Incubed Documentation

Release 1.2

Slock.it GmbH

Oct 08, 2019

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	450 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:

```
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	Docker-Hub	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	Docker-Hub	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

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).

3.1 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)

3.2 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

3.3 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.

3.4 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.

3.5 V2.4 Substrate: Q1 2021

Supports Polkadot or any substrate-based chains.

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

3.6 V2.5 Services: Q3 2021

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

CHAPTER 4

Threat Model for Incubed

4.1 Registry Issues

4.1.1 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.

4.1.2 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.

4.1.3 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: 0x2134124213534653463 4634,0xabf2134124213534653463 4634,
↪0xdef2134124213534653463 4634
```

Only if this DNS record contains the signer-address, the client should communicate with this node.

4.1.4 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

4.1.5 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

4.2 Network Attacks

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.3 Privacy

4.3.1 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.

4.3.2 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.

4.3.3 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:

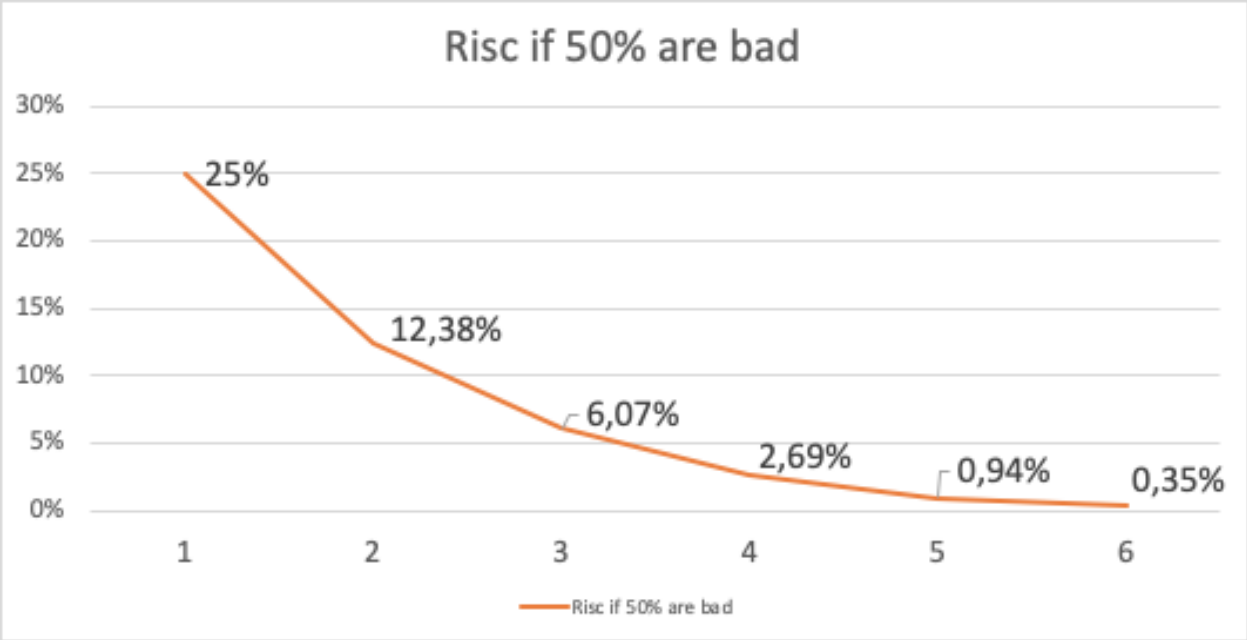
> Encrypt the data by using the public key of the server. This can only be decrypted by the target server with the private key.

4.4 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.

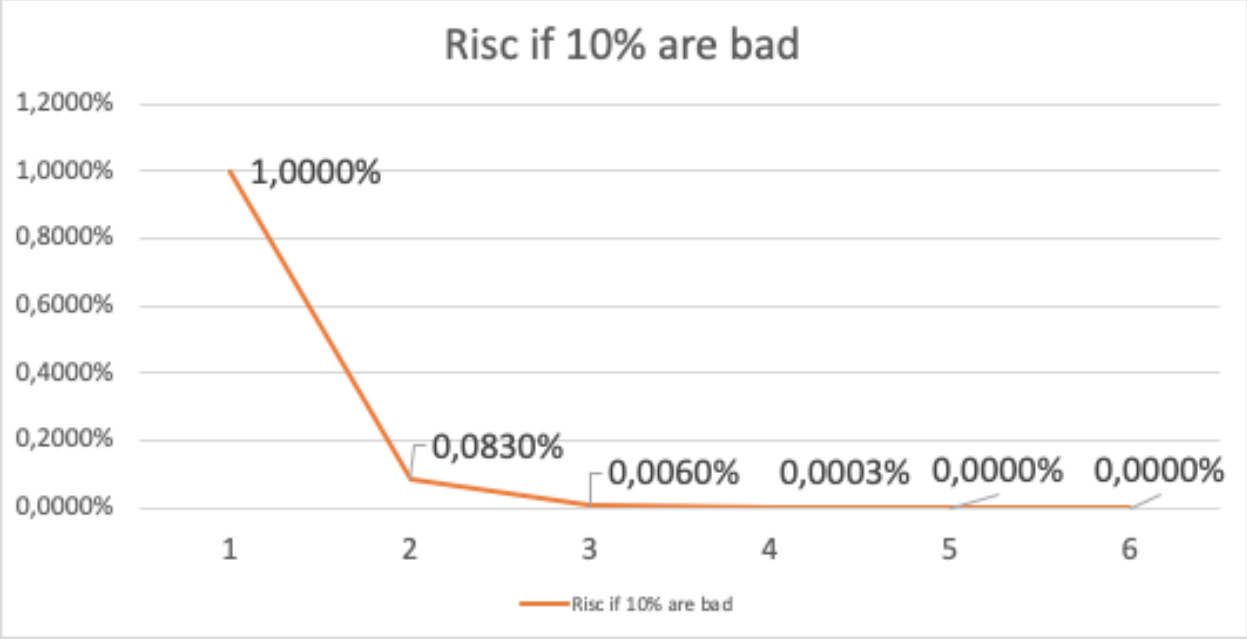
The risk of losing the deposit is significantly lower if the DataProvider node and the signing nodes are run by the same attacker. In this case, they will not only skip over checks, but also prepare the data, the proof, and a blockhash that matches the blockheader. If this were the only request and the 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.

CHAPTER 5

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.

5.1 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.

5.2 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.

5.3 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 6

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.

6.1 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.
  - `'proofWithSignature'` : The returned proof will also include signed blockhashes as required in `signatures`.
- **signatures** `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": "proofWithSignature",
        "signatures":["0x784bfa9eb182C3a02DbeB5285e3dBa92d717E07a"]
  }
}
```

6.2 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*
- **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",
```

(continues on next page)

(continued from previous page)


```
    "chainId": "0x1",
    "condition": null,
    "creates": null,
    "from": "0x2c5811cb45ba9387f2e7c227193ad10014960bfc",
    "gas": "0x186a0",
    "gasPrice": "0x4a817c800",
    "hash": "0xf84cfb78971ebd940d7e4375b077244e93db2c3f88443bb93c561812cfed055c",
    "input":
↪"0xa9059cbb000000000000000000000000290648fc6f2cb27a2a81dc35a429090872991b920000000000000000000000000000
↪",
    "nonce": "0xa8",
    "publicKey":
↪"0x6b30c392dda89d58866bf2c1bedf8229d12c6ae3589d82d0f52ae588838a475aacda64775b7a1b376935d732bb802263
↪",
    "r": "0x4666976b528fc7802edd9330b935c7d48fce0144ce97ade8236da29878c1aa96",
    "raw":
↪"0xf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb000000000000
↪",
    "s": "0x5089dca7ecf7b061bec3cca7726aab1fcb4c8beb51517886f91c9b0ca710b09d",
    "standardV": "0x0",
    "to": "0xd3ebdaea9aeac98de723f640bce4aa07e2e44192",
    "transactionIndex": "0x3e",
    "v": "0x25",
    "value": "0x0"
  },
  "id": 2,
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block":
↪"0xf90219a03d050deecd980b16cad9752133333ccdface463cc69e784f32dd981e2e751e34a01dcc4de8dec75d7aab85b5
↪",
      "merkleProof": [

↪"0xf90131a00150ff50e29f3df34b89870f183c85a82a73f21722d7e6c787e663159f165010a0b8c56f207a223067c7ae5c
↪",

↪"0xf90211a0f4a5e4a1197190f910e4a026f50bd6a169716b52be42c99ddb043ad9b4da6117a09ad1def70dd1d991331d01
↪",

↪"0xf8b020b8adf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb00
↪"
      ],
      "txIndex": 62,
      "signatures": [
        {
          "blockHash":
↪"0x2dbbac3abe47a1d0a7843d378fe3b8701ca7892f530fd1d2b13a46b202af4297",
          "block": 7994038,
          "r": "0xef73a527ae8d38b595437e6436bd4fa037d50550bf3840ad0cd3c6ca641a951e",
          "s": "0x6a5815db16c12b890347d42c014d19b60e1605d2e8e64b729f89e662f9ce706b",
          "v": 27,
          "msgHash":
↪"0xa8fc6e2564e496efc5fd7db8e70f03fd50af53e092f47c98329c84c96026fdff"
        }
      ]
    },
```


(continues on next page)

(continued from previous page)

```
    "currentBlock": 7994124,
    "lastValidatorChange": 0,
    "lastNodeList": 6619795
  }
}
```

6.3 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.

6.4 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.

6.4.1 Node structure

Each Incubed node must be registered in the ServerRegistry 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** `uint64` - A bitmask defining the capabilities of the node:

  - `0x01` : **proof** : 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.
  - `0x02` : **multichain** : The same RPC endpoint may also accept requests for different chains.
  - `0x04` : **archive** : If set, the node is able to support archive requests returning older states. If not, only a pruned node is running.
  - `0x08` : **http** : 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.
  - `0x10` : **binary** : If set, the node accepts request with `binary:true`. This reduces the payload to about 30% for embedded devices.

