRequest tx = new TransactionRequest();
    tx.setFrom(account);
    tx.setTo("0x1234567890123456789012345678901234567890");
    tx.setFunction("transfer(address,uint256)");
    tx.setParams(new Object[] {receipient, value});

    String txHash = in3.getEth1API().sendTransaction(tx);

    System.out.println("Transaction sent with hash = " + txHash);
  }
}
```

13.2.9 Building

In order to run those examples, you only need a Java SDK installed.

```
./build.sh
```

will build all examples in this directory.

In order to run a example use

```
java -cp $IN3/build/lib/in3.jar:. GetBlockAPI
```

13.3 Package in3

13.3.1 class BlockID

fromHash

public static *BlockID* fromHash(*String* hash);

arguments:

String	**hash**

fromNumber

public static *BlockID* fromNumber(*long* number);

arguments:

long	**number**

getNumber

public Long getNumber();

setNumber

public void setNumber(*long* block);

arguments:

long	**block**

getHash

public String getHash();

setHash

public void setHash(*String* hash);

arguments:

String	**hash**

toJSON

public String toJSON();

toString

public String toString();

13.3.2 class Chain

Constants for Chain-specs.

MULTICHAIN

support for multiple chains, a client can then switch between different chains (but consumes more memory)

Type: static `final long`

MAINNET

use mainnet

Type: static `final long`

KOVAN

use kovan testnet

Type: static `final long`

TOBALABA

use tobalaba testnet

Type: static `final long`

GOERLI

use goerli testnet

Type: static `final long`

EWC

use ewf chain

Type: static `final long`

EVAN

use evan testnet

Type: static `final long`

IPFS

use ipfs

Type: static `final long`

VOLTA

use volta test net

Type: static `final long`

LOCAL

use local client

Type: static `final long`

BTC

use bitcoin client

Type: static `final long`

13.3.3 class IN3

This is the main class creating the incubed client.

The client can then be configured.

IN3

public IN3();

getConfig

returns the current configuration.

any changes to the configuration will be applied witth the next request.

public *ClientConfiguration* getConfig();

setSigner

sets the signer or wallet.

public `void` setSigner(*Signer* signer);

arguments:

Signer	**signer**

getSigner

returns the signer or wallet.

public *Signer* getSigner();

getIpfs

gets the ipfs-api

public *in3.ipfs.API* getIpfs();

getBtcAPI

gets the btc-api

public *in3.btc.API* getBtcAPI();

getEth1API

gets the ethereum-api

public *in3.eth1.API* getEth1API();

getCrypto

gets the utils/crypto-api

public *Crypto* getCrypto();

setStorageProvider

provides the ability to cache content like nodelists, contract codes and validatorlists

public void setStorageProvider(*StorageProvider* val);

arguments:

StorageProvider	**val**

getStorageProvider

provides the ability to cache content

public *StorageProvider* getStorageProvider();

setTransport

sets The transport interface.

This allows to fetch the result of the incubed in a different way.

public void setTransport(*IN3Transport* newTransport);

arguments:

IN3Transport	**newTransport**

getTransport

returns the current transport implementation.

public IN3Transport getTransport();

getChainId

servers to filter for the given chain.

The chain-id based on EIP-155.

public `native long` getChainId();

setChainId

sets the chain to be used.

The chain-id based on EIP-155.

public `native void` setChainId(*long* val);

arguments:

`long`	**val**

send

send a request.

The request must a valid json-string with method and params

public `String` send(*String* request);

arguments:

`String`	**request**

sendobject

send a request but returns a object like array or map with the parsed response.

The request must a valid json-string with method and params

public `Object` sendobject(*String* request);

arguments:

`String`	**request**

sendRPC

send a RPC request by only passing the method and params.

It will create the raw request from it and return the result.

public `String` sendRPC(*String* method, *Object[]* params);

arguments:

`String`	**method**
`Object[]`	**params**

sendRPCasObject

public `Object` sendRPCasObject(*String* method, *Object[]* params, *boolean* useEnsResolver);

arguments:

`String`	**method**
`Object[]`	**params**
`boolean`	**useEnsResolver**

sendRPCasObject

send a RPC request by only passing the method and params.

It will create the raw request from it and return the result.

public `Object` sendRPCasObject(*String* method, *Object[]* params);

arguments:

`String`	**method**
`Object[]`	**params**

cacheClear

clears the cache.

public `boolean` cacheClear();

nodeList

restrieves the node list

public *IN3Node[]* nodeList();

sign

request for a signature of an already verified hash.

public *SignedBlockHash[]* sign(*BlockID[]* blocks, *String[]* dataNodeAdresses);

arguments:

BlockID[]	**blocks**
`String[]`	**dataNodeAdresses**

forChain

create a Incubed client using the chain-config.

if chainId is Chain.MULTICHAIN, the client can later be switched between different chains, for all other chains, it will be initialized only with the chainspec for this one chain (safes memory)

public static *IN3* forChain(*long* chainId);

arguments:

`long`	**chainId**

getVersion

returns the current incubed version.

> public static `native String` getVersion();

main

> public static `void` main(*`String[]`* args);

arguments:

`String[]`	**args**

13.3.4 class IN3DefaultTransport

handle

> public `byte[][]` handle(*`String[]`* urls, *`byte[]`* payload);

arguments:

`String[]`	**urls**
`byte[]`	**payload**

13.3.5 class IN3Node

getUrl

> public `String` getUrl();

getAddress

> public `String` getAddress();

getIndex

> public `int` getIndex();

getDeposit

> public `String` getDeposit();

getProps

public `long` getProps();

getTimeout

public `int` getTimeout();

getRegisterTime

public `int` getRegisterTime();

getWeight

public `int` getWeight();

13.3.6 class IN3Props

IN3Props

public IN3Props();

setDataNodes

public `void` setDataNodes(*`String[]`* adresses);

arguments:

`String[]`	**adresses**

setSignerNodes

public `void` setSignerNodes(*`String[]`* adresses);

arguments:

`String[]`	**adresses**

toString

public `String` toString();

toJSON

public `String` toJSON();

13.3.7 class Loader

loadLibrary

public static `void` loadLibrary();

13.3.8 class NodeList

getNodes

returns an array of IN3Node

public *IN3Node[]* getNodes();

13.3.9 class NodeProps

NODE_PROP_PROOF

Type: static `final long`

NODE_PROP_MULTICHAIN

Type: static `final long`

NODE_PROP_ARCHIVE

Type: static `final long`

NODE_PROP_HTTP

Type: static `final long`

NODE_PROP_BINARY

Type: static `final long`

NODE_PROP_ONION

Type: static `final long`

NODE_PROP_STATS

Type: static `final long`

13.3.10 class SignedBlockHash

getBlockHash

public `String` getBlockHash();

getBlock

public `long` getBlock();

getR

public `String` getR();

getS

public `String` getS();

getV

public `long` getV();

getMsgHash

public `String` getMsgHash();

13.3.11 enum Proof

The Proof type indicating how much proof is required.

The enum type contains the following values:

none	0	No Verification.
standard	1	Standard Verification of the important properties.
full	2	Full Verification including even uncles wich leads to higher payload.

13.3.12 interface IN3Transport

handle

public `byte[][]` handle(*`String[]`* urls, *`byte[]`* payload);

arguments:

`String[]`	**urls**
`byte[]`	**payload**

13.4 Package in3.btc

13.4.1 class API

API for handling BitCoin data.

Use it when connected to Chain.BTC.

API

creates a btc.API using the given incubed instance.

public API(*IN3* in3);

arguments:

IN3	**in3**

getTransaction

Retrieves the transaction and returns the data as json.

public `Transaction` getTransaction(`String` txid);

arguments:

`String`	**txid**

getTransactionBytes

Retrieves the serialized transaction (bytes).

public `byte[]` getTransactionBytes(`String` txid);

arguments:

`String`	**txid**

getBlockHeader

Retrieves the blockheader.

public `BlockHeader` getBlockHeader(`String` blockHash);

arguments:

`String`	**blockHash**

getBlockHeaderBytes

Retrieves the byte array representing teh serialized blockheader data.

public `byte[]` getBlockHeaderBytes(*String* blockHash);

arguments:

`String`	**blockHash**

getBlockWithTxData

Retrieves the block including the full transaction data.

Use Api::GetBlockWithTxIds" for only the transaction ids.

public *Block* getBlockWithTxData(*String* blockHash);

arguments:

`String`	**blockHash**

getBlockWithTxIds

Retrieves the block including only transaction ids.

Use Api::GetBlockWithTxData for the full transaction data.

public *Block* getBlockWithTxIds(*String* blockHash);

arguments:

`String`	**blockHash**

getBlockBytes

Retrieves the serialized block in bytes.

public `byte[]` getBlockBytes(*String* blockHash);

arguments:

`String`	**blockHash**

13.4.2 class Block

A Block.

getTransactions

Transactions or Transaction of a block.

public *Transaction[]* getTransactions();

getTransactionHashes

Transactions or Transaction ids of a block.

public `String[]` getTransactionHashes();

getSize

Size of this block in bytes.

public `long` getSize();

getWeight

Weight of this block in bytes.

public `long` getWeight();

13.4.3 class BlockHeader

A Block header.

getHash

The hash of the blockheader.

public `String` getHash();

getConfirmations

Number of confirmations or blocks mined on top of the containing block.

public `long` getConfirmations();

getHeight

Block number.

public `long` getHeight();

getVersion

Used version.

public `long` getVersion();

getVersionHex

Version as hex.

public `String` getVersionHex();

getMerkleroot

Merkle root of the trie of all transactions in the block.

public `String` getMerkleroot();

getTime

Unix timestamp in seconds since 1970.

public `long` getTime();

getMediantime

Unix timestamp in seconds since 1970.

public `long` getMediantime();

getNonce

Nonce-field of the block.

public `long` getNonce();

getBits

Bits (target) for the block as hex.

public `String` getBits();

getDifficulty

Difficulty of the block.

public `float` getDifficulty();

getChainwork

Total amount of work since genesis.

public `String` getChainwork();

getNTx

Number of transactions in the block.

public `long` getNTx();

getPreviousblockhash

Hash of the parent blockheader.

public `String` getPreviousblockhash();

getNextblockhash

Hash of the next blockheader.

public `String` getNextblockhash();

13.4.4 class ScriptPubKey

Script on a transaction output.

getAsm

The hash of the blockheader.

public `String` getAsm();

getHex

The raw hex data.

public `String` getHex();

getReqSigs

The required sigs.

public `long` getReqSigs();

getType

The type e.g.

: pubkeyhash.

public `String` getType();

getAddresses

List of addresses.

public `String[]` getAddresses();

13.4.5 class ScriptSig

Script on a transaction input.

ScriptSig

public ScriptSig(*JSON* data);

arguments:

JSON	**data**

getAsm

The asm data.

public `String` getAsm();

getHex

The raw hex data.

public `String` getHex();

13.4.6 class Transaction

A BitCoin Transaction.

asTransaction

public static *Transaction* asTransaction(*Object* o);

arguments:

`Object`	**o**

asTransactions

public static *Transaction[]* asTransactions(*Object* o);

arguments:

`Object`	**o**

getTxid

Transaction Id.

public `String` getTxid();

getHash

The transaction hash (differs from txid for witness transactions).

public `String` getHash();

getVersion

The version.

public `long` getVersion();

getSize

The serialized transaction size.

public `long` getSize();

getVsize

The virtual transaction size (differs from size for witness transactions).

public `long` getVsize();

getWeight

The transactions weight (between vsize4-3 and vsize4).

public `long` getWeight();

getLocktime

The locktime.

public `long` getLocktime();

getHex

The hex representation of raw data.

public `String` getHex();

getBlockhash

The block hash of the block containing this transaction.

public `String` getBlockhash();

getConfirmations

The confirmations.

public `long` getConfirmations();

getTime

The transaction time in seconds since epoch (Jan 1 1970 GMT).

public `long` getTime();

getBlocktime

The block time in seconds since epoch (Jan 1 1970 GMT).

public `long` getBlocktime();

getVin

The transaction inputs.

public *`TransactionInput[]`* getVin();

getVout

The transaction outputs.

public *`TransactionOutput[]`* getVout();

13.4.7 class TransactionInput

Input of a transaction.

getTxid

The transaction id.

public `String` getTxid();

getYout

The index of the transactionoutput.

public `long` getYout();

getScriptSig

The script.

public *`ScriptSig`* getScriptSig();

getTxinwitness

Hex-encoded witness data (if any).

public `String[]` getTxinwitness();

getSequence

The script sequence number.

public `long` getSequence();

13.4.8 class TransactionOutput

A BitCoin Transaction.

TransactionOutput

public TransactionOutput(*JSON* data);

arguments:

JSON	**data**

getValue

The value in bitcoins.

public `float` getValue();

getN

The index in the transaction.

public `long` getN();

getScriptPubKey

The script of the transaction.

public *ScriptPubKey* getScriptPubKey();

13.5 Package in3.config

13.5.1 class ChainConfiguration

Part of the configuration hierarchy for IN3 Client.

Holds the configuration a node group in a particular Chain.

nodesConfig

Type: *NodeConfigurationArrayList< , >*

ChainConfiguration

public ChainConfiguration(*long* chain, *ClientConfiguration* config);

arguments:

`long`	**chain**
ClientConfiguration	**config**

getChain

public `long` getChain();

isNeedsUpdate

public `Boolean` isNeedsUpdate();

setNeedsUpdate

public `void` setNeedsUpdate(*`boolean`* needsUpdate);

arguments:

`boolean`	**needsUpdate**

getContract

public `String` getContract();

setContract

public `void` setContract(*`String`* contract);

arguments:

`String`	**contract**

getRegistryId

public `String` getRegistryId();

setRegistryId

public `void` setRegistryId(*`String`* registryId);

arguments:

`String`	**registryId**

getWhiteListContract

public `String` getWhiteListContract();

setWhiteListContract

public `void` setWhiteListContract(*`String`* whiteListContract);

arguments:

`String`	**whiteListContract**

getWhiteList

public `String[]` getWhiteList();

setWhiteList

public `void` setWhiteList(*`String[]`* whiteList);

arguments:

`String[]`	**whiteList**

toJSON

generates a json-string based on the internal data.

public `String` toJSON();

toString

public `String` toString();

13.5.2 class ClientConfiguration

Configuration Object for Incubed Client.

It holds the state for the root of the configuration tree. Should be retrieved from the client instance as IN3::getConfig()

getRequestCount

public `Integer` getRequestCount();

setRequestCount

sets the number of requests send when getting a first answer

public `void` setRequestCount(*`int`* requestCount);

arguments:

`int`	**requestCount**

isAutoUpdateList

public `Boolean` isAutoUpdateList();

setAutoUpdateList

activates the auto update.if true the nodelist will be automaticly updated if the lastBlock is newer

public `void` setAutoUpdateList(*boolean* autoUpdateList);

arguments:

`boolean`	**autoUpdateList**

getProof

public *Proof* getProof();

setProof

sets the type of proof used

public `void` setProof(*Proof* proof);

arguments:

Proof	**proof**

getMaxAttempts

public `Integer` getMaxAttempts();

setMaxAttempts

sets the max number of attempts before giving up

public `void` setMaxAttempts(*int* maxAttempts);

arguments:

`int`	**maxAttempts**

getSignatureCount

public `Integer` getSignatureCount();

setSignatureCount

sets the number of signatures used to proof the blockhash.

public `void` setSignatureCount(*`int`* signatureCount);

arguments:

`int`	**signatureCount**

isStats

public `Boolean` isStats();

setStats

if true (default) the request will be counted as part of the regular stats, if not they are not shown as part of the dashboard.

public `void` setStats(*`boolean`* stats);

arguments:

`boolean`	**stats**

getFinality

public `Integer` getFinality();

setFinality

sets the number of signatures in percent required for the request

public `void` setFinality(*`int`* finality);

arguments:

`int`	**finality**

isIncludeCode

public `Boolean` isIncludeCode();

setIncludeCode

public `void` setIncludeCode(*`boolean`* includeCode);

arguments:

`boolean`	**includeCode**

isBootWeights

public `Boolean` isBootWeights();

setBootWeights

if true, the first request (updating the nodelist) will also fetch the current health status and use it for blacklisting unhealthy nodes.

This is used only if no nodelist is availabkle from cache.

public `void` setBootWeights(*`boolean`* value);

arguments:

`boolean`	**value**

isKeepIn3

public `Boolean` isKeepIn3();

setKeepIn3

public `void` setKeepIn3(*`boolean`* keepIn3);

arguments:

`boolean`	**keepIn3**

isUseHttp

public `Boolean` isUseHttp();

setUseHttp

public `void` setUseHttp(*`boolean`* useHttp);

arguments:

`boolean`	**useHttp**

getTimeout

public `Long` getTimeout();

setTimeout

specifies the number of milliseconds before the request times out.

increasing may be helpful if the device uses a slow connection.

public `void` setTimeout(*long* timeout);

arguments:

`long`	**timeout**

getMinDeposit

public `Long` getMinDeposit();

setMinDeposit

sets min stake of the server.

Only nodes owning at least this amount will be chosen.

public `void` setMinDeposit(*long* minDeposit);

arguments:

`long`	**minDeposit**

getNodeProps

public `Long` getNodeProps();

setNodeProps

public `void` setNodeProps(*long* nodeProps);

arguments:

`long`	**nodeProps**

getNodeLimit

public `Long` getNodeLimit();

setNodeLimit

sets the limit of nodes to store in the client.

public `void` setNodeLimit(*long* nodeLimit);

arguments:

long	**nodeLimit**

getReplaceLatestBlock

public `Integer` getReplaceLatestBlock();

setReplaceLatestBlock

replaces the *latest* with blockNumber- specified value

public `void` setReplaceLatestBlock(*int* replaceLatestBlock);

arguments:

int	**replaceLatestBlock**

getRpc

public `String` getRpc();

setRpc

setup an custom rpc source for requests by setting Chain to local and proof to none

public `void` setRpc(*String* rpc);

arguments:

String	**rpc**

getNodesConfig

public *ChainConfigurationHashMap< Long, , >* getNodesConfig();

setChainsConfig

public `void` setChainsConfig(*HashMap*< Long, *ChainConfiguration* >);

arguments:

ChainConfigurationHashMap< Long, , >	**chainsConfig**

markAsSynced

public `void` markAsSynced();

isSynced

public `boolean` isSynced();

toString

public `String` toString();

toJSON

generates a json-string based on the internal data.

public `String` toJSON();

13.5.3 class NodeConfiguration

Configuration Object for Incubed Client.

It represents the node of a nodelist.

NodeConfiguration

public NodeConfiguration(*ChainConfiguration* config);

arguments:

ChainConfiguration	**config**

getUrl

public `String` getUrl();

setUrl

public `void` setUrl(*String* url);

arguments:

`String`	**url**

getProps

public `long` getProps();

setProps

public `void` setProps(*long* props);

arguments:

`long`	**props**

getAddress

public `String` getAddress();

setAddress

public `void` setAddress(*String* address);

arguments:

`String`	**address**

toString

public `String` toString();

13.5.4 interface Configuration

an Interface class, which is able to generate a JSON-String.

toJSON

generates a json-string based on the internal data.

public `String` toJSON();

13.6 Package in3.eth1

13.6.1 class API

a Wrapper for the incubed client offering Type-safe Access and additional helper functions.

API

creates an eth1.API using the given incubed instance.

public API(*IN3* in3);

arguments:

IN3	**in3**

getBlockByNumber

finds the Block as specified by the number.

use `Block.LATEST` for getting the lastest block.

> public *Block* getBlockByNumber(*long* block, *boolean* includeTransactions);

arguments:

`long`	**block**	
`boolean`	**includeTransactions**	< the Blocknumber < if true all Transactions will be includes, if not only the transactionhashes

getBlockByHash

Returns information about a block by hash.

> public *Block* getBlockByHash(*String* blockHash, *boolean* includeTransactions);

arguments:

`String`	**blockHash**	
`boolean`	**includeTransactions**	< the Blocknumber < if true all Transactions will be includes, if not only the transactionhashes

getBlockNumber

the current BlockNumber.

> public `long` getBlockNumber();

getGasPrice

the current Gas Price.

> public `long` getGasPrice();

getChainId

Returns the EIP155 chain ID used for transaction signing at the current best block.

Null is returned if not available.

> public `String` getChainId();

call

calls a function of a smart contract and returns the result.

> public `Object` call(*TransactionRequest* request, *long* block);

arguments:

TransactionRequest	**request**	
`long`	**block**	< the transaction to call. < the Block used to for the state.

returns: `Object` : the decoded result. if only one return value is expected the Object will be returned, if not an array of objects will be the result.

estimateGas

Makes a call or transaction, which won't be added to the blockchain and returns the used gas, which can be used for estimating the used gas.

public `long` estimateGas(*TransactionRequest* request, *long* block);

arguments:

TransactionRequest	**request**	
`long`	**block**	< the transaction to call. < the Block used to for the state.

returns: `long` : the gas required to call the function.

getBalance

Returns the balance of the account of given address in wei.

public `BigInteger` getBalance(*String* address, *long* block);

arguments:

`String`	**address**
`long`	**block**

getCode

Returns code at a given address.

public `String` getCode(*String* address, *long* block);

arguments:

`String`	**address**
`long`	**block**

getStorageAt

Returns the value from a storage position at a given address.

public `String` getStorageAt(*String* address, *BigInteger* position, *long* block);

arguments:

`String`	**address**
`BigInteger`	**position**
`long`	**block**

getBlockTransactionCountByHash

Returns the number of transactions in a block from a block matching the given block hash.

public `long` getBlockTransactionCountByHash(*String* blockHash);

arguments:

`String`	**blockHash**

getBlockTransactionCountByNumber

Returns the number of transactions in a block from a block matching the given block number.

public `long` getBlockTransactionCountByNumber(*long* block);

arguments:

`long`	**block**

getFilterChangesFromLogs

Polling method for a filter, which returns an array of logs which occurred since last poll.

public *Log[]* getFilterChangesFromLogs(*long* id);

arguments:

`long`	**id**

getFilterChangesFromBlocks

Polling method for a filter, which returns an array of logs which occurred since last poll.

public `String[]` getFilterChangesFromBlocks(*long* id);

arguments:

`long`	**id**

getFilterLogs

Polling method for a filter, which returns an array of logs which occurred since last poll.

public *Log[]* getFilterLogs(*long* id);

arguments:

`long`	**id**

getLogs

Polling method for a filter, which returns an array of logs which occurred since last poll.

> public *Log[]* getLogs(*LogFilter* filter);

arguments:

LogFilter	**filter**

getTransactionByBlockHashAndIndex

Returns information about a transaction by block hash and transaction index position.

> public *Transaction* getTransactionByBlockHashAndIndex(*String* blockHash, *int* index);

arguments:

`String`	**blockHash**
`int`	**index**

getTransactionByBlockNumberAndIndex

Returns information about a transaction by block number and transaction index position.

> public *Transaction* getTransactionByBlockNumberAndIndex(*long* block, *int* index);

arguments:

`long`	**block**
`int`	**index**

getTransactionByHash

Returns the information about a transaction requested by transaction hash.

> public *Transaction* getTransactionByHash(*String* transactionHash);

arguments:

`String`	**transactionHash**

getTransactionCount

Returns the number of transactions sent from an address.

> public `BigInteger` getTransactionCount(*String* address, *long* block);

arguments:

`String`	**address**
`long`	**block**

getTransactionReceipt

Returns the number of transactions sent from an address.

public *TransactionReceipt* getTransactionReceipt(*String* transactionHash);

arguments:

`String`	**transactionHash**

getUncleByBlockNumberAndIndex

Returns information about a uncle of a block number and uncle index position.

Note: An uncle doesn't contain individual transactions.

public *Block* getUncleByBlockNumberAndIndex(*long* block, *int* pos);

arguments:

`long`	**block**
`int`	**pos**

getUncleCountByBlockHash

Returns the number of uncles in a block from a block matching the given block hash.

public `long` getUncleCountByBlockHash(*String* block);

arguments:

`String`	**block**

getUncleCountByBlockNumber

Returns the number of uncles in a block from a block matching the given block hash.

public `long` getUncleCountByBlockNumber(*long* block);

arguments:

`long`	**block**

newBlockFilter

Creates a filter in the node, to notify when a new block arrives.

To check if the state has changed, call eth_getFilterChanges.

> public `long` newBlockFilter();

newLogFilter

Creates a filter object, based on filter options, to notify when the state changes (logs).

To check if the state has changed, call eth_getFilterChanges.

A note on specifying topic filters: Topics are order-dependent. A transaction with a log with topics [A, B] will be matched by the following topic filters:

[] "anything" [A] "A in first position (and anything after)" [null, B] "anything in first position AND B in second position (and anything after)" [A, B] "A in first position AND B in second position (and anything after)" [[A, B], [A, B]] "(A OR B) in first position AND (A OR B) in second position (and anything after)"

> public `long` newLogFilter(*LogFilter* filter);

arguments:

LogFilter	**filter**

uninstallFilter

uninstall filter.

> public `boolean` uninstallFilter(*long* filter);

arguments:

`long`	**filter**

sendRawTransaction

Creates new message call transaction or a contract creation for signed transactions.

> public `String` sendRawTransaction(*String* data);

arguments:

`String`	**data**

returns: `String` : transactionHash

abiEncode

encodes the arguments as described in the method signature using ABI-Encoding.

> public `String` abiEncode(*String* signature, *String[]* params);

arguments:

`String`	**signature**
`String[]`	**params**

abiDecode

decodes the data based on the signature.

public `String[]` abiDecode(*String* signature, *String* encoded);

arguments:

`String`	**signature**
`String`	**encoded**

checksumAddress

converts the given address to a checksum address.

public `String` checksumAddress(*String* address);

arguments:

`String`	**address**

checksumAddress

converts the given address to a checksum address.

Second parameter includes the chainId.

public `String` checksumAddress(*String* address, *Boolean* useChainId);

arguments:

`String`	**address**
`Boolean`	**useChainId**

ens

resolve ens-name.

public `String` ens(*String* name);

arguments:

`String`	**name**

ens

resolve ens-name.

Second parameter especifies if it is an address, owner, resolver or hash.

public `String` ens(*`String`* name, *`ENSMethod`* type);

arguments:

`String`	**name**
ENSMethod	**type**

sendTransaction

sends a Transaction as described by the TransactionRequest.

This will require a signer to be set in order to sign the transaction.

public `String` sendTransaction(*`TransactionRequest`* tx);

arguments:

TransactionRequest	**tx**

call

the current Gas Price.

public `Object` call(*`String`* to, *`String`* function, *`Object...`* params);

arguments:

`String`	**to**
`String`	**function**
`Object...`	**params**

returns: `Object` : the decoded result. if only one return value is expected the Object will be returned, if not an array of objects will be the result.

13.6.2 class Block

represents a Block in ethereum.

LATEST

The latest Block Number.

Type: static `long`

EARLIEST

The Genesis Block.

Type: static `long`

getTotalDifficulty

returns the total Difficulty as a sum of all difficulties starting from genesis.

public `BigInteger` getTotalDifficulty();

getGasLimit

the gas limit of the block.

public `BigInteger` getGasLimit();

getExtraData

the extra data of the block.

public `String` getExtraData();

getDifficulty

the difficulty of the block.

public `BigInteger` getDifficulty();

getAuthor

the author or miner of the block.

public `String` getAuthor();

getTransactionsRoot

the roothash of the merkletree containing all transaction of the block.

public `String` getTransactionsRoot();

getTransactionReceiptsRoot

the roothash of the merkletree containing all transaction receipts of the block.

public `String` getTransactionReceiptsRoot();

getStateRoot

the roothash of the merkletree containing the complete state.

public `String` getStateRoot();

getTransactionHashes

the transaction hashes of the transactions in the block.

public `String[]` getTransactionHashes();

getTransactions

the transactions of the block.

public *`Transaction[]`* getTransactions();

getTimeStamp

the unix timestamp in seconds since 1970.

public `long` getTimeStamp();

getSha3Uncles

the roothash of the merkletree containing all uncles of the block.

public `String` getSha3Uncles();

getSize

the size of the block.

public `long` getSize();

getSealFields

the seal fields used for proof of authority.

public `String[]` getSealFields();

getHash

the block hash of the of the header.

public `String` getHash();

getLogsBloom

the bloom filter of the block.

public `String` getLogsBloom();

getMixHash

the mix hash of the block.

(only valid of proof of work)

public `String` getMixHash();

getNonce

the mix hash of the block.

(only valid of proof of work)

public `String` getNonce();

getNumber

the block number

public `long` getNumber();

getParentHash

the hash of the parent-block.

public `String` getParentHash();

getUncles

returns the blockhashes of all uncles-blocks.

public `String[]` getUncles();

hashCode

public `int` hashCode();

equals

public `boolean` equals(*`Object`* obj);

arguments:

`Object`	**obj**

13.6.3 class Log

a log entry of a transaction receipt.

isRemoved

true when the log was removed, due to a chain reorganization.

false if its a valid log.

public `boolean` isRemoved();

getLogIndex

integer of the log index position in the block.

null when its pending log.

public `int` getLogIndex();

gettTansactionIndex

integer of the transactions index position log was created from.

null when its pending log.

public `int` gettTansactionIndex();

getTransactionHash

Hash, 32 Bytes - hash of the transactions this log was created from.

null when its pending log.

public `String` getTransactionHash();

getBlockHash

Hash, 32 Bytes - hash of the block where this log was in.

null when its pending. null when its pending log.

public `String` getBlockHash();

getBlockNumber

the block number where this log was in.

null when its pending. null when its pending log.

public `long` getBlockNumber();

getAddress

20 Bytes - address from which this log originated.

public `String` getAddress();

getTopics

Array of 0 to 4 32 Bytes DATA of indexed log arguments.

(In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier.)

public `String[]` getTopics();

13.6.4 class LogFilter

Log configuration for search logs.

getFromBlock

public `long` getFromBlock();

setFromBlock

public `void` setFromBlock(*`long`* fromBlock);

arguments:

`long`	**fromBlock**

getToBlock

public `long` getToBlock();

setToBlock

public `void` setToBlock(*`long`* toBlock);

arguments:

`long`	**toBlock**

getAddress

public `String` getAddress();

setAddress

public `void` setAddress(*`String`* address);

arguments:

`String`	**address**

getTopics

public `Object[]` getTopics();

setTopics

public `void` setTopics(*`Object[]`* topics);

arguments:

`Object[]`	**topics**

getLimit

public `int` getLimit();

setLimit

public `void` setLimit(*`int`* limit);

arguments:

`int`	**limit**

toString

creates a JSON-String.

public `String` toString();

13.6.5 class SimpleWallet

a simple Implementation for holding private keys to sing data or transactions.

addRawKey

adds a key to the wallet and returns its public address.

public `String` addRawKey(*`String`* data);

arguments:

`String`	**data**

addKeyStore

adds a key to the wallet and returns its public address.

public `String` addKeyStore(*String* jsonData, *String* passphrase);

arguments:

`String`	**jsonData**
`String`	**passphrase**

prepareTransaction

optiional method which allows to change the transaction-data before sending it.

This can be used for redirecting it through a multisig.

public *TransactionRequest* prepareTransaction(*IN3* in3, *TransactionRequest* tx);

arguments:

IN3	**in3**
TransactionRequest	**tx**

canSign

returns true if the account is supported (or unlocked)

public `boolean` canSign(*String* address);

arguments:

`String`	**address**

sign

signing of the raw data.

public `String` sign(*String* data, *String* address);

arguments:

`String`	**data**
`String`	**address**

13.6.6 class Transaction

represents a Transaction in ethereum.

asTransaction

public static `Transaction` asTransaction(`Object` o);

arguments:

`Object`	**o**

getBlockHash

the blockhash of the block containing this transaction.

public `String` getBlockHash();

getBlockNumber

the block number of the block containing this transaction.

public `long` getBlockNumber();

getChainId

the chainId of this transaction.

public `String` getChainId();

getCreatedContractAddress

the address of the deployed contract (if successfull)

public `String` getCreatedContractAddress();

getFrom

the address of the sender.

public `String` getFrom();

getHash

the Transaction hash.

public `String` getHash();

getData

the Transaction data or input data.

public `String` getData();

getNonce

the nonce used in the transaction.

public `long` getNonce();

getPublicKey

the public key of the sender.

public `String` getPublicKey();

getValue

the value send in wei.

public `BigInteger` getValue();

getRaw

the raw transaction as rlp encoded data.

public `String` getRaw();

getTo

the address of the receipient or contract.

public `String` getTo();

getSignature

the signature of the sender - a array of the [r, s, v]

public `String[]` getSignature();

getGasPrice

the gas price provided by the sender.

public `long` getGasPrice();

getGas

the gas provided by the sender.

public `long` getGas();

13.6.7 class TransactionReceipt

represents a Transaction receipt in ethereum.

getBlockHash

the blockhash of the block containing this transaction.

> public `String` getBlockHash();

getBlockNumber

the block number of the block containing this transaction.

> public `long` getBlockNumber();

getCreatedContractAddress

the address of the deployed contract (if successfull)

> public `String` getCreatedContractAddress();

getFrom

the address of the sender.

> public `String` getFrom();

getTransactionHash

the Transaction hash.

> public `String` getTransactionHash();

getTransactionIndex

the Transaction index.

> public `int` getTransactionIndex();

getTo

20 Bytes - The address of the receiver.

null when it's a contract creation transaction.

> public `String` getTo();

getGasUsed

The amount of gas used by this specific transaction alone.

> public `long` getGasUsed();

getLogs

Array of log objects, which this transaction generated.

public `Log[]` getLogs();

getLogsBloom

256 Bytes - A bloom filter of logs/events generated by contracts during transaction execution.

Used to efficiently rule out transactions without expected logs

public `String` getLogsBloom();

getRoot

32 Bytes - Merkle root of the state trie after the transaction has been executed (optional after Byzantium hard fork EIP609).

public `String` getRoot();

getStatus

success of a Transaction.

true indicates transaction failure , false indicates transaction success. Set for blocks mined after Byzantium hard fork EIP609, null before.

public `boolean` getStatus();

13.6.8 class TransactionRequest

represents a Transaction Request which should be send or called.

getFrom

public `String` getFrom();

setFrom

public `void` setFrom(*`String`* from);

arguments:

`String`	**from**

getTo

public `String` getTo();

setTo

public `void` setTo(*`String`* to);

arguments:

`String`	**to**

getValue

public `BigInteger` getValue();

setValue

public `void` setValue(*`BigInteger`* value);

arguments:

`BigInteger`	**value**

getNonce

public `long` getNonce();

setNonce

public `void` setNonce(*`long`* nonce);

arguments:

`long`	**nonce**

getGas

public `long` getGas();

setGas

public `void` setGas(*`long`* gas);

arguments:

`long`	**gas**

getGasPrice

public `long` getGasPrice();

setGasPrice

public `void` setGasPrice(*`long`* gasPrice);

arguments:

`long`	**gasPrice**

getFunction

public `String` getFunction();

setFunction

public `void` setFunction(*`String`* function);

arguments:

`String`	**function**

getParams

public `Object[]` getParams();

setParams

public `void` setParams(*`Object[]`* params);

arguments:

`Object[]`	**params**

setData

public `void` setData(*`String`* data);

arguments:

`String`	**data**

getData

creates the data based on the function/params values.

public `String` getData();

getTransactionJson

public `String` getTransactionJson();

getResult

public `Object` getResult(*String* data);

arguments:

`String`	**data**

13.6.9 enum ENSMethod

The enum type contains the following values:

addr	0
resolver	1
hash	2
owner	3

13.7 Package in3.ipfs

13.7.1 class API

API for ipfs custom methods.

To be used along with "Chain.IPFS" on in3 instance.

API

creates a ipfs.API using the given incubed instance.

public API(*IN3* in3);

arguments:

IN3	**in3**

get

Returns the content associated with specified multihash on success OR NULL on error.

public `byte[]` get(*String* multihash);

arguments:

`String`	**multihash**

put

Returns the IPFS multihash of stored content on success OR NULL on error.

public `String` put(*String* content);

arguments:

`String`	**content**

put

Returns the IPFS multihash of stored content on success OR NULL on error.

public `String` put(*byte[]* content);

arguments:

`byte[]`	**content**

13.8 Package in3.ipfs.API

13.8.1 enum Encoding

The enum type contains the following values:

base64	0
hex	1
utf8	2

13.9 Package in3.utils

13.9.1 class Account

Pojo that represents the result of an ecrecover operation (see: Crypto class).

getAddress

address from ecrecover operation.

public `String` getAddress();

getPublicKey

public key from ecrecover operation.

public `String` getPublicKey();

13.9.2 class Crypto

a Wrapper for crypto-related helper functions.

Crypto

public Crypto(*IN3* in3);

arguments:

IN3	**in3**

signData

returns a signature given a message and a key.

public *Signature* signData(*String* msg, *String* key, *SignatureType* sigType);

arguments:

`String`	**msg**
`String`	**key**
SignatureType	**sigType**

decryptKey

public `String` decryptKey(*String* key, *String* passphrase);

arguments:

`String`	**key**
`String`	**passphrase**

pk2address

extracts the public address from a private key.

public `String` pk2address(*String* key);

arguments:

`String`	**key**

pk2public

extracts the public key from a private key.

public `String` pk2public(*String* key);

arguments:

`String`	**key**

ecrecover

extracts the address and public key from a signature.

public *Account* ecrecover(*String* msg, *String* sig);

arguments:

String	**msg**
String	**sig**

ecrecover

extracts the address and public key from a signature.

public *Account* ecrecover(*String* msg, *String* sig, *SignatureType* sigType);

arguments:

String	**msg**
String	**sig**
SignatureType	**sigType**

signData

returns a signature given a message and a key.

public *Signature* signData(*String* msg, *String* key);

arguments:

String	**msg**
String	**key**

13.9.3 class JSON

internal helper tool to represent a JSON-Object.

Since the internal representation of JSON in incubed uses hashes instead of name, the getter will creates these hashes.

get

gets the property

public Object get(*String* prop);

arguments:

String	**prop**	the name of the property.

returns: Object : the raw object.

put

adds values.

This function will be called from the JNI-Iterface.

Internal use only!

public `void` put(*int* key, *Object* val);

arguments:

`int`	**key**	the hash of the key
`Object`	**val**	the value object

getLong

returns the property as long

public `long` getLong(*String* key);

arguments:

`String`	**key**	the propertyName

returns: `long` : the long value

getBigInteger

returns the property as BigInteger

public `BigInteger` getBigInteger(*String* key);

arguments:

`String`	**key**	the propertyName

returns: `BigInteger` : the BigInteger value

getStringArray

returns the property as StringArray

public `String[]` getStringArray(*String* key);

arguments:

`String`	**key**	the propertyName

returns: `String[]` : the array or null

getString

returns the property as String or in case of a number as hexstring.

public `String` getString(*`String`* key);

arguments:

`String`	**key**	the propertyName

returns: `String` : the hexstring

toString

public `String` toString();

hashCode

public `int` hashCode();

equals

public `boolean` equals(*`Object`* obj);

arguments:

`Object`	**obj**

asStringArray

casts the object to a String[]

public static `String[]` asStringArray(*`Object`* o);

arguments:

`Object`	**o**

asBigInteger

public static `BigInteger` asBigInteger(*`Object`* o);

arguments:

`Object`	**o**

asLong

public static `long` asLong(*Object* o);

arguments:

`Object`	**o**

asInt

public static `int` asInt(*Object* o);

arguments:

`Object`	**o**

asString

public static `String` asString(*Object* o);

arguments:

`Object`	**o**

toJson

public static `String` toJson(*Object* ob);

arguments:

`Object`	**ob**

appendKey

public static `void` appendKey(*StringBuilder* sb, *String* key, *Object* value);

arguments:

`StringBuilder`	**sb**
`String`	**key**
`Object`	**value**

13.9.4 class Signature

getMessage

public `String` getMessage();

getMessageHash

public `String` getMessageHash();

getSignature

public `String` getSignature();

getR

public `String` getR();

getS

public `String` getS();

getV

public `long` getV();

13.9.5 class TempStorageProvider

a simple Storage Provider storing the cache in the temp-folder.

getItem

returns a item from cache ()

public `byte[]` getItem(*`String`* key);

arguments:

`String`	**key**	the key for the item

returns: `byte[]` : the bytes or null if not found.

setItem

stores a item in the cache.

public `void` setItem(*`String`* key, *`byte[]`* content);

arguments:

`String`	**key**	the key for the item
`byte[]`	**content**	the value to store

clear

clear the cache.

public `boolean` clear();

13.9.6 enum SignatureType

The enum type contains the following values:

eth_sign	0
raw	1
hash	2

13.9.7 interface Signer

a Interface responsible for signing data or transactions.

prepareTransaction

optiional method which allows to change the transaction-data before sending it.

This can be used for redirecting it through a multisig.

public *TransactionRequest* prepareTransaction(*IN3* in3, *TransactionRequest* tx);

arguments:

IN3	**in3**
TransactionRequest	**tx**

canSign

returns true if the account is supported (or unlocked)

public `boolean` canSign(*String* address);

arguments:

`String`	**address**

sign

signing of the raw data.

public `String` sign(*String* data, *String* address);

arguments:

`String`	**data**
`String`	**address**

13.9.8 interface StorageProvider

Provider methods to cache data.

These data could be nodelists, contract codes or validator changes.

getItem

returns a item from cache ()

public `byte[]` getItem(*`String`* key);

arguments:

`String`	**key**	the key for the item

returns: `byte[]` : the bytes or null if not found.

setItem

stores a item in the cache.

public `void` setItem(*`String`* key, *`byte[]`* content);

arguments:

`String`	**key**	the key for the item
`byte[]`	**content**	the value to store

clear

clear the cache.

public `boolean` clear();

CHAPTER 14

API Reference Dotnet

Dotnet bindings and library for in3. Go to our readthedocs page for more on usage.

This library is based on the C version of Incubed.

14.1 Runtimes

Since this is built on top of the native library, it is limited to the followin runtimes (RID)

- osx-x64
- linux-x86
- linux-x64
- win-x64
- linux-arm64

For more information, see Rid Catalog.

14.2 Quickstart

14.2.1 Install with nuget