  More properties 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.

The following functions are offered within the registry:

6.4.2 NodeRegistry functions

constructor

constructor

Development notice: *cannot be deployed in a genesis block*

Parameters:

- _blockRegistry `BlockhashRegistry`: *address of a BlockhashRegistry-contract*

convict

must be called before revealConvict commits a blocknumber and a hash

Development notice: *The v,r,s paramaters are from the signature of the wrong blockhash that the node provided*

Parameters:

- _blockNumber `uint`: *the blocknumber of the wrong blockhash*
- _hash `bytes32`: *keccak256(wrong blockhash, msg.sender, v, r, s); used to prevent frontrunning.*

registerNode

register a new node with the sender as owner

Development notice: *will call the registerNodeInteral function*

Parameters:

- _url `string`: *the url of the node, has to be unique*

- _props `uint64`: *properties of the node*
- _timeout `uint64`: *timespan of how long the node of a deposit will be locked. Will be at least for 1h*
- _weight `uint64`: *how many requests per second the node is able to handle*

registerNodeFor

register 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, timeout, 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

Parameters:

- _url `string`: *the url of the node, has to be unique*
- _props `uint64`: *properties of the node*
- _timeout `uint64`: *timespan of how long the node of a deposit will be locked. Will be at least for 1h*
- _signer `address`: *the signer of the in3-node*
- _weight `uint64`: *how many requests per second the node is able to handle*
- _v `uint8`: *v of the signed message*
- _r `bytes32`: *r of the signed message*
- _s `bytes32`: *s of the signed message*

removeNodeFromRegistry

removes an in3-server from the registry

Development notice: *only callable in the 1st year after deployment*

only callable by the unregisterKey-account

Parameters:

- _signer `address`: *the signer-address of the in3-node*

returnDeposit

only callable after the timeout of the deposit is over returns the deposit after a node has been removed

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*

revealConvict

reveals the wrongly provided blockhash, so that the node-owner will lose its 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*

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*

unregisteringNode

doing so will also lock his deposit for the timeout of the node a node owner can unregister a node, removing it from the nodeList

Development notice: *reverts when not called by the owner of the node*

reverts when the provided address is not an in3-signer

Parameters:

- _signer `address`: *the signer of the in3-node*

updateNode

updates a node by adding the msg.value to the deposit and setting the props or timeout

Development notice: *reverts when trying to change the url to an already existing one*

reverts when trying to increase the timeout above 10 years

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 onec*
- _timeout `uint64`: *the new timeout of the node, cannot be decreased. Has to be at least 1h*
- _weight `uint64`: *the amount of requests per second the node is able to handle*

totalNodes

length of the nodelist **Return Parameters:**

- `uint` the number of currently active nodes

calcProofHash

calculates the sha3 hash of the most important properties in order to make the proof faster

Parameters:

- _node `In3Node`: *the in3 node to calculate the hash from*

Return Parameters:

- `bytes32` the hash of the properties to prove with in3

checkNodeProperties

function to check whether the allowed amount of ether as deposit per server has been reached

Development notice: *will fail when the provided timeout is greater then 1 year*

will fail when the deposit is greater then 50 ether in the 1st year

Parameters:

- _deposit `uint256`: *the new amount of deposit a server has*
- _timeout `uint64`: *the timeout until a server can receive his deposit after unregister*

registerNodeInternal

registers a node

Development notice: *reverts when either the owner or the url is already in use*

reverts when trying to register a node with more then 50 ether in the 1st year after deployment

reverts when provided not enough deposit

reverts when time timeout exceed the MAXDEPOSITTIMEOUT

Parameters:

- _url `string`: *the url of a node*
- _props `uint64`: *properties of a node*
- _timeout `uint64`: *the time before the owner can access the deposit after unregister a node*
- _signer `address`: *the address that signs the answers of the node*
- _owner `address`: *the owner address of the node*
- _deposit `uint`: *the deposit of a node*
- _weight `uint64`: *the amount of requests per second a node is able to handle*

unregisterNodeInternal

handles the setting of the unregister values for a node internally

Parameters:

- _si `SignerInformation`: *information of the signer*
- _n `In3Node`: *information of the in3-node*

removeNode

removes a node from the node-array

6.4.3 BlockHashRegistry functions

constructor

constructor

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.

recreateBlockheaders

if successfull 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 only usable when the given blocknumber is already in the smart contract starts with a given blocknumber and its header and tries to recreate a (reverse) chain of blocks

Development notice: *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)*

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*

snapshot

stores the currentBlock-1 in the smart contract

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`

reCalculateBlockheaders

the array of the blockheaders have to be in reverse order (e.g. [100,99,98,97]) starts with a given blockhash and its header and tries to recreate a (reverse) chain of blocks

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*

6.5 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).

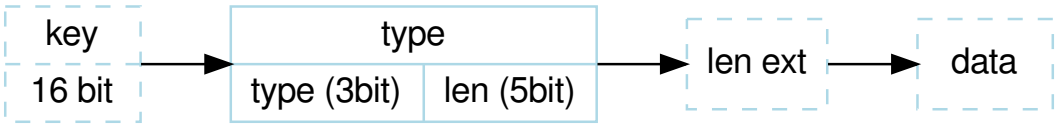
  The hash is calculated very easily like this:

```
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.

6.6 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:

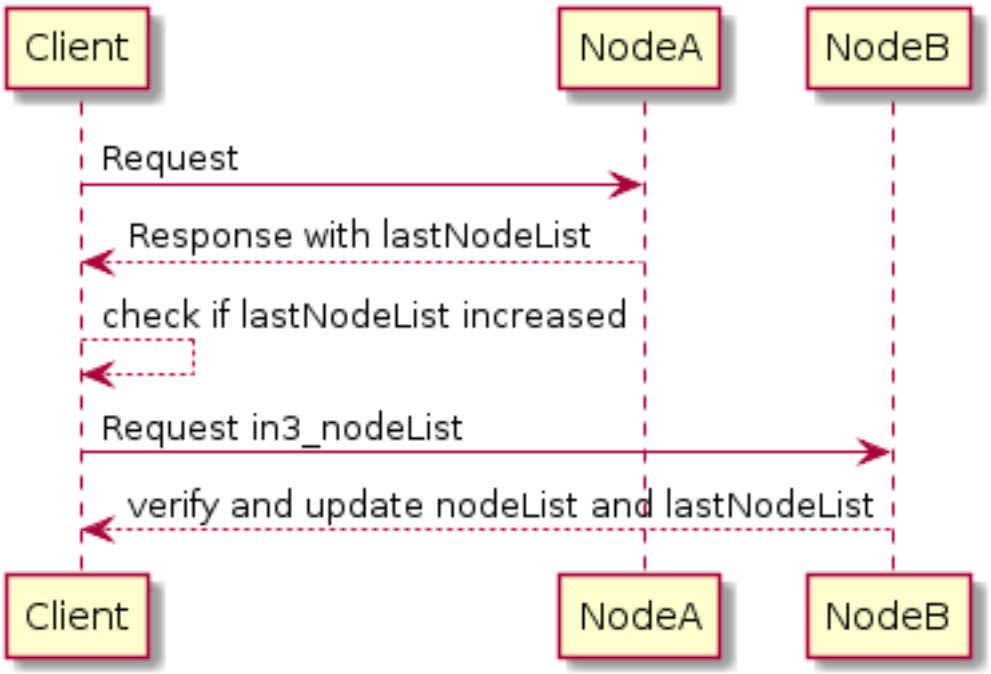
1. **Changes in the NodeRegistry**

   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.

6.7 Proofs

Proofs are a crucial part of the security concept for Incubed. 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.

6.7.1 Ethereum

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.

- ***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 an `eth_call`-response.

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.

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 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')
```

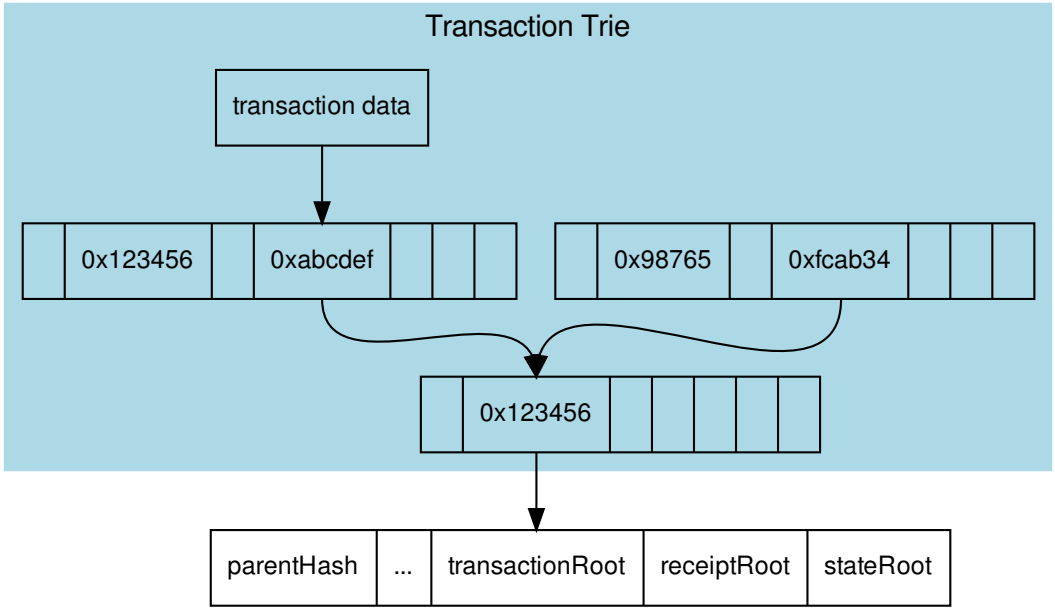
In case of the `eth_getBlockTransactionCountBy...`, the proof contains the full blockHeader already serialized plus all transactionHashes. This is needed in order to verify them in a merkle tree and compare them with the `transactionRoot`.