```
dotnet add package Blockchains.In3
```

14.3 Examples

14.3.1 CallSmartContractFunction

source : in3-c/dotnet/Examples/CallSmartContractFunction//CallSmartContractFunction

```
using System;
using System.Numerics;
using System.Threading.Tasks;
using In3;
using In3.Configuration;
using In3.Eth1;
using In3.Utils;

namespace CallSmartContractFunction
{
    public class Program
    {
        public static async Task Main()
        {
            // Set it to mainnet
            IN3 mainnetClient = IN3.ForChain(Chain.Mainnet);
            ClientConfiguration cfg = mainnetClient.Configuration;
            cfg.Proof = Proof.Standard;

            string contractAddress = "0x2736D225f85740f42D17987100dc8d58e9e16252";

            // Create the query transaction
            TransactionRequest serverCountQuery = new TransactionRequest();
            serverCountQuery.To = contractAddress;

            // Define the function and the parameters to query the total in3 servers
            serverCountQuery.Function = "totalServers():uint256";
            serverCountQuery.Params = new object[0];

            string[] serverCountResult = (string[])await mainnetClient.Eth1.
↪Call(serverCountQuery, BlockParameter.Latest);
            BigInteger servers = DataTypeConverter.
↪HexStringToBigint(serverCountResult[0]);

            for (int i = 0; i < servers; i++)
            {
                TransactionRequest serverDetailQuery = new TransactionRequest();
                serverDetailQuery.To = contractAddress;

                // Define the function and the parameters to query the in3 servers␣
↪detail
                serverDetailQuery.Function = "servers(uint256):(string,address,uint32,
↪uint256,uint256,address)";
                serverDetailQuery.Params = new object[] { i }; // index of the server␣
↪(uint256) as per solidity function signature

                string[] serverDetailResult = (string[])await mainnetClient.Eth1.
↪Call(serverDetailQuery, BlockParameter.Latest);
                Console.Out.WriteLine($"Server url: {serverDetailResult[0]}");
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

14.3.2 ConnectToEthereum

source : in3-c/dotnet/Examples/ConnectToEthereum//ConnectToEthereum

```
using System;
using System.Numerics;
using System.Threading.Tasks;
using In3;

namespace ConnectToEthereum
{
    class Program
    {
        static async Task Main()
        {
            Console.Out.WriteLine("Ethereum Main Network");
            IN3 mainnetClient = IN3.ForChain(Chain.Mainnet);
            BigInteger mainnetLatest = await mainnetClient.Eth1.BlockNumber();
            BigInteger mainnetCurrentGasPrice = await mainnetClient.Eth1.
↪GetGasPrice();
            Console.Out.WriteLine($"Latest Block Number: {mainnetLatest}");
            Console.Out.WriteLine($"Gas Price: {mainnetCurrentGasPrice} Wei");

            Console.Out.WriteLine("Ethereum Kovan Test Network");
            IN3 kovanClient = IN3.ForChain(Chain.Kovan);
            BigInteger kovanLatest = await kovanClient.Eth1.BlockNumber();
            BigInteger kovanCurrentGasPrice = await kovanClient.Eth1.GetGasPrice();
            Console.Out.WriteLine($"Latest Block Number: {kovanLatest}");
            Console.Out.WriteLine($"Gas Price: {kovanCurrentGasPrice} Wei");

            Console.Out.WriteLine("Ethereum Goerli Test Network");
            IN3 goerliClient = IN3.ForChain(Chain.Goerli);
            BigInteger goerliLatest = await goerliClient.Eth1.BlockNumber();
            BigInteger clientCurrentGasPrice = await goerliClient.Eth1.GetGasPrice();
            Console.Out.WriteLine($"Latest Block Number: {goerliLatest}");
            Console.Out.WriteLine($"Gas Price: {clientCurrentGasPrice} Wei");
        }
    }
}
```

14.3.3 EnsResolver

source : in3-c/dotnet/Examples/EnsResolver//EnsResolver

```
using System;
using System.Threading.Tasks;
using In3;

namespace EnsResolver
{
```

(continues on next page)

(continued from previous page)

```
    public class Program
    {
        static async Task Main()
        {
            IN3 in3 = IN3.ForChain(Chain.Mainnet);

            string cryptoKittiesDomain = "cryptokitties.eth";
            string resolver = await in3.Eth1.Ens(cryptoKittiesDomain, ENSParameter.
↪Resolver);
            string owner = await in3.Eth1.Ens(cryptoKittiesDomain, ENSParameter.
↪Owner);

            Console.Out.WriteLine($"The owner of {cryptoKittiesDomain} is {owner},␣
↪resolver is {resolver}.");
        }
    }
}
```

14.3.4 Ipfs

source : in3-c/dotnet/Examples/Ipfs//Ipfs

```
using System;
using System.Text;
using System.Threading.Tasks;
using In3;

namespace Ipfs
{
    class Program
    {
        static async Task Main()
        {
            // Content to be stored
            string toStore = "LOREM_IPSUM";

            // Connect to ipfs.
            IN3 ipfsClient = IN3.ForChain(Chain.Ipfs);

            // Store the hash since it will be needed to fetch the content back.
            string hash = await ipfsClient.Ipfs.Put(toStore);

            //
            byte[] storedBytes = await ipfsClient.Ipfs.Get(hash);
            string storedStging = Encoding.UTF8.GetString(storedBytes, 0, storedBytes.
↪Length);
            Console.Out.WriteLine($"The stored string is: {storedStging}");
        }
    }
}
```

14.3.5 Logs

source : in3-c/dotnet/Examples/Logs//Logs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using In3;
using In3.Eth1;

namespace Logs
{
    class Program
    {
        static async Task Main()
        {
            // Define an upper limit for poll since we dont want our application␣
↪potentially running forever.
            int maxIterations = 500;
            int oneSecond = 1000; // in ms

            // Connect to mainnet.
            IN3 mainnetClient = IN3.ForChain(Chain.Mainnet);

            // Create a filter object pointing, in this case, to an "eventful"␣
↪contract address.
            LogFilter tetherUsFilter = new LogFilter {Address =
↪"0xdAC17F958D2ee523a2206206994597C13D831ec7"};

            // Create the filter to be polled for logs.
            long filterId = await mainnetClient.Eth1.NewLogFilter(tetherUsFilter);

            // Loop to initiate the poll for the logs.
            for (int i = 0; i < maxIterations; i++)
            {
                // Query for the log events since the creation of the filter or the␣
↪previous poll (this method in NOT idempotent as it retrieves a diff).
                Log[] tetherLogs = await mainnetClient.Eth1.
↪GetFilterChangesFromLogs(filterId);
                if (tetherLogs.Length > 0)
                {
                    Console.Out.WriteLine("Logs found: " + tetherLogs.Length);
                    break;
                }

                // Wait before next query.
                Thread.Sleep(oneSecond);
            }
        }
    }
}
```

14.3.6 SendTransaction

source : in3-c/dotnet/Examples/SendTransaction//SendTransaction

```
using System;
using System.Threading;
using System.Threading.Tasks;
using In3;
```

(continues on next page)

(continued from previous page)

```
using In3.Crypto;
using In3.Eth1;

namespace SendTransaction
{
    public class Program
    {
        static async Task Main()
        {
            IN3 goerliClient = IN3.ForChain(Chain.Goerli);

            string myPrivateKey =
↪"0x0829B3C639A3A8F2226C8057F100128D4F7AE8102C92048BA6DE38CF4D3BC6F1";
            string receivingAddress = "0x6FA33809667A99A805b610C49EE2042863b1bb83";

            // Get the wallet, which is the default signer.
            SimpleWallet myAccountWallet = (SimpleWallet)goerliClient.Signer;

            string myAccount = myAccountWallet.AddRawKey(myPrivateKey);

            // Create the transaction request
            TransactionRequest transferWei = new TransactionRequest();
            transferWei.To = receivingAddress;
            transferWei.From = myAccount;
            transferWei.Value = 300;

            // Get the current gas prices
            long currentGasPrice = await goerliClient.Eth1.GetGasPrice();
            transferWei.GasPrice = currentGasPrice;

            long estimatedSpentGas = await goerliClient.Eth1.EstimateGas(transferWei,␣
↪BlockParameter.Latest);
            Console.Out.WriteLine($"Estimated gas to spend: {estimatedSpentGas}");

            string transactionHash = await goerliClient.Eth1.
↪SendTransaction(transferWei);
            Console.Out.WriteLine($"Transaction {transactionHash} sent.");
            Thread.Sleep(30000);

            TransactionReceipt receipt = await goerliClient.Eth1.
↪GetTransactionReceipt(transactionHash);
            Console.Out.WriteLine($"Transaction {transactionHash} mined on block
↪{receipt.BlockNumber}.");
        }
    }
}
```

14.3.7 Build Examples

To setup and run the example projects, simply run on the respective project folder:

```
dotnet run
```

To build all of them, on the solution folder, run:

```
dotnet build
```

14.4 Index

- *Account*
  - *Address*
  - *PublicKey*
- *Api*
- *Api*
- *Api*
- *Api*
  - *GetBlockBytes(blockHash)*
  - *GetBlockHeader(blockHash)*
  - *GetBlockHeaderBytes(blockHash)*
  - *GetBlockWithTxData(blockHash)*
  - *GetBlockWithTxIds(blockHash)*
  - *GetTransaction(txid)*
  - *GetTransactionBytes(txid)*
  - *DecryptKey(pk,passphrase)*
  - *EcRecover(signedData,signature,signatureType)*
  - *Pk2Address(pk)*
  - *Pk2Public(pk)*
  - *Sha3(data)*
  - *SignData(msg,pk,sigType)*
  - *AbiDecode(signature,encodedData)*
  - *AbiEncode(signature,args)*
  - *BlockNumber()*
  - *Call(request,blockNumber)*
  - *ChecksumAddress(address,shouldUseChainId)*
  - *Ens(name,type)*
  - *EstimateGas(request,blockNumber)*
  - *GetBalance(address,blockNumber)*
  - *GetBlockByHash(blockHash,shouldIncludeTransactions)*
  - *GetBlockByNumber(blockNumber,shouldIncludeTransactions)*
  - *GetBlockTransactionCountByHash(blockHash)*
  - *GetBlockTransactionCountByNumber(blockNumber)*

  - *GetItem(key)*
  - *SetItem(key,content)*
- *Log*
  - *Address*
  - *BlockHash*
  - *BlockNumber*
  - *Data*
  - *LogIndex*
  - *Removed*
  - *Topics*
  - *TransactionHash*
  - *TransactionIndex*
  - *Type*
- *LogFilter*
  - *#ctor()*
  - *Address*
  - *BlockHash*
  - *FromBlock*
  - *ToBlock*
  - *Topics*
- *NodeConfiguration*
  - *#ctor(config)*
  - *Address*
  - *Props*
  - *Url*
- *Proof*
  - *Full*
  - *None*
  - *Standard*
- *Props*
  - *NodePropArchive*
  - *NodePropBinary*
  - *NodePropData*
  - *NodePropHttp*
  - *NodePropMinblockheight*
  - *NodePropMultichain*

   - *Params*
   - *To*
   - *Value*
- *Transport*
   - *Handle(url,payload)*

14.4.1 Account `type`

In3.Crypto

Composite entity that holds address and public key. It represents and Ethereum acount. Entity returned from *EcRecover*.

Address `property`

The address.

PublicKey `property`

The public key.

14.4.2 Api `type`

In3.Btc

API for handling BitCoin data. Use it when connected to *Btc*.

14.4.3 Api `type`

In3.Crypto

Class that exposes utility methods for cryptographic utilities. Relies on *IN3* functionality.

14.4.4 Api `type`

In3.Eth1

Module based on Ethereum's api and web3. Works as a general parent for all Ethereum-specific operations.

14.4.5 Api `type`

In3.Ipfs

API for ipfs realted methods. To be used along with *Ipfs* on *IN3*. Ipfs stands for and is a peer-to-peer hypermedia protocol designed to make the web faster, safer, and more open.

GetBlockBytes(blockHash) method

Retrieves the serialized block in bytes.

Returns

The bytes of the block.

Parameters

- System.String **blockHash** - The hash of the Block.

Example

```
byte[] blockBytes = in3.Btc.GetBlockBytes(
↪"00000000000000000064ba7512ecc70cabd7ed17e31c06f2205d5ecdadd6d22");
```

GetBlockHeader(blockHash) method

Retrieves the blockheader.

Returns

The Block header.

Parameters

- System.String **blockHash** - The hash of the Block.

Example

```
BlockHeader header = in3.Btc.GetBlockHeader(
↪"0000000000000000000cd3c5d7638014e78a5fba33be5fa5cb10ef9f03d99e60");
```

GetBlockHeaderBytes(blockHash) method

Retrieves the byte array representing teh serialized blockheader data.

Returns

The Block header in bytes.

Parameters

- System.String **blockHash** - The hash of the Block.

Example

```
byte[] header = in3.Btc.GetBlockHeaderBytes(
↪"000000000000000000cd3c5d7638014e78a5fba33be5fa5cb10ef9f03d99e60");
```

GetBlockWithTxData(blockHash) method

Retrieves the block including the full transaction data. Use *GetBlockWithTxIds* for only the transaction ids.

Returns

The block of type *Block'1*.

Parameters

- System.String **blockHash** - The hash of the Block.

Example

```
Block{Transaction} block = in3.Btc.GetBlockWithTxData(
↪"0000000000000000000064ba7512ecc70cabd7ed17e31c06f2205d5ecdadd6d22");
Transaction t1 = block.Tx[0];
```

GetBlockWithTxIds(blockHash) method

Retrieves the block including only transaction ids. Use *GetBlockWithTxData* for the full transaction data.

Returns

The block of type *Block'1*.

Parameters

- System.String **blockHash** - The hash of the Block.

Example

```
Block{string} block = in3.Btc.GetBlockWithTxIds(
↪"0000000000000000000064ba7512ecc70cabd7ed17e31c06f2205d5ecdadd6d22");
string t1 = block.Tx[0];
```

GetTransaction(txid) `method`

Retrieves the transaction and returns the data as json.

Returns

The transaction object.

Parameters

- System.String **txid** - The transaction Id.

Example

```
Transaction desiredTransaction = in3.Btc.GetTransaction(
↪"1427c7d1698e61afe061950226f1c149990b8c1e1b157320b0c4acf7d6b5605d");
```

GetTransactionBytes(txid) `method`

Retrieves the serialized transaction (bytes).

Returns

The byte array for the Transaction.

Parameters

- System.String **txid** - The transaction Id.

Example

```
byte[] serializedTransaction = in3.Btc.GetTransactionBytes(
↪"1427c7d1698e61afe061950226f1c149990b8c1e1b157320b0c4acf7d6b5605d");
```

DecryptKey(pk,passphrase) `method`

Decryot an encrypted private key.

Returns

Decrypted key.

Parameters

- System.String **pk** - Private key.
- System.String **passphrase** - Passphrase whose `pk`.

EcRecover(signedData,signature,signatureType) `method`

Recovers the account associated with the signed data.

Returns

The account.

Parameters

- System.String **signedData** - Data that was signed with.
- System.String **signature** - The signature.
- *In3.Crypto.SignatureType* **signatureType** - One of *SignatureType*.

Pk2Address(pk) `method`

Derives an address from the given private (`pk`) key using SHA-3 algorithm.

Returns

The address.

Parameters

- System.String **pk** - Private key.

Pk2Public(pk) `method`

Derives public key from the given private (`pk`) key using SHA-3 algorithm.

Returns

The public key.

Parameters

- System.String **pk** - Private key.

Sha3(data) `method`

Hash the input data using sha3 algorithm.

Returns

Hashed output.

Parameters

- System.String **data** - Content to be hashed.

SignData(msg,pk,sigType) `method`

Signs the data `msg` with a given private key. Refer to *SignedData* for more information.

Returns

The signed data.

Parameters

- System.String **msg** - Data to be signed.
- System.String **pk** - Private key.
- *In3.Crypto.SignatureType* **sigType** - Type of signature, one of *SignatureType*.

AbiDecode(signature,encodedData) `method`

ABI decoder. Used to parse rpc responses from the EVM. Based on the Solidity specification .

Returns

The decoded argugments for the function call given the encded data.

Parameters

- System.String **signature** - Function signature i.e. or . In case of the latter, the function signature will be ignored and only the return types will be parsed.
- System.String **encodedData** - Abi encoded values. Usually the string returned from a rpc to the EVM.

AbiEncode(signature,args) `method`

ABI encoder. Used to serialize a rpc to the EVM. Based on the Solidity specification . Note: Parameters refers to the list of variables in a method declaration. Arguments are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Returns

The encoded data.

Parameters

- System.String **signature** - Function signature, with parameters. i.e. , can contain the return types but will be ignored.
- System.Object[] **args** - Function parameters, in the same order as in passed on to .

BlockNumber() `method`

Returns the number of the most recent block the in3 network can collect signatures to verify. Can be changed by *ReplaceLatestBlock*. If you need the very latest block, change *SignatureCount* to `0`.

Returns

The number of the block.

Parameters

This method has no parameters.

Call(request,blockNumber) `method`

Calls a smart-contract method. Will be executed locally by Incubed's EVM or signed and sent over to save the state changes. Check https://ethereum.stackexchange.com/questions/3514/how-to-call-a-contract-method-using-the-eth-call-json-rpc-api for more.

Returns

Ddecoded result. If only one return value is expected the Object will be returned, if not an array of objects will be the result.

Parameters

- *In3.Eth1.TransactionRequest* **request** - The transaction request to be processed.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

ChecksumAddress(address,shouldUseChainId) `method`

Will convert an upper or lowercase Ethereum `address` to a checksum address, that uses case to encode values. See EIP55.

Returns

EIP-55 compliant, mixed-case address.

Parameters

- System.String **address** - Ethereum address.
- System.Nullable{System.Boolean} **shouldUseChainId** - If `true`, the chain id is integrated as well. Default being `false`.

Ens(name,type) `method`

Resolves ENS domain name.

Returns

The resolved entity for the domain.

Parameters

- System.String **name** - ENS domain name.
- *In3.ENSParameter* **type** - One of *ENSParameter*.

Remarks

The actual semantics of the returning value changes according to `type`.

EstimateGas(request,blockNumber) `method`

Gas estimation for transaction. Used to fill transaction.gas field. Check RawTransaction docs for more on gas.

Returns

Estimated gas in Wei.

Parameters

- *In3.Eth1.TransactionRequest* **request** - The transaction request whose cost will be estimated.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetBalance(address,blockNumber) `method`

Returns the balance of the account of given `address`.

Returns

The current balance in wei.

Parameters

- System.String **address** - Address to check for balance.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetBlockByHash(blockHash,shouldIncludeTransactions) `method`

Blocks can be identified by root hash of the block merkle tree (this), or sequential number in which it was mined *GetBlockByNumber*.

Returns

The *Block* of the requested (if exists).

Parameters

- System.String **blockHash** - Desired block hash.
- System.Boolean **shouldIncludeTransactions** - If true, returns the full transaction objects, otherwise only its hashes. The default value is `false`.

Remarks

Returning *Block* must be cast to *TransactionBlock* or *TransactionHashBlock* to access the transaction data.

GetBlockByNumber(blockNumber,shouldIncludeTransactions) `method`

Blocks can be identified by sequential number in which it was mined, or root hash of the block merkle tree *GetBlock-ByHash*.

Returns

The *Block* of the requested (if exists).

Parameters

- System.Numerics.BigInteger **blockNumber** - Desired block number or *Latest* or *Earliest*.
- System.Boolean **shouldIncludeTransactions** - If `true`, returns the full transaction objects, otherwise only its hashes. The default value is `true`.

Example