Transaction Proof

TransactionProofs are used for the following transaction-methods:

- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex

- eth_getTransactionByBlockNumberAndIndex



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 ),
  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.

The proof-data will look like these:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "blockHash": "0xf1a2fd6a36f27950c78ce559b1dc4e991d46590683cb8cb84804fa672bca395b",
    "blockNumber": "0xca",
    "from": "0x7e5f4552091a69125d5dfcb7b8c2659029395bdf",
    "gas": "0x55f0",
    "gasPrice": "0x0",
```

(continues on next page)

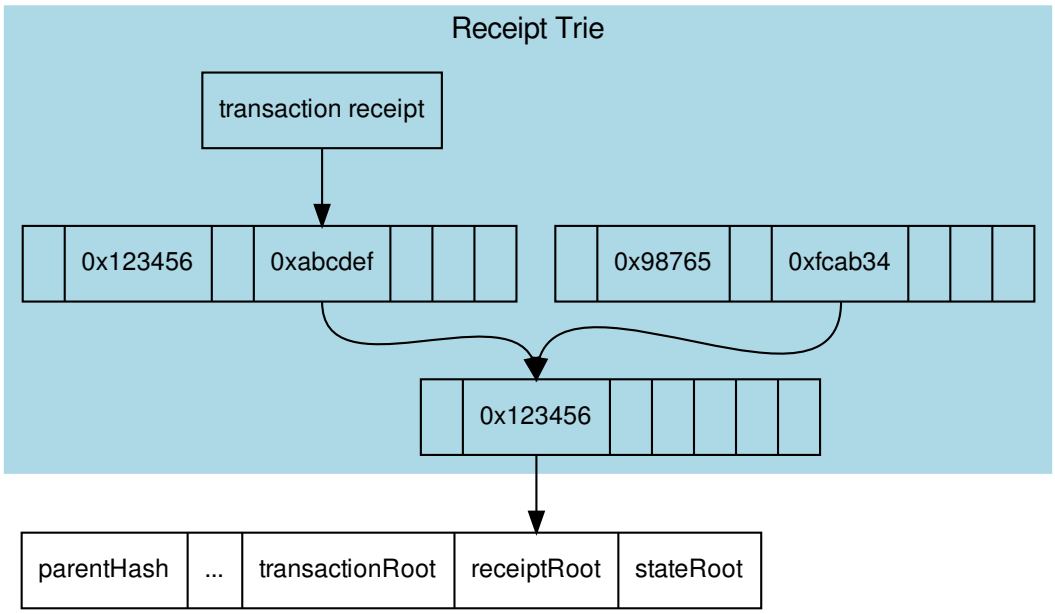
(continued from previous page)

```
    "hash": "0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b",
    "input": "0x00",
    "value": "0x3e8"
    ...
  },
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block": "0xf901e6a040997a53895b48...", // serialized blockheader
      "merkleProof": [  /* serialized nodes starting with the root-node */
        "0xf868822080b863f86136808255f0942b5ad5c4795c026514f8317c7a215e218dc..."
        "0xcd6cf8203e8001ca0dc967310342af5042bb64c34d3b92799345401b26713b43f..."
      ],
      "txIndex": 0,
      "signatures": [...]
    }
  }
}
```

Receipt Proof

Proofs for the transactionReceipt are used for the following method:

- eth_getTransactionReceipt



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

and store them in a merkle tree with `elp.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*.

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 TransactionReceipt 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*.

Account Proof

Proofing an account-value applies to these functions:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt

Each of these 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
)
```

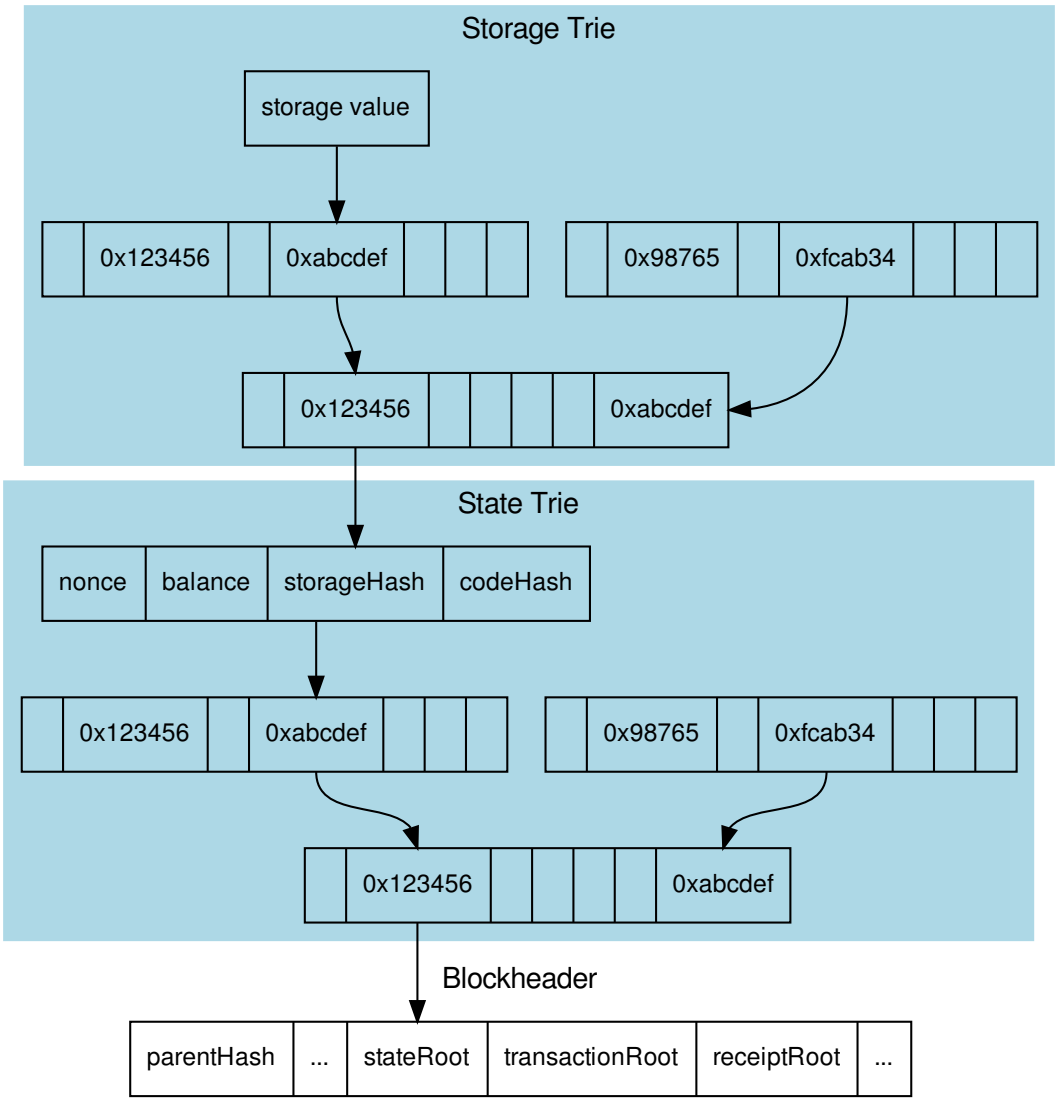
In case the account does not exist yet (which is the case if `none == startNonce` and `codeHash == '0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470'`), the proof may end with one of these nodes:

- 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
))
```



Call Proof

Call proofs are used whenever you are calling a read-only function of a smart contract:

- eth_call

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 )))

  // 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.

6.8 RPC-Methods Ethereum

This section describes the behavior for each standard-RPC-method.

6.8.1 web3_clientVersion

Returns the underlying client version.

See web3_clientversion for spec. No proof or verification possible.

6.8.2 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.

6.8.3 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.

6.8.4 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.

6.8.5 eth_getBalance

Returns the balance of the account of a given address.

See eth_getBalance for spec.

An AccountProof, 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.

6.9 PoA Validations

CHAPTER 7

API Reference TS

This page contains a list of all Datastructures and Classes used within the TypeScript IN3 Client.

7.1 Examples

This is a collection of different incubed-examples.

7.1.1 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')
...
```

7.1.2 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
})

...
```

7.1.3 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()}`)

...
```

7.2 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:

‘

- ***AccountProof*** : `interface` - the Proof-for a single Account
- ***AuraValidatoryProof*** : `interface` - a Object holding proofs for validator logs. The key is the blockNumber as hex
- ***ChainSpec*** : `interface` - describes the chainspecific consensus params
- ***IN3Client*** : `class` - Client for N3.
- ***IN3Config*** : `interface` - the iguration of the IN3-Client. This can be paritally overriden for every 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)
- ***IN3RPCConfig*** : `interface` - the configuration for the rpc-handler
- ***IN3RPCHandlerConfig*** : `interface` - the configuration for the rpc-handler
- ***IN3RPCRequestConfig*** : `interface` - additional config for a IN3 RPC-Request
- ***IN3ResponseConfig*** : `interface` - additional data returned from a IN3 Server
- ***LogProof*** : `interface` - a Object holding proofs for event logs. The key is the blockNumber as hex
- ***Proof*** : `interface` - the Proof-data as part of the in3-section
- ***RPCRequest*** : `interface` - a JSONRPC-Request with N3-Extension
- ***RPCResponse*** : `interface` - a JSONRPC-Responset with N3-Extension
- ***ServerList*** : `interface` - a List of nodes
- ***Signature*** : `interface` - 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.
- ***EthAPI*** : `class`
- **chainAliases**
  - **goerli** :`string`
  - **ipfs** :`string`
  - **kovan** :`string`
  - **main** :`string`
  - **mainnet** :`string`

  - **tobalaba** :`string`

- **chainData**
  - **callContract**(client :*Client*, contract :`string`, chainId :`string`, signature :`string`, args :`any`[], config :*IN3Config*) :`Promise<any>`
  - **getChainData**(client :*Client*, chainId :`string`, config :*IN3Config*) :`Promise<>`
- **createRandomIndexes**(len :`number`, limit :`number`, seed :*Buffer*, result :`number`[] = []) :`number`[] - helper function creating deterministic random indexes used for limited nodelists
- **header**
  - ***AuthSpec*** :`interface` - Authority specification for proof of authority chains
  - **checkBlockSignatures**(blockHeaders :`any`[], getChainSpec :) :`Promise<number>` - verify a Blockheader and returns the percentage of finality
  - **getChainSpec**(b :*Block*, ctx :*ChainContext*) :*Promise<AuthSpec>*
  - **getSigner**(data :*Block*) :*Buffer*
- **typeDefs**
  - **AccountProof** : `Object`
  - **AuraValidatoryProof** : `Object`
  - **ChainSpec** : `Object`
  - **IN3Config** : `Object`
  - **IN3NodeConfig** : `Object`
  - **IN3NodeWeight** : `Object`
  - **IN3RPCConfig** : `Object`
  - **IN3RPCHandlerConfig** : `Object`
  - **IN3RPCRequestConfig** : `Object`
  - **IN3ResponseConfig** : `Object`
  - **LogProof** : `Object`
  - **Proof** : `Object`
  - **RPCRequest** : `Object`
  - **RPCResponse** : `Object`
  - **ServerList** : `Object`
  - **Signature** : `Object`
- **util** :`any`

7.3 Package client

7.3.1 Type Client

Client for N3.