```
TransactionBlock latest = (TransactionBlock) _client.Eth1.
↪GetBlockByNumber(BlockParameter.Latest, true);
TransactionHashBlock earliest = (TransactionHashBlock) _client.Eth1.
↪GetBlockByNumber(BlockParameter.Earliest, false);
```

Remarks

Returning *Block* must be cast to *TransactionBlock* or *TransactionHashBlock* to access the transaction data.

GetBlockTransactionCountByHash(blockHash) `method`

The total transactions on a block. See also *GetBlockTransactionCountByNumber*.

Returns

The number (count) of *Transaction*.

Parameters

- System.String **blockHash** - Desired block hash.

GetBlockTransactionCountByNumber(blockNumber) `method`

The total transactions on a block. See also *GetBlockTransactionCountByHash*.

Returns

The number (count) of *Transaction*.

Parameters

- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetChainId() `method`

Get the *Chain* which the client is currently connected to.

Returns

The *Chain*.

Parameters

This method has no parameters.

GetCode(address,blockNumber) `method`

Smart-Contract bytecode in hexadecimal. If the account is a simple wallet the function will return '0x'.

Returns

Smart-Contract bytecode in hexadecimal.

Parameters

- System.String **address** - Ethereum address.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetFilterChangesFromLogs(filterId) `method`

Retrieve the logs for a certain filter. Logs marks changes of state on the chan for events. Equivalent to *GetFilterLogs*.

Returns

Array of logs which occurred since last poll.

Parameters

- System.Int64 **filterId** - Id returned during the filter creation.

Remarks

Since the return is the since last poll, executing this multiple times changes the state making this a "non-idempotent" getter.

GetFilterLogs(filterId) `method`

Retrieve the logs for a certain filter. Logs marks changes of state on the blockchain for events. Equivalent to *GetFilterChangesFromLogs*.

Returns

Array of logs which occurred since last poll.

Parameters

- System.Int64 **filterId** - Id returned during the filter creation.

Remarks

Since the return is the `Log[]` since last poll, executing this multiple times changes the state making this a "non-idempotent" getter.

GetGasPrice() `method`

The current gas price in Wei (1 ETH equals 1000000000000000000 Wei).

Returns

The gas price.

Parameters

This method has no parameters.

GetLogs(filter) `method`

Retrieve the logs for a certain filter. Logs marks changes of state on the blockchain for events. Unlike *GetFilterChangesFromLogs* or *GetFilterLogs* this is made to be used in a non-incremental manner (aka no poll) and will return the Logs that satisfy the filter condition.

Returns

Logs that satisfy the `filter`.

Parameters

- *In3.Eth1.LogFilter* **filter** - Filter conditions.

GetStorageAt(address,position,blockNumber) `method`

Stored value in designed position at a given `address`. Storage can be used to store a smart contract state, constructor or just any data. Each contract consists of a EVM bytecode handling the execution and a storage to save the state of the contract.

Returns

Stored value in designed position.

Parameters

- System.String **address** - Ethereum account address.
- System.Numerics.BigInteger **position** - Position index, 0x0 up to 100.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetTransactionByBlockHashAndIndex(blockHash,index) `method`

Transactions can be identified by root hash of the transaction merkle tree (this) or by its position in the block transactions merkle tree. Every transaction hash is unique for the whole chain. Collision could in theory happen, chances are 67148E-63%. See also *GetTransactionByBlockNumberAndIndex*.

Returns

The *Transaction* (if it exists).

Parameters

- System.String **blockHash** - Desired block hash.
- System.Int32 **index** - The index of the *Transaction* in a *Block*

GetTransactionByBlockNumberAndIndex(blockNumber,index) `method`

Transactions can be identified by root hash of the transaction merkle tree (this) or by its position in the block transactions merkle tree. Every transaction hash is unique for the whole chain. Collision could in theory happen, chances are 67148E-63%.

Returns

The *Transaction* (if it exists).

Parameters

- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.
- System.Int32 **index** - The index of the *Transaction* in a *Block*

GetTransactionByHash(transactionHash) `method`

Transactions can be identified by root hash of the transaction merkle tree (this) or by its position in the block transactions merkle tree. Every transaction hash is unique for the whole chain. Collision could in theory happen, chances are 67148E-63%.

Returns

The *Transaction* (if it exists).

Parameters

- System.String **transactionHash** - Desired transaction hash.

GetTransactionCount(address,blockNumber) `method`

Number of transactions mined from this `address`. Used to set transaction nonce. Nonce is a value that will make a transaction fail in case it is different from (transaction count + 1). It exists to mitigate replay attacks.

Returns

Number of transactions mined from this address.

Parameters

- System.String **address** - Ethereum account address.
- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

GetTransactionReceipt(transactionHash) `method`

After a transaction is received the by the client, it returns the transaction hash. With it, it is possible to gather the receipt, once a miner has mined and it is part of an acknowledged block. Because how it is possible, in distributed systems, that data is asymmetric in different parts of the system, the transaction is only "final" once a certain number of blocks was mined after it, and still it can be possible that the transaction is discarded after some time. But, in general terms, it is accepted that after 6 to 8 blocks from latest, that it is very likely that the transaction will stay in the chain.

Returns

The mined transaction data including event logs.

Parameters

- System.String **transactionHash** - Desired transaction hash.

GetUncleByBlockNumberAndIndex(blockNumber,position) `method`

Retrieve the of uncle of a block for the given `blockNumber` and a position. Uncle blocks are valid blocks and are mined in a genuine manner, but get rejected from the main blockchain.

Returns

The uncle block.

Parameters

- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.
- System.Int32 **position** - Position of the block.

GetUncleCountByBlockHash(blockHash) `method`

Retrieve the total of uncles of a block for the given `blockHash`. Uncle blocks are valid blocks and are mined in a genuine manner, but get rejected from the main blockchain. See *GetUncleCountByBlockNumber*.

Returns

The number of uncles in a block.

Parameters

- System.String **blockHash** - Desired block hash.

GetUncleCountByBlockNumber(blockNumber) `method`

Retrieve the total of uncles of a block for the given `blockNumber`. Uncle blocks are valid and are mined in a genuine manner, but get rejected from the main blockchain. See *GetUncleCountByBlockHash*.

Returns

The number of uncles in a block.

Parameters

- System.Numerics.BigInteger **blockNumber** - Block number or *Latest* or *Earliest*.

NewBlockFilter() `method`

Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call *GetFilterChangesFromLogs*. Filters are event catchers running on the Ethereum Client. Incubed has a client-side implementation. An event will be stored in case it is within to and from blocks, or in the block of blockhash, contains a transaction to the designed address, and has a word listed on topics.

Returns

The filter id.

Parameters

This method has no parameters.

Remarks

Use the returned filter id to perform other filter operations.

NewLogFilter(filter) `method`

Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call *GetFilterChangesFromLogs*. Filters are event catchers running on the Ethereum Client. Incubed has a client-side implementation. An event will be stored in case it is within to and from blocks, or in the block of blockhash, contains a transaction to the designed address, and has a word listed on topics.

Returns

The filter id.

Parameters

- *In3.Eth1.LogFilter* **filter** - Model that holds the data for the filter creation.

Remarks

Use the returned filter id to perform other filter operations.

SendRawTransaction(transactionData) `method`

Sends a signed and encoded transaction.

Returns

Transaction hash, used to get the receipt and check if the transaction was mined.

Parameters

- System.String **transactionData** - Signed keccak hash of the serialized transaction.

Remarks

Client will add the other required fields, gas and chaind id.

SendTransaction(tx) `method`

Signs and sends the assigned transaction. The *Signer* used to sign the transaction is the one set by *Signer*. Transactions change the state of an account, just the balance, or additionally, the storage and the code. Every transaction has a cost, gas, paid in Wei. The transaction gas is calculated over estimated gas times the gas cost, plus an additional miner fee, if the sender wants to be sure that the transaction will be mined in the latest block.

Returns

Transaction hash, used to get the receipt and check if the transaction was mined.

Parameters

- *In3.Eth1.TransactionRequest* **tx** - All information needed to perform a transaction.

Example

```
SimpleWallet wallet = (SimpleWallet) client.Signer;
TransactionRequest tx = new TransactionRequest();
tx.From = wallet.AddRawKey(pk);;
tx.To = "0x3940256B93c4BE0B1d5931A6A036608c25706B0c";
tx.Gas = 21000;
tx.Value = 100000000;
client.Eth1.SendTransaction(tx);
```

UninstallFilter(filterId) `method`

Uninstalls a previously created filter.

Returns

The result of the operation, `true` on success or `false` on failure.

Parameters

- System.Int64 **filterId** - The filter id returned by *NewBlockFilter*.

Get(multihash) `method`

Returns the content associated with specified multihash on success OR on error.

Returns

The content that was stored by *Put* or *Put*.

Parameters

- System.String **multihash** - The multihash.

Put(content) `method`

Stores content on ipfs.

Returns

The multihash.

Parameters

- System.String **content** - The content that will be stored via ipfs.

Put(content) `method`

Stores content on ipfs. The content is encoded as base64 before storing.

Returns

The multihash.

Parameters

- System.Byte[] **content** - The content that will be stored via ipfs.

14.4.6 BaseConfiguration `type`

In3.Configuration

Base class for all configuration classes.

14.4.7 Block `type`

In3.Eth1

Class that represents as Ethereum block.

Author `property`

The miner of the block.

Difficulty `property`

Dificulty of the block.

ExtraData `property`

Extra data.

GasLimit `property`

Gas limit.

Hash property

The block hash.

LogsBloom property

The logsBloom data of the block.

MixHash property

The mix hash of the block. (only valid of proof of work).

Nonce property

The nonce.

Number property

The index of the block.

ParentHash property

The parent block's hash.

ReceiptsRoot property

The roothash of the merkletree containing all transaction receipts of the block.

Sha3Uncles property

The roothash of the merkletree containing all uncles of the block.

Size property

Size of the block.

StateRoot property

The roothash of the merkletree containing the complete state.

Timestamp property

Epoch timestamp when the block was created.

TotalDifficulty property

Total Difficulty as a sum of all difficulties starting from genesis.

TransactionsRoot `property`

The roothash of the merkletree containing all transaction of the block.

Uncles `property`

List of uncle hashes.

14.4.8 BlockHeader `type`

In3.Btc

A Block header.

Bits `property`

Bits (target) for the block as hex.

Chainwork `property`

Total amount of work since genesis.

Confirmations `property`

Number of confirmations or blocks mined on top of the containing block.

Difficulty `property`

Difficulty of the block.

Hash `property`

The hash of the blockheader.

Height `property`

Block number.

Mediantime `property`

Unix timestamp in seconds since 1970.

Merkleroot `property`

Merkle root of the trie of all transactions in the block.

NTx property

Number of transactions in the block.

Nextblockhash property

Hash of the next blockheader.

Nonce property

Nonce-field of the block.

Previousblockhash property

Hash of the parent blockheader.

Time property

Unix timestamp in seconds since 1970.

Version property

Used version.

VersionHex property

Version as hex.

14.4.9 BlockParameter type

In3

Enum-like class that defines constants to be used with *Api*.

Earliest property

Genesis block.

Latest property

Constant associated with the latest mined block in the chain.

Remarks

While the parameter itself is constant the current "latest" block changes everytime a new block is mined. The result of the operations are also related to `ReplaceLatestBlock` on *ClientConfiguration*.

14.4.10 Block‘1 type

In3.Btc

A Block.

Size property

Size of this block in bytes.

Tx property

Transactions or Transaction ids of a block. *GetBlockWithTxData* or *GetBlockWithTxIds*.

Weight property

Weight of this block in bytes.

14.4.11 Chain type

In3

Represents the multiple chains supported by Incubed.

Btc constants

Bitcoin chain.

Evan constants

Evan testnet.

Ewc constants

Ewf chain.

Goerli constants

Goerli testnet.

Ipfs constants

Ipfs (InterPlanetary File System).

Kovan constants

Kovan testnet.

Local constants

Local client.

Mainnet constants

Ethereum mainnet.

Multichain constants

Support for multiple chains, a client can then switch between different chains (but consumes more memory).

Tobalaba constants

Tobalaba testnet.

Volta constants

Volta testnet.

14.4.12 ChainConfiguration type

In3.Configuration

Class that represents part of the configuration to be applied on the *IN3* (in particular to each chain). This is a child of *ClientConfiguration* and have many *NodeConfiguration*.

#ctor(chain,clientConfiguration) constructor

Constructor.

Parameters

- *In3.Chain* **chain** - One of *Chain*. The chain that this configuration is related to.
- *In3.Configuration.ClientConfiguration* **clientConfiguration** - The configuration for the client whose the chain configuration belongs to.

Example

```
ChainConfiguration goerliConfiguration = new ChainConfiguration(Chain.Goerli,␣
↪in3Client.Configuration);
```

Contract property

Incubed registry contract from which the list was taken.

NeedsUpdate `property`

Preemptively update the node list.

NodesConfiguration `property`

Getter for the list of elements that represent the configuration for each node.

Remarks

This is a read-only property. To add configuration for nodes, Use *NodeConfiguration* constructor.

RegistryId `property`

Uuid of this incubed network. one chain could contain more than one incubed networks.

WhiteList `property`

Node addresses that constitute the white list of nodes.

WhiteListContract `property`

Address of whiteList contract.

14.4.13 ClientConfiguration `type`

In3.Configuration

Class that represents the configuration to be applied on *IN3*. Due to the 1-to-1 relationship with the client, this class should never be instantiated. To obtain a reference of the client configuration use *Configuration* instead.

Remarks

Use in conjunction with *ChainConfiguration* and *NodeConfiguration*.

AutoUpdateList `property`

If `true` the nodelist will be automatically updated. False may compromise data security.

BootWeights `property`

if true, the first request (updating the nodelist) will also fetch the current health status and use it for blacklisting unhealthy nodes. This is used only if no nodelist is availabkle from cache.

ChainsConfiguration `property`

Configuration for the chains. Read-only attribute.

Finality `property`

Remarks

Beware that the semantics of the values change greatly from chain to chain. The value of `8` would mean 8 blocks mined on top of the requested one while with the POW algorithm while, for POA, it would mean 8% of validators.

IncludeCode `property`

Code is included when sending eth_call-requests.

KeepIn3 `property`

Tthe in3-section (custom node on the RPC call) with the proof will also returned.

MaxAttempts `property`

Maximum times the client will retry to contact a certain node.

MinDeposit `property`

Only nodes owning at least this amount will be chosen to sign responses to your requests.

NodeLimit `property`

Limit nodes stored in the client.

NodeProps `property`

Props define the capabilities of the nodes. Accepts a combination of values.

Example

```
clientConfiguration.NodeProps = Props.NodePropProof | Props.NodePropArchive;
```

Proof `property`

One of *Proof*. *Full* gets the whole block Patricia-Merkle-Tree, *Standard* only verifies the specific tree branch concerning the request, *None* only verifies the root hashes, like a light-client does.

ReplaceLatestBlock `property`

Distance considered safe, consensus wise, from the very latest block. Higher values exponentially increases state finality, and therefore data security, as well guaranteeded responses from in3 nodes.

RequestCount `property`

Useful when *SignatureCount* is less then `1`. The client will check for consensus in responses.

Rpc `property`

Setup an custom rpc source for requests by setting chain to *Local* and proof to *None*.

SignatureCount `property`

Node signatures attesting the response to your request. Will send a separate request for each.

Example

When set to `3`, 3 nodes will have to sign the response.

Timeout `property`

Milliseconds before a request times out.

UseHttp `property`

Disable ssl on the Http connection.

14.4.14 Context `type`

In3.Context

Acts as the main orchestrator for the execution of an rpc. Holds a reference to the native context (ctx) and wraps behavior around it.

#ctor(ctx,nativeClient) `constructor`

Standard constructor, private so people use *FromRpc*.

Parameters

- System.IntPtr **ctx** - The native rpc context.
- *In3.Native.NativeClient* **nativeClient** - Object that encapsulates the native client.

CreateNativeCtx(nativeIn3Ptr,rpc) `method`

Method to manage the creation of the native ctx request.

Returns

Native rpc pointer

Parameters

- System.IntPtr **nativeIn3Ptr** - Native client pointer.
- System.String **rpc** - The rpc request

Exceptions

| Name | Description |

| *In3.Exceptions.RpcException* | |

Dispose() `method`

Destructor method for the native ctx encapsulated by the *Context* object.

Parameters

This method has no parameters.

Execute() `method`

Proxy to in3_ctx_execute, every invocation generates a new state.

Returns

The state as computed by in3_ctx_execute.

Parameters

This method has no parameters.

FromRpc(wrapper,rpc) `method`

Factory-like method to build a Context object from an rpc request.

Returns

An instance of context.

Parameters

- *In3.Native.NativeClient* **wrapper** - The object that encapsulates the native client pointer.
- System.String **rpc** - The rpc request

GetErrorMessage() `method`

Retrieve the error result on the context.

Returns

A string describing the encountered error.

Parameters

This method has no parameters.

GetLastWaiting() `method`

Method responsible to fetch the pending context references in the current context.

Returns

A context object.

Parameters

This method has no parameters.

GetResponse() `method`

Method to get the consolidated response of a request.

Returns

The final result.

Parameters

This method has no parameters.

GetType() `method`

Method to get the consolidated response of a request.

Returns

The final result.

Parameters

This method has no parameters.

HandleRequest() `method`

Handle rpc request in an asynchronous manner.

Parameters

This method has no parameters.

HandleSign() `method`

Handle signing request in an asynchronous manner.

Parameters

This method has no parameters.

IsValid() `method`

Conditional to verify if the encapsulated pointer actually points to something.

Returns

if its valid, `false` if it is not.

Parameters

This method has no parameters.

ReportError() `method`

Setter for the error on the current context. Proxies it to the native context.

Parameters

This method has no parameters.

14.4.15 DataTypeConverter `type`

In3.Utils

General util class for conversion between blockchain types.

HexStringToBigint(source) `method`

Converts a zero-prefixed hex (e.g.: 0x05) to BigInteger

Returns

The number representation of `source`.

Parameters

- System.String **source** - The hex number string.

14.4.16 DefaultTransport `type`

In3.Transport

Basic implementation for synchronous http transport for Incubed client.

#ctor() `constructor`

Standard construction.

Parameters

This constructor has no parameters.

Handle(url,payload) `method`

Method that handles, sychronously the http requests.

Returns

The http json response.

Parameters

- System.String **url** - The url of the node.
- System.String **payload** - Json for the body of the POST request to the node.

14.4.17 ENSParameter `type`

In3

Defines the kind of entity associated with the ENS Resolved. Used along with *Ens*.

Addr `property`

Address.

Hash `property`

Hash.

Owner `property`

Owner.

Resolver `property`

Resolver.

14.4.18 IN3 `type`

In3

Incubed network client. Connect to the blockchain via a list of bootnodes, then gets the latest list of nodes in the network and ask a certain number of the to sign the block header of given list, putting their deposit at stake. Once with the latest list at hand, the client can request any other on-chain information using the same scheme.

#ctor(chainId) `constructor`

Standard constructor, use *ForChain* instead.

Parameters

- *In3.Chain* **chainId** - The chainId to connect to.

Btc `property`

Gets *Api* object.

Configuration `property`

Gets *ClientConfiguration* object. Any changes in the object will be automaticaly applied to the client before each method invocation.

Crypto `property`

Gets *Api* object.

Eth1 `property`

Gets *Api* object.

Ipfs `property`

Gets *Api* object.

Signer `property`

Get or Sets *Signer* object. If not set *SimpleWallet* will be used.

Storage `property`

Get or Sets *Storage* object. If not set *InMemoryStorage* will be used.

Transport `property`

Gets or sets *Transport* object. If not set *DefaultTransport* will be used.

Finalize() `method`

Finalizer for the client.

Parameters

This method has no parameters.

ForChain(chain) `method`

Creates a new instance of `IN3`.

Returns

An Incubed instance.

Parameters

- *In3.Chain* **chain** - *Chain* that Incubed will connect to.

Example

```
IN3 client = IN3.ForChain(Chain.Mainnet);
```

SendRpc(method,args,in3) `method`

Method used to communicate with the client. In general, its preferably to use the API.

Returns

The result of the Rpc operation as JSON.

Parameters

- System.String **method** - Rpc method.
- System.Object[] **args** - Arguments to the operation.
- System.Collections.Generic.Dictionary{System.String,System.Object} **in3** - Internal parameters to be repassed to the server or to change the client behavior.

14.4.19 InMemoryStorage `type`

In3.Storage

Default implementation of *Storage*. It caches all cacheable data in memory.

#ctor() `constructor`

Standard constructor.

Parameters

This constructor has no parameters.

Clear() `method`

Empty the in-memory cache.

Returns

Result for the clear operation.

Parameters

This method has no parameters.

GetItem(key) `method`

Fetches the data from memory.

Returns

The cached value as a `byte[]`.

Parameters

- System.String **key** - Key

SetItem(key,content) `method`

Stores a value in memory for a given key.

Parameters

- System.String **key** - A unique identifier for the data that is being cached.
- System.Byte[] **content** - The value that is being cached.

14.4.20 Log `type`

In3.Eth1

Logs marks changes of state on the blockchain for events. The *Log* is a data object with information from logs.

Address `property`

Address from which this log originated.

BlockHash `property`

Hash of the block this log was in. null when its pending log.

BlockNumber `property`

Number of the block this log was in.

Data `property`

Data associated with the log.

LogIndex `property`

Index position in the block.

Removed `property`

Flags log removal (due to chain reorganization).

Topics `property`

Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier).

TransactionHash `property`

Hash of the transactions this log was created from. null when its pending log.

TransactionIndex `property`

index position log was created from.

Type `property`

Address from which this log originated.

14.4.21 LogFilter `type`

In3.Eth1

Filter configuration for search logs. To be used along with the *Api* filter and methods.

#ctor() `constructor`

Standard constructor.

Parameters

This constructor has no parameters.

Address `property`

Address for the filter.

BlockHash `property`

Blcok hash of the filtered blocks.

Remarks

If present, *FromBlock* and *ToBlock* will be ignored.

FromBlock `property`

Starting block for the filter.

ToBlock `property`

End block for the filter.

Topics `property`

Array of 32 Bytes Data topics. Topics are order-dependent. It's possible to pass in null to match any topic, or a subarray of multiple topics of which one should be matching.

14.4.22 NodeConfiguration `type`

In3.Configuration

Class that represents part of the configuration to be applied on the *IN3* (in particular to each boot node). This is a child of *ChainConfiguration*.

#ctor(config) `constructor`

Constructor for the node configuration.

Parameters

- *In3.Configuration.ChainConfiguration* **config** - The *ChainConfiguration* of which this node belongs to.

Example

```
NodeConfiguration myDeployedNode = new NodeConfiguration(mainnetChainConfiguration);
```

Address `property`

Address of the node, which is the public address it is signing with.

Props `property`

Props define the capabilities of the node. Accepts a combination of values.

Example

```
nodeConfiguration.Props = Props.NodePropProof | Props.NodePropArchive;
```

Url `property`

Url of the bootnode which the client can connect to.

14.4.23 Proof `type`

In3.Configuration

Alias for verification levels. Verification is done by calculating Ethereum Trie states requested by the Incubed network ans signed as proofs of a certain state.

Full `property`

All fields will be validated (including uncles).

None `property`

No Verification.

Standard `property`

Standard Verification of the important properties.

14.4.24 Props `type`

In3.Configuration

`Enum` that defines the capabilities an incubed node.

NodePropArchive `constants`

filter out non-archive supporting nodes.

NodePropBinary `constants`

filter out nodes that don't support binary encoding.

NodePropData `constants`

filter out non-data provider nodes.

NodePropHttp `constants`

filter out non-http nodes.

NodePropMinblockheight `constants`

filter out nodes that will sign blocks with lower min block height than specified.

NodePropMultichain `constants`

filter out nodes other then which have capability of the same RPC endpoint may also accept requests for different chains.

NodePropOnion `constants`

filter out non-onion nodes.

NodePropProof `constants`

filter out nodes which are providing no proof.

NodePropSigner `constants`

filter out non-signer nodes.

NodePropStats `constants`

filter out nodes that do not provide stats.

14.4.25 RpcException `type`

In3.Exceptions

Custom Exception to be thrown during the

14.4.26 ScriptPubKey `type`

In3.Btc

Script on a transaction output.

Addresses `property`

List of addresses.

Asm `property`

The asm data,

Hex `property`

The raw hex data.

ReqSigs `property`

The required sigs.

Type `property`

The type.

Example

pubkeyhash

14.4.27 ScriptSig `type`

In3.Btc

Script on a transaction input.

Asm `property`

The asm data.

Hex `property`

The raw hex data.

14.4.28 SignatureType `type`

In3.Crypto

Group of constants to be used along with the methods of *Api*.

EthSign `property`

For hashes of the RLP prefixed.

Hash `property`

For data that was hashed and then signed.

Raw `property`

For data that was signed directly.

14.4.29 SignedData `type`

In3.Crypto

Output of *SignData*.

Message `property`

Signed message.

MessageHash `property`

Hash of (*Message*.

R `property`

Part of the ECDSA signature.

S `property`

Part of the ECDSA signature.

Signature `property`

ECDSA calculated r, s, and parity v, concatenated.

V `property`

27 + (*R* % 2).

14.4.30 Signer `type`

In3.Crypto

Minimum interface to be implemented by a kind of signer. Used by *SendTransaction*. Set it with *Signer*.

CanSign(account) `method`

Queries the Signer if it can sign for a certain key.

Returns

`true` if it can sign, `false` if it cant.

Parameters

- System.String **account** - The account derived from the private key used to sign transactions.

Remarks

This method is invoked internaly by *SendTransaction* using *From* and will throw a `SystemException` in case `false` is returned.

PrepareTransaction() `method`

Optional method which allows to change the transaction-data before sending it. This can be used for redirecting it through a multisig. Invoked just before sending a transaction through *SendTransaction*.

Returns

Modified transaction request.

Parameters

This method has no parameters.

Sign(data,account) `method`

Signs the transaction data with the private key associated with the invoked account. Both arguments are automaticaly passed by Incubed client base on *TransactionRequest* data during a *SendTransaction*.

Returns

The signed transaction data.

Parameters

- System.String **data** - Data to be signed.
- System.String **account** - The account that will sign the transaction.

14.4.31 SimpleWallet `type`

In3.Crypto

Default implementation of the *Signer*. Works as an orchestration of the in order to manage multiple accounts.

#ctor(in3) `constructor`

Basic constructor.

Parameters

- *In3.IN3* **in3** - A client instance.

AddRawKey(privateKey) `method`

Adds a private key to be managed by the wallet and sign transactions.

Returns

The address derived from the `privateKey`

Parameters

- System.String **privateKey** - The private key to be stored by the wallet.

CanSign(address) `method`

Check if this address is managed by this wallet.

Returns

`true` if the address is managed by this wallter, `false` if not.

Parameters

- System.String **address** - The address. Value returned by *AddRawKey*.

PrepareTransaction(tx) `method`

Identity function-like method.

Returns

`tx`

Parameters

- *In3.Eth1.TransactionRequest* **tx** - A transaction object.

Sign(data,address) `method`

Signs the transaction data by invoking *SignData*.

Returns

Signed transaction data.

Parameters

- System.String **data** - Data to be signed.
- System.String **address** - Address managed by the wallet, see *AddRawKey*

14.4.32 Storage `type`

In3.Storage

Provider methods to cache data. These data could be nodelists, contract codes or validator changes. Any form of cache should implement *Storage* and be set with *Storage*.

Clear() `method`

Clear the cache.

Returns

The result of the operation: `true` for success and `false` for failure.

Parameters

This method has no parameters.

GetItem(key) `method`

returns a item from cache.

Returns

The bytes or `null` if not found.

Parameters

- System.String **key** - The key for the item.

SetItem(key,content) `method`

Stores an item to cache.

Parameters

- System.String **key** - The key for the item.
- System.Byte[] **content** - The value to store.

14.4.33 Transaction `type`

In3.Btc

A BitCoin Transaction.

14.4.34 Transaction `type`

In3.Eth1

Class representing a transaction that was accepted by the Ethereum chain.

Blockhash `property`

The block hash of the block containing this transaction.

Blocktime `property`

The block time in seconds since epoch (Jan 1 1970 GMT).

Confirmations `property`

The confirmations.

Hash `property`

The transaction hash (differs from txid for witness transactions).

Hex `property`

The hex representation of raw data.

Locktime `property`

The locktime.

Size `property`

The serialized transaction size.

Time `property`

The transaction time in seconds since epoch (Jan 1 1970 GMT).

Txid `property`

Transaction Id.

Version `property`

The version.

Vin `property`

The transaction inputs.

Vout `property`

The transaction outputs.

Vsize `property`

The virtual transaction size (differs from size for witness transactions).

Weight `property`

The transaction's weight (between vsize4-3 and vsize4).

BlockHash `property`

Hash of the block that this transaction belongs to.

BlockNumber `property`

Number of the block that this transaction belongs to.

ChainId `property`

Chain id that this transaction belongs to.

Creates `property`

Address of the deployed contract (if successfull).

From `property`

Address whose private key signed this transaction with.

Gas `property`

Gas for the transaction.

GasPrice `property`

Gas price (in wei) for each unit of gas.

Hash `property`

Transaction hash.

Input `property`

Transaction data.

Nonce `property`

Nonce for this transaction.

PublicKey `property`

Public key.

R `property`

Part of the transaction signature.

Raw `property`

Transaction as rlp encoded data.

S `property`

Part of the transaction signature.

StandardV `property`

Part of the transaction signature. V is parity set by v = 27 + (r % 2).

To `property`

To address of the transaction.

TransactionIndex `property`

Transaction index.

V `property`

The *StandardV* plus the chain.

Value `property`

Value of the transaction.

14.4.35 TransactionBlock `type`

In3.Eth1

Class that holds a block with its full transaction array: *Transaction*.

Transactions `property`

Array with the full transactions containing on this block.

Remarks

Returned when `shouldIncludeTransactions` on *Api* get block methods are set to `true`.

14.4.36 TransactionHashBlock `type`

In3.Eth1

Class that holds a block with its transaction hash array.

Transactions `property`

Array with the full transactions containing on this block.

Remarks

Returned when `shouldIncludeTransactions` on *Api* get block methods are set to `false`.

14.4.37 TransactionInput `type`

In3.Btc

Input of a transaction.

ScriptSig `property`

The script.

Sequence `property`

The script sequence number.

Txid `property`

The transaction id.

Txinwitness `property`

Hex-encoded witness data (if any).

Yout `property`

The index of the transactionoutput.

14.4.38 TransactionOutput `type`

In3.Btc

Output of a transaction.

N `property`

The index in the transaction.

ScriptPubKey `property`

The script of the transaction.

Value `property`

The value in bitcoins.

14.4.39 TransactionReceipt `type`

In3.Eth1

Class that represents a transaction receipt. See *GetTransactionReceipt*.

BlockHash `property`

Hash of the block with the transaction which this receipt is associated with.

BlockNumber `property`

Number of the block with the transaction which this receipt is associated with.

ContractAddress `property`

Address of the smart contract invoked in the transaction (if any).

From `property`

Address of the account that signed the transaction.

GasUsed `property`

Gas used on this transaction.

Logs `property`

Logs/events for this transaction.

LogsBloom `property`

A bloom filter of logs/events generated by contracts during transaction execution. Used to efficiently rule out transactions without expected logs.

Root `property`

Merkle root of the state trie after the transaction has been executed (optional after Byzantium hard fork EIP609).

Status `property`

Status of the transaction.

To `property`

Address whose value will be transfered to.

TransactionHash `property`

Hash of the transaction.

TransactionIndex `property`

Number of the transaction on the block.

14.4.40 TransactionRequest `type`

In3.Eth1

Class that holds the state for the transaction request to be submited via *SendTransaction*.

Data `property`

Data of the transaction (in the case of a smart contract deployment for exemple).

From `property`

Address derivated from the private key that will sign the transaction. See *Signer*.

Function `property`

Function of the smart contract to be invoked.

Gas `property`

Gas cost for the transaction. Can be estimated via *EstimateGas*.

GasPrice `property`

Gas price (in wei). Can be obtained via *GetGasPrice*.

Nonce `property`

Nonce of the transaction.

Params `property`

Array of parameters for the function (in the same order of its signature), see *Function*

To `property`

Address to whom the transaction value will be transfered to or the smart contract address whose function will be invoked.

Value `property`

Value of the transaction.

14.4.41 Transport `type`

In3.Transport

Minimum interface for a custom transport. Transport is a mean of communication with the Incubed server.

Handle(url,payload) `method`

Method to be implemented that will handle the requests to the server. This method may be called once for each url on each batch of requests.

Returns

The rpc response.

Parameters

- System.String **url** - Url of the node.
- System.String **payload** - Content for the RPC request.

CHAPTER 15

API Reference Rust

15.1 IN3 Rust API features:

- Cross-platform support tested with cross-rs.
- Unsafe code is isolated to a small subset of the API and should not be required for most use cases.
- The C sources are bundled with the crate and we leverage the rust-bindgen and cmake-rs projects to auto-generate bindings.
- Leak-free verified with Valgrind-memcheck.
- Well-documented API support for Ethereum, Bitcoin, and IPFS.
- Customizable storage, transport, and signing.
- All of IN3's verification capabilities with examples and much more!

15.2 Quickstart

15.2.1 Add in3 to Cargo manifest

Add IN3 and futures_executor (or just any executor of your choice) to your cargo manifest. The in3-rs API is asynchronous and Rust doesn't have any built-in executors so we need to choose one, and we decided futures_executor is a very good option as it is lightweight and practical to use.

```
[package]
name = "in3-tutorial"
version = "0.0.1"
authors = ["reader@medium.com"]
edition = "2018"

[dependencies]
```

(continues on next page)

(continued from previous page)

```
in3 = "1.0.0"
futures-executor = "0.3.5"
```

Let's begin with the 'hello-world' equivalent of the Ethereum JSON-RPC API - eth_blockNumber. This call returns the number of the most recent block in the blockchain. Here's the complete program:

```
use in3::eth1;
use in3::prelude::*;

fn main() -> In3Result<()> {
   let client = Client::new(chain::MAINNET);
   let mut eth_api = eth1::Api::new(client);
   let number = futures_executor::block_on(eth_api.block_number())?;
   println!("Latest block number => {:?}", number);
   Ok(())
}
```

Now, let's go through this program line-by-line. We start by creating a JSON-RPC capable Incubed Client instance for the Ethereum mainnet chain.

```
let client = Client::new(chain::MAINNET);
```

This client is then used to instantiate an Ethereum Api instance which implements the Ethereum JSON-RPC API spec.

```
let mut eth_api = eth1::Api::new(client);
```

From here, getting the latest block number is as simple as calling the block_number() function on the Ethereum Api instance. As specified before, we need to use futures_executor::block_on to run the future returned by block_number() to completion on the current thread.

```
let number = futures_executor::block_on(eth_api.block_number())?;
```

A complete list of supported functions can be found on the in3-rs crate documentation page at docs.rs.

15.2.2 Get an Ethereum block by number

```

use async_std::task;
use in3::prelude::*;

fn main() {
    // configure client and API
    let mut eth_api = Api::new(Client::new(chain::MAINNET));
    // get latest block
    let block: Block = block_on(eth_api.get_block_by_number(BlockNumber::Latest,␣
↪false))?;
    println!("Block => {:?}", block);
}
```

15.2.3 An Ethereum contract call

In this case, we are reading the number of nodes that are registered in the IN3 network deployed on the Ethereum Mainnet at `0x2736D225f85740f42D17987100dc8d58e9e16252`

```
use async_std::task;
use in3::prelude::*;
fn main() {
    // configure client and API
    let mut eth_api = Api::new(Client::new(chain::MAINNET));
    // Setup Incubed contract address
    let contract: Address =
        serde_json::from_str(r#""0x2736D225f85740f42D17987100dc8d58e9e16252""#).
↪unwrap(); // cannot fail
    // Instantiate an abi encoder for the contract call
    let mut abi = abi::In3EthAbi::new();
    // Setup the signature to call in this case we are calling totalServers():uint256␣
↪from in3-nodes contract
    let params = task::block_on(abi.encode("totalServers():uint256", serde_json::json!
↪([])))
        .expect("failed to ABI encode params");
    // Setup the transaction with contract and signature data
    let txn = CallTransaction {
        to: Some(contract),
        data: Some(params),
        ..Default::default()
    };
    // Execute asynchronous api call.
    let output: Bytes =
        task::block_on(eth_api.call(txn, BlockNumber::Latest)).expect("ETH call failed
↪");
    // Decode the Bytes output and get the result
    let output =
        task::block_on(abi.decode("uint256", output)).expect("failed to ABI decode␣
↪output");
    let total_servers: U256 = serde_json::from_value(output).unwrap(); // cannot fail␣
↪if ABI decode succeeds
    println!("{:?}", total_servers);

}
```

15.2.4 Store a string in IPFS

IPFS is a protocol and peer-to-peer network for storing and sharing data in a distributed file system.

```
fn main() {
    let mut ipfs_api = Api::new(Client::new(chain::IPFS));
    //`put` is an asynchrous request (due to the internal C library). Therefore to␣
↪block execution
    //we use async_std's block_on function
    match task::block_on(ipfs_api.put("incubed meets rust".as_bytes().into())) {
        Ok(res) => println!("The hash is {:?}", res),
        Err(err) => println!("Failed with error: {}", err),
    }
}
```

15.2.5 Ready-To-Run Example

Head over to our sample project on GitHub or simply run:

```
$ git clone https://github.com/hu55a1n1/in3-examples.rs
$ cd in3-examples.rs
$ cargo run
```

15.3 Crate

In3 crate can be found in crates.io/crates/in3

15.4 Api Documentation

Api reference information can be found in docs.rs/in3/0.0.2/in3

CHAPTER 16

API Reference CMD

Incubed can be used as a command-line utility or as a tool in Bash scripts. This tool will execute a JSON-RPC request and write the result to standard output.

16.1 Usage

```
in3 [options] method [arguments]
```

-c, -chain The chain to use currently:

- **mainnet** Mainnet
- **kovan** Kovan testnet
- **tobalaba** EWF testchain
- **goerli** Goerli testchain using Clique
- **btc** Bitcoin (still experimental)
- **local** Use the local client on http://localhost:8545
- **RPCURL** If any other RPC-URL is passed as chain name, this is used but without verification

-p, -proof Specifies the verification level:

- **none** No proof
- **standard** Standard verification (default)
- **full** Full verification

-np Short for `-p none`.

-s, -signs Number of signatures to use when verifying.

-b, -block	The block number to use when making calls. Could be either `latest` (default), `earliest`, or a hex number.
-l, -latest	replaces `latest` with latest BlockNumber - the number of blocks given.
-pk	The path to the private key as keystore file.
-pwd	Password to unlock the key. (Warning: since the passphrase must be kept private, make sure that this key may not appear in the bash_history)
-to	The target address of the call.
-st, -sigtype	the type of the signature data : `eth_sign` (use the prefix and hash it), `raw` (hash the raw data), `hash` (use the already hashed data). Default: raw
-port	specifies the port to run incubed as a server. Opening port 8545 may replace a local parity or geth client.
-d, -data	The data for a transaction. This can be a file path, a 0x-hexvalue, or `-` to read it from standard input. If a method signature is given with the data, they will be combined and used as constructor arguments when deploying.
-gas	The gas limit to use when sending transactions (default: 100000).
-value	The value to send when conducting a transaction. Can be a hex value or a float/integer with the suffix `eth` or `wei` like `1.8eth` (default: 0).
-w, -wait	If given, `eth_sendTransaction` or `eth_sendRawTransaction` will not only return the transaction hash after sending but also wait until the transaction is mined and returned to the transaction receipt.
-json	If given, the result will be returned as JSON, which is especially important for `eth_call`, which results in complex structres.
-hex	If given, the result will be returned as hex.
-debug	If given, Incubed will output debug information when executing.
-q	quiet. no warnings or log to stderr.
-ri	Reads the response from standard input instead of sending the request, allowing for offline use cases.
-ro	Writes the raw response from the node to standard output.

16.2 Install

16.2.1 From Binaries

You can download the from the latest release-page:

https://github.com/slockit/in3-c/releases

These release files contain the sources, precompiled libraries and executables, headerfiles and documentation.

16.2.2 From Package Managers

We currently support

Ubuntu Launchpad (Linux)

Installs libs and binaries on IoT devices or Linux-Systems

```
# Add the slock.it ppa to your system
sudo add-apt-repository ppa:devops-slock-it/in3

# install the commandline tool in3
apt-get install in3

# install shared and static libs and header files
apt-get install in3-dev
```

Brew (MacOS)

This is the easiest way to install it on your mac using brew

```
# Add a brew tap
brew tap slockit/in3

# install all binaries and libraries
brew install in3
```

16.2.3 From Sources

Before building, make sure you have these components installed:

- CMake (should be installed as part of the build-essential: `apt-get install build-essential`)
- libcurl (for Ubuntu, use either `sudo apt-get install libcurl4-gnutls-dev` or `apt-get install libcurl4-openssl-dev`)
- If libcurl cannot be found, Conan is used to fetch and build curl

```
# clone the sources
git clone https://github.com/slockit/in3-c.git

# create build-folder
cd in3-c
mkdir build && cd build

# configure and build
cmake -DCMAKE_BUILD_TYPE=Release .. && make in3

# install
sudo make install
```

When building from source, CMake accepts the flags which help to optimize. For more details just look at the CMake-Options .

16.2.4 From Docker

Incubed can be run as docker container. For this pull the container:

```
# run a simple statement
docker run slockit/in3:latest eth_blockNumber

# to start it as a server
docker run -p 8545:8545 slockit/in3:latest -port 8545

# mount the cache in order to cache nodelists, validatorlists and contract code.
docker run -v $(pwd)/cache:/root/.in3 -p 8545:8545 slockit/in3:latest -port 8545
```

16.3 Environment Variables

The following environment variables may be used to define defaults:

IN3_PK The raw private key used for signing. This should be used with caution, since all subprocesses have access to it!

IN3_CHAIN The chain to use (default: mainnet) (same as -c). If a URL is passed, this server will be used instead.

16.4 Methods

As methods, the following can be used:

<JSON-RPC>-method All officially supported JSON-RPC methods may be used.

send <signature> ...args Based on the `-to`, `-value`, and `-pk`, a transaction is built, signed, and sent. If there is another argument after *send*, this would be taken as a function signature of the smart contract followed by optional arguments of the function.

```
# Send some ETH (requires setting the IN3_PK-variable before).
in3 send -to 0x1234556 -value 0.5eth
# Send a text to a function.
in3 -to 0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c  -gas 1000000 send
↪"registerServer(string,uint256)" "https://in3.slock.it/kovan1" 0xFF
```

sign <data> signs the data and returns the signature (65byte as hex). Use the -sigtype to specify the creation of the hash.

call <signature> ...args `eth_call` to call a function. After the `call` argument, the function signature and its arguments must follow.

in3_nodeList Returns the NodeList of the Incubed NodeRegistry as JSON.

in3_sign <blocknumber> Requests a node to sign. To specify the signer, you need to pass the URL like this:

```
# Send a text to a function.
in3 in3_sign -c https://in3.slock.it/mainnet/nd-1 6000000
```

in3_stats Returns the stats of a node. Unless you specify the node with `-c <rpcurl>`, it will pick a random node.

abi_encode <signature> ...args Encodes the arguments as described in the method signature using ABI encoding.

abi_decode <signature> data Decodes the data based on the signature.

pk2address <privatekey> Extracts the public address from a private key.

pk2public <privatekey> Extracts the public key from a private key.

ecrecover <msg> <signature> Extracts the address and public key from a signature.

createkey Generates a random raw private key.

key <keyfile> Reads the private key from JSON keystore file from the first argument and returns the private key. This may ask the user to enter the passphrase (unless provided with -pwd). To unlock the key to reuse it within the shell, you can set the environment variable like this:

```
export IN3_PK=`in3 keystore mykeyfile.json`
```

if no method is passed, this tool will read json-rpc-requests from stdin and response on stdout until stdin is closed.

```
echo '{"method":"eth_blockNumber","params":[]}' | in3 -q -c goerli
```

This can also be used process to communicate with by startiing a in3-process and send rpc-comands through stdin and read the responses from stout. if multiple requests are passed in the input stream, they will executed in the same order. The result will be terminated by a newline-character.

16.5 Running as Server

While you can use in3 to execute a request, return a result and quit, you can also start it as a server using the specified port (-port 8545) to serve RPC-requests. Thiss way you can replace your local parity or geth with a incubed client. All Dapps can then connect to http://localhost:8545.

```
# starts a server at the standard port for kovan.
in3 -c kovan -port 8545
```

16.6 Cache

Even though Incubed does not need a configuration or setup and runs completely statelessly, caching already verified data can boost the performance. That's why in3 uses a cache to store.

NodeLists List of all nodes as verified from the registry.

Reputations Holding the score for each node to improve weights for honest nodes.

Code For eth_call, Incubed needs the code of the contract, but this can be taken from a cache if possible.

Validators For PoA changes, the validators and their changes over time will be stored.

By default, Incubed will use ~/.in3 as a folder to cache data.

If you run the docker container, you need to mount /root/.in3 in to persist the cache.

16.7 Signing

While Incubed itself uses an abstract definition for signing, at the moment, the command-line utility only supports raw private keys. There are two ways you can specify the private keys that Incubed should use to sign transactions:

1. Use the environment variable IN3_PK. This makes it easier to run multiple transaction.

> **Warning:** Since the key is stored in an envirmoent variable all subpoccess have access to this. That's why this method is potentially unsafe.

```
#!/bin/sh

# reads the key from the keyfile and asks the user for the passphrase.
IN3_PK = `in3 key my_keyfile.json`

# you can can now use this private keys since it is stored in a enviroment-
↪variable
in3 -to 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1 -value 3.5eth -wait send
in3 -to 0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c  -gas 1000000 send
↪"registerServer(string,uint256)" "https://in3.slock.it/kovan1" 0xFF
```

2. Use the -pk option

   This option takes the path to the keystore-file and will ask the user to unlock as needed. It will not store the unlocked key anywhere.

```
in3 -pk my_keyfile.json -to 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1 -value␣
↪200eth -wait send
```

16.8 Autocompletion

If you want autocompletion, simply add these lines to your *.bashrc* or *.bash_profile*:

```
_IN3_WORDS=`in3 autocompletelist`
complete -W "$_IN3_WORDS" in3
```

16.9 Function Signatures

When using send or call, the next optional parameter is the function signature. This signature describes not only the name of the function to call but also the types of arguments and return values.

In general, the signature is built by simply removing all names and only holding onto the types:

```
<FUNCTION_NAME>(<ARGUMENT_TYPES>):(<RETURN_TYPES>)
```

It is important to mention that the type names must always be the full Solidity names. Most Solidity functions use aliases. They would need to be replaced with the full type name.

e.g., uint -> uint256

16.10 Examples

16.10.1 Getting the Current Block

```
# On a command line:
in3 eth_blockNumber
> 8035324

# For a different chain:
in3 -c kovan eth_blockNumber
```

(continues on next page)

(continued from previous page)

```
> 11834906

# Getting it as hex:
in3 -c kovan -hex eth_blockNumber
> 0xb49625

# As part of shell script:
BLOCK_NUMBER=`in3 eth_blockNumber`
```

16.10.2 Using jq to Filter JSON

```
# Get the timestamp of the latest block:
in3 eth_getBlockByNumber latest false | jq -r .timestamp
> 0x5d162a47

# Get the first transaction of the last block:
in3 eth_getBlockByNumber latest true | jq  '.transactions[0]'
> {
   "blockHash": "0xe4edd75bf43cd8e334ca756c4df1605d8056974e2575f5ea835038c6d724ab14",
   "blockNumber": "0x7ac96d",
   "chainId": "0x1",
   "condition": null,
   "creates": null,
   "from": "0x91fdebe2e1b68da999cb7d634fe693359659d967",
   "gas": "0x5208",
   "gasPrice": "0xba43b7400",
   "hash": "0x4b0fe62b30780d089a3318f0e5e71f2b905d62111a4effe48992fcfda36b197f",
   "input": "0x",
   "nonce": "0x8b7",
   "publicKey":
↪"0x17f6413717c12dab2f0d4f4a033b77b4252204bfe4ae229a608ed724292d7172a19758e84110a2a926842457c351f803
↪",
   "r": "0x1d04ee9e31727824a19a4fcd0c29c0ba5dd74a2f25c701bd5fdabbf5542c014c",
   "raw":
↪"0xf86e8208b7850ba43b7400825208947fb38d6a092bbdd476e80f00800b03c3f1b2d332883aefa89df48ed4008026a01c
↪",
   "s": "0x43f8df6c171e51bf05036c8fe8d978e182316785d0aace8ecc56d2add157a635",
   "standardV": "0x1",
   "to": "0x7fb38d6a092bbdd476e80f00800b03c3f1b2d332",
   "transactionIndex": "0x0",
   "v": "0x26",
   "value": "0x3aefa89df48ed400"
  }
```

16.10.3 Calling a Function of a Smart Contract

```
# Without arguments:
in3 -to 0x2736D225f85740f42D17987100dc8d58e9e16252 call "totalServers():uint256"
> 5

# With arguments returning an array of values:
in3 -to 0x2736D225f85740f42D17987100dc8d58e9e16252 call "servers(uint256):(string,
↪address,uint256,uint256,uint256,address)" 1
```

(continues on next page)

(continued from previous page)

```
 > https://in3.slock.it/mainnet/nd-1
 > 0x784bfa9eb182c3a02dbeb5285e3dba92d717e07a
 > 65535
 > 65535
 > 0
 > 0x0000000000000000000000000000000000000000

# With arguments returning an array of values as JSON:
 in3 -to 0x2736D225f85740f42D17987100dc8d58e9e16252 -json call
↪"servers(uint256):(string,address,uint256,uint256,uint256,address)" 1
 > ["https://in3.slock.it/mainnet/nd-4","0xbc0ea09c1651a3d5d40bacb4356fb59159a99564",
↪"0xffff","0xffff","0x00","0x0000000000000000000000000000000000000000"]
```

16.10.4 Sending a Transaction

```
# Sends a transaction to a register server function and signs it with the private key␣
↪given :
in3 -pk mykeyfile.json -to 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1  -gas 1000000 ␣
↪send "registerServer(string,uint256)" "https://in3.slock.it/kovan1" 0xFF
```

16.10.5 Deploying a Contract

```
# Compiling the Solidity code, filtering the binary, and sending it as a transaction␣
↪returning the txhash:
solc --bin ServerRegistry.sol | in3 -gas 5000000 -pk my_private_key.json -d - send

# If you want the address, you would need to wait until the text is mined before␣
↪obtaining the receipt:
solc --bin ServerRegistry.sol | in3 -gas 5000000 -pk my_private_key.json -d - -wait␣
↪send | jq -r .contractAddress
```

CHAPTER 17

API Reference Node/Server

The term in3-server and in3-node are used interchangeably.

Nodes are the backend of Incubed. Each node serves RPC requests to Incubed clients. The node itself runs like a proxy for an Ethereum client (Geth, Parity, etc.), but instead of simply passing the raw response, it will add the required proof needed by the client to verify the response.

To run such a node, you need to have an Ethereum client running where you want to forward the request to. At the moment, the minimum requirement is that this client needs to support `eth_getProof` (see http://eips.ethereum.org/EIPS/eip-1186).

You can create your own docker compose file/docker command using our command line descriptions below. But you can also use our tool in3-server-setup to help you through the process.

17.1 Command-line Arguments

--autoRegistry-capabilities-multiChain If true, this node is able to deliver multiple chains.

--autoRegistry-capabilities-proof If true, this node is able to deliver proofs.

--autoRegistry-capacity Max number of parallel requests.

--autoRegistry-deposit The deposit you want to store.

--autoRegistry-depositUnit Unit of the deposit value.

--autoRegistry-url The public URL to reach this node.

--cache Cache Merkle tries.

--chain ChainId.

--clientKeys A comma-separated list of client keys to use for simulating clients for the watchdog.

--db-database Name of the database.

--db-host Db-host (default: local host).

--db-password Password for db-access.

--db-user Username for the db.

--defaultChain The default chainId in case the request does not contain one.

--freeScore The score for requests without a valid signature.

--handler The implementation used to handle the calls.

--help Output usage information.

--id An identifier used in log files for reading the configuration from the database.

--ipfsUrl The URL of the IPFS client.

--logging-colors If true, colors will be used.

--logging-file The path to the log file.

--logging-host The host for custom logging.

--logging-level Log level.

--logging-name The name of the provider.

--logging-type The module of the provider.

--maxThreads The maximal number of threads running parallel to the processes.

--maxPointsPerMinute The Score for one client able to use within one minute, which is used as DOS-Protection.

--maxBlocksSigned The max number of blocks signed per in3_sign-request

--maxSignatures The max number of signatures to sign per request

--minBlockHeight The minimal block height needed to sign.

--persistentFile The file name of the file keeping track of the last handled blockNumber.

--privateKey The path to the keystore-file for the signer key used to sign blockhashes.

--privateKeyPassphrase The password used to decrypt the private key.

--profile-comment Comments for the node.

--profile-icon URL to an icon or logo of a company offering this node.

--profile-name Name of the node or company.

--profile-noStats If active, the stats will not be shown (default: false).

--profile-url URL of the website of the company.

--profile-prometheus URL of the prometheus gateway to report stats

--registry The address of the server registry used to update the NodeList.

--registryRPC The URL of the client in case the registry is not on the same chain.

--rpcUrl The URL of the client. User can specify multiple clients for higher security and data availability. If multiple URLs are used server will check block hash on all RPC clients before signing. Also server will only switch to another node when any request will fail on previous. Format for using multple clients is: –rpcUrl=http://rpc1.com –rpcUrl=http://rpc2.com

--startBlock BlockNumber to start watching the registry.

--timeout Number of milliseconds needed to wait before a request times out.

--version Output of the version number.

--watchInterval The number of seconds before a new event.

--watchdogInterval Average time between sending requests to the same node. 0 turns it off (default).

17.2 in3-server-setup tool

The in3-server-setup tool can be found both [online](https://in3-setup.slock.it) and on [DockerHub](https://hub.docker.com/r/slockit/in3-server-setup). The DockerHub version can be used to avoid relying on our online service, a full source will be released soon.

The tool can be used to generate the private key as well as the docker-compose file for use on the server.

Note: The below guide is a basic example of how to setup and in3 node, no assurances are made as to the security of the setup. Please take measures to protect your private key and server.

Setting up a server on AWS:

1. Create an account on AWS and create a new EC2 instance
2. Save the key and SSH into the machine with `` `ssh -i "SSH_KEY.pem" user@IP` ``
3. Install docker and docker-compose on the EC2 instance `` `apt-get install docker docker-compose` ``
4. Use scp to transfer the docker-compose file and private key, `` `scp -i "SSH_KEY" FILE user@IP:.` ``
5. Run the Ethereum client, for example parity and allow it to sync
6. Once the client is synced, run the docker-compose file with `` `docker-compose up` ``
7. **Test the in3 node by making a request to the address**

```
curl -X POST -H 'Content-Type:application/json' \
--data '{"id":1,"jsonrpc":"2.0","method":"in3_nodeList", \
"params":[],"in3":{"version": "0x2","chainId":"0x1","verification":"proof
↪"}}' \
    <MY_NODE_URL>
```

8. Consider using tools such as AWS Shield to protect your server from DOS attacks

17.3 Registering Your Own Incubed Node

If you want to participate in this network and register a node, you need to send a transaction to the registry contract, calling *registerServer(string _url, uint _props)*.

To run an Incubed node, you simply use docker-compose:

First run partiy, and allow the client to sync:

```
version: '2'
services:
incubed-parity:
    image: parity:latest                                        # Parity image with␣
↪the proof function implemented.
    command:
```

(continues on next page)

(continued from previous page)

```
    - --auto-update=none                                            # Do not␣
↪automatically update the client.
    - --pruning=archive
    - --pruning-memory=30000                                        # Limit storage.
    - --jsonrpc-experimental                                        # Currently still␣
↪needed until EIP 1186 is finalized.
```

Then run in3 with the below docker-compose file:

```
version: '2'
      services:
      incubed-server:
          image: slockit/in3-server:latest
          volumes:
          - $PWD/keys:/secure                                             # Directory␣
↪where the private key is stored.
          ports:
          - 8500:8500/tcp                                                 # Open the port␣
↪8500 to be accessed by the public.
          command:
          - --privateKey=/secure/myKey.json                               # Internal path␣
↪to the key.
          - --privateKeyPassphrase=dummy                                  # Passphrase to␣
↪unlock the key.
          - --chain=0x1                                                   # Chain (Kovan).
          - --rpcUrl=http://incubed-parity:8545                           # URL of the␣
↪Kovan client.
          - --registry=0xFdb0eA8AB08212A1fFfDB35aFacf37C3857083ca # URL of the␣
↪Incubed registry.
          - --autoRegistry-url=http://in3.server:8500                     # Check or␣
↪register this node for this URL.
          - --autoRegistry-deposit=2                                      # Deposit to␣
↪use when registering.
```

CHAPTER 18

API Reference Solidity

This page contains a list of function for the registry contracts.

18.1 NodeRegistryData functions

18.1.1 adminRemoveNodeFromRegistry

Removes an in3-node from the nodeList

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _signer `address`: the signer

18.1.2 adminSetLogic

Sets the new Logic-contract as owner of the contract.

Development notice:

- only callable by the current Logic-contract / owner
- the `0x00`-address as owner is not supported

Return Parameters:

- true when successful

18.1.3 adminSetNodeDeposit

Sets the deposit of an existing in3-node

Development notice:

- only callable by the NodeRegistryLogic-contract
- used to remove the deposit of a node after he had been convicted

Parameters:

- _signer `address`: the signer of the in3-node
- _newDeposit `uint`: the new deposit

Return Parameters:

- true when successful

18.1.4 adminSetStage

Sets the stage of a signer

Development notice:

- only callable by the current Logic-contract / owner

Parameters:

- _signer `address`: the signer of the in3-node
- *stage* `uint`: the new stage

Return Parameters:

- true when successful

18.1.5 adminSetSupportedToken

Sets a new erc20-token as supported token for the in3-nodes.

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _newToken `address`: the address of the new supported token

Return Parameters:

- true when successful

18.1.6 adminSetTimeout

Sets the new timeout until the deposit of a node can be accessed after he was unregistered.

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _newTimeout `uint`: the new timeout

Return Parameters:

- true when successful

18.1.7 adminTransferDeposit

Transfers a certain amount of ERC20-tokens to the provided address

Development notice:

- only callable by the NodeRegistryLogic-contract
- reverts when the transfer failed

Parameters:

- _to `address`: the address to receive the tokens
- _amount: `uint`: the amount of tokens to be transferred

Return Parameters:

- true when successful

18.1.8 setConvict

Writes a value to te convictMapping to be used later for revealConvict in the logic contract.

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _hash `bytes32`: the data to be written
- _caller `address`: the address for that called convict in the logic-contract

Development notice:

- only callable by the NodeRegistryLogic-contract

18.1.9 registerNodeFor

Registers a new node in the nodeList

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _url `string`: the url of the in3-node
- _props `uint192`: the properties of the in3-node
- _signer `address`: the signer address
- _weight `uit64`: the weight
- _owner `address`: the address of the owner

- _deposit `uint`: the deposit in erc20 tokens
- _stage `uint`: the stage the in3-node should have

Return Parameters:

- true when successful

18.1.10 transferOwnership

Transfers the ownership of an active in3-node

Development notice:

- only callable by the NodeRegistryLogic-contract

Parameters:

- _signer `address`: the signer of the in3-node
- _newOwner `address`: the address of the new owner

Return Parameters:

- true when successful

18.1.11 unregisteringNode

Removes a node from the nodeList

Development notice:

- only callable by the NodeRegistryLogic-contract
- calls `_unregisterNodeInternal()`

Parameters:

- _signer `address`: the signer of the in3-node

Return Parameters:

- true when successful

18.1.12 updateNode

Updates an existing in3-node

Development notice:

- only callable by the NodeRegistryLogic-contract
- reverts when the an updated url already exists

Parameters:

- _signer `address`: the signer of the in3-node
- _url `string`: the new url
- _props `uint192` the new properties
- _weight `uint64` the new weight

- _deposit `uint` the new deposit

Return Parameters:

- true when successful

18.1.13 getIn3NodeInformation

Returns the In3Node-struct of a certain index

Parameters:

- index `uint`: the index-position in the nodes-array

Return Parameters:

- the In3Node-struct

18.1.14 getSignerInformation

Returns the SignerInformation of a signer

Parameters:

- _signer `address`: the signer

Return Parameters: the SignerInformation of a signer

18.1.15 totalNodes

Returns the length of the nodeList

Return Parameters: The length of the nodeList

18.1.16 adminSetSignerInfo

Sets the SignerInformation-struct for a signer

Development notice:

- only callable by the NodeRegistryLogic-contract
- gets used for updating the information after returning the deposit

Parameters:

- _signer `address`: the signer
- _si: `SignerInformation` the struct to be set

Return Parameters:

- true when successful

18.2 NodeRegistryLogic functions

18.2.1 activateNewLogic

Applies a new update to the logic-contract by setting the pending NodeRegistryLogic-contract as owner to the NodeRegistryData-conract

Development notice:

- Only callable after 47 days have passed since the latest update has been proposed

18.2.2 adminRemoveNodeFromRegistry

Removes an malicious in3-node from the nodeList

Development notice:

- only callable by the admin of the smart contract
- only callable in the 1st year after deployment
- ony usable on registered in3-nodes

Parameters:

- _signer `address`: the malicious signer

18.2.3 adminUpdateLogic

Proposes an update to the logic contract which can only be applied after 47 days. This will allow all nodes that don't approve the update to unregister from the registry

Development notice:

- only callable by the admin of the smart contract
- does not allow for the 0x0-address to be set as new logic

Parameters:

- _newLogic `address`: the malicious signer

18.2.4 convict

Must be called before revealConvict and commits a blocknumber and a hash.

Development notice:

- The `v`,`r`,`s` parameters are from the signature of the wrong blockhash that the node provided

Parameters:

- _hash `bytes32`: `keccak256(wrong blockhash, msg.sender, v, r, s)`; used to prevent frontrunning.

18.2.5 registerNode

Registers a new node with the sender as owner

Development notice:

- will call the registerNodeInteral function
- the amount of `_deposit` token have be approved by the signer in order for them to be transferred by the logic contract

Parameters:

- _url `string`: the url of the node, has to be unique
- _props `uint64`: properties of the node
- _weight `uint64`: how many requests per second the node is able to handle
- _deposit `uint`: amount of supported ERC20 tokens as deposit

18.2.6 registerNodeFor

Registers a new node as a owner using a different signer address*

Development notice:

- will revert when a wrong signature has been provided which is calculated by the hash of the url, properties, weight and the owner in order to prove that the owner has control over the signer-address he has to sign a message
- will call the registerNodeInteral function
- the amount of `_deposit` token have be approved by the in3-node-owner in order for them to be transferred by the logic contract

Parameters:

- _url `string`: the url of the node, has to be unique
- _props `uint64`: properties of the node
- _signer `address`: the signer of the in3-node
- _weight `uint64`: how many requests per second the node is able to handle
- _depositAmount `uint`: the amount of supported ERC20 tokens as deposit
- _v `uint8`: v of the signed message
- _r `bytes32`: r of the signed message
- _s `bytes32`: s of the signed message

18.2.7 returnDeposit

Returns the deposit after a node has been removed and it's timeout is over.

Development notice:

- reverts if the deposit is still locked
- reverts when there is nothing to transfer
- reverts when not the owner of the former in3-node

Parameters:

- _signer `address`: the signer-address of a former in3-node

18.2.8 revealConvict

Reveals the wrongly provided blockhash, so that the node-owner will lose its deposit while the sender will get half of the deposit

Development notice:

- reverts when the wrong convict hash (see convict-function) is used
- reverts when the _signer did not sign the block
- reverts when trying to reveal immediately after calling convict
- reverts when trying to convict someone with a correct blockhash
- reverts if a block with that number cannot be found in either the latest 256 blocks or the blockhash registry

Parameters:

- _signer `address`: the address that signed the wrong blockhash
- _blockhash `bytes32`: the wrongly provided blockhash
- _blockNumber `uint`: number of the wrongly provided blockhash
- _v `uint8`: v of the signature
- _r `bytes32`: r of the signature
- _s `bytes32`: s of the signature

18.2.9 transferOwnership

Changes the ownership of an in3-node.

Development notice:

- reverts when the sender is not the current owner
- reverts when trying to pass ownership to `0x0`
- reverts when trying to change ownership of an inactive node

Parameters:

- _signer `address`: the signer-address of the in3-node, used as an identifier
- _newOwner `address`: the new owner

18.2.10 unregisteringNode

A node owner can unregister a node, removing it from the nodeList. Doing so will also lock his deposit for the timeout of the node.

Development notice:

- reverts when not called by the owner of the node
- reverts when the provided address is not an in3-signer

- reverts when node is not active

Parameters:

- _signer `address`: the signer of the in3-node

18.2.11 updateNode

Updates a node by changing its props

Development notice:

- if there is an additional deposit the owner has to approve the tokenTransfer before
- reverts when trying to change the url to an already existing one
- reverts when the signer does not own a node
- reverts when the sender is not the owner of the node

Parameters:

- _signer `address`: the signer-address of the in3-node, used as an identifier
- _url `string`: the url, will be changed if different from the current one
- _props `uint64`: the new properties, will be changed if different from the current one
- _weight `uint64`: the amount of requests per second the node is able to handle
- _additionalDeposit `uint`: additional deposit in supported erc20 tokens

18.2.12 maxDepositFirstYear

Returns the current maximum amount of deposit allowed for registering or updating a node

Return Parameters:

- `uint` the maximum amount of tokens

18.2.13 minDeposit

Returns the current minimal amount of deposit required for registering a new node

Return Parameters:

- `uint` the minimal amount of tokens needed for registering a new node

18.2.14 supportedToken

Returns the current supported ERC20 token-address

Return Parameters:

- `address` the address of the currently supported erc20 token

18.3 BlockHashRegistry functions

18.3.1 searchForAvailableBlock

Searches for an already existing snapshot

Parameters:

- _startNumber `uint`: the blocknumber to start searching
- _numBlocks `uint`: the number of blocks to search for

Return Parameters:

- `uint` returns a blocknumber when a snapshot had been found. It will return 0 if no blocknumber was found.

18.3.2 recreateBlockheaders

Starts with a given blocknumber and its header and tries to recreate a (reverse) chain of blocks. If this has been successful the last blockhash of the header will be added to the smart. contract. It will be checked whether the provided chain is correct by using the reCalculateBlockheaders function.

Development notice:

- only usable when the given blocknumber is already in the smart contract
- function is public due to the usage of a dynamic bytes array (not yet supported for external functions)
- reverts when the chain of headers is incorrect
- reverts when there is not parent block already stored in the contract

Parameters:

- _blockNumber `uint`: the block number to start recreation from
- _blockheaders `bytes[]`: array with serialized blockheaders in reverse order (youngest -> oldest) => (e.g. 100, 99, 98)

18.3.3 saveBlockNumber

Stores a certain blockhash to the state

Development notice:

- reverts if the block can't be found inside the evm

Parameters:

- _blockNumber `uint`: the blocknumber to be stored

18.3.4 snapshot

Stores the currentBlock-1 in the smart contract

18.3.5 getRlpUint

Returns the value from the rlp encoded data

Development notice: *This function is limited to only value up to 32 bytes length!

Parameters:

- _data `bytes`: the rlp encoded data
- _offset `uint`: the offset

Return Parameters:

- value `uint` the value

18.3.6 getParentAndBlockhash

Returns the blockhash and the parent blockhash from the provided blockheader

Parameters:

- _blockheader `bytes`: a serialized (rlp-encoded) blockheader

Return Parameters:

- parentHash `bytes32`
- bhash `bytes32`

18.3.7 reCalculateBlockheaders

Starts with a given blockhash and its header and tries to recreate a (reverse) chain of blocks. The array of the blockheaders have to be in reverse order (e.g. [100,99,98,97]).

Parameters:

- _blockheaders `bytes[]`: array with serialized blockheaders in reverse order, i.e. from youngest to oldest
- _bHash `bytes32`: blockhash of the 1st element of the _blockheaders-array

CHAPTER 19

Concept

To enable smart devices of the internet of things to be connected to the Ethereum blockchain, an Ethereum client needs to run on this hardware. The same applies to other blockchains, whether based on Ethereum or not. While current notebooks or desktop computers with a broadband Internet connection are able to run a full node without any problems, smaller devices such as tablets and smartphones with less powerful hardware or more restricted Internet connection are capable of running a light node. However, many IoT devices are severely limited in terms of computing capacity, connectivity and often also power supply. Connecting an IoT device to a remote node enables even low-performance devices to be connected to blockchain. By using distinct remote nodes, the advantages of a decentralized network are undermined without being forced to trust single players or there is a risk of malfunction or attack because there is a single point of failure.

With the presented Trustless Incentivized Remote Node Network, in short INCUBED, it will be possible to establish a decentralized and secure network of remote nodes, which enables trustworthy and fast access to blockchain for a large number of low-performance IoT devices.

19.1 Situation

The number of IoT devices is increasing rapidly. This opens up many new possibilities for equipping these devices with payment or sharing functionality. While desktop computers can run an Ethereum full client without any problems, small devices are limited in terms of computing power, available memory, Internet connectivity and bandwidth. The development of Ethereum light clients has significantly contributed to the connection of smaller devices with the blockchain. Devices like smartphones or computers like Raspberry PI or Samsung Artik 5/7/10 are able to run light clients. However, the requirements regarding the mentioned resources and the available power supply are not met by a large number of IoT devices.

One option is to run the client on an external server, which is then used by the device as a remote client. However, central advantages of the blockchain technology - decentralization rather than having to trust individual players - are lost this way. There is also a risk that the service will fail due to the failure of individual nodes.

A possible solution for this may be a decentralized network of remote-nodes (netservice nodes) combined with a protocol to secure access.

19.2 Low-Performance Hardware

There are several classes of IoT devices, for which running a full or light client is somehow problematic and a INNN can be a real benefit or even a job enabler:

- **Devices with insufficient calculation power or memory space**

  Today, the majority of IoT devices do not have processors capable of running a full client or a light client. To run such a client, the computer needs to be able to synchronize the blockchain and calculate the state (or at least the needed part thereof).

- **Devices with insufficient power supply**

  If devices are mobile (for instance a bike lock or an environment sensor) and rely on a battery for power supply, running a full or a light light, which needs to be constantly synchronized, is not possible.

- **Devices which are not permanently connected to the Internet**

  Devices which are not permantently connected to the Internet, also have trouble running a full or a light client as these clients need to be in sync before they can be used.

19.3 Scalability

One of the most important topics discussed regarding blockchain technology is scalability. Of course, a working INCUBED does not solve the scaling problems that more transactions can be executed per second. However, it does contribute to providing access to the Ethereum network for devices that could not be integrated into existing clients (full client, light client) due to their lack of performance or availability of a continuous Internet connection with sufficient bandwidth.

19.4 Use Cases

With the following use cases, some realistic scenarios should be designed in which the use of INCUBED will be at least useful. These use cases are intended as real-life relevant examples only to envision the potential of this technology but are by no means a somehow complete list of possible applications.

19.4.1 Publicly Accessible Environment Sensor

Description

An environment sensor, which measures some air quality characteristics, is installed in the city of Stuttgart. All measuring data is stored locally and can be accessed via the Internet by paying a small fee. Also a hash of the current data set is published to the public Ethereum blockchain to validate the integrity of the data.

The computational power of the control unit is restricted to collecting the measuring data from the sensors and storing these data to the local storage. It is able to encrypt or cryptographically sign messages. As this sensor is one of thousands throughout Europe, the energy consumption must be as low as possible. A special low-performance hardware is installed. An Internet connection is provided, but the available bandwidth is not sufficient to synchrone a blockchain client.

Blockchain Integration

The connection to the blockchain is only needed if someone requests the data and sends the validation hash code to the smart contract.

The installed hardware (available computational power) and the requirement to minimize energy consumption disable the installation and operation of a light client without installing addition hardware (like a Samsung Artik 7) as PBCD (Physical Blockchain Connection Device/Ethereum computer). Also, the available Internet bandwidth would need to be enhanced to be able to synchronize properly with the blockchain.

Using a netservice-client connected to the INCUBED can be realized using the existing hardware and Internet connection. No additional hardware or Internet bandwidth is needed. The netservice-client connects to the INCUBED only to send signed messages, to trigger transactions or to request information from the blockchain.

19.4.2 Smart Bike Lock

Description

A smart bike lock which enables sharing is installed on an e-bike. It is able to connect to the Internet to check if renting is allowed and the current user is authorized to open the lock.

The computational power of the control unit is restricted to the control of the lock. Because the energy is provided by the e-bike's battery, the controller runs only when needed in order to save energy. For this reason, it is also not possible to maintain a permanent Internet connection.

Blockchain Integration

Running a light-client on such a platform would consume far too much energy, but even synchronizing the client only when needed would take too much time and require an Internet connection with the corresponding bandwidth, which is not always the case. With a netservice-client running on the lock, a secure connection to the blockchain can be established at the required times, even if the Internet connection only allows limited bandwidth. In times when there is no rental process in action, neither computing power is needed nor data is transferred.

19.4.3 Smart Home - Smart Thermostat

Description

With smart home devices it is possible to realize new business models, e. g. for the energy supply. With smart thermostats it is possible to bill heating energy pay-per-use. During operation, the thermostat must only be connected to the blockchain if there is a heating requirement and a demand exists. Then the thermostat must check whether the user is authorized and then also perform the transactions for payment.

Blockchain Integration

Similar to the cycle lock application, a thermostat does not need to be permanently connected to the blockchain to keep a client in sync. Furthermore, its hardware is not able to run a full or light client. Here, too, it makes sense to use a netservice-client. Such a client can be developed especially for this hardware.

19.4.4 Smartphone App

Description

The range of smartphone apps that can or should be connected to the blockchain is widely diversified. These can be any apps with payment functions, apps that use blockchain as a notary service, apps that control or lend IoT devices, apps that visualize data from the blockchain, and much more.

Often these apps only need sporadic access to the blockchain. Due to the limited battery power and limited data volume, neither a full client nor a light client is really suitable for such applications, as these clients require a permanent connection to keep the blockchain up-to-date.

Blockchain Integration

In order to minimize energy consumption and the amount of data to be transferred, it makes sense to implement smartphone applications that do not necessarily require a permanent connection to the Internet and thus also to the blockchain with a netservice-client. This makes it possible to dispense with a centralized remote server solution, but only have access to the blockchain when it is needed without having to wait long before the client is synchronized.

19.4.5 Advantages

As has already been pointed out in the use cases, there are various advantages that speak in favor of using INCUBED:

- Devices with low computing power can communicate with the blockchain.
- Devices with a poor Internet connection or limited bandwidth can communicate with the blockchain.
- Devices with a limited power supply can be integrated.
- It is a decentralized solution that does not require a central service provider for remote nodes.
- A remote node does not need to be trusted, as there is a verification facility.
- Existing centralized remote services can be easily integrated.
- Net service clients for special and proprietary hardware can be implemented independently of current Ethereum developments.

19.4.6 Challenges

Of course, there are several challenges that need to be solved in order to implement a working INCUBED.

Security

The biggest challenge for a decentralized and trust-free system is to ensure that one can make sure that the information supplied is actually correct. If a full client runs on a device and is synchronized with the network, it can check the correctness itself. A light client can also check if the block headers match, but does not have the transactions available and requires a connection to a full client for this information. A remote client that communicates with a full client via the REST API has no direct way to verify that the answer is correct. In a decentralized network of netservice-nodes whose trustworthiness is not known, a way to be certain with a high probability that the answer is correct is required. The INCUBED system provides the nodes that supply the information with additional nodes that serve as validators.

Business models

In order to provide an incentive to provide nodes for a decentralized solution, any transaction or query that passes through such a node would have to be remunerated with an additional fee for the operator of the node. However, this would further increase the transaction costs, which are already a real problem for micro-payments. However, there are also numerous non-monetary incentives that encourage participation in this infrastructure.

19.5 Architecture

19.5.1 Overview

An INCUBED network consists of several components:

1. The INCUBED registry (later called registry). This is a Smart Contract deployed on the Ethereum Main-Net where all nodes that want to participate in the network must register and, if desired, store a security deposit.
2. The INCUBED or Netservice node (later called node), which are also full nodes for the blockchain. The nodes act as information providers and validators.
3. The INCUBED or Netservice clients (later called client), which are installed e.g. in the IoT devices.
4. Watchdogs who as autonomous authorities (bots) ensure that misbehavior of nodes is uncovered and punished.

Initialization of a Client

Each client gets an initial list of boot nodes by default. Before its first "real" communication with the network, the current list of nodes must be queried as they are registered in the registry (see section [subsec:IN3-Registry-Smart-Contract]). Initially, this can only be done using an invalidated query (see figure [fig:unvalidated request]). In order to have the maximum possible security, this query can and should be made to several or even all boot nodes in order to obtain a valid list with great certainty.

This list must be updated at regular intervals to ensure that the current network is always available.

Unvalidated Requests / Transactions

Unvalidated queries and transactions are performed by the client by selecting one or more nodes from the registry and sending them the request (see figure [fig:unvalidated request]). Although the responses cannot be verified directly, the option to send the request to multiple nodes in parallel remains. The returned results can then be checked for consistency by the client. Assuming that the majority will deliver the correct result (or execute the transaction correctly), this will at least increase the likelihood of receiving the correct response (Proof of Majority).

There are other requests too that can only be returned as an unverified response. This could be the case, for example:

- Current block number (the node may not have synchronized the latest block yet or may be in a micro fork,. . .)
- Information from a block that has not yet been finalized
- Gas price

The multiple parallel query of several nodes and the verification of the results according to the majority principle is a standard functionality of the client. With the number of nodes requested in parallel, a suitable compromise must be made between increased data traffic, effort for processing the data (comparison) and higher security.

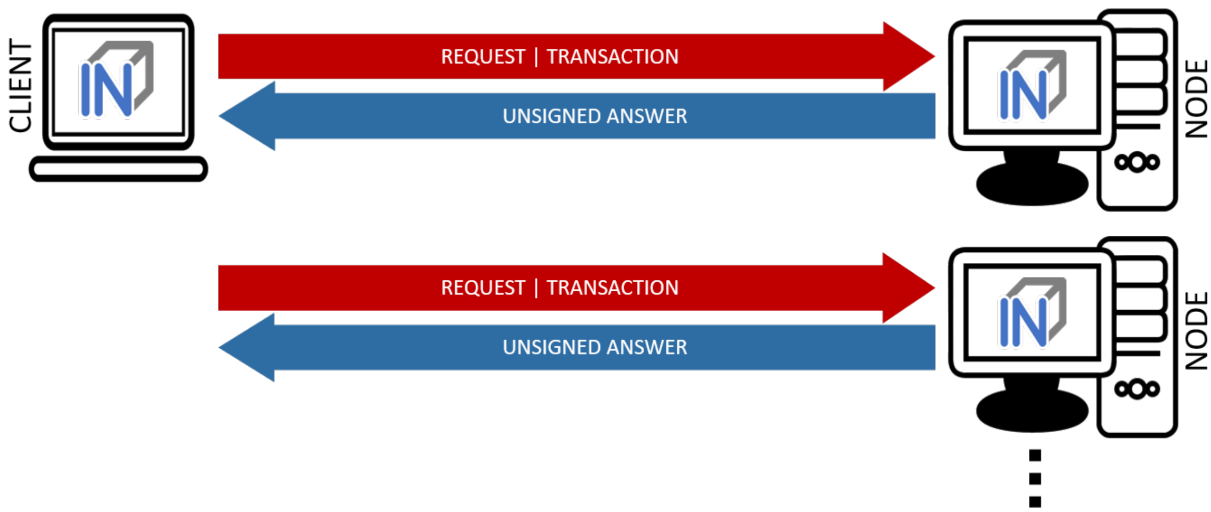
The selection of the nodes to be queried must be made at random. In particular, successive queries should always be sent to different nodes. This way it is not possible, or at least only very difficult, for a possibly misbehaving node to send specific incorrect answers to a certain client, since it cannot be foreseen at any time that the same client will also send a follow-up query to the same node, for example, and thus the danger is high that the misbehavior will be uncovered.

In the case of a misbehavior, the client can blacklist this node or at least reduce the internal rating of this node. However, inconsistent responses can also be provided unintentionally by a node, i.e. without the intention of spreading false information. This can happen, for example, if the node has not yet synchronized the current block or is running on a micro fork. These possibilities must therefore always be taken into consideration when the client "reacts" to such a response.

An unvalidated answer will be returned unsigned. Thus, it is not possible to punish the sender in case of an incorrect response, except that the client can blacklist or downgrade the sender in the above-mentioned form.

Validated Requests

The second form of queries are validated requests. The nodes must be able to provide various verification options and proofs in addition to the result of the request. With validated requests, it is possible to achieve a similar level of security with an INCUBED client as with a light or even full client, without having to blindly trust a centralized middleman (as is the case with a remote client). Depending on the security requirements and the available resources (e.g. computing power), different validations and proofs are possible.



As with an invalidated query, the node to be queried should be selected randomly. However, there are various criteria, such as the deposited security deposit, reliability and performance from previous requests, etc., which can or must also be included in the selection.

Call Parameter

A validated request consists of the parts:

- Actual request
- List of validators
- Proof request
- List of already known validations and proofs (optional).

Return values

The return depends on the request:

- The requested information (signed by the node)
- The signed answers of the validators (block hash) - 1 or more

- The Merkle Proof
- Request for a payment.

Validation

Validation refers to the checking of a block hash by one or more additional nodes. A client cannot perform this check on its own. To check the credibility of a node (information provider), the block hash it returns is checked by one or more independent nodes (validators). If a validator node can detect the malfunction of the originally requested node (delivery of an incorrect block), it can receive its security deposit and the compromised node is removed from the registry. The same applies to a validator node.

Since the network connection and bandwidth of a node is often better than that of a client, and the number of client requests should be as small as possible, the validation requests are sent from the requested node (information provider) to the validators. These return the signed answer, so that there is no possibility for the information provider to manipulate the answer. Since the selection of nodes to act as validators is made only by the client, a potentially malfunctioning node cannot influence it or select a validator to participate in a conspiracy with it.

If the selected validator is not available or does not respond, the client can specify several validators in the request, which are then contacted instead of the failed node. For example, if multiple nodes are involved in a conspiracy, the requested misbehaving node could only send the validation requests to the nodes that support the wrong response.

Proof

The validators only confirm that the block hash of the block from which the requested information originates is correct. The consistency of the returned response cannot be checked in this way.

Optionally, this information can be checked directly by the client. However, this is obligatory, but considerably increases safety. On the other hand, more information has to be transferred and a computationally complex check has to be performed by the client.

When a proof is requested, the node provides the Merkle Tree of the response so that the client can calculate and check the Merkle Root for the result itself.

Payment and Incentives

As an incentive system for the return of verified responses, the node can request a payment. For this, however, the node must guarantee with its security deposit that the answer is correct.

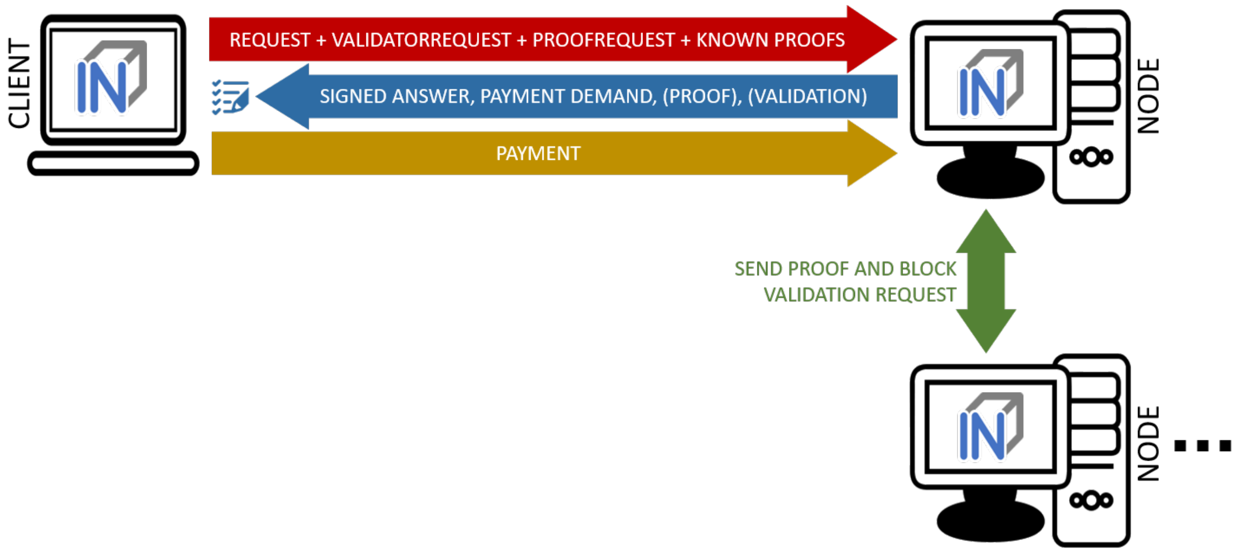
There are two strong incentives for the node to provide the correct response with high performance since it loses its deposit when a validator (wrong block hash) detects misbehavior and is eliminated from the registry, and receives a reward for this if it provides a correct response.

If a client refuses payment after receiving the correctly validated information which it requested, it can be blacklisted or downgraded by the node so that it will no longer receive responses to its requests.

If a node refuses to provide the information for no reason, it is blacklisted by the client in return or is at least downgraded in rating, which means that it may no longer receive any requests and therefore no remuneration in the future.

If the client detects that the Merkle Proof is not correct (although the validated block hash is correct), it cannot attack the node's deposit but has the option to blacklist or downgrade the node to no longer ask it. A node caught this way of misbehavior does not receive any more requests and therefore cannot make any profits.

The security deposit of the node has a decisive influence on how much trust is placed in it. When selecting the node, a client chooses those nodes that have a corresponding deposit (stake), depending on the security requirements (e.g. high value of a transaction). Conversely, nodes with a high deposit will also charge higher fees, so that a market with supply and demand for different security requirements will develop.



19.5.2 IN3-Registry Smart Contract

Each client is able to fetch the complete list including the deposit and other information from the contract, which is required in order to operate. The client must update the list of nodes logged into the registry during initialization and regularly during operation to notice changes (e.g. if a node is removed from the registry intentionally or due to misbehavior detected).

In order to maintain a list of network nodes offering INCUBED-services a smart contract IN3Registry in the Ethereum Main-Net is deployed. This contract is used to manage ownership and deposit for each node.

```
contract ServerRegistry {

    /// server has been registered or updated its registry props or deposit
    event LogServerRegistered(string url, uint props, address owner, uint deposit);

    ///  a caller requested to unregister a server.
    event LogServerUnregisterRequested(string url, address owner, address caller);

    /// the owner canceled the unregister-proccess
    event LogServerUnregisterCanceled(string url, address owner);

    /// a Server was convicted
    event LogServerConvicted(string url, address owner);

    /// a Server is removed
    event LogServerRemoved(string url, address owner);

    struct In3Server {
        string url;  // the url of the server
        address owner; // the owner, which is also the key to sign blockhashes
        uint deposit; // stored deposit
        uint props; // a list of properties-flags representing the capabilities of␣
↪the server

        // unregister state
        uint128 unregisterTime; // earliest timestamp in to to call unregister
        uint128 unregisterDeposit; // Deposit for unregistering
```

(continues on next page)

(continued from previous page)

```
        address unregisterCaller; // address of the caller requesting the unregister
    }

    /// server list of incubed nodes
    In3Server[] public servers;

    /// length of the serverlist
    function totalServers() public view returns (uint) ;

    /// register a new Server with the sender as owner
    function registerServer(string _url, uint _props) public payable;

    /// updates a Server by adding the msg.value to the deposit and setting the props␣
↪
    function updateServer(uint _serverIndex, uint _props) public payable;

    /// this should be called before unregistering a server.
    /// there are 2 use cases:
    /// a) the owner wants to stop offering the service and remove the server.
    ///    in this case he has to wait for one hour before actually removing the␣
↪server.
    ///    This is needed in order to give others a chance to convict it in case this␣
↪server signs wrong hashes
    /// b) anybody can request to remove a server because it has been inactive.
    ///    in this case he needs to pay a small deposit, which he will lose
    //       if the owner become active again
    //       or the caller will receive 20% of the deposit in case the owner does not␣
↪react.
    function requestUnregisteringServer(uint _serverIndex) payable public;

    /// this function must be called by the caller of the requestUnregisteringServer-
↪function after 28 days
    /// if the owner did not cancel, the caller will receive 20% of the server␣
↪deposit + his own deposit.
    /// the owner will receive 80% of the server deposit before the server will be␣
↪removed.
    function confirmUnregisteringServer(uint _serverIndex) public ;

    /// this function must be called by the owner to cancel the unregister-process.
    /// if the caller is not the owner, then he will also get the deposit paid by the␣
↪caller.
    function cancelUnregisteringServer(uint _serverIndex) public;


    /// convicts a server that signed a wrong blockhash
    function convict(uint _serverIndex, bytes32 _blockhash, uint _blocknumber, uint8 _
↪v, bytes32 _r, bytes32 _s) public ;