Source: client/Client.ts

- **defaultMaxListeners** :number
- static **listenerCount**(emitter :*EventEmitter*, event :string|symbol) :number
- constructor **constructor**(config :*Partial<IN3Config>* = { }, transport :*Transport*) :*Client* - creates a new Client.
- **defConfig** :*IN3Config* - the iguration of the IN3-Client. This can be paritally overriden for every request.
- **eth** :*EthAPI*
- **ipfs** :*IpfsAPI* - simple API for IPFS
- **config**()
- **addListener**(event :string|symbol, listener :) :this
- **call**(method :string, params :any, chain :string, config :*Partial<IN3Config>*) :Promise<any> - sends a simply RPC-Request
- **clearStats**() :void - clears all stats and weights, like blocklisted nodes
- **createWeb3Provider**() :any
- **emit**(event :string|symbol, args :any[]) :boolean
- **eventNames**() :*Array<>*
- **getChainContext**(chainId :string) :*ChainContext*
- **getMaxListeners**() :number
- **listenerCount**(type :string|symbol) :number
- **listeners**(event :string|symbol) :*Function*[]
- **off**(event :string|symbol, listener :) :this
- **on**(event :string|symbol, listener :) :this
- **once**(event :string|symbol, listener :) :this
- **prependListener**(event :string|symbol, listener :) :this
- **prependOnceListener**(event :string|symbol, listener :) :this
- **rawListeners**(event :string|symbol) :*Function*[]
- **removeAllListeners**(event :string|symbol) :this
- **removeListener**(event :string|symbol, listener :) :this
- **send**(request :*RPCRequest*[]|*RPCRequest*, callback :, config :*Partial<IN3Config>*) :Promise<> - 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.
- **sendRPC**(method :string, params :any[] = [], chain :string, config :*Partial<IN3Config>*) :*Promise<RPCResponse>* - sends a simply RPC-Request
- **setMaxListeners**(n :number) :this
- **updateNodeList**(chainId :string, conf :*Partial<IN3Config>*, retryCount :number = 5) :Promise<void> - fetches the nodeList from the servers.

7.3.2 Type ChainContext

Context for a specific chain including cache and chainSpecs.

Source: client/ChainContext.ts

- `constructor` **constructor**(client :*Client*, chainId :`string`, chainSpec :*ChainSpec*[]) :*ChainContext*
- **chainId** :`string`
- **chainSpec** :*ChainSpec*[]
- **client** :*Client* - Client for N3.
- **genericCache**
- **lastValidatorChange** :`number`
- **module** :*Module*
- **registryId** :`string` *(optional)*
- **clearCache**(prefix :`string`) :`void`
- **getChainSpec**(block :`number`) :*ChainSpec* - returns the chainspec for th given block number
- **getFromCache**(key :`string`) :`string`
- **handleIntern**(request :*RPCRequest*) :*Promise<RPCResponse>* - 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
- **initCache**() :`void`
- **putInCache**(key :`string`, value :`string`) :`void`
- **updateCache**() :`void`

7.3.3 Type Module

Source: client/modules.ts

- **name** :`string`
- **createChainContext**(client :*Client*, chainId :`string`, spec :*ChainSpec*[]) :*ChainContext*
- **verifyProof**(request :*RPCRequest*, response :*RPCResponse*, allowWithoutProof :`boolean`, ctx :*ChainContext*) :`Promise<boolean>` - general verification-function which handles it according to its given type.

7.4 Package modules/eth

7.4.1 Type EthAPI

Source: modules/eth/api.ts

- `constructor` **constructor**(client :*Client*) :*EthAPI*
- **client** :*Client* - Client for N3.
- **signer** :*Signer* *(optional)*

- **blockNumber**() :Promise<number> - Returns the number of most recent block. (as number)
- **call**(tx :*Transaction*, block :*BlockType* = "latest") :Promise<string> - Executes a new message call immediately without creating a transaction on the block chain.
- **callFn**(to :*Address*, method :string, args :any[]) :Promise<any> - 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.
- **chainId**() :Promise<string> - 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*) :
- **decodeEventData**(log :*Log*, d :*ABI*) :any
- **estimateGas**(tx :*Transaction*) :Promise<number> - 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.
- **gasPrice**() :Promise<number> - Returns the current price per gas in wei. (as number)
- **getBalance**(address :*Address*, block :*BlockType* = "latest") :*Promise<BN>* - Returns the balance of the account of given address in wei (as hex).
- **getBlockByHash**(hash :*Hash*, includeTransactions :boolean = false) :*Promise<Block>* - Returns information about a block by hash.
- **getBlockByNumber**(block :*BlockType* = "latest", includeTransactions :boolean = false) :*Promise<Block>* - Returns information about a block by block number.
- **getBlockTransactionCountByHash**(block :*Hash*) :Promise<number> - Returns the number of transactions in a block from a block matching the given block hash.
- **getBlockTransactionCountByNumber**(block :*Hash*) :Promise<number> - Returns the number of transactions in a block from a block matching the given block number.
- **getCode**(address :*Address*, block :*BlockType* = "latest") :Promise<string> - Returns code at a given address.
- **getFilterChanges**(id :*Quantity*) :Promise<> - Polling method for a filter, which returns an array of logs which occurred since last poll.
- **getFilterLogs**(id :*Quantity*) :Promise<> - Returns an array of all logs matching filter with given id.
- **getLogs**(filter :*LogFilter*) :Promise<> - Returns an array of all logs matching a given filter object.
- **getStorageAt**(address :*Address*, pos :*Quantity*, block :*BlockType* = "latest") :Promise<string> - Returns the value from a storage position at a given address.
- **getTransactionByBlockHashAndIndex**(hash :*Hash*, pos :*Quantity*) :*Promise<TransactionDetail>* - Returns information about a transaction by block hash and transaction index position.
- **getTransactionByBlockNumberAndIndex**(block :*BlockType*, pos :*Quantity*) :*Promise<TransactionDetail>* - Returns information about a transaction by block number and transaction index position.
- **getTransactionByHash**(hash :*Hash*) :*Promise<TransactionDetail>* - Returns the information about a transaction requested by transaction hash.
- **getTransactionCount**(address :*Address*, block :*BlockType* = "latest") :Promise<number> - Returns the number of transactions sent from an address. (as number)
- **getTransactionReceipt**(hash :*Hash*) :*Promise<TransactionReceipt>* - Returns the receipt of a transaction by transaction hash. Note That the receipt is available even for pending transactions.

- **getUncleByBlockHashAndIndex**(hash :*Hash*, pos :*Quantity*) :*Promise<Block>* - Returns information about a uncle of a block by hash and uncle index position. Note: An uncle doesn't contain individual transactions.
- **getUncleByBlockNumberAndIndex**(block :*BlockType*, pos :*Quantity*) :*Promise<Block>* - Returns information about a uncle of a block number and uncle index position. Note: An uncle doesn't contain individual transactions.
- **getUncleCountByBlockHash**(hash :*Hash*) :Promise<number> - Returns the number of uncles in a block from a block matching the given block hash.
- **getUncleCountByBlockNumber**(block :*BlockType*) :Promise<number> - Returns the number of uncles in a block from a block matching the given block hash.
- **hashMessage**(data :*Data*|*Buffer*) :*Buffer*
- **newBlockFilter**() :Promise<string> - Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.
- **newFilter**(filter :*LogFilter*) :Promise<string> - 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.
- **newPendingTransactionFilter**() :Promise<string> - Creates a filter in the node, to notify when new pending transactions arrive.
- **protocolVersion**() :Promise<string> - Returns the current ethereum protocol version.
- **sendRawTransaction**(data :*Data*) :Promise<string> - Creates new message call transaction or a contract creation for signed transactions.
- **sendTransaction**(args :*TxRequest*) :Promise<> - sends a Transaction
- **sign**(account :*Address*, data :*Data*) :*Promise<Signature>* - signs any kind of message using the \x19Ethereum Signed Message:\n-prefix
- **syncing**() :Promise<> - Returns the current ethereum protocol version.
- **uninstallFilter**(id :*Quantity*) :*Promise<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.