}
```

To register, the owner of the node needs to provide the following data:

- **props** : a bitmask holding properties like.
- **url** : the public url of the server.
- **msg.value** : the value sent during this transaction is stored as deposit in the contract.
- **msg.sender** : the sender of the transaction is set as owner of the node and therefore able to manage it at any

given time.

Deposit

The deposit is an important incentive for the secure operation of the INCUBED network. The risk of losing the deposit if misconduct is detected motivates the nodes to provide correct and verifiable answers.

The amount of the deposit can be part of the decision criterion for the clients when selecting the node for a request. The "value" of the request can therefore influence the selection of the node (as information provider). For example, a request that is associated with a high value may not be sent to a node that has a very low deposit. On the other hand, for a request for a dashboard, which only provides an overview of some information, the size of the deposit may play a subordinate role.

19.5.3 Netservice-Node

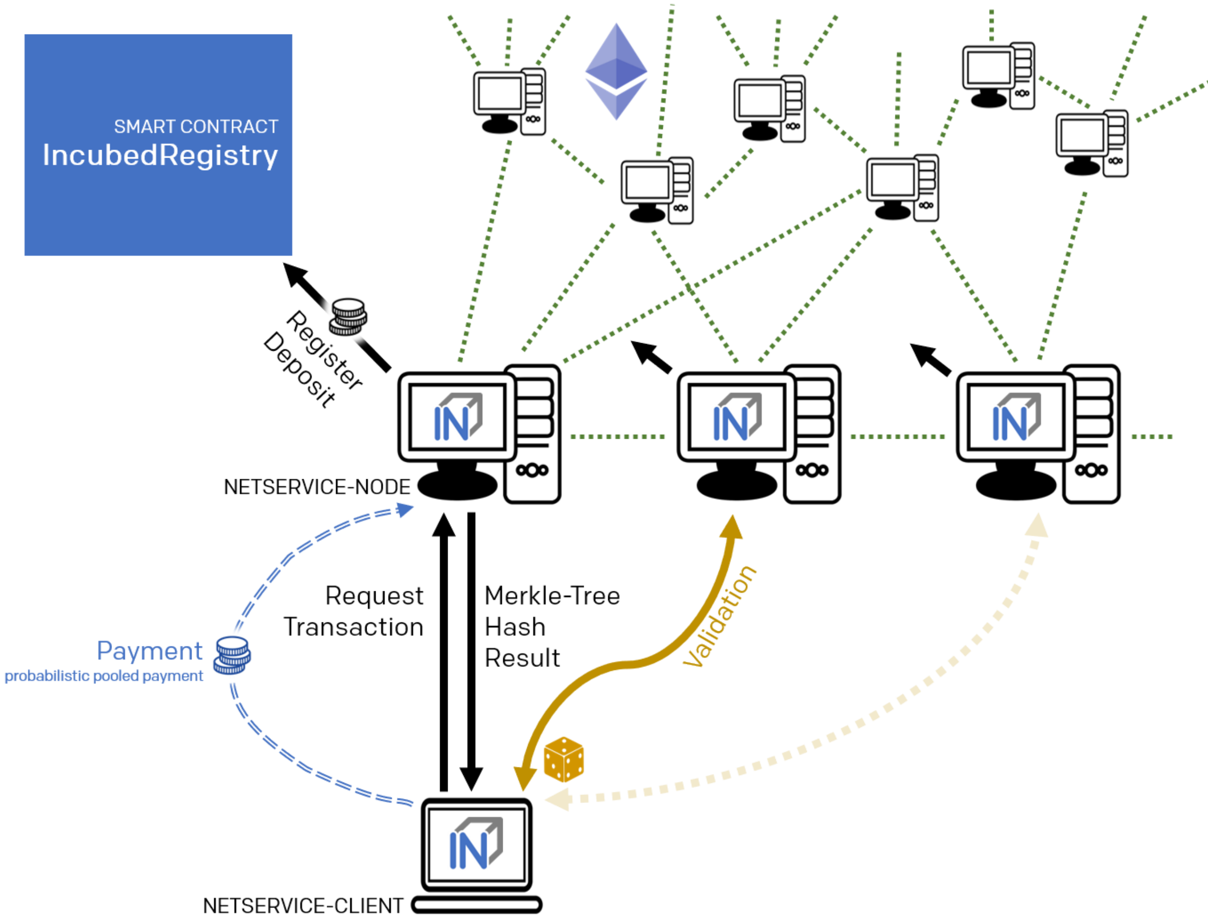
The net service node (short: node) is the communication interface for the client to the blockchain client. It can be implemented as a separate application or as an integrated module of a blockchain client (such as Geth or Parity).

Nodes must provide two different services:

- Information Provider
- Validator.

Information Provider

A client directly addresses a node (information provider) to retrieve the desired information. Similar to a remote client, the node interacts with the blockchain via its blockchain client and returns the information to the requesting client. Furthermore, the node (information provider) provides the information the client needs to verify the result of the query (validation and proof). For the service, it can request payment when it returns a validated response.



If an information provider is found to return incorrect information as a validated response, it loses its deposit and is removed from the registry. It can be transferred by a validator or watchdog.

Validator

The second service that a node has to provide is validation. When a client submits a validated request to the information provider, it also specifies the node(s) that are designated as validators. Each node that is logged on to the registry must also accept the task as validator.

If a validator is found to return false information as validation, it loses its deposit and is removed from the registry. It can be transferred by another validator or a watchdog.

Watchdog

Watchdogs are independent bots whose random validators logged in to the registry are checked by specific queries to detect misbehavior. In order to provide an incentive for validator activity, watchdogs can also deliberately pretend misbehavior and thus give the validator the opportunity to claim the security deposit.

19.5.4 Netservice-Client

The netservice client (short client) is the instance running on the device that needs the connection to the blockchain. It communicates with the nodes of the INCUBED network via a REST API.

The client can decide autonomously whether it wants to request an unvalidated or a validated answer (see section…). In addition to communicating with the nodes, the client has the ability to verify the responses by evaluating the majority (unvalidated request) or validations and proofs (validated requests).

The client receives the list of available nodes of the INCUBED network from the registry and ensures that this list is always kept up-to-date. Based on the list, the client also manages a local reputation system of nodes to take into account performance, reliability, trustworthiness and security when selecting a node.

A client can communicate with different blockchains at the same time. In the registry, nodes of different blockchains (identified by their ID) are registered so that the client can and must filter the list to identify the nodes that can process (and validate, if necessary) its request.

Local Reputation System

The local reputations system aims to support the selection of a node.

The reputation system is also the only way for a client to blacklist nodes that are unreliable or classified as fraudulent. This can happen, for example, in the case of an unvalidated query if the results of a node do not match those of the majority, or in the case of validated queries, if the validation is correct but the proof is incorrect.

Performance-Weighting

In order to balance the network, each client may weight each node by:

$$weight = \frac{\max(\lg(deposit), 1)}{\max(avgResponseTime, 100)}$$

Based on the weight of each node a random node is chosen for each request. While the deposit is read by the contract, the avgResponseTime is managed by the client himself. The does so by measuring the time between request and response and calculate the average (in ms) within the last 24 hours. This way the load is balanced and faster servers will get more traffic.

19.5.5 Payment / Incentives

To build an incentive-based network, it is necessary to have appropriate technologies to process payments. The payments to be made in INCUBED (e.g. as a fee for a validated answer) are, without exception micro payments (other than the deposit of the deposit, which is part of the registration of a node and which is not mentioned here, however). When designing a suitable payment solution, it must therefore be ensured that a reasonable balance is always found between the actual fee, transaction costs and transaction times.

Direct Transaction Payment

Direct payment by transaction is of course possible, but this is not possible due to the high transaction costs. Exceptions to this could be transactions with a high value, so that corresponding transaction costs would be acceptable.

However, such payments are not practical for general use.

State Channels

State channels are well-suited for the processing of micropayments. A decisive point of the protocol is that the node must always be selected randomly (albeit weighted according to further criteria). However, it is not practical for a client to open a separate state channel (including deposit) with each potential node that it wants to use for a request. To establish a suitable micropayment system based on state channels, a state channel network such as Raiden is required. If enough partners are interconnected in such a network and a path can be found between two partners, payments can also be exchanged between these participants.

Probabilistic Payment

Another way of making small payments is probabilistic micropayments. The idea is based on issuing probabilistic lottery tickets instead of very small direct payments, which, with a certain probability, promise to pay out a higher amount. The probability distribution is adjusted so that the expected value corresponds to the payment to be made.

For a probabilistic payment, an amount corresponding to the value of the lottery ticket is deposited. Instead of direct payment, tickets are now issued that have a high likelihood of winning. If a ticket is not a winning ticket, it expires and does not entitle the recipient to receive a payment. Winning tickets, on the other hand, entitle the recipient to receive the full value of the ticket.

Since this value is so high that a transaction is worthwhile, the ticket can be redeemed in exchange for a payment.

Probabilistic payments are particularly suitable for combining a continuous, preferably evenly distributed flow of small payments into individual larger payments (e.g. for streaming data).

Similar to state channels, a type of payment channel is created between two partners (with an appropriate deposit).

For the application in the INCUBED protocol, it is not practical to establish individual probabilistic payment channels between each client and requested node, since on the one hand the prerequisite of a continuous and evenly distributed payment stream is not given and, on the other hand, payments may be very irregularly required (e.g. if a client only rarely sends queries).

The analog to a state channel network is pooled probabilistic payments. Payers can be pooled and recipients can also be connected in a pool, or both.

19.6 Scaling

The interface between client and node is independent of the blockchain with which the node communicates. This allows a client to communicate with multiple blockchains / networks simultaneously as long as suitable nodes are registered in the registry.

For example, a payment transaction can take place on the Ethereum Mainnet and access authorization can be triggered in a special application chain.

19.6.1 Multi Chain Support

Each node may support one or more network or chains. The supported list can be read by filtering the list of all servers in the contract.

The ChainId refers to a list based on EIP-155. The ChainIds defined there will be extended by enabling even custom chains to register a new chainId.

19.6.2 Conclusion

INCUBED establishes a decentralized network of validatable remote nodes, which enables IoT devices in particular to gain secure and reliable access to the blockchain. The demands on the client's computing and storage capacity can be reduced to a minimum, as can the requirements on connectivity and network traffic.

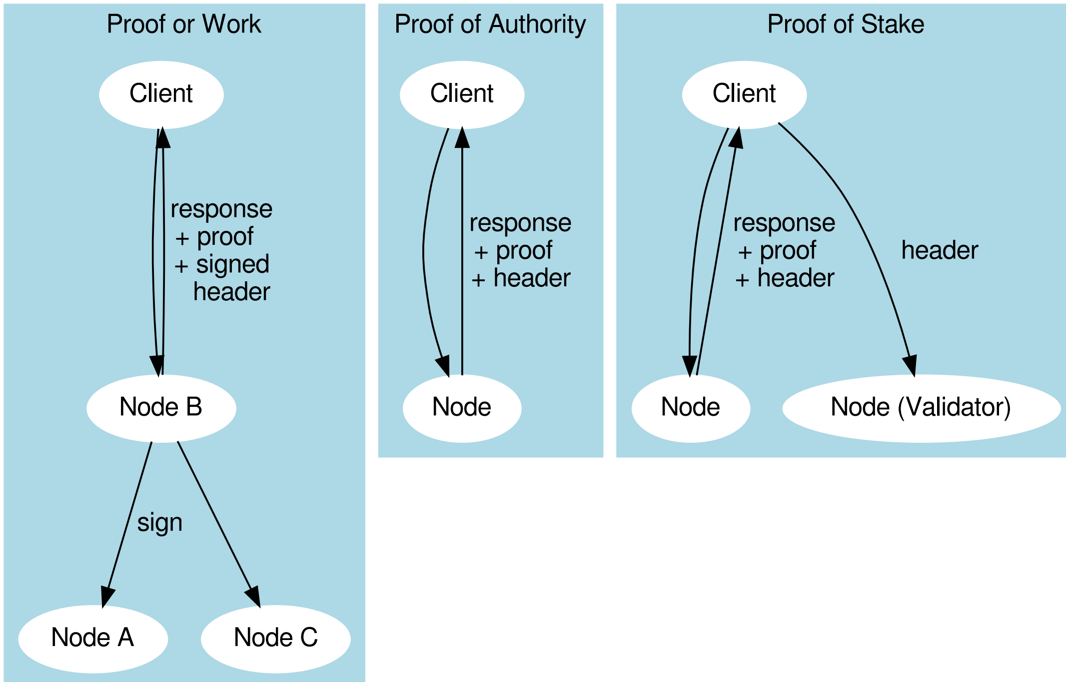
INCUBED also provides a platform for scaling by allowing multiple blockchains to be accessed in parallel from the same client. Although INCUBED is designed in the first instance for the Ethereum network (and other chains using the Ethereum protocol), in principle other networks and blockchains can also be integrated, as long as it is possible to realize a node that can work as information provider (incl. proof) and validator.

CHAPTER 20

Ethereum

20.1 Blockheader Verification

Since all proofs always include the blockheader it is crucial to verify the correctness of these data as well. But verification depends on the consensus of the underlying blockchain. (For details, see Ethereum Verification and MerkleProof.)



20.2 Proof of Work

Currently, the public chain uses proof of work. This makes it very hard to verify the header since anybody can produce such a header. So the only way to verify that the block in question is an accepted block is to let registered nodes sign the blockhash. If they are wrong, they lose their previously stored deposit. For the client, this means that the required security depends on the deposit stored by the nodes.

This is why a client may be configured to require multiple signatures and even a minimal deposit:

```
client.sendRPC('eth_getBalance', [account, 'latest'], chain, {
  minDeposit: web3.utils.toWei(10,'ether'),
  signatureCount: 3
})
```

The `minDeposit` lets the client preselect only nodes with at least that much deposit. The `signatureCount` asks for multiple signatures and so increases the security.

Since most clients are small devices with limited bandwith, the client is not asking for the signatures directly from the nodes but, rather, chooses one node and lets this node run a subrequest to get the signatures. This means not only fewer requests for the clients but also that at least one node checks the signatures and “convicts” another if it lied.

20.3 Proof of Authority

The good thing about proof of authority is that there is already a signature included in the blockheader. So if we know who is allowed to sign a block, we do not need an additional blockhash signed. The only critical information we rely on is the list of validators.

Currently, there are two consensus algorithms:

20.3.1 Aura

Aura is only used by Parity, and there are two ways to configure it:

- **static list of nodes** (like the Kovan network): in this case, the validatorlist is included in the chain-spec and cannot change, which makes it very easy for a client to verify blockheaders.
- **validator contract**: a contract that offers the function `getValidators()`. Depending on the chain, this contract may contain rules that define how validators may change. But this flexibility comes with a price. It makes it harder for a client to find a secure way to detect validator changes. This is why the proof for such a contract depends on the rules laid out in the contract and is chain-specific.

20.3.2 Clique

Clique is a protocol developed by the Geth team and is now also supported by Parity (see Görli testnet).

Instead of relying on a contract, Clique defines a protocol of how validator nodes may change. All votes are done directly in the blockheader. This makes it easier to prove since it does not rely on any contract.

The Incubed nodes will check all the blocks for votes and create a `validatorlist` that defines the validatorset for any given blockNumber. This also includes the proof in form of all blockheaders that either voted the new node in or out. This way, the client can ask for the list and automatically update the internal list after it has verified each blockheader and vote. Even though malicious nodes cannot forge the signatures of a validator, they may skip votes in the validatorlist. This is why a validatorlist update should always be done by running multiple requests and merging them together.

20.4 Ethereum Verification

The Incubed is also often called Minimal Verifying Client because it may not sync, but still is able to verify all incoming data. This is possible since ethereum is based on a technology allowing to verify almost any value.

Our goal was to verify at least all standard `eth_...` rpc methods as described in the Specification.

In order to prove something, you always need a starting value. In our case this is the BlockHash. Why do we use the BlockHash? If you know the BlockHash of a block, you can easily verify the full BlockHeader. And since the BlockHeader contains the stateRoot, transationRoot and receiptRoot, these can be verified as well. And the rest will simply depend on them.

There is also another reason why the BlockHash is so important. This is the only value you are able to access from within a SmartContract, because the evm supports a OpCode (`BLOCKHASH`), which allows you to read the last 256 Blockhashes, which gives us the chance to even verify the blockhash onchain.

Depending on the method, different proofs are needed, which are described in this document.

- ***Block Proof*** - verifies the content of the BlockHeader
- ***Transaction Proof*** - verifies the input data of a transaction

- ***Receipt Proof*** - verifies the outcome of a transaction
- ***Log Proof*** - verifies the response of `eth_getPastLogs`
- ***Account Proof*** - verifies the state of an account
- ***Call Proof*** - verifies the result of a `eth_call` - response

20.4.1 BlockProof

BlockProofs are used whenever you want to read data of a Block and verify them. This would be:

- eth_getBlockTransactionCountByHash
- eth_getBlockTransactionCountByNumber
- eth_getBlockByHash
- eth_getBlockByNumber

The `eth_getBlockBy...` methods return the Block-Data. In this case all we need is somebody verifying the blockhash, which is don by requiring somebody who stored a deposit and would lose it, to sign this blockhash.

The Verification is then simply by creating the blockhash and comparing this to the signed one.

The Blockhash is calculated by serializing the blockdata with rlp and hashing it:

```
blockHeader = rlp.encode([
  bytes32( parentHash ),
  bytes32( sha3Uncles ),
  address( miner || coinbase ),
  bytes32( stateRoot ),
  bytes32( transactionsRoot ),
  bytes32( receiptsRoot || receiptRoot ),
  bytes256( logsBloom ),
  uint( difficulty ),
  uint( number ),
  uint( gasLimit ),
  uint( gasUsed ),
  uint( timestamp ),
  bytes( extraData ),

  ... sealFields
    ? sealFields.map( rlp.decode )
    : [
      bytes32( b.mixHash ),
      bytes8( b.nonce )
    ]
])
```

For POA-Chains the blockheader will use the `sealFields` (instead of mixHash and nonce) which are already rlp-encoded and should be added as raw data when using rlp.encode.

```
if (keccak256(blockHeader) !== singedBlockHash)
  throw new Error('Invalid Block')
```

In case of the `eth_getBlockTransactionCountBy...` the proof contains the full blockHeader already serialized + all transactionHashes. This is needed in order to verify them in a merkleTree and compare them with the `transactionRoot`

20.4.2 Transaction Proof

TransactionProofs are used for the following transaction-methods:

- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex

In order to verify we need :

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)
2. serialize the transaction-data

```
transaction = rlp.encode([
  uint( tx.nonce ),
  uint( tx.gasPrice ),
  uint( tx.gas || tx.gasLimit ),
  address( tx.to ),
  uint( tx.value ),
  bytes( tx.input || tx.data ),
  uint( tx.v ),
  uint( tx.r ),
  uint( tx.s )
])
```

1. verify the merkleProof of the transaction with

```
verifyMerkleProof(
  blockHeader.transactionRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transaction /* expected value */
)
```

The Proof-Data will look like these:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "blockHash": "0xf1a2fd6a36f27950c78ce559b1dc4e991d46590683cb8cb84804fa672bca395b",
    "blockNumber": "0xca",
    "from": "0x7e5f4552091a69125d5dfcb7b8c2659029395bdf",
    "gas": "0x55f0",
    "gasPrice": "0x0",
    "hash": "0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b",
    "input": "0x00",
    "value": "0x3e8"
    ...
  },
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block": "0xf901e6a040997a53895b48...", // serialized blockheader
      "merkleProof": [  /* serialized nodes starting with the root-node */

↪"f868822080b863f86136808255f0942b5ad5c4795c026514f8317c7a215e218dccd6cf8283e8001ca0dc967310342af504
↪"
```

(continues on next page)

(continued from previous page)

```
      ],
      "txIndex": 0,
      "signatures": [...]
    }
  }
}
```

20.4.3 Receipt Proof

Proofs for the transactionReceipt are used for the following transaction-method:

- eth_getTransactionReceipt

In order to verify we need :

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)
2. serialize the transaction receipt

```
transactionReceipt = rlp.encode([
  uint( r.status || r.root ),
  uint( r.cumulativeGasUsed ),
  bytes256( r.logsBloom ),
  r.logs.map(l => [
    address( l.address ),
    l.topics.map( bytes32 ),
    bytes( l.data )
  ])
].slice(r.status === null && r.root === null ? 1 : 0))
```

1. verify the merkleProof of the transaction receipt with

```
verifyMerkleProof(
  blockHeader.transactionReceiptRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transactionReceipt /* expected value */
)
```

1. Since the merkle-Proof is only proving the value for the given transactionIndex, we also need to prove that the transactionIndex matches the transactionHash requested. This is done by adding another MerkleProof for the Transaction itself as described in the *Transaction Proof*

20.4.4 Log Proof

Proofs for logs are only for the one rpc-method:

- eth_getLogs

Since logs or events are based on the TransactionReceipts, the only way to prove them is by proving the Transaction-Receipt each event belongs to.

That's why this proof needs to provide

- all blockheaders where these events occured

- all TransactionReceipts + their MerkleProof of the logs
- all MerkleProofs for the transactions in order to prove the transactionIndex

The Proof data structure will look like this:

```
Proof {
  type: 'logProof',
  logProof: {
    [blockNr: string]: {  // the blockNumber in hex as key
      block : string  // serialized blockheader
      receipts: {
        [txHash: string]: {  // the transactionHash as key
          txIndex: number // transactionIndex within the block
          txProof: string[] // the merkle Proof-Array for the transaction
          proof: string[] // the merkle Proof-Array for the receipts
        }
      }
    }
  }
}
```

In order to verify we need :

1. deserialize each blockheader and compare the blockhash with the signed hashes. (See *BlockProof*)
2. for each blockheader we verify all receipts by using

```
verifyMerkleProof(
  blockHeader.transactionReceiptRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transactionReceipt /* expected value */
)
```

1. The resulting values are the receipts. For each log-entry, we are comparing the verified values of the receipt with the returned logs to ensure that they are correct.

20.4.5 Account Proof

Prooving an account-value applies to these functions:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt

eth_getProof

For the Transaction or Block Proofs all needed data can be found in the block itself and retrieved through standard rpc calls, but if we want to approve the values of an account, we need the MerkleTree of the state, which is not accessable through the standard rpc. That's why we have created a EIP to add this function and also implemented this in geth and parity:

- parity (Status: pending pull request) - Docker
- geth (Status: pending pull request) - Docker

This function accepts 3 parameter :

1. `account` - the address of the account to proof
2. `storage` - a array of storage-keys to include in the proof.
3. `block` - integer block number, or the string "latest", "earliest" or "pending"

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "eth_getProof",
  "params": [
    "0x7F0d15C7FAae65896648C8273B6d7E43f58Fa842",
    [  "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421" ],
    "latest"
  ]
}
```

The result will look like this:

```
{
  "jsonrpc": "2.0",
  "result": {
    "accountProof": [
      "0xf90211a...0701bc80",
      "0xf90211a...0d832380",
      "0xf90211a...5fb20c80",
      "0xf90211a...0675b80",
      "0xf90151a0...ca08080"
    ],
    "address": "0x7f0d15c7faae65896648c8273b6d7e43f58fa842",
    "balance": "0x0",
    "codeHash": "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
    "nonce": "0x0",
    "storageHash": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
↪",
    "storageProof": [
      {
        "key": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
        "proof": [
          "0xf90211a...0701bc80",
          "0xf90211a...0d832380"
        ],
        "value": "0x1"
      }
    ]
  },
  "id": 1
}
```

In order to run the verification the blockheader is needed as well.

The Verification of such a proof is done in the following steps:

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)
2. Serialize the account, which holds the 4 values:

```
account = rlp.encode([
  uint( nonce),
  uint( balance),
  bytes32( storageHash || ethUtil.KECCAK256_RLP),
  bytes32( codeHash || ethUtil.KECCAK256_NULL)
])
```

1. verify the merkle Proof for the account using the stateRoot of the blockHeader:

```
verifyMerkleProof(
 block.stateRoot, // expected merkle root
 util.keccak(accountProof.address), // path, which is the hashed address
 accountProof.accountProof.map(bytes), // array of Buffer with the merkle-proof-data
 isNotExistend(accountProof) ? null : serializeAccount(accountProof), // the expected␣
↪serialized account
)
```

In case the account does exist yet, (which is the case if `none == startNonce` and `codeHash == '0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470'`), the proof may end with one of these nodes:

- the last node is a branch, where the child of the next step does not exist.
- the last node is a leaf with different relative key

Both would prove, that this key does not exist.

1. Verify each merkle Proof for the storage using the storageHash of the account:

```
verifyMerkleProof(
  bytes32( accountProof.storageHash ),   // the storageRoot of the account
  util.keccak(bytes32(s.key)),  // the path, which is the hash of the key
  s.proof.map(bytes), // array of Buffer with the merkle-proof-data
  s.value === '0x0' ? null : util.rlp.encode(s.value) // the expected value or none␣
↪to proof non-existence
))
```

20.4.6 Call Proof

Call Proofs are used whenever you are calling a read-only function of smart contract:

- eth_call

Verifying the result of a `eth_call` is a little bit more complex. Because the response is a result of executing opcodes in the vm. The only way to do so, is to reproduce it and execute the same code. That's why a Call Proof needs to provide all data used within the call. This means :

- all referred accounts including the code (if it is a contract), storageHash, nonce and balance.
- all storage keys, which are used (This can be found by tracing the transaction and collecting data based on th `SLOAD`-opcode)
- all blockdata, which are referred at (besides the current one, also the `BLOCKHASH`-opcodes are referring to former blocks)

For Verifying you need to follow these steps:

1. serialize all used blockheaders and compare the blockhash with the signed hashes. (See *BlockProof*)
2. Verify all used accounts and their storage as showed in *Account Proof*

3. create a new VM with a MerkleTree as state and fill in all used value in the state:

```
// create new state for a vm
const state = new Trie()
const vm = new VM({ state })

// fill in values
for (const adr of Object.keys(accounts)) {
  const ac = accounts[adr]

  // create an account-object
  const account = new Account([ac.nonce, ac.balance, ac.stateRoot, ac.codeHash])

  // if we have a code, we will set the code
  if (ac.code) account.setCode( state, bytes( ac.code ))

  // set all storage-values
  for (const s of ac.storageProof)
    account.setStorage( state, bytes32( s.key ), rlp.encode( bytes32( s.value )))

  // set the account data
  state.put( address( adr ), account.serialize())
}

// add listener on each step to make sure it uses only values found in the proof
vm.on('step', ev => {
   if (ev.opcode.name === 'SLOAD') {
      const contract = toHex( ev.address ) // address of the current code
      const storageKey = bytes32( ev.stack[ev.stack.length - 1] ) // last element on the stack is the key
      if (!getStorageValue(contract, storageKey))
        throw new Error(`incomplete data: missing key ${storageKey}`)
   }
   /// ... check other opcodes as well
})

// create a transaction
const tx = new Transaction(txData)

// run it
const result = await vm.runTx({ tx, block: new Block([block, [], []]) })

// use the return value
return result.vm.return
```

In the future we will be using the same approach to verify calls with ewasm.

CHAPTER 21

Bitcoin

Bitcoin may be a complete different chain but there are ways to verify a Bitcoin block header within an Ethereum Smart Contract and Bitcoin data in general on the client-side as well. This requires a little bit more effort but you can use all the features of Incubed.

21.1 Concept

For the verification of Bitcoin we make use of the Simplified Payment Verification (SPV) proposed in the Bitcoin paper by Satoshi Nakamoto.

> It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it. As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network.

In contrast to SPV-clients an Incubed client does not keep a copy of all block headers, instead the client is stateless and only requests required block headers. We are following a simple process: A client requests certain data, the server sends a response with proof data in adition to the actual result, the client verifies the result by using the proof data. We rely on the fact that it is extremly expensive to deliver a wrong block (wrong data) which still has following blocks referring the wrong block (i.e. delivering a chain of fake-blocks). This does not really work for very old blocks. Beside the very low difficulty at this time, the miner has many years of time to pre-mine a wrong chain of blocks. Therefore, a different approach is required which will be explained here

21.1.1 Bitcoin Block Header

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Parent Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Bits	The Proof-of-Work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

21.1.2 Finality in Bitcoin

In terms of Bitcoin, finality is the assurance or guarantee that a block and its included transactions will not be revoked once committed to the blockchain. Bitcoin uses a probabilistic finality in which the probability that a block will not be reverted increases as the block sinks deeper into the chain. The deeper the block, the more likely that the fork containing that block is the longest chain. After being 6 blocks deep into the Bitcoin blockchain it is very unlikely (but not impossible) for that block to be reverted. (For more information see here)

21.1.3 Mining in Bitcoin

The process of trying to add a new block of transactions to the Bitcoin blockchain is called *mining*. Miners are competing in a network-wide competition, each trying to find a new block faster than anyone else. The first miner who finds a block broadcasts it across the network and other miners are adding it to their blockchain after verifying the block. Miners restart the mining-process after a new block was added to the blockchain to build on top of this block. As a result, the blockchain is constantly growing – one block every 10 minutes on average.

> But how can miners *find* a block?

They start by filling a candidate block with transactions from their memory pool. Next they construct a block header for this block, which is a summary of all the data in the block including a reference to a block that is already part of the blockchain (known as the parent hash). Now the actual mining happens: miners put the block header through the SHA256 hash function and hope that the resulting hash is below the current target. If this is not the case, miners keep trying by incrementing a number in the block header resulting in a completely different hash. This process is referred to as proof-of-work.

21.1.4 Difficulty Adjustment Period

This section is important to understand how the adjustment of the difficulty (and therefore the target) works. The knowledge of this section serves as the basis for the remaining part.

The white paper of Bitcoin specifies the block time as 10 minutes. Due to the fact that Bitcoin is a decentralized network that can be entered and exited by miners at any time, the computing power in the network constantly changes depending on the number of miners and their computing power. In order to still achieve an average block time of 10 minutes a mechanism to adjust the difficulty of finding a block is required: the `difficulty`.

The adjustment of the difficulty happens every 2016 blocks - roughly every two weeks and (which is one epoch/period). Since Bitcoin is a decentralized network there is no authority which adjusts the difficulty. Instead every miner calculates the expected time to mine 2016 blocks (20160 minutes) and compares it with the actual time it took to mine the last 2016 blocks (using timestamps). The difficulty increases when the blocks were mined faster than expected and vice versa. Although the computing power increased heavily since the introduction of Bitcoin in 2009 the average block time is still 10 minutes due to this mechanism.

> What is the difference between the `difficulty` and the `target`?

The difficulty is a big number used for the adjustment process. The target is used for the mining process and for the verification of a block hash. As mentioned above the hash of a block has to be smaller than the target to be accepted across the network. The target can be calculated using the difficulty and the constant value `targetmax`:

```
target = targetmax / difficulty

targetmax = 0x00000000FFFF0000000000000000000000000000000000000000000000000000
```

21.2 Security Calculation

How secure is the Incubed Bitcoin Verification?

21.2.1 Blocks Before 227,836 (BIP34)

The verification of blocks before BIP34 relies on checkpoints as explained here.

Although one checkpoint is only 12 bytes in total, it provides a security of 16 bytes because we know that the first 4 bytes are *always* zeros. An attacker has to test 2128 possibilities to (propably) find a hash whose leading 16 bytes are equal to the checkpoint's.

With a current total hash rate of 120 EH/s in the Bitcoin network:

```
2^128 = 3.4 * 10^38 possible hashes

3.4 * 10^38 H / 120 EH/s = 2,835,686,391,007,820,529 s

= 89,919,025,590 years
```

It would take up to **89,919,025,590** years if the whole bitcoin network with its total hash rate would try to find such a hash.

> Does the security increase if the requested block is further away from a checkpoint?

Slighlty - but actually not. The further away a requested block is from a checkpoint, the more proof-of-work has to be done. Due to the low difficulty in the early days of Bitcoin this is not a problem with the today's computing power. Solving the proof-of-work does not really have to be taken into account - because this is "nothing" compared to the many years to brute force a hash whose leading 16 bytes are equal to the checkpoint's. Therefore, the security does not really increase with a greater distance to a checkpoint.

21.2.2 Blocks After 227,836 (BIP34)

The highest risk is a situation, where a malicious node could provide a manipulated or fake block header (i.e. changing the data to gain benefits) and finality block headers which fullfill the rules but are not actually valid (i.e. not part of the longest chain / chain of fake-blocks). The client would trust this data in case he has no other information to check against. The following calculation outlines the security (in terms of $) when the client is requesting one of the newer blocks and 6 finality headers. This results in a total of 7 fake-blocks that an atacker has to calculate to fool the client. The calculation is based on assumptions and averages.

Assume that the attacker has 10% of the total mining power. This would mean he needs around 100 minutes to mine 1 block (average block time of Bitcoin is 10 minutes) and around 700 minutes to mine 7 blocks. While mining fake-blocks, the attacker loses his chance of earning block rewards. Assuming that we would have been able to mine 7 blocks, with a current block reward of 6.25 BTC and $11,400 per Bitcoin at the time of writing:

```
7 * 6.25 BTC = 43.75 BTC

43.75 BTC * ($11,400 / 1 BTC) = $498,750
```

Furthermore, the attacker needs to achieve 10% of the mining power. With a current total hash rate of 120 EH/s, this would mean 12 EH/s. There are two options: buying the hardware or renting the mining power from others. A new Antminer S9 with 16 TH/s can be bought for ~$100. This would mean an attacker has to pay $75,000,000 to buy so many of these miners to reach 12 EH/s. The costs for electricity, storage room and cooling still needs to be added.

Hashing power can also be rented online. Obviously nobody is offering to lend 12 EH/s of hashing power – but for this calculation we assume that an attacker is still able to rent this amount of hashing power. The website nicehash.com is offering 1 PH/s for 0.0098 BTC (for 24 hours).

```
1 PH/s = 0.0098 BTC

12 EH/s = 117.6 BTC
```

Assuming it is possible to rent it for 700 minutes only (which would be 48.6% of one day).

```
117.6 BTC * 0.486 = 57.15 BTC

57.15 BTC * ($11,400 / 1 BTC) = $651,510

Total: $498,750 + $651,510 = $1,150,260
```

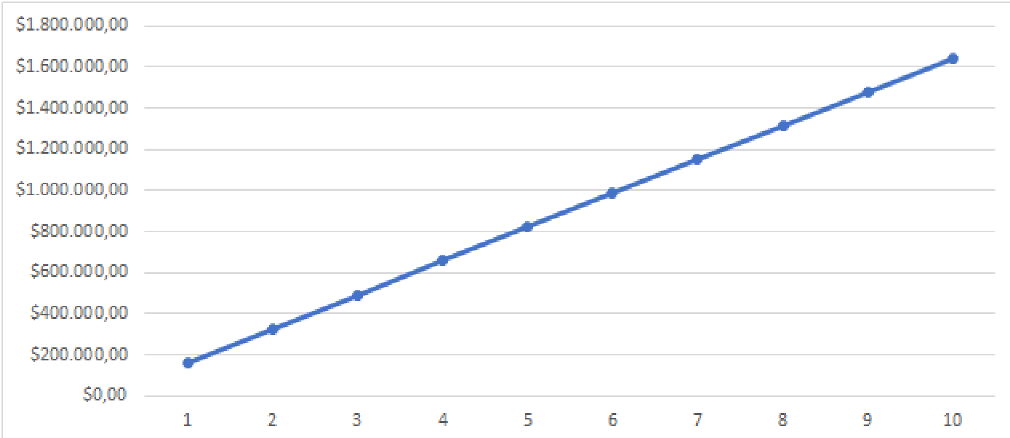
Therefore, 6 finality headers provide a security of estimated **$1,150,260** in total.

> What does that mean for the client?

A rental car is equipped with an Incubed client running on a microship to perform authorization checks and activate the ignition if necessary. The car is its own owner and it has a Bitcoin address to receive payments to rent itself to customers. Part of the authorization check is the verification of the existence and correctness of the payment (using the Incubed client). Therefore, a customers sends the hash of the payment transaction to the car to be authorized in case the transaction gets verified.

Assuming that a customer (Bob) runs a malicious Incubed node and the car randomly asks exactly this node for the verification of the transaction. Bob could fool the car by creating a fake-transaction in a fake-block. To prove the correctness of the fake-transaction, Bob needs to calculate a chain of fake-blocks as well (to prove the finality). In this case the car would authorize Bob because it was able to verify the transaction, even though the transaction is fake.

Bob would be able to use the car without having to pay for it, **but** performing such an attack (calculate a wrong block and 6 finality headers) is very expensive as shown above. And this is what is meant by *security in terms of $* - fooling the client in such a scenario is definitely not worth it (since paying the actual fees for the car would be a *far* less than the cost of performing such an attack). Hence, Incubed clients can trust in the correctness of a transaction (with a high probability) if the value is less than $1,150,260 and the server is able to provide 6 finality headers for the block that transaction is included. The higher the number of finality blocks, the higher the security (i.e. the higher the costs for an attack). The following figure shows the cost to mine *n* fake-blocks based on the numbers mentioned above.



21.3 Proofs

21.3.1 Target Proof

Having a verified target on the client-side is important to verify the proof of work and therefore the data itself (assuming that the data is correct when someone put a lot of work into it). Since the target is part of a block header (`bits`-field) we can verify the target by verifying the block header. This is a dilemma since we want to verify the target by verifying the block header but we need a verified target to verify the block header (as shown in block proof). You will read about two different options to verify a target.

Verification using finality headers

The client maintains a cache with the number of a difficulty adjustment period (dap) and the corresponding target - which stays the same for the duration of one period. This cache was filled with default values at the time of the release of the Bitcoin implementation. If a target is not yet part of the cache it needs to be verified first and added to the cache afterwards.

> How does the verification works?

We completely rely on the finality of a block. We can verify the target of a block (and therefore for a whole period) by requesting a block header (`getblockheader`) and `n`-amount of finality headers. If we are able to prove the finality using the finality proof we can consider the target as verified as mentioned earlier.

The client sets a limit in his configuration regarding the maximum change of the target from a verified one to the one he wants to verify. The client will not trust the changes of the target when they are too big (i.e. greater than the limit). In this case the client will use the proofTarget-method to verify the big changes in smaller steps.

Verification using signatures

Important: This concept is still in development and discussion and is not yet fully implemented.

This approach uses signatures of Incubed nodes to verify the target.

Since the target is part of the block header we just have to be very sure that the block header is correct - which leads us to a correct target. The client fetches the node list and chooses n nodes which will provide signature. Afterwards he sends a `getblockheader`-request (also containing the addresses of the selected nodes) to a random provider node. This node asks the signatures nodes to sign his result (the block header). The response will include the block header itself and all the signatures as well. The client can verify all signatures by using the node list and therefore verifying the actual result (a verified block header and therefore a verified target). The incentivation for the nodes to act honest is their deposit which they will loose in case they act malicious. (see here for more details of this process)

The amount of signatures nodes n should be chosen with the Risk Calculation in mind.

21.3.2 Block Proof

Verifying a Bitcoin block is quite easy when you already have a verified block hash.

1. We take the first 80 bytes of the block data - which is the block header - and hash it with `sha256` twice. Since Bitcoin stores the hashes in little endian we have to reverse the order of the bytes afterwards:

```
// btc hash = sha256(sha256(data))
const hash(data: Buffer) => crypto.createHash('sha256').update(crypto.createHash(
↪'sha256').update(data).digest()).digest()

const blockData:Buffer = ...
// take the first 80 bytes, hash them and reverse the order
const blockHash = hash(blockData.slice(0,80)).reverse()
```

2. In order to check the proof of work in the block header we compare the target with the hash:

```
const target = Buffer.alloc(32)
// we take the first 3 bytes from the bits-field and use the 4th byte as exponent
blockData.copy(target, blockData[75]-3,72,75);

// the hash must be lower than the target
if (target.reverse().compare(blockHash) < 0)
   throw new Error('blockHash must be smaller than the target')
```

21.3.3 Finality Proof

Necessary data to perform this proof:

- Block header (block `X`)
- Finality block header (block `X+1`, . . . , `X+n`)

The finality for block `X` can be proven as follows:

The proof data contains the block header of block `X` as well as `n` following block headers as finality headers. In Bitcoin every block header includes a `parentHash`-field which contains the block hash of its predecessor. By checking this linking the finality can be proven for block `X`. Meaning the block hash of block `X` is the `parentHash` of block `X+1`, the hash of block `X+1` is the `parentHash` of block `X+2`, and so on. If this linking correct until block `X+n` (i.e. the last finality header) then block `X` can be considered as final (Hint: as mentioned above Bitcoin uses a probabilistic finality, meaning a higher `n` increases the probability of being actual final).

Example

This example will use two finality headers to demonstrate the process:

Hash: 0000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7 (block #625000)

Finality Headers:

```
(1)␣
↪00e00020d7cd60cab00255bb13bbb023fd5d85daaaf389720a14000000000000000000000040273a5828953c61554c98540f

(2)␣
↪00e0ff7fc78d20fab2c28de35d00f7ec5fb269a63d597146d9b3100000000000000000000052960bb1aa3c23581ab3c233a2
```

```
Hash (reversed): d7cd60cab00255bb13bbb023fd5d85daaaf389720a1400000000000000000000
Parent Hash (1): d7cd60cab00255bb13bbb023fd5d85daaaf389720a1400000000000000000000

Hash of (1):     c78d20fab2c28de35d00f7ec5fb269a63d597146d9b310000000000000000000
Parent Hash (2): c78d20fab2c28de35d00f7ec5fb269a63d597146d9b310000000000000000000
```

21.3.4 Transaction Proof (Merkle Proof)

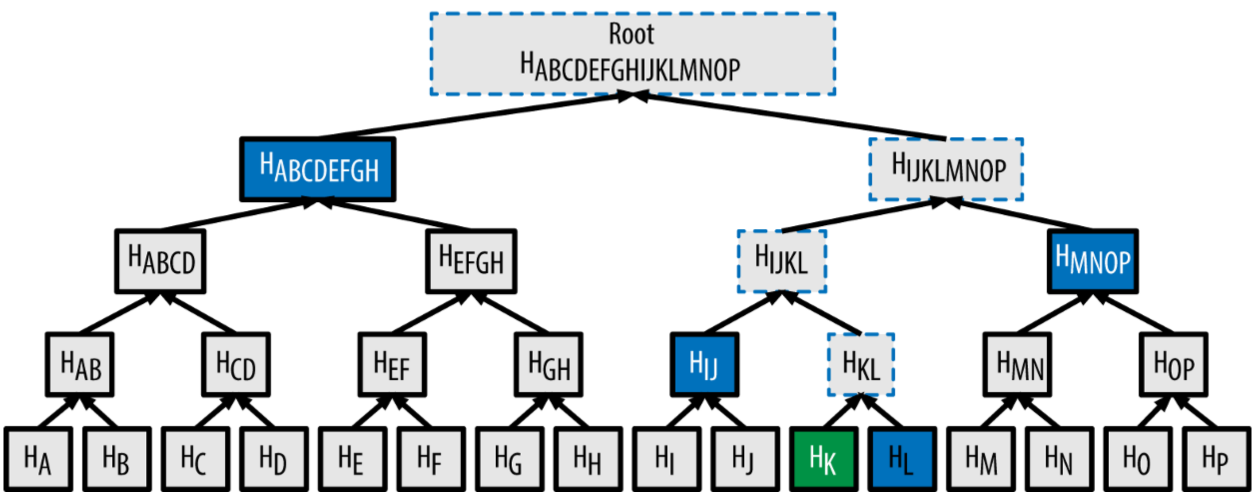
Necessary data to perform this proof:

- Block header
- Transaction
- Merkle proof (for this transaction)
- Index (of this transaction)

All transactions of a Bitcoin block are stored in a **merkle tree**. Every leaf node is labelled with with the hash of a transaction, and every non-leaf node is labelled with the hash of the labels of its two child nodes. This results in one single hash - the **merkle root** - which is part of the block header. Attempts to change or remove a leaf node after the block was mined (i.e. changing or removing a transaction) will not be possible since this will cause changes in the merkle root, thereby changes in the block header and therefore changes in the hash of this block. By checking the block header against the block hash such an attempt will be discovered.

Having a verified block header and therefore a verified merkle root allows us to perform a merkle root proving the existence and correctness of a certain transaction.

The following example explains a merkle proof (for more details see here):



In order to verify the existence and correctness of transaction [K] we use `sha256` to hash [K] twice to obtain H(K). For this example the merkle proof data will contain the hashes H(L), H(IJ), H(MNOP) and H(ABCDEFGH). These hashes can be used to calculate the merkle root as shown in the picture. The hash of the next level can be calculated by concatenating the two hashes of the level below and then hashing this hash with `sha256` twice. The index determines which of the hashes is on the right and which one on the left side for the concatenation (Hint: the placement is important, since swaped hashes will result in a completely different hash). When the calculated merkle root appears to be equal to the one contained by the block header we've hence proven the existence and correctness of transaction [K].

This can be done for every transaction of a block by simply hashing the transaction and then keep on hashing this result with the next hash from the merkle proof data. The last hash must match the merkle root. (Hint: obviously the merkle proof data will be different for different transactions).

21.3.5 Block Number Proof

Necessary data to perform this proof:

- Block header
- Coinbase transaction (first transaction of the block)
- Merkle proof (for the coinbase transaction)

In comparison to Ethereum there is no block number in a Bitcoin block header. Bitcoin uses the height of a block, which is the number of predecessors. The genesis block is at height 0 since there are no predecessors (the block with 100 predecessors is at height 100). Therefore, you need to know the complete Bitcoin blockchain to verify the height of a block (by counting the links back to the genesis block). Hence, actors that do not store the complete chain (like an Incubed client) are not able to verify the height of a block. To change that Gavin Andresen proposed a change to the Bitcoin protocol in 2012.

> Bitcoin Improvement Proposal 34 (BIP-34) introduces an upgrade path for versioned transactions and blocks. A unique value is added to newly produced coinbase transactions, and blocks are updated to version 2. After block number 227,835 all blocks must include the block height in their coinbase transaction.

For all blocks after block number 227,835 the block number can be proven as follows:

1.) Extract block number out of the coinbase transaction

Coinbase transaction of block 624692

```
03348809041f4e8b5e7669702f7777772e6f6b65782e636f6d2ffabe6d6db388905769d4e3720b1e59081407ea75173ba3ed6
```

Decode:

a) `03`: first byte signals the length of the block number (push the following 3 bytes) b) `348809`: the block number in big endian (convert to little endian) c) `098834`: the block number in little endian (convert to decimal) d) **`624692`**: the actual block number e) `041f4e...` : the rest can be anything

2.) Prove the existence and correctness of the coinbase transaction

To trust the extracted block number it's necessary to verify the existence and correctness of the coinbase transaction. This can be done by performing a merkle proof using the provided block header and the merkle proof data.

Size of a block number proof

As mentioned above three things are required to perform this proof:

- block header (fixed size): 80 bytes
- coinbase transaction (variable size): 300 bytes on average (there are some extra ordinary large ones: e.g. of block #376992 with 9,534 bytes)
- merkle proof (variable size): block limit of 1 MB, a maximum of approximately 3500 transactions in one block, maximum of 12 hashes needed in the merkle proof = 12 * 32 bytes = 384 bytes

Conclusion: a block number proof will be **764 bytes** on average (the size of this proof can be much smaller - but can also be much bigger - depending on the size of the coinbase transaction and the total amount of transaction)

21.3.6 Blocks Before 227,836 (BIP34)

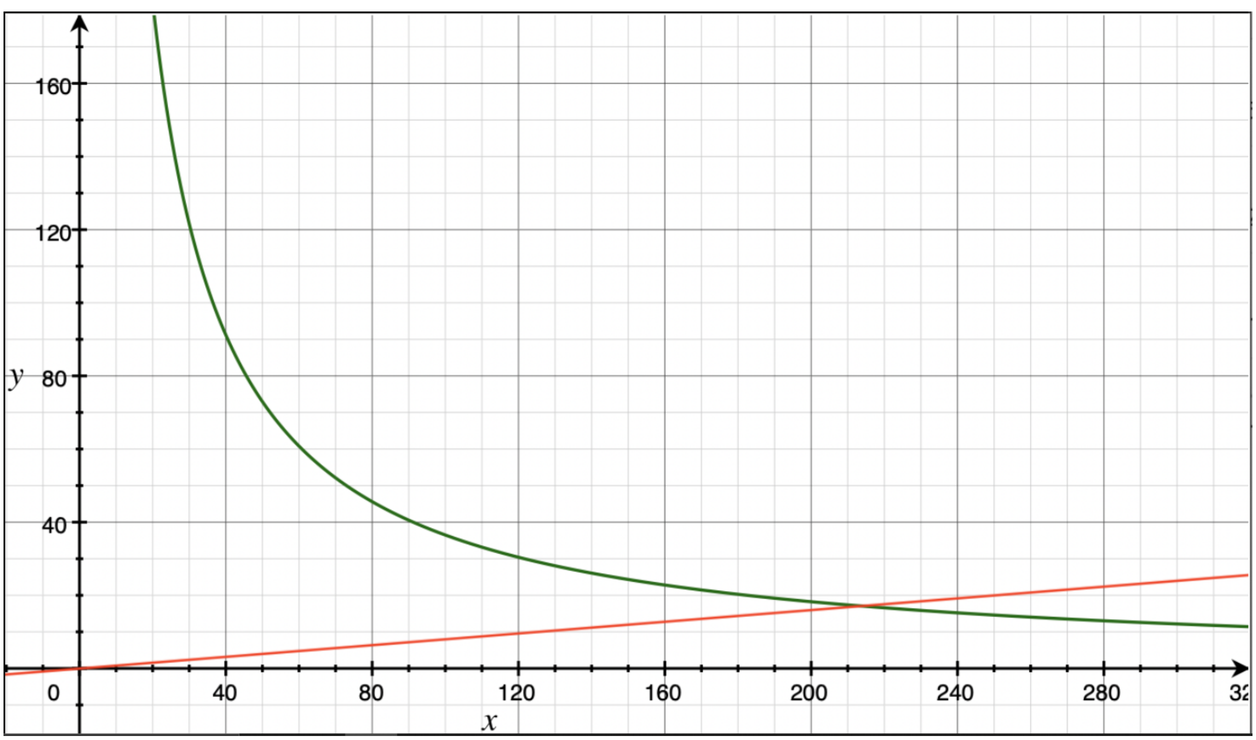
As mentioned in the introduction, relying on the finality does not really work for very old blocks (old in this context always means before BIP34, block number < 227,836) due to the following problems:

- **low difficulty** The total hash rate of the bitcoin network was around 1-10 TH/s in 2011, whereas today the total hash rate is around 130 EH/s and a single Antminer S9 is capable of running at 14 TH/s (which is more than the total hash rate back in 2011). Therefore, an attacker can easily mine a chain of fake-blocks with today's computing power and finality blocks provide almost no security. See here for the evolution of the total hash rate.
- **missing BIP34** The verification of the block number is an important part of the verification of bitcoin data in general. Since the block number is not part of the block header in Bitcoin the client needs a different way to verify the block number to make sure that a requested block X really is block X. For every block after block number 227,835 the block number is part of the coinbase transaction due to BIP34. The verification described in Block Number Proof obviosuly does not work for very old blocks (before the introduction of BIP34).

The verification of blocks before BIP34 relies on hard-coded checkpoints of hashes of bygone blocks on the client-side. The server needs to provide the corresponding finality headers from a requested block up to the next checkpoint. By checking the linking the client is able to verify the existence and correctness of the requested block. The only way for an attacker to fool the client would be by finding a hash collision (find different inputs that produce the same hash) of a certain checkpoint (the attacker could provide a chain of fake-blocks and the client accepts it because he was able to verify the chain against a checkpoint). The client has the opportunity to decide whether he wants to verify old blocks or not. By turning on this option the checkpoints will be included in the client and the server will provide the corresponding finality headers in each request of old blocks.

Creation of the checkpoints

The reason why we need checkpoints is that it is not feasable for the client to save every single hash from the genesis block up to the introduction of BIP34. The checkpoints are hashes of bygone blocks, and to save on space the checkpoints have a distance X. The larger this distance is, the smaller is the amount of checkpoints and the larger is the amount of necessary finality headers to reach a checkpoint (maximum X finality headers). Therefore, having a large distance requires less storage space to save the checkpoints BUT the amount of finality headers per request will be very big (resulting in a lot of data to transfer). The following graph should help to decide where the sweetspot is.



```
y: size in kB
x: distance between checkpoints (blocks)
green: size of record of checkpoints
red: size of finality headers per request (maximum)
```

As you can see in the graph the distance of **200** is the sweetspot we were looking for. This means the record of checkpoints includes the hash of every 200th block of the Bitcoin blockchain starting with block 200 (storing the genesis block is not necessary since a checkpoint always has to be in the future of a requested block). It takes 32 bytes to store a block hash. To save on space we decided to store the first 16 bytes only - and to save even more space we removed the first 4 bytes of every hash because each hash started with at least 4 bytes of zeros (storing only 12 bytes is still very secure). The record of checkpoints needs a total of **13680 bytes**. Depending on the distance from a requested block to the next checkpoint a response will include a maximum of 199 finality headers which is a total of around *16 kB*.

> Why is it necessary having checkpoints in the future (from the view of a requested block)? Why can a checkpoint not be in past to have a maximum distance of 100 (either forwards or backwards to the next checkpoint)?

Simple answer: Since the hash of block X-1 is part of block X (not not vice versa) checking the links backward does not provide any security. An attacker can simply modify block X and refer to block X-1 (using the hash of block X-1 as the parent hash of block X). The attacker just have to solve the proof-of-work again for block X (which should not be too hard with the today's computing power and the low difficulty at that time). To verify that block X is correct the client always needs a chain of blocks **up to** the next checkpoint.

21.4 Conviction

Important: This concept is still in development and discussion and is not yet fully implemented.

Just as the Incubed Client can ask for signed block hashes in Ethereum, he can do this in Bitcoin as well. The signed payload from the node will have to contain these data:

```
bytes32 blockhash;
uint256 timestamp;
bytes32 registryId;
```

Client requires a Signed Blockhash

and the Data Provider Node will ask the chosen node to sign.

The Data Provider Node (or Watchdog) will then check the signature

If the signed blockhash is wrong it will start the conviting process:

Convict with BlockHeaders

In order to convict, the Node needs to provide proof, which is the correct blockheader.

But since the BlockHeader does not contain the BlockNumber, we have to use the timestamp. So the correct block as proof must have either the same timestamp or a the last block before the timestamp. Additionally the Node may provide FinalityBlockHeaders. As many as possible, but at least one in order to prove, that the timestamp of the correct block is the closest one.

The Registry Contract will then verify

- the Signature of the convited Node.
- the BlockHeaders gives as Proof

The Verification of the BlockHeader can be done directly in Solitidy, because the EVM offers a precompiled Contract at address `0x2` : sha256, which is needed to calculate the Blockhash. With this in mind we can follow the steps as described in Block Proof implemented in Solidity.

While doing so we need to add the difficulties of each block and store the last blockHash and the `totalDifficulty` for later.

Challenge the longest chain

Now the convited Server has the chance to also deliver blockheaders to proof that he has signed the correct one.

The simple rule is:

> If the other node (convited or convitor) is not able to add enough verified BlockHeaders with a higher `totalDifficulty` within 1 hour, the other party can get the deposit and kick the malicious node out.

Even though this game could go for a while, if the convicted Node signed a hash, which is not part of the longest chain, it will not be possible to create enough mining power to continue mining enough blocks to keep up with the longest chain in the mainnet. Therefore he will most likely give up right after the first transaction.

CHAPTER 22

Incentivization

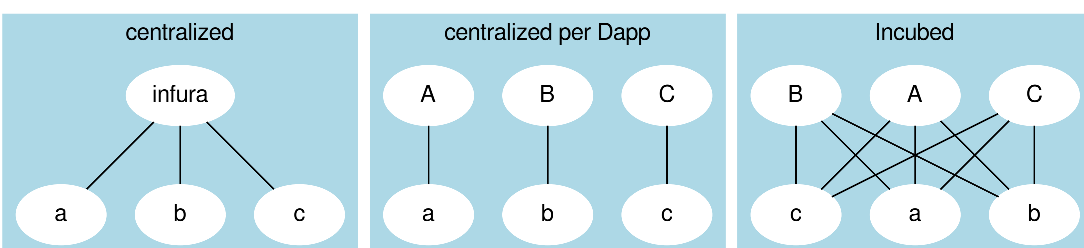
Important: This concept is still in development and discussion and is not yet fully implemented.

The original idea of blockchain is a permissionless peer-to-peer network in which anybody can participate if they run a node and sync with other peers. Since this is still true, we know that such a node won't run on a small IoT-device.

22.1 Decentralizing Access

This is why a lot of users try remote-nodes to serve their devices. However, this introduces a new single point of failure and the risk of man-in-the-middle attacks.

So the first step is to decentralize remote nodes by sharing rpc-nodes with other apps.



22.2 Incentivization for Nodes

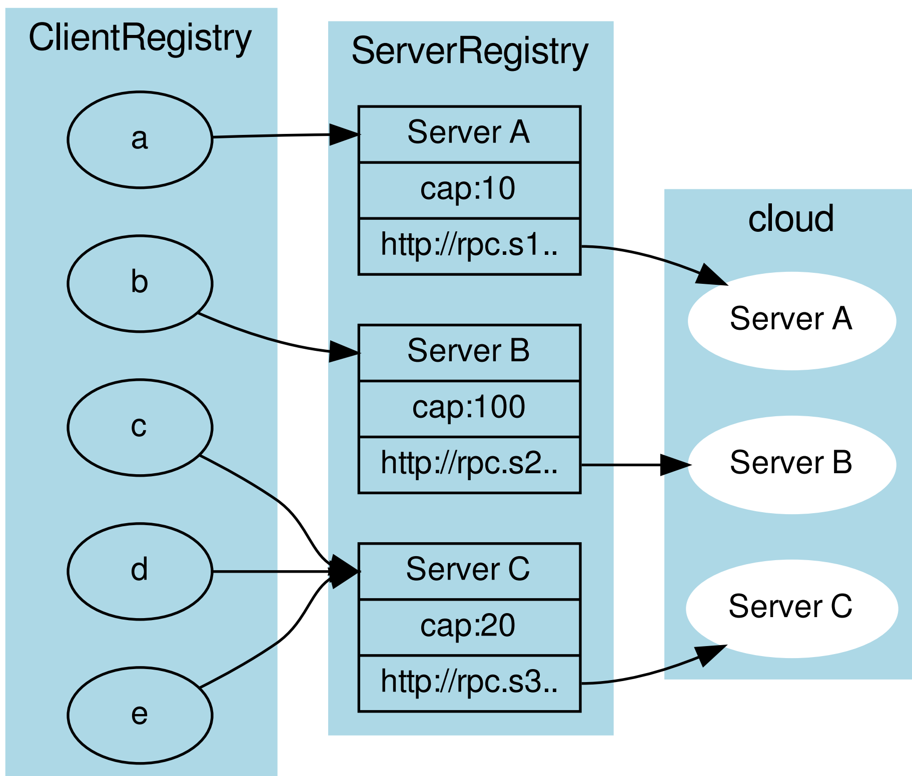
In order to incentivize a node to serve requests to clients, there must be something to gain (payment) or to lose (access to other nodes for its clients).

22.3 Connecting Clients and Server

As a simple rule, we can define this as:

> **The Incubed network will serve your client requests if you also run an honest node.**

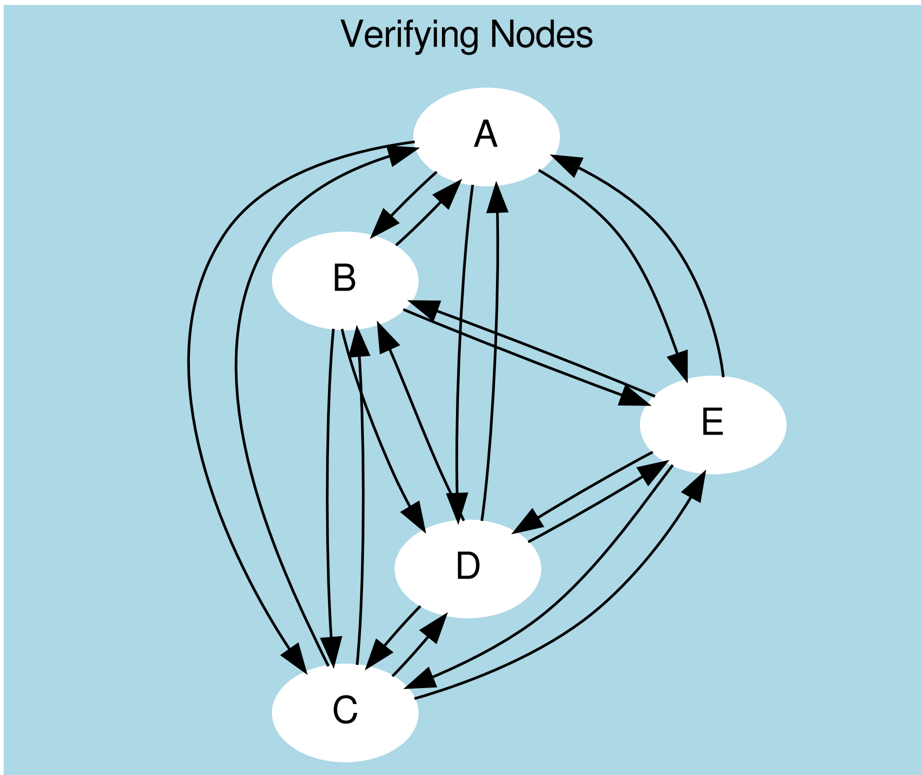
This requires a user to connect a client key (used to sign their requests) with a registered server. Clients are able to share keys as long as the owner of the node is able to ensure their security. This makes it possible to use one key for the same mobile app or device. The owner may also register as many keys as they want for their server or even change them from time to time (as long as only one client key points to one server). The key is registered in a client-contract, holding a mapping of the key to the server address.



22.4 Ensuring Client Access

Connecting a client key to a server does not mean the key relies on that server. Instead, the requests are simply served in the same quality as the connected node serves other clients. This creates a very strong incentive to deliver to all clients, because if a server node were offline or refused to deliver, eventually other nodes would deliver less or even stop responding to requests coming from the connected clients.

To actually find out which node delivers to clients, each server node uses one of the client keys to send test requests and measure the availability based on verified responses.



The servers measure the $A_{availability}$ by checking periodically (about every hour in order to make sure a malicious server is not only responding to test requests). These requests may be sent through an anonymous network like tor.

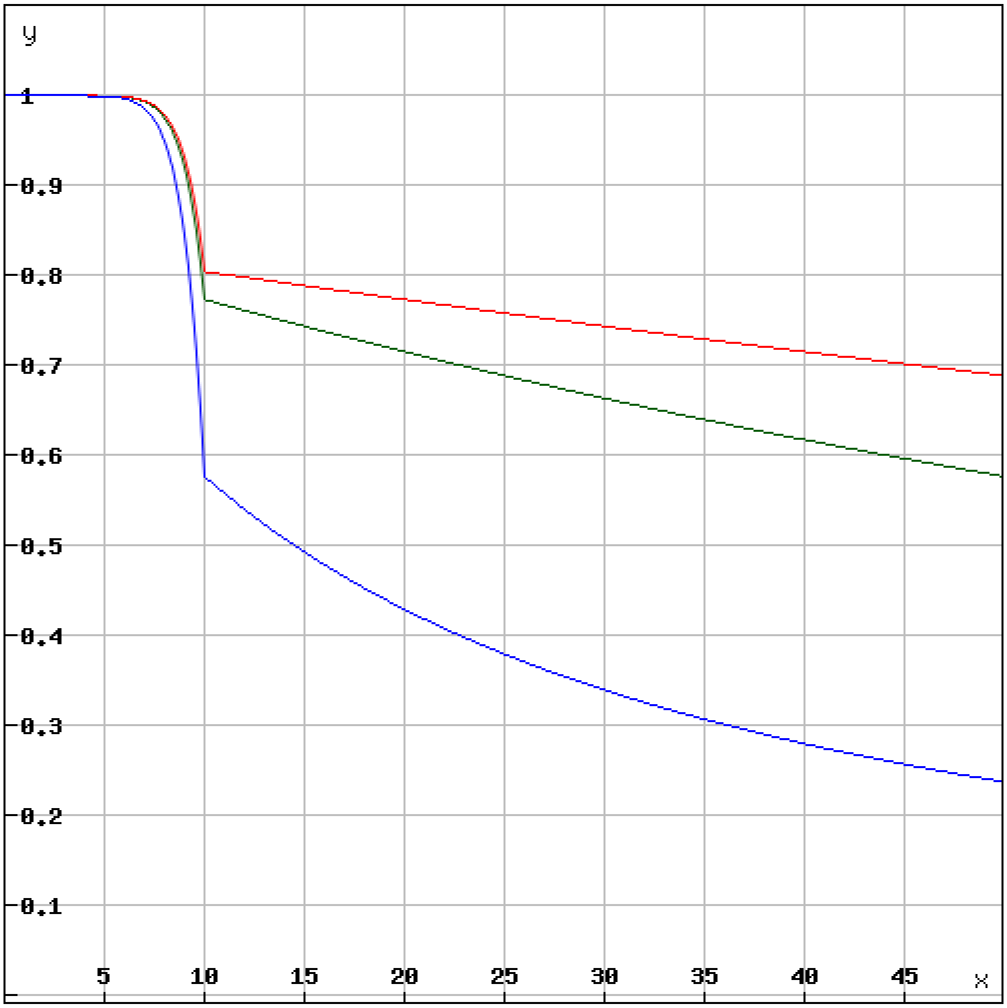
Based on the long-term (>1 day) and short-term (<1 day) availibility, the score is calculated as:

$$A = \frac{R_{received}}{R_{sent}}$$

In order to balance long-term and short-term availability, each node measures both and calculates a factor for the score. This factor should ensure that short-term avilability will not drop the score immediately, but keep it up for a while before dropping. Long-term availibility will be rewarded by dropping the score slowly.

$$A = 1 - (1 - \frac{A_{long} + 5 \cdot A_{short}}{6})^{10}$$

- A_{long} - The ratio between valid requests received and sent within the last month.
- A_{short} - The ratio between valid requests received and sent within the last 24 hours.



Depending on the long-term availibility the disconnected node will lose its score over time.

The final score is then calulated:

$$score = \frac{A \cdot D_{weight} \cdot C_{max}}{weight}$$

- A - The availibility of the node.
- $weight$ - The weight of the incoming request from that server's clients (see LoadBalancing).
- C_{max} - The maximal number of open or parallel requests the server can handle (will be taken from the registry).
- D_{weight} - The weight of the deposit of the node.

This score is then used as the priority for incoming requests. This is done by keeping track of the number of currently open or serving requests. Whenever a new request comes in, the node does the following:

1. Checks the signature.
2. Calculates the score based on the score of the node it is connected to.
3. Accepts or rejects the request.