7.4.2 Type AuthSpec

Authority specification for proof of authority chains

Source: modules/eth/header.ts

- **authorities** :*Buffer*[] - List of validator addresses storead as an buffer array
- **proposer** :*Buffer* - proposer of the block this authspec belongs
- **spec** :*ChainSpec* - chain specification

7.4.3 Type Block

Source: modules/eth/api.ts

- **Block**
  - **Hex** :string
  - **Quantity** :number|*Hex*

  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Hex** :`string`
  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Hex** :`string`
  - **sealFields** :*Data*[] - PoA-Fields
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Quantity** :`number`|*Hex*
  - **transactions** :`string`|[] - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter
  - **Hex** :`string`
  - **uncles** :*Hash*[] - Array of uncle hashes

7.4.4 Type Signer

Source: modules/eth/api.ts

- **prepareTransaction** *(optional)* - optiional method which allows to change the transaction-data before sending it. This can be used for redirecting it through a multisig.
- **sign** - signing of any data.
- **hasAccount**(account :*Address*) :`Promise<boolean>` - returns true if the account is supported (or unlocked)

7.4.5 Type Transaction

Source: modules/eth/api.ts

- **Transaction**
  - **chainId** :`any` *(optional)* - optional chain id
  - **data** :`string` - 4 byte hash of the method signature followed by encoded parameters. For details see Ethereum Contract ABI.
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*

  - **Quantity** :number|*Hex*
  - **Quantity** :number|*Hex*
  - **Hex** :string
  - **Quantity** :number|*Hex*

7.4.6 Type BlockType

Source: modules/eth/api.ts

- **BlockType** :number|'latest'|'earliest'|'pending'

7.4.7 Type Address

Source: modules/eth/api.ts

- **Hex** :string

7.4.8 Type ABI

Source: modules/eth/api.ts

- **ABI**
  - **anonymous** :boolean *(optional)*
  - **constant** :boolean *(optional)*
  - **inputs** :*ABIField*[] *(optional)*
  - **name** :string *(optional)*
  - **outputs** :*ABIField*[] *(optional)*
  - **payable** :boolean *(optional)*
  - **stateMutability** :'nonpayable'|'payable'|'view'|'pure' *(optional)*
  - **type** :'event'|'function'|'constructor'|'fallback'

7.4.9 Type Log

Source: modules/eth/api.ts

- **Log**
  - **Hex** :string
  - **Hex** :string
  - **Quantity** :number|*Hex*
  - **Hex** :string
  - **Quantity** :number|*Hex*
  - **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.)
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*

7.4.10 Type Hash

Source: modules/eth/api.ts

- **Hex** :`string`

7.4.11 Type Quantity

Source: modules/eth/api.ts

- **Quantity** :`number`|*Hex*

7.4.12 Type LogFilter

Source: modules/eth/api.ts

- **LogFilter**
  - **Hex** :`string`
  - **BlockType** :`number`|`'latest'`|`'earliest'`|`'pending'`
  - **Quantity** :`number`|*Hex*
  - **BlockType** :`number`|`'latest'`|`'earliest'`|`'pending'`
  - **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.

7.4.13 Type TransactionDetail

Source: modules/eth/api.ts

- **TransactionDetail**
  - **Hex** :`string`
  - **BlockType** :`number`|`'latest'`|`'earliest'`|`'pending'`
  - **Quantity** :`number`|*Hex*
  - **condition** :`any` - (optional) conditional submission, Block number in block or timestamp in time or null. (parity-feature)
  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Quantity** :`number`|*Hex*

  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **pk** :`any` *(optional)* - optional: the private key to use for signing
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Quantity** :`number`|*Hex*
  - **Quantity** :`number`|*Hex*

7.4.14 Type TransactionReceipt

Source: modules/eth/api.ts

- **TransactionReceipt**
  - **Hex** :`string`
  - **BlockType** :`number`|`'latest'`|`'earliest'`|`'pending'`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **logs** :*Log*[] - Array of log objects, which this transaction generated.
  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*
  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*

7.4.15 Type Data

Source: modules/eth/api.ts

- **Hex** :`string`

7.4.16 Type TxRequest

Source: modules/eth/api.ts

- **TxRequest**
  - **args** :`any`[] *(optional)* - the argument to pass to the method
  - **confirmations** :`number` *(optional)* - number of block to wait before confirming
  - **Hex** :`string`
  - **Hex** :`string`
  - **gas** :`number` *(optional)* - the gas needed
  - **gasPrice** :`number` *(optional)* - the gasPrice used
  - **method** :`string` *(optional)* - the ABI of the method to be used
  - **nonce** :`number` *(optional)* - the nonce
  - **Hex** :`string`
  - **Hex** :`string`
  - **Quantity** :`number`|*Hex*

7.4.17 Type Hex

Source: modules/eth/api.ts

- **Hex** :`string`

7.4.18 Type ABIField

Source: modules/eth/api.ts

- **ABIField**
  - **indexed** :`boolean` *(optional)*
  - **name** :`string`
  - **type** :`string`

7.5 Package modules/ipfs

7.5.1 Type IpfsAPI

simple API for IPFS

Source: modules/ipfs/api.ts

- `constructor` **constructor**(_client :*Client*) :*IpfsAPI*
- **client** :*Client* - Client for N3.
- **get**(hash :`string`, resultEncoding :`string`) :*Promise<Buffer>* - retrieves the conent for a hash from IPFS.

- **put**(data :*Buffer*, dataEncoding :`string`) :`Promise<string>` - stores the data on ipfs and returns the IPFS-Hash.

7.6 Package types

7.6.1 Type AccountProof

the Proof-for a single Account

Source: types/types.ts

- **accountProof** :`string`[] - the serialized merle-noodes beginning with the root-node
- **address** :`string` - the address of this account
- **balance** :`string` - the balance of this account as hex
- **code** :`string` *(optional)* - the code of this account as hex (if required)
- **codeHash** :`string` - the codeHash of this account as hex
- **nonce** :`string` - the nonce of this account as hex
- **storageHash** :`string` - the storageHash of this account as hex
- **storageProof** :[] - proof for requested storage-data

7.6.2 Type AuraValidatoryProof

a Object holding proofs for validator logs. The key is the blockNumber as hex

Source: types/types.ts

- **block** :`string` - the serialized blockheader example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec
- **finalityBlocks** :`any`[] *(optional)* - the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec1ffee98c0437b4ac839d8a2ece1b18166da704b86d8f4
- **logIndex** :`number` - the transaction log index
- **proof** :`string`[] - the merkleProof
- **txIndex** :`number` - the transactionIndex within the block

7.6.3 Type ChainSpec

describes the chainspecific consensus params

Source: types/types.ts

- **block** :`number` *(optional)* - the blocknumnber when this configuration should apply
- **bypassFinality** :`number` *(optional)* - 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
- **contract** :`string` *(optional)* - The validator contract at the block
- **engine** :`'ethHash'|'authorityRound'|'clique'` *(optional)* - the engine type (like Ethhash, authorityRound, ...)
- **list** :`string`[] *(optional)* - The list of validators at the particular block

- **requiresFinality** :`boolean` *(optional)* - indicates whether the transition requires a finality check example: true

7.6.4 Type IN3Config

the iguration of the IN3-Client. This can be paritally overriden for every request.

Source: types/types.ts

- **autoConfig** :`boolean` *(optional)* - if true the config will be adjusted depending on the request
- **autoUpdateList** :`boolean` *(optional)* - if true the nodelist will be automaticly updated if the lastBlock is newer example: true
- **cacheStorage** :`any` *(optional)* - a cache handler offering 2 functions (setItem(string,string), getItem(string))
- **cacheTimeout** :`number` *(optional)* - number of seconds requests can be cached.
- **chainId** :`string` - servers to filter for the given chain. The chain-id based on EIP-155. example: 0x1
- **chainRegistry** :`string` *(optional)* - main chain-registry contract example: 0xe36179e2286ef405e929C90ad3E70E649B22a945
- **finality** :`number` *(optional)* - the number in percent needed in order reach finality (% of signature of the validators) example: 50
- **format** :`'json'|'jsonRef'|'cbor'` *(optional)* - 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
- **includeCode** :`boolean` *(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
- **keepIn3** :`boolean` *(optional)* - 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.
- **key** :`any` *(optional)* - the client key to sign requests example: 0x387a8233c96e1fc0ad5e284353276177af2186e7afa85296f106336
- **loggerUrl** :`string` *(optional)* - 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? }
- **mainChain** :`string` *(optional)* - main chain-id, where the chain registry is running. example: 0x1
- **maxAttempts** :`number` *(optional)* - max number of attempts in case a response is rejected example: 10
- **maxBlockCache** :`number` *(optional)* - number of number of blocks cached in memory example: 100
- **maxCodeCache** :`number` *(optional)* - number of max bytes used to cache the code in memory example: 100000
- **minDeposit** :`number` - min stake of the server. Only nodes owning at least this amount will be chosen.
- **nodeLimit** :`number` *(optional)* - the limit of nodes to store in the client. example: 150
- **proof** :`'none'|'standard'|'full'` *(optional)* - if true the nodes should send a proof of the response example: true
- **replaceLatestBlock** :`number` *(optional)* - if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6
- **requestCount** :`number` - the number of request send when getting a first answer example: 3
- **retryWithoutProof** :`boolean` *(optional)* - if true the the request may be handled without proof in case of an error. (use with care!)
- **rpc** :`string` *(optional)* - url of one or more rpc-endpoints to use. (list can be comma seperated)

- **servers** *(optional)* - the nodelist per chain
- **signatureCount** :`number` *(optional)* - number of signatures requested example: 2
- **timeout** :`number` *(optional)* - specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection. example: 3000
- **verifiedHashes** :`string`[] *(optional)* - 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.

7.6.5 Type IN3NodeConfig

a configuration of a in3-server.