```
if ( score < openRequests ) reject()
```

This way, nodes reject requests with a lower score when the load increases. For a client, this means if you have a low score and the load in the network is high, your clients may get rejected often and will have to wait longer for responses. If the node has a score of 0, they are blacklisted.

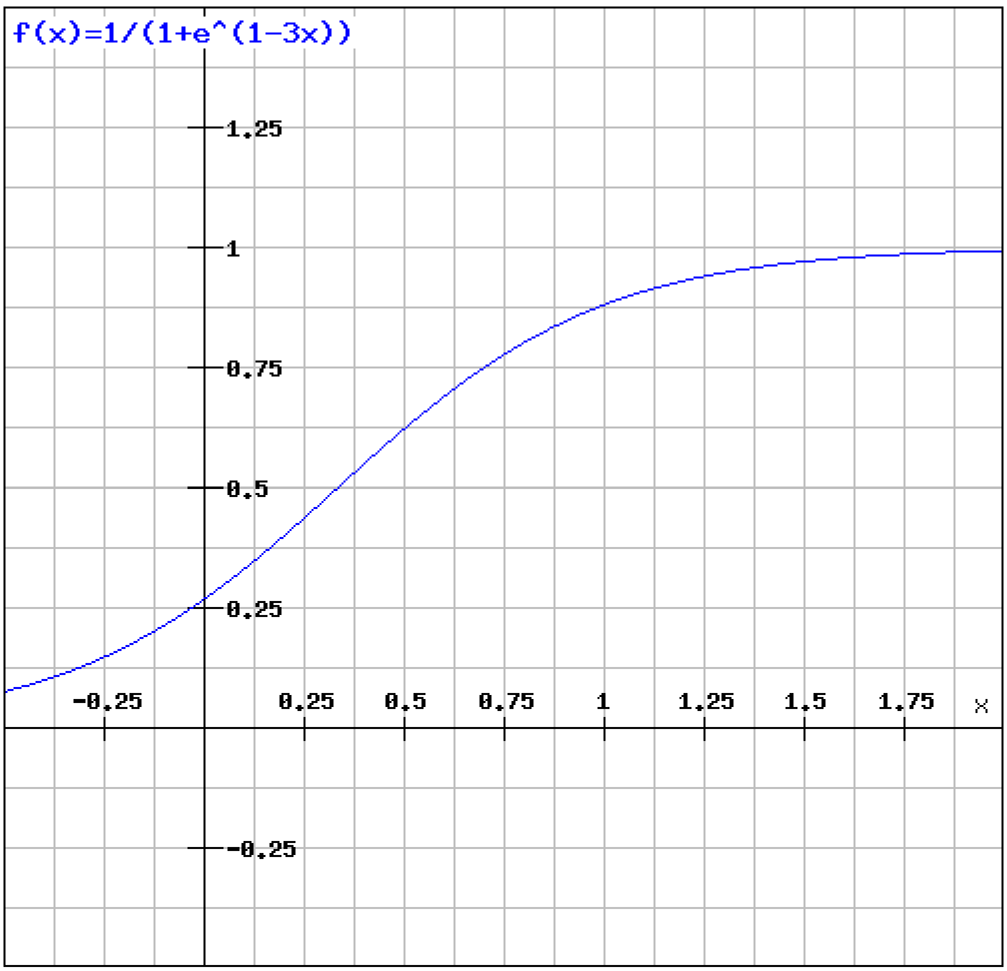
22.5 Deposit

Storing a high deposit brings more security to the network. This is important for proof-of-work chains. In order to reflect the benefit in the score, the client multiplies it with the D_{weight} (the deposit weight).

$$D_{weight} = \frac{1}{1 + e^{1 - \frac{3D}{D_{avg}}}}$$

- D - The stored deposit of the node.
- D_{avg} - The average deposit of all nodes.

A node without any deposit will only receive 26.8% of the max cap, while any node with an average deposit gets 88% and above and quickly reaches 99%.



22.6 LoadBalancing

In an optimal network, each server would handle an equal amount and all clients would have an equal share. In order to prevent situations where 80% of the requests come from clients belonging to the same node, we need to decrease the score for clients sending more requests than their shares. Thus, for each node the weight can be calculated by:

$$weight_n = \frac{\sum_{i=0}^{n} C_i \cdot R_n}{\sum_{i=0}^{n} R_i \cdot C_n}$$

- R_n - The number of requests served to one of the clients connected to the node.
- $\sum_{i=0}^{n} R_i$ - The total number of requests served.
- $\sum_{i=0}^{n} C_i$ - The total number of capacities of the registered servers.
- C_n - The capacity of the registered node.

Each node will update the $score$ and the $weight$ for the other nodes after each check in order to prioritize incoming requests.

The capacity of a node is the maximal number of parallel requests it can handle and is stored in the ServerRegistry. This way, all clients know the cap and will weigh the nodes accordingly, which leads to stronger servers. A node declaring a high capacity will gain a higher score, and its clients will receive more reliable responses. On the other hand, if a node cannot deliver the load, it may lose its availability as well as its score.

22.7 Free Access

Each node may allow free access for clients without any signature. A special option `--freeScore=2` is used when starting the server. For any client requests without a signature, this $score$ is used. Setting this value to 0 would not allow any free clients.

```
if (!signature) score = conf.freeScore
```

A low value for freeScore would serve requests only if the current load or the open requests are less than this number, which would mean that getting a response from the network without signing may take longer as the client would have to send a lot of requests until they are lucky enough to get a response if the load is high. Chances are higher if the load is very low.

22.8 Convict

Even though servers are allowed to register without a deposit, convicting is still a hard punishment. In this case, the server is not part of the registry anymore and all its connected clients are treated as not having a signature. The device or app will likely stop working or be extremely slow (depending on the freeScore configured in all the nodes).

22.9 Handling conflicts

In case of a conflict, each client now has at least one server it knows it can trust since it is run by the same owner. This makes it impossible for attackers to use blacklist-attacks or other threats which can be solved by requiring a response from the "home"-node.

22.10 Payment

Each registered node creates its own ecosystem with its own score. All the clients belonging to this ecosystem will be served only as well as the score of the ecosystem allows. However, a good score can not only be achieved with a good performance, but also by paying for it.

For all the payments, a special contract is created. Here, anybody can create their own ecosystem even without running a node. Instead, they can pay for it. The payment will work as follows:

The user will choose a price and time range (these values can always be increased later). Depending on the price, they also achieve voting power, thus creating a reputation for the registered nodes.

Each node is entitled to its portion of the balance in the payment contract, and can, at any given time, send a transaction to extract its share. The share depends on the current reputation of the node.

$$payment_n = \frac{weight_n \cdot reputation_n \cdot balance_{total}}{weight_{total}}$$

Why should a node treat a paying client better than others?

Because the higher the price a user paid, the higher the voting power, which they may use to upgrade or downgrade the reputation of the node. This reputation will directly influence the payment to the node.

That's why, for a node, the score of a client depends on what follows:

$$score_c = \frac{paid_c \cdot requests_{total}}{requests_c \cdot paid_{total} + 1}$$

The score would be 1 if the payment a node receives has the same percentage of requests from an ecosystem as the payment of the ecosystem represented relative to the total payment per month. So, paying a higher price would increase its score.

22.11 Client Identification

As a requirement for identification, each client needs to generate a unique private key, which must never leave the device.

In order to securely identify a client as belonging to an ecosystem, each request needs two signatures:

1. **The Ecosystem-Proof**This proof consists of the following information:

```
proof = rlp.encode(
   bytes32(registry_id),       // The unique ID of the registry.
   address(client_address),    // The public address of a client.
   uint(ttl),                  // Unix timestamp when this proof expires.
   bytes(signature)            // The signature with the signer-key of the␣
↪ecosystem. The message hash is created by rlp.encode, the client_address, and␣
↪the ttl.
)
```

For the client, this means they should always store such a proof on the device. If the ttl expires, they need to renew it. If the ecosystem is a server, it may send a request to the server. If the ecosystem is a payer, this needs to happen in a custom way.

2. **The Client-Proof**This must be created for each request. Here the client will create a hash of the request (simply by adding the `method`, `params` and a `timestamp`-field) and sign this with its private key.

```
message_hash = keccack(
   request.method
   + JSON.stringify(request.params)
   + request.timestamp
)
```

With each request, the client needs to send both proofs.

The server may cache the ecosystem-proof, but it needs to verify the client signature with each request, thus ensuring the identity of the sending client.

CHAPTER 23

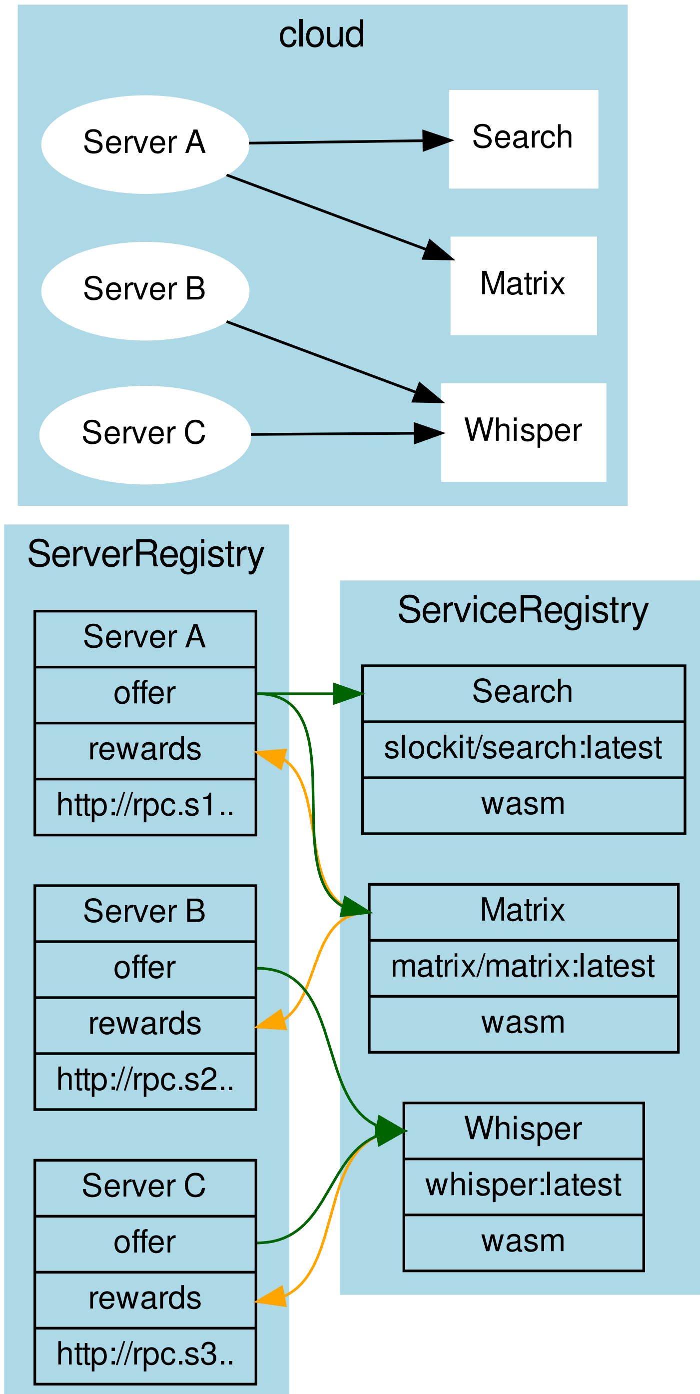
Decentralizing Central Services

Important: This concept is still in early development, meaning it has not been implemented yet.

Many dApps still require some off-chain services, such as search services running on a server, which, of course, can be seen as a single point of failure. To decentralize these dApp-specific services, they must fulfill the following criteria:

1. **Stateless**: Since requests may be sent to different servers, they cannot hold a user's state, which would only be available on one node.
2. **Deterministic**: All servers need to produce the exact same result.

If these requirements are met, the service can be registered, defining the server behavior in a docker image.


cloud
Server A
Search
Matrix
Server B
Server C
Whisper
ServerRegistry
Server A
offer
rewards
http://rpc.s1..
Server B
offer
rewards
http://rpc.s2..
Server C
offer
rewards
http://rpc.s3..
ServiceRegistry
Search
slockit/search:latest
wasm
Matrix
matrix/matrix:latest
wasm
Whisper
whisper:latest
wasm

23.1 Incentivization

Each server can define (1) a list of services to offer or (2) a list of services to reward.

The main idea is simply the following:

> **If you run my service, I will run yours.**

Each server can specifiy which services we would like to see used. If another server offers them, we will also run at least as many rewarded services as the other node.

23.2 Verification

Each service specifies a verifier, which is a Wasm module (specified through an IPFS hash). This Wasm offers two functions:

```
function minRequests():number

function verify(request:RPCRequest[], responses:RPCResponse[])
```

A minimal version could simply ensure that two requests were running and then compare them. If different, the Wasm could check with the home server and "convict" the nodes.

23.2.1 Convicting

Convicting on chain cannot be done, but each server is able to verify the result and, if false, downgrade the score.

CHAPTER 24

Threat Model for Incubed

24.1 Registry Issues

24.1.1 Long Time Attack

Status: open

A client is offline for a long time and does not update the NodeList. During this time, a server is convicted and/or removed from the list. The client may now send a request to this server, which means it cannot be convicted anymore and the client has no way to know that.

Solutions:

> CHR: I think that the fallback is often "out of service." What will happen is that those random nodes (A, C) will not respond. We (slock.it) could help them update the list in a centralized way.
>
> But I think the best way is the following: Allow nodes to commit to stay in the registry for a fixed amount of time. In that time, they cannot withdraw their funds. The client will most likely look for those first, especially those who only occasionally need data from the chain.
>
> SIM: Yes, this could help, but it only protects from regular unregistering. If you convict a server, then this timeout does not help.
>
> To remove this issue completely, you would need a trusted authority where you could update the NodeList first. But for the 100% decentralized way, you can only reduce it by asking multiple servers. Since they will also pass the latest block number when the NodeList changes, the client will find out that it needs to update the NodeList, and by having multiple requests in parallel, it reduces the risk of relying on a manipulated NodeList. The malicious server may return a correct NodeList for an older block when this server was still valid and even receive signatures for this, but the server cannot do so for a newer block number, which can only be found out by asking as many servers as needed.
>
> Another point is that as long as the signature does not come from the same server, the DataProvider will always check, so even if you request a signature from a server that is not part of the list anymore, the DataProvider will reject this. To use this attack, both the DataProvider and the BlockHashSigner must work together to provide a proof that matches the wrong blockhash.

CHR: Correct. I think the strategy for clients who have been offline for a while is to first get multiple signed blockhashes from different sources (ideally from bootstrap nodes similar to light clients and then ask for the current list). Actually, we could define the same bootstrap nodes as those currently hard-coded in Parity and Geth.

24.1.2 Inactive Server Spam Attack

Status: partially solved

Everyone can register a lot of servers that don't even exist or aren't running. Somebody may even put in a decent deposit. Of course, the client would try to find out whether these nodes were inactive. If an attacker were able to onboard enough inactive servers, the chances for an Incubed client to find a working server would decrease.

Solutions:

1. **Static Min Deposit**

   There is a min deposit required to register a new node. Even though this may not entirely stop any attacker, but it makes it expensive to register a high number of nodes.

   Desicion :

   Will be implemented in the first release, since it does not create new Riscs.

2. **Unregister Key**

   At least in the beginning we may give us (for example for the first year) the right to remove inactive nodes. While this goes against the principle of a fully decentralized system, it will help us to learn. If this key has a timeout coded into the smart contract all users can rely on the fact that we will not be able to do this after one year.

   Desicion :

   Will be implemented in the first release, at least as a workaround limited to one year.

3. **Dynamic Min Deposit**

   To register a server, the owner has to pay a deposit calculated by the formula:

   $$deposit_{min} = \frac{86400 \cdot deposit_{average}}{(t_{now} - t_{lastRegistered})}$$

   To avoid some exploitation of the formula, the `deposit_average` gets capped at 50 Ether. This means that the maximum `deposit_min` calculated by this formula is about 4.3 million Ether when trying to register two servers within one block. In the first year, there will also be an enforced deposit limit of 50 Ether, so it will be impossible to rapidly register new servers, giving us more time to react to possible spam attacks (e.g., through voting).

   Desicion :

   This dynamic deposit creates new Threads, because an attacker can stop other nodes from registering honest nodes by adding a lot of nodes and so increasing the min deposit. That's why this will not be implemented right now.

4. **Voting**

   In addition, the smart contract provides a voting function for removing inactive servers: To vote, a server has to sign a message with a current block and the owner of the server they want to get voted out. Only the latest 256 blockhashes are allowed, so every signature will effectively expire after roughly 1 hour. The power of each vote will be calculated by the amount of time when the server was registered. To make sure that the oldest servers won't get too powerful, the voting power gets capped at one year and won't increase further. The server being voted out will also get an oppositional voting power that is capped at two years.

For the server to be voted out, the combined voting power of all the servers has to be greater than the oppositional voting power. Also, the accumulated voting power has to be greater than at least 50% of all the chosen voters.

As with a high amount of registered in3-servers, the handling of all votes would become impossible. We cap the maximum amount of signatures at 24. This means to vote out a server that has been active for more then two years, 24 in3-servers with a lifetime of one month are required to vote. This number decreases when more older servers are voting. This mechanism will prevent the rapid onboarding of many malicious in3-servers that would vote out all regular servers and take control of the in3-nodelist.

Additionally, we do not allow all servers to vote. Instead, we choose up to 24 servers randomly with the blockhash as a seed. For the vote to succeed, they have to sign on the same blockhash and have enough voting power.

To "punish" a server owner for having an inactive server, after a successful vote, that individual will lose 1% of their deposit while the rest is locked until their deposit timeout expires, ensuring possible liabilities. Part of this 1% deposit will be used to reimburse the transaction costs; the rest will be burned. To make sure that the transaction will always be paid, a minimum deposit of 10 finney (equal to 0.01 Ether) will be enforced.

Desicion:

Voting will also create the risc of also Voting against honest nodes. Any node can act honest for a long time and then become a malicious node using their voting power to vote against the remaining honest nodes and so end up kicking all other nodes out. That's why voting will be removed for the first release.

24.1.3 DDOS Attack to uncontrolled urls

Status: not implemented yet

As a owner I can register any url even a server which I don't own. By doing this I can also add a high weight, which increases the chances to get request. This way I can get potentially a lot of clients to send many requests to a node, which is not expecting it. Even though clients may blacklist this node, it would be to easy to create a DDOS-Atack.

Solution:

Whenever there is a new node the client has never communicated to, we should should check using a DNS-Entry if this node is controlled by the owner. The Entry may look like this:

```
in3-signer: 0x21341242135346534634634,0xabf21341242135346534634634,
↪0xdef21341242135346534634634
```

Only if this DNS record contains the signer-address, the client should communicate with this node.

24.1.4 Self-Convict Attack

Status: solved

A user may register a mailcious server and even store a deposit, but as soon as they sign a wrong blockhash, they use a second account to convict themself to get the deposit before somebody else can.

Solution:

SIM: We burn 50% of the depoist. In this case, the attacker would lose 50% of the deposit. But this also means the attacker would get the other half, so the price they would have to pay for lying is up to 50% of their deposit. This should be considered by clients when picking nodes for signatures.

Desicion: Accepted and implemented

24.1.5 Convict Frontrunner Attack

Status: solved

Servers act as watchdogs and automatically call convict if they receive a wrong blockhash. This will cost them some gas to send the transaction. If the block is older than 256 blocks, this may even cost a lot of gas since the server needs to put blockhashes into the BlockhashRegistry first. But they are incentivized to do so, because after successfully convicting, they receive a reward of 50% of the deposit.

A miner or other attacker could now wait for a pending transaction for convict and simply use the data and send the same transaction with a high gas price, which means the transaction would eventually be mined first and the server, after putting so much work into preparing the convict, would get nothing.

Solution:

Convicting a server requires two steps: The first is calling the `convict` function with the block number of the wrongly signed block `keccak256(_blockhash, sender, v, r, s)`. Both the real blockhash and the provided hash will be stored in the smart contract. In the second step, the function `revealConvict` has to be called. The missing information is revealed there, but only the previous sender is able to reproduce the provided hash of the first transaction, thus being able to convict a server.

Desicion: Accepted and implemented

24.2 Network Attacks

24.2.1 Blacklist Attack

Status: partially solved

If the client has no direct internet connection and must rely on a proxy or a phone to make requests, this would give the intermediary the chance to set up a malicious server.

This is done by simply forwarding the request to its own server instead of the requested one. Of course, they may prepare a wrong answer, but they cannot fake the signatures of the blockhash. Instead of sending back any signed hashes, they may return no signatures, which indicates to the client that the chosen nodes were not willing to sign them. The client will then blacklist them and request the signature from other nodes. The proxy or relay could return no signature and repeat that until all are blacklisted and the client finally asks for the signature from a malicious node, which would then give the signature and the response. Since both come from a bad-acting server, they will not convict themself and will thus prepare a proof for a wrong response.

Solutions:

1. **Signing Responses**

    SIM: First, we may consider signing the response of the DataProvider node, even if this signature cannot be used to convict. However, the client then knows that this response came from the client they requested and was also checked by them. This would reduce the chances of this attack since this would mean that the client picked two random servers that were acting malicious together.

   Decision:

   Not implemented yet. Maybe later.

2. **Reject responses when 50% are blacklisted**

    If the client blacklisted more than 50% of the nodes, we should stop. The only issue here is that the client does not know whether this is an 'Inactive Server Spam Attack' or not. In case of an 'Inactive Server Spam Attack,' it would actually be good to blacklist 90% of the servers and still be able to work with the remaining 10%, but if the proxy is the problem, then the client needs to stop blacklisting.

CHR: I think the client needs a list of nodes (bootstrape nodes) that should be signed in case the response is no signature at all. No signature at all should default to an untrusted relayer. In this case, it needs to go to trusted relayers. Or ask the untrusted relayer to get a signature from one of the trusted relayers. If they forward the signed reponse, they should become trusted again.

SIM: We will allow the client to configure optional trusted nodes, which will always be part of the nodelist and used in case of a blacklist attack. This means in case more than 50% are blacklisted the client may only ask trusted nodes and if they don't respond, instead of blacklisting it will reject the request. While this may work in case of such a attack, it becomes an issue if more than 50% of the registered nodes are inactive and blacklisted.

Decision:

The option of allowing trusted nodes is implemented.

24.2.2 DDoS Attacks

Status: solved (as much as possible)

Since the URLs of the network are known, they may be targets for DDoS attacks.

Solution:

SIM: Each node is reponsible for protecting itself with services like Cloudflare. Also, the nodes should have an upper limit of concurrent requests they can handle. The response with status 500 should indicate reaching this limit. This will still lead to blacklisting, but this protects the node by not sending more requests.

CHR: The same is true for bootstrapping nodes of the foundation.

24.2.3 None Verifying DataProvider

Status: solved (more signatures = more security)

A DataProvider should always check the signatures of the blockhash they received from the signers. Of course, the DataProvider is incentivized to do so because then they can get 50% of their deposit, but after getting the deposit, they are not incentivized to report this to the client. There are two scenarios:

1. The DataProvider receives the signature but does not check it.

   In this case, at least the verification inside the client will fail since the provided blockheader does not match.

2. The DataProvider works together with the signer.

   In this case, the DataProvider would prepare a wrong blockheader that fits the wrong blockhash and would pass the verification inside the client.

Solution:

SIM: In this case, only a higher number of signatures could increase security.

24.3 Privacy

24.3.1 Private Keys as API Keys

Status: solved

For the scoring model, we are using private keys. The perfect security model would register each client, which is almost impossible on mainnet, especially if you have a lot of devices. Using shared keys will very likely happen, but this a nightmare for security experts.

Solution:

1. Limit the power of such a key so that the worst thing that can happen is a leaked key that can be used by another client, which would then be able to use the score of the server the key is assigned to.
2. Keep the private key secret and manage the connection to the server only off chain.
3. Instead of using a private key as API-Key, we keep the private key private and only get a signature from the node of the ecosystem confirming this relationship. This may happen completly offchain and scales much better.

Desicion: clients will not share private keys, but work with a signed approval from the node.

24.3.2 Filtering of Nodes

Status: partially solved

All nodes are known with their URLs in the NodeRegistry-contract. For countries trying to filter blockchain requests, this makes it easy to add these URLs to blacklists of firewalls, which would stop the Incubed network.

Solution:

> Support Onion-URLs, dynamic IPs, LORA, BLE, and other protocols. The registry may even use the props to indicate the capabilities, so the client can choose which protocol to he is capable to use.

Decision: Accepted and prepared, but not fully implemented yet.

24.3.3 Inspecting Data in Relays or Proxies

For a device like a BLE, a relay (for example, a phone) is used to connect to the internet. Since a relay is able to read the content, it is possible to read the data or even pretend the server is not responding. (See Blacklist Attack above.)

Solution:

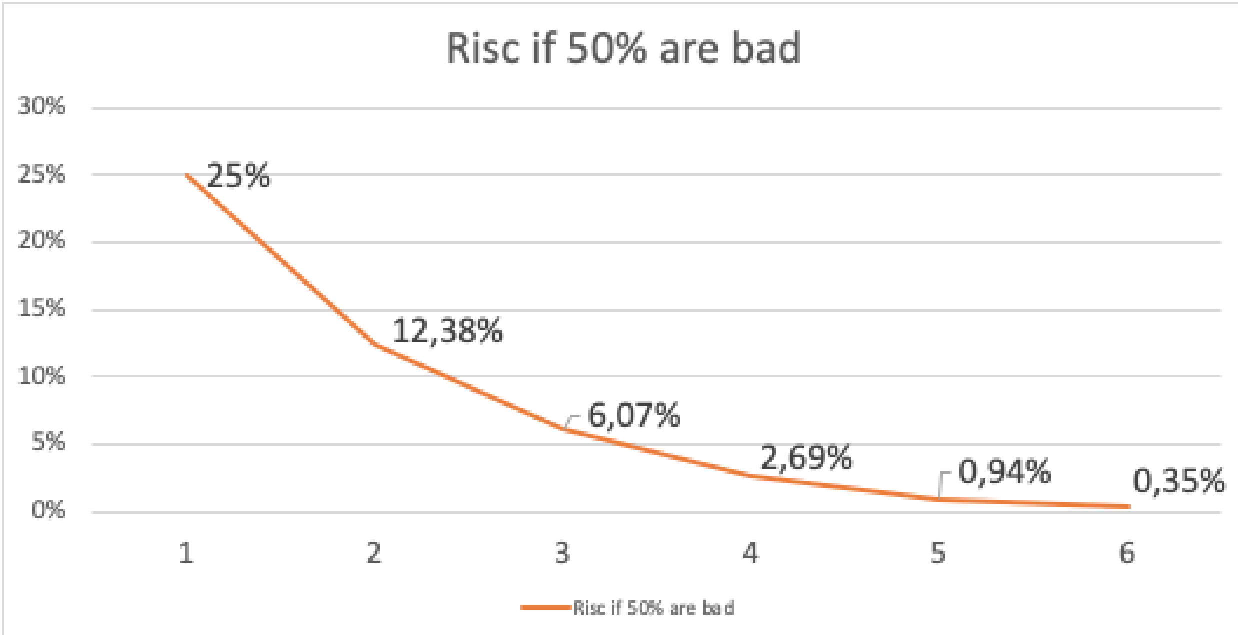
> Encrypt the data by using the public key of the server. This can only be decrypted by the target server with the private key.

24.4 Risk Calculation

Just like the light client there is not 100% protection from malicious servers. The only way to reach this would be to trust special authority nodes to sign the blockhash. For all other nodes, we must always assume they are trying to find ways to cheat.

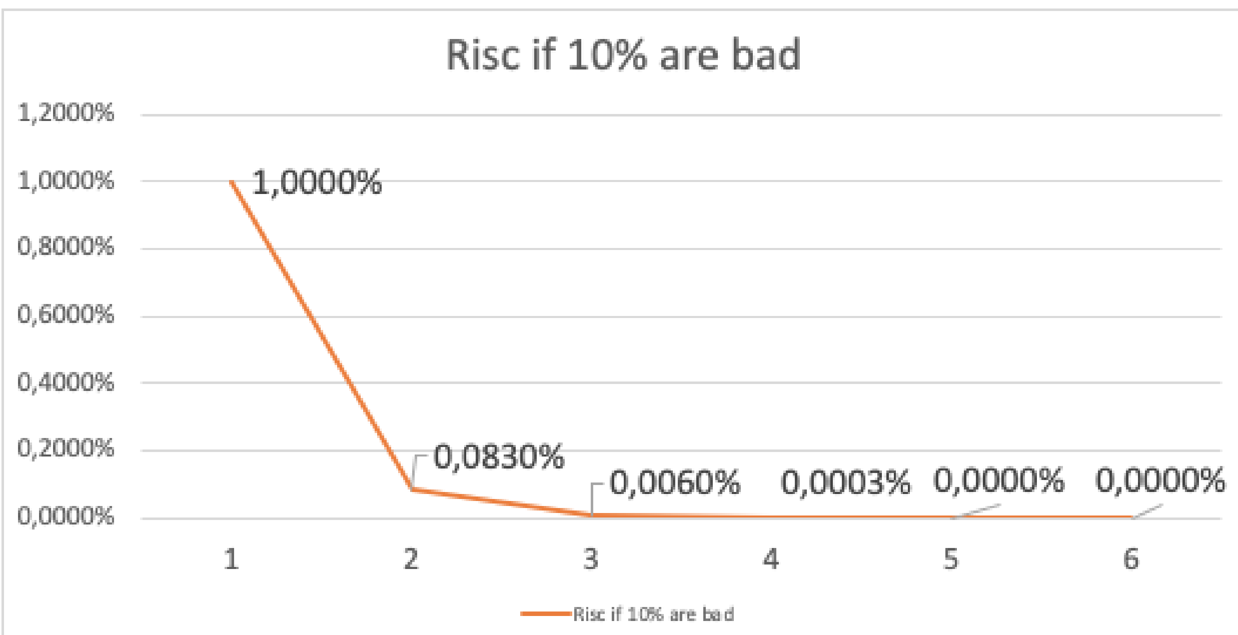
The risk of losing the deposit is significantly lower if the DataProvider node and the signing nodes are run by the same attacker. In this case, they will not only skip over checks, but also prepare the data, the proof, and a blockhash that matches the blockheader. If this were the only request and the client had no other anchor, they would accept a malicious response.

Depending on how many malicious nodes have registered themselves and are working together, the risk can be calculated. If 10% of all registered nodes would be run by an attacker (with the same deposit as the rest), the risk of getting a malicious response would be 1% with only one signature. The risk would go down to 0.006% with three signatures:



50% bad

In case of an attacker controlling 50% of all nodes, it looks a bit different. Here, one signature would give you a risk of 25% to get a bad response, and it would take more than four signatures to reduce this to under 1%.



10% bad

Solution:

> The risk can be reduced by sending two requests in parallel. This way the attacker cannot be sure that their attack would be successful because chances are higher to detect this. If both requests lead to a different result, this conflict can be forwarded to as many servers as possible, where these servers can then check

the blockhash and possibly convict the malicious server.

- genindex

Index

Symbols

A

C

E

I

K

N

P

R

S

V