Source: types/types.ts

- **address** :`string` - the address of the node, which is the public address it iis signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **capacity** :`number` *(optional)* - the capacity of the node. example: 100
- **chainIds** :`string`[] - the list of supported chains example: 0x1
- **deposit** :`number` - the deposit of the node in wei example: 12350000
- **index** :`number` *(optional)* - the index within the contract example: 13
- **props** :`number` *(optional)* - the properties of the node. example: 3
- **registerTime** :`number` *(optional)* - the UNIX-timestamp when the node was registered example: 1563279168
- **timeout** :`number` *(optional)* - the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600
- **unregisterTime** :`number` *(optional)* - the UNIX-timestamp when the node is allowed to be deregister example: 1563279168
- **url** :`string` - the endpoint to post to example: https://in3.slock.it

7.6.6 Type IN3NodeWeight

a local weight of a n3-node. (This is used internally to weight the requests)

Source: types/types.ts

- **avgResponseTime** :`number` *(optional)* - average time of a response in ms example: 240
- **blacklistedUntil** :`number` *(optional)* - blacklisted because of failed requests until the timestamp example: 1529074639623
- **lastRequest** :`number` *(optional)* - timestamp of the last request in ms example: 1529074632623
- **pricePerRequest** :`number` *(optional)* - last price
- **responseCount** :`number` *(optional)* - number of uses. example: 147
- **weight** :`number` *(optional)* - factor the weight this noe (default 1.0) example: 0.5

7.6.7 Type IN3RPCConfig

the configuration for the rpc-handler

Source: types/types.ts

- **chains** *(optional)* - a definition of the Handler per chain
- **db** *(optional)*
  - **database** :`string` *(optional)* - name of the database
  - **host** :`string` *(optional)* - db-host (default = localhost)
  - **password** :`string` *(optional)* - password for db-access
  - **port** :`number` *(optional)* - the database port
  - **user** :`string` *(optional)* - username for the db
- **defaultChain** :`string` *(optional)* - the default chainId in case the request does not contain one.
- **id** :`string` *(optional)* - a identifier used in logfiles as also for reading the config from the database
- **logging** *(optional)* - logger config
  - **colors** :`boolean` *(optional)* - if true colors will be used
  - **file** :`string` *(optional)* - the path to the logile
  - **host** :`string` *(optional)* - the host for custom logging
  - **level** :`string` *(optional)* - Loglevel
  - **name** :`string` *(optional)* - the name of the provider
  - **port** :`number` *(optional)* - the port for custom logging
  - **type** :`string` *(optional)* - the module of the provider
- **port** :`number` *(optional)* - the listeneing port for the server
- **profile** *(optional)*
  - **comment** :`string` *(optional)* - comments for the node
  - **icon** :`string` *(optional)* - url to a icon or logo of company offering this node
  - **name** :`string` *(optional)* - name of the node or company
  - **noStats** :`boolean` *(optional)* - if active the stats will not be shown (default:false)
  - **url** :`string` *(optional)* - url of the website of the company

7.6.8 Type IN3RPCHandlerConfig

the configuration for the rpc-handler

Source: types/types.ts

- **autoRegistry** *(optional)*
  - **capabilities** *(optional)*
    - **multiChain** :`boolean` *(optional)* - if true, this node is able to deliver multiple chains
    - **proof** :`boolean` *(optional)* - if true, this node is able to deliver proofs

  - **capacity** :`number` *(optional)* - max number of parallel requests
  - **deposit** :`number` - the deposit you want ot store
  - **depositUnit** :`'ether'|'finney'|'szabo'|'wei'` *(optional)* - unit of the deposit value
  - **url** :`string` - the public url to reach this node
- **clientKeys** :`string` *(optional)* - a comma sepearted list of client keys to use for simulating clients for the watchdog
- **freeScore** :`number` *(optional)* - the score for requests without a valid signature
- **handler** :`'eth'|'ipfs'|'btc'` *(optional)* - the impl used to handle the calls
- **ipfsUrl** :`string` *(optional)* - the url of the ipfs-client
- **maxThreads** :`number` *(optional)* - the maximal number of threads ofr running parallel processes
- **minBlockHeight** :`number` *(optional)* - the minimal blockheight in order to sign
- **persistentFile** :`string` *(optional)* - the filename of the file keeping track of the last handled blocknumber
- **privateKey** :`string` - 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.
- **privateKeyPassphrase** :`string` *(optional)* - the password used to decrpyt the private key
- **registry** :`string` - the address of the server registry used in order to update the nodeList
- **registryRPC** :`string` *(optional)* - the url of the client in case the registry is not on the same chain.
- **rpcUrl** :`string` - the url of the client
- **startBlock** :`number` *(optional)* - blocknumber to start watching the registry
- **timeout** :`number` *(optional)* - number of milliseconds to wait before a request gets a timeout
- **watchInterval** :`number` *(optional)* - the number of seconds of the interval for checking for new events
- **watchdogInterval** :`number` *(optional)* - average time between sending requests to the same node. 0 turns it off (default)

7.6.9 Type IN3RPCRequestConfig

additional config for a IN3 RPC-Request

Source: types/types.ts

- **chainId** :`string` - the requested chainId example: 0x1
- **clientSignature** :`any` *(optional)* - the signature of the client
- **finality** :`number` *(optional)* - if given the server will deliver the blockheaders of the following blocks until at least the number in percent of the validators is reached.
- **includeCode** :`boolean` *(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
- **latestBlock** :`number` *(optional)* - if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6
- **signatures** :`string`[] *(optional)* - a list of addresses requested to sign the blockhash example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **useBinary** :`boolean` *(optional)* - if true binary-data will be used.

- **useFullProof** :`boolean` *(optional)* - if true all data in the response will be proven, which leads to a higher payload.
- **useRef** :`boolean` *(optional)* - if true binary-data (starting with a 0x) will be refered if occuring again.
- **verification** :`'never'|'proof'|'proofWithSignature'` *(optional)* - defines the kind of proof the client is asking for example: proof
- **verifiedHashes** :`string`[] *(optional)* - 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.

7.6.10 Type IN3ResponseConfig

additional data returned from a IN3 Server

Source: types/types.ts

- **currentBlock** :`number` *(optional)* - the current blocknumber. example: 320126478
- **lastNodeList** :`number` *(optional)* - the blocknumber for the last block updating the nodelist. If the client has a smaller blocknumber he should update the nodeList. example: 326478
- **lastValidatorChange** :`number` *(optional)* - the blocknumber of gthe last change of the validatorList
- **proof** :*`Proof`* *(optional)* - the Proof-data

7.6.11 Type LogProof

a Object holding proofs for event logs. The key is the blockNumber as hex

Source: types/types.ts

7.6.12 Type Proof

the Proof-data as part of the in3-section

Source: types/types.ts

- **accounts** *(optional)* - a map of addresses and their AccountProof
- **block** :`string` *(optional)* - the serialized blockheader as hex, required in most proofs example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec1ffee98c0437b4ac839d8a2ece1b18166da704b86d8f4
- **finalityBlocks** :`any`[] *(optional)* - the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec1ffee98c0437b4ac839d8a2ece1b18166da704b86d8f4
- **logProof** :*`LogProof`* *(optional)* - the Log Proof in case of a Log-Request
- **merkleProof** :`string`[] *(optional)* - the serialized merle-noodes beginning with the root-node
- **merkleProofPrev** :`string`[] *(optional)* - the serialized merkle-noodes beginning with the root-node of the previous entry (only for full proof of receipts)
- **signatures** :*`Signature`*[] *(optional)* - requested signatures
- **transactions** :`any`[] *(optional)* - the list of transactions of the block example:
- **txIndex** :`number` *(optional)* - the transactionIndex within the block example: 4
- **txProof** :`string`[] *(optional)* - the serialized merkle-nodes beginning with the root-node in order to prrof the transactionIndex

- **type** :`'transactionProof'|'receiptProof'|'blockProof'|'accountProof'|'callProof'|'logProof'` - the type of the proof example: accountProof
- **uncles** :`any`[] *(optional)* - the list of uncle-headers of the block example:

7.6.13 Type RPCRequest

a JSONRPC-Request with N3-Extension

Source: types/types.ts

- **id** :`number|string` *(optional)* - the identifier of the request example: 2
- **in3** :*IN3RPCRequestConfig* *(optional)* - the IN3-Config
- **jsonrpc** :`'2.0'` - the version
- **method** :`string` - the method to call example: eth_getBalance
- **params** :`any`[] *(optional)* - the params example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,latest

7.6.14 Type RPCResponse

a JSONRPC-Responset with N3-Extension

Source: types/types.ts

- **error** :`string` *(optional)* - in case of an error this needs to be set
- **id** :`string|number` - the id matching the request example: 2
- **in3** :*IN3ResponseConfig* *(optional)* - the IN3-Result
- **in3Node** :*IN3NodeConfig* *(optional)* - the node handling this response (internal only)
- **jsonrpc** :`'2.0'` - the version
- **result** :`any` *(optional)* - the params example: 0xa35bc

7.6.15 Type ServerList

a List of nodes

Source: types/types.ts

- **contract** :`string` *(optional)* - IN3 Registry
- **lastBlockNumber** :`number` *(optional)* - last Block number
- **nodes** :*IN3NodeConfig*[] - the list of nodes
- **proof** :*Proof* *(optional)* - the Proof-data as part of the in3-section
- **registryId** :`string` *(optional)* - registry id of the contract
- **totalServers** :`number` *(optional)* - number of servers

7.6.16 Type Signature

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.

Source: types/types.ts

- **address** :`string` *(optional)* - the address of the signing node example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **block** :`number` - the blocknumber example: 3123874
- **blockHash** :`string` - the hash of the block example: 0x6C1a01C2aB554930A937B0a212346037E8105fB47946c679
- **msgHash** :`string` - hash of the message example: 0x9C1a01C2aB554930A937B0a212346037E8105fB47946AB5D
- **r** :`string` - Positive non-zero Integer signature.r example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2
- **s** :`string` - Positive non-zero Integer signature.s example: 0x6d17b34aeaf95fee98c0437b4ac839d8a2ece1b18166da704b86d8f42
- **v** :`number` - Calculated curve point, or identity element O. example: 28

7.7 Common Module

The common module (in3-common) contains all the typedefs used in the node and server.

- ***BlockData*** : `interface` - Block as returned by eth_getBlockByNumber
- ***LogData*** : `interface` - LogData as part of the TransactionReceipt
- **Receipt** :*`_serialize.Receipt`*
- ***ReceiptData*** : `interface` - TransactionReceipt as returned by eth_getTransactionReceipt
- **Transaction** :*`_serialize.Transaction`*
- ***TransactionData*** : `interface` - Transaction as returned by eth_getTransactionByHash
- ***Transport*** : `interface` - A Transport-object responsible to transport the message to the handler.
- ***AxiosTransport*** : `class` - Default Transport impl sending http-requests.
- ***Block*** : `class` - encodes and decodes the blockheader
- **blockFromHex**(hex :`string`) :*`Block`* - converts a hexstring to a block-object
- **cbor**
  - **createRefs**(val :*`T`*, cache :`string`[] = []) :*`T`*
  - **decodeRequests**(request :*`Buffer`*) :*`RPCRequest`*[]
  - **decodeResponses**(responses :*`Buffer`*) :*`RPCResponse`*[]
  - **encodeRequests**(requests :*`RPCRequest`*[]) :*`Buffer`* - turn
  - **encodeResponses**(responses :*`RPCResponse`*[]) :*`Buffer`*
  - **resolveRefs**(val :*`T`*, cache :`string`[] = []) :*`T`*
- **chainAliases**
  - **goerli** :`string`
  - **ipfs** :`string`
  - **kovan** :`string`

  - **main** :string
  - **mainnet** :string
  - **tobalaba** :string
- **createRandomIndexes**(len :number, limit :number, seed :*Buffer*, result :number[] = []) :number[]
- **createTx**(transaction :any) :any - creates a Transaction-object from the rpc-transaction-data
- **getSigner**(data :*Block*) :*Buffer*
- **rlp**
- **serialize**
  - ***Block*** :class - encodes and decodes the blockheader
  - ***AccountData*** :interface - Account-Object
  - ***BlockData*** :interface - Block as returned by eth_getBlockByNumber
  - ***LogData*** :interface - LogData as part of the TransactionReceipt
  - ***ReceiptData*** :interface - TransactionReceipt as returned by eth_getTransactionReceipt
  - ***TransactionData*** :interface - Transaction as returned by eth_getTransactionByHash
  - **Account** :*Buffer*[] - Buffer[] of the Account
  - **BlockHeader** :*Buffer*[] - Buffer[] of the header
  - **Receipt** : - Buffer[] of the Receipt
  - **Transaction** :*Buffer*[] - Buffer[] of the transaction
  - **rlp** - RLP-functions
  - **address**(val :any) :any - converts it to a Buffer with 20 bytes length
  - **blockFromHex**(hex :string) :*Block* - converts a hexstring to a block-object
  - **blockToHex**(block :any) :string - converts blockdata to a hexstring
  - **bytes**(val :any) :any - converts it to a Buffer
  - **bytes256**(val :any) :any - converts it to a Buffer with 256 bytes length
  - **bytes32**(val :any) :any - converts it to a Buffer with 32 bytes length
  - **bytes8**(val :any) :any - converts it to a Buffer with 8 bytes length
  - **createTx**(transaction :any) :any - creates a Transaction-object from the rpc-transaction-data
  - **hash**(val :*Block*|*Transaction*|*Receipt*|*Account*|*Buffer*) :*Buffer* - returns the hash of the object
  - **serialize**(val :*Block*|*Transaction*|*Receipt*|*Account*|any) :*Buffer* - serialize the data
  - **toAccount**(account :*AccountData*) :*Buffer*[]
  - **toBlockHeader**(block :*BlockData*) :*Buffer*[] - create a Buffer[] from RPC-Response
  - **toReceipt**(r :*ReceiptData*) :Object - create a Buffer[] from RPC-Response
  - **toTransaction**(tx :*TransactionData*) :*Buffer*[] - create a Buffer[] from RPC-Response
  - **uint**(val :any) :any - converts it to a Buffer with a variable length. 0 = length 0
  - **uint128**(val :any) :any

  - **uint64**(val :any) :any

- **storage**
  - **getStorageArrayKey**(pos :number, arrayIndex :number, structSize :number = 1, structPos :number = 0) :any - calc the storrage array key
  - **getStorageMapKey**(pos :number, key :string, structPos :number = 0) :any - calcs the storage Map key.
  - **getStorageValue**(rpc :string, contract :string, pos :number, type :'address'|'bytes32'|'bytes16'|'bytes4'|'int'|'string', keyOrIndex :number|string, structSize :number, structPos :number) :Promise<any> - get a storage value from the server
  - **getStringValue**(data :*Buffer*, storageKey :*Buffer*) :string| - creates a string from storage.
  - **getStringValueFromList**(values :*Buffer*[], len :number) :string - concats the storage values to a string.
  - **toBN**(val :any) :any - converts any value to BN

- **transport**
  - ***AxiosTransport*** :class - Default Transport impl sending http-requests.
  - ***Transport*** :interface - A Transport-object responsible to transport the message to the handler.

- **util**
  - **checkForError**(res :*T*) :*T* - check a RPC-Response for errors and rejects the promise if found
  - **createRandomIndexes**(len :number, limit :number, seed :*Buffer*, result :number[] = []) :number[]
  - **getAddress**(pk :string) :string - returns a address from a private key
  - **getSigner**(data :*Block*) :*Buffer*
  - **padEnd**(val :string, minLength :number, fill :string = " ") :string - padEnd for legacy
  - **padStart**(val :string, minLength :number, fill :string = " ") :string - padStart for legacy
  - **promisify**(self :any, fn :any, args :any[]) :Promise<any> - simple promisy-function
  - **toBN**(val :any) :any - convert to BigNumber
  - **toBuffer**(val :any, len :number = -1) :any - converts any value as Buffer if len === 0 it will return an empty Buffer if the value is 0 or '0x00', since this is the way rlpencode works wit 0-values.
  - **toHex**(val :any, bytes :number) :string - converts any value as hex-string
  - **toMinHex**(key :string|*Buffer*|number) :string - removes all leading 0 in the hexstring
  - **toNumber**(val :any) :number - converts to a js-number
  - **toSimpleHex**(val :string) :string - removes all leading 0 in a hex-string
  - **toUtf8**(val :any) :string
  - **aliases** : Object
    - **goerli** :string
    - **ipfs** :string
    - **kovan** :string
    - **main** :string

        * **mainnet** :string
        * **tobalaba** :string

- **validate**
  - **ajv** :*Ajv* - the ajv instance with custom formatters and keywords
  - **validate**(ob :any, def :any) :void
  - **validateAndThrow**(fn :*Ajv.ValidateFunction*, ob :any) :void - validates the data and throws an error in case they are not valid.

7.8 Package modules/eth

7.8.1 Type BlockData

Block as returned by eth_getBlockByNumber

Source: modules/eth/serialize.ts

- **coinbase** :string *(optional)*
- **difficulty** :string|number
- **extraData** :string
- **gasLimit** :string|number
- **gasUsed** :string|number
- **hash** :string
- **logsBloom** :string
- **miner** :string
- **mixHash** :string *(optional)*
- **nonce** :string|number *(optional)*
- **number** :string|number
- **parentHash** :string
- **receiptRoot** :string *(optional)*
- **receiptsRoot** :string
- **sealFields** :string[] *(optional)*
- **sha3Uncles** :string
- **stateRoot** :string
- **timestamp** :string|number
- **transactions** :any[] *(optional)*
- **transactionsRoot** :string
- **uncles** :string[] *(optional)*

7.8.2 Type LogData

LogData as part of the TransactionReceipt

Source: modules/eth/serialize.ts

- **address** :`string`
- **blockHash** :`string`
- **blockNumber** :`string`
- **data** :`string`
- **logIndex** :`string`
- **removed** :`boolean`
- **topics** :`string`[]
- **transactionHash** :`string`
- **transactionIndex** :`string`
- **transactionLogIndex** :`string`

7.8.3 Type ReceiptData

TransactionReceipt as returned by eth_getTransactionReceipt

Source: modules/eth/serialize.ts

- **blockHash** :`string` *(optional)*
- **blockNumber** :`string`|`number` *(optional)*
- **cumulativeGasUsed** :`string`|`number` *(optional)*
- **gasUsed** :`string`|`number` *(optional)*
- **logs** :*LogData*[]
- **logsBloom** :`string` *(optional)*
- **root** :`string` *(optional)*
- **status** :`string`|`boolean` *(optional)*
- **transactionHash** :`string` *(optional)*
- **transactionIndex** :`number` *(optional)*

7.8.4 Type TransactionData

Transaction as returned by eth_getTransactionByHash

Source: modules/eth/serialize.ts

- **blockHash** :`string` *(optional)*
- **blockNumber** :`number`|`string` *(optional)*
- **chainId** :`number`|`string` *(optional)*
- **condition** :`string` *(optional)*

- **creates** :`string` *(optional)*
- **data** :`string` *(optional)*
- **from** :`string` *(optional)*
- **gas** :`number|string` *(optional)*
- **gasLimit** :`number|string` *(optional)*
- **gasPrice** :`number|string` *(optional)*
- **hash** :`string`
- **input** :`string`
- **nonce** :`number|string`
- **publicKey** :`string` *(optional)*
- **r** :`string` *(optional)*
- **raw** :`string` *(optional)*
- **s** :`string` *(optional)*
- **standardV** :`string` *(optional)*
- **to** :`string`
- **transactionIndex** :`number`
- **v** :`string` *(optional)*
- **value** :`number|string`

7.8.5 Type Block

encodes and decodes the blockheader

Source: modules/eth/serialize.ts

- `constructor` **constructor**(data :*`Buffer`*|`string`|*`BlockData`*) :*`Block`* - creates a Block-Onject from either the block-data as returned from rpc, a buffer or a hex-string of the encoded blockheader
- **raw** :*`BlockHeader`* - the raw Buffer fields of the BlockHeader
- **transactions** :*`Tx`*[] - the transaction-Object (if given)
- **bloom()**
- **coinbase()**
- **difficulty()**
- **extra()**
- **gasLimit()**
- **gasUsed()**
- **number()**
- **parentHash()**
- **receiptTrie()**
- **sealedFields()**

- **stateRoot()**
- **timestamp()**
- **transactionsTrie()**
- **uncleHash()**
- **bareHash**() :*Buffer* - the blockhash as buffer without the seal fields
- **hash**() :*Buffer* - the blockhash as buffer
- **serializeHeader**() :*Buffer* - the serialized header as buffer

7.8.6 Type AccountData

Account-Object

Source: modules/eth/serialize.ts

- **balance** :`string`
- **code** :`string` *(optional)*
- **codeHash** :`string`
- **nonce** :`string`
- **storageHash** :`string`

7.8.7 Type Transaction

Buffer[] of the transaction

Source: modules/eth/serialize.ts

- **Transaction** :*Buffer*[] - Buffer[] of the transaction

7.8.8 Type Receipt

Buffer[] of the Receipt

Source: modules/eth/serialize.ts

- **Receipt** : - Buffer[] of the Receipt

7.8.9 Type Account

Buffer[] of the Account

Source: modules/eth/serialize.ts

- **Account** :*Buffer*[] - Buffer[] of the Account

7.8.10 Type BlockHeader

Buffer[] of the header

Source: modules/eth/serialize.ts

- **BlockHeader** :*Buffer*[] - Buffer[] of the header

7.9 Package types

7.9.1 Type RPCRequest

a JSONRPC-Request with N3-Extension

Source: types/types.ts

- **id** :number|string *(optional)* - the identifier of the request example: 2
- **in3** :*IN3RPCRequestConfig* *(optional)* - the IN3-Config
- **jsonrpc** :'2.0' - the version
- **method** :string - the method to call example: eth_getBalance
- **params** :any[] *(optional)* - the params example: 0xe36179e2286ef405e929C90ad3E70E649B22a945,latest

7.9.2 Type RPCResponse

a JSONRPC-Responset with N3-Extension

Source: types/types.ts

- **error** :string *(optional)* - in case of an error this needs to be set
- **id** :string|number - the id matching the request example: 2
- **in3** :*IN3ResponseConfig* *(optional)* - the IN3-Result
- **in3Node** :*IN3NodeConfig* *(optional)* - the node handling this response (internal only)
- **jsonrpc** :'2.0' - the version
- **result** :any *(optional)* - the params example: 0xa35bc

7.9.3 Type IN3RPCRequestConfig

additional config for a IN3 RPC-Request

Source: types/types.ts

- **chainId** :string - the requested chainId example: 0x1
- **clientSignature** :any *(optional)* - the signature of the client
- **finality** :number *(optional)* - if given the server will deliver the blockheaders of the following blocks until at least the number in percent of the validators is reached.
- **includeCode** :boolean *(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

- **latestBlock** :`number` *(optional)* - if specified, the blocknumber *latest* will be replaced by blockNumber- specified value example: 6
- **signatures** :`string`[] *(optional)* - a list of addresses requested to sign the blockhash example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **useBinary** :`boolean` *(optional)* - if true binary-data will be used.
- **useFullProof** :`boolean` *(optional)* - if true all data in the response will be proven, which leads to a higher payload.
- **useRef** :`boolean` *(optional)* - if true binary-data (starting with a 0x) will be refered if occuring again.
- **verification** :`'never'|'proof'|'proofWithSignature'` *(optional)* - defines the kind of proof the client is asking for example: proof
- **verifiedHashes** :`string`[] *(optional)* - 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.

7.9.4 Type IN3ResponseConfig

additional data returned from a IN3 Server

Source: types/types.ts

- **currentBlock** :`number` *(optional)* - the current blocknumber. example: 320126478
- **lastNodeList** :`number` *(optional)* - the blocknumber for the last block updating the nodelist. If the client has a smaller blocknumber he should update the nodeList. example: 326478
- **lastValidatorChange** :`number` *(optional)* - the blocknumber of gthe last change of the validatorList
- **proof** :*`Proof`* *(optional)* - the Proof-data

7.9.5 Type IN3NodeConfig

a configuration of a in3-server.

Source: types/types.ts

- **address** :`string` - the address of the node, which is the public address it iis signing with. example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **capacity** :`number` *(optional)* - the capacity of the node. example: 100
- **chainIds** :`string`[] - the list of supported chains example: 0x1
- **deposit** :`number` - the deposit of the node in wei example: 12350000
- **index** :`number` *(optional)* - the index within the contract example: 13
- **props** :`number` *(optional)* - the properties of the node. example: 3
- **registerTime** :`number` *(optional)* - the UNIX-timestamp when the node was registered example: 1563279168
- **timeout** :`number` *(optional)* - the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600
- **unregisterTime** :`number` *(optional)* - the UNIX-timestamp when the node is allowed to be deregister example: 1563279168
- **url** :`string` - the endpoint to post to example: https://in3.slock.it

7.9.6 Type Proof

the Proof-data as part of the in3-section

Source: types/types.ts

- **accounts** *(optional)* - a map of addresses and their AccountProof
- **block** :`string` *(optional)* - the serialized blockheader as hex, required in most proofs example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec1ffee98c0437b4ac839d8a2ece1b18166da704b86d8f42
- **finalityBlocks** :`any`[] *(optional)* - the serialized blockheader as hex, required in case of finality asked example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2b8ec1ffee98c0437b4ac839d8a2ece1b18166da704b86d8f42
- **logProof** :*`LogProof`* *(optional)* - the Log Proof in case of a Log-Request
- **merkleProof** :`string`[] *(optional)* - the serialized merle-noodes beginning with the root-node
- **merkleProofPrev** :`string`[] *(optional)* - the serialized merkle-noodes beginning with the root-node of the previous entry (only for full proof of receipts)
- **signatures** :*`Signature`*[] *(optional)* - requested signatures
- **transactions** :`any`[] *(optional)* - the list of transactions of the block example:
- **txIndex** :`number` *(optional)* - the transactionIndex within the block example: 4
- **txProof** :`string`[] *(optional)* - the serialized merkle-nodes beginning with the root-node in order to prrof the transactionIndex
- **type** :`'transactionProof'|'receiptProof'|'blockProof'|'accountProof'|'callProof'|'logProof'` - the type of the proof example: accountProof
- **uncles** :`any`[] *(optional)* - the list of uncle-headers of the block example:

7.9.7 Type LogProof

a Object holding proofs for event logs. The key is the blockNumber as hex

Source: types/types.ts

7.9.8 Type Signature

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.

Source: types/types.ts

- **address** :`string` *(optional)* - the address of the signing node example: 0x6C1a01C2aB554930A937B0a2E8105fB47946c679
- **block** :`number` - the blocknumber example: 3123874
- **blockHash** :`string` - the hash of the block example: 0x6C1a01C2aB554930A937B0a212346037E8105fB47946c679
- **msgHash** :`string` - hash of the message example: 0x9C1a01C2aB554930A937B0a212346037E8105fB47946AB5D
- **r** :`string` - Positive non-zero Integer signature.r example: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf2
- **s** :`string` - Positive non-zero Integer signature.s example: 0x6d17b34aeaf95fee98c0437b4ac839d8a2ece1b18166da704b86d8f42
- **v** :`number` - Calculated curve point, or identity element O. example: 28

7.10 Package util

7.10.1 Type Transport

A Transport-object responsible to transport the message to the handler.

Source: util/transport.ts

- **handle**(url :`string`, data :*RPCRequest*|*RPCRequest*[], timeout :`number`) :`Promise<>` - handles a request by passing the data to the handler
- **isOnline**() :`Promise<boolean>` - check whether the handler is onlne.
- **random**(count :`number`) :`number`[] - generates random numbers (between 0-1)

7.10.2 Type AxiosTransport

Default Transport impl sending http-requests.

Source: util/transport.ts

- `constructor` **constructor**(format :`'json'|'cbor'|'jsonRef'` = "json") :*AxiosTransport*
- **format** :`'json'|'cbor'|'jsonRef'`
- **handle**(url :`string`, data :*RPCRequest*|*RPCRequest*[], timeout :`number`) :`Promise<>`
- **isOnline**() :`Promise<boolean>`
- **random**(count :`number`) :`number`[]

CHAPTER 8

API Reference C

8.1 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.

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

8.1.1 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. *nano*: a minimal module only able to verify transaction receipts (`eth_getTransactionReceipt`).
2. *basic*: module able to verify almost all other standard RPC functions (except `eth_call`).
3. *full*: module able to verify standard RPC functions. It also implements a full EVM to handle `eth_call`.

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()*.

8.1.2 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. *curl*: module with a dependency on curl, which executes these requests and supports HTTPS. This module runs a standard OS with curl installed.

8.1.3 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.

8.2 Building

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

8.2.1 requirements

- cmake
- curl : curl is used as transport for command-line tools.
- optional: libsycrypt, which would be used for unlocking keystore files using `scrypt` as kdf method. if it does not exist you can still build, but not decrypt such keys.

for osx `brew install libscrypt` and for debian `sudo apt-get install libscrypt-dev`

Incubed uses cmake for configuring:

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

8.2.2 CMake options

CMD

build the comandline utils

Type: `BOOL` , Default-Value: `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.

Type: `BOOL` , Default-Value: `ON`

FAST_MATH

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

Type: `BOOL` , Default-Value: `OFF`

IN3API

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

Type: `BOOL` , Default-Value: `ON`

IN3_LIB

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

Type: `BOOL` , Default-Value: `ON`

IN3_SERVER

support proxy server

Type: `BOOL` , Default-Value: `OFF`

IN3_STAGING

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

Type: `BOOL` , Default-Value: `ON`

JAVA

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

Type: `BOOL` , Default-Value: `OFF`

SEGGER_RTT

Use the segger real time transfer terminal as the logging mechanism

Type: `BOOL` , Default-Value: `OFF`

TAG_VERSION

the tagged version, which should be used

Type: `BOOL` , Default-Value: `OFF`

TEST

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

Type: `BOOL` , Default-Value: `OFF`

TRANSPORTS

builds transports, which may require extra libraries.

Type: `BOOL` , Default-Value: `ON`

USE_CURL

if true

Type: `BOOL` , Default-Value: `ON`

USE_SCRYPT

if scrypt is installed, it will link dynamicly to the shared scrypt lib.

Type: `BOOL` , Default-Value: `OFF`

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.

Type: `BOOL` , Default-Value: `OFF`

8.3 Examples

The full list of examples can be found here: https://git.slock.it/in3/c/in3-core/tree/develop/examples/c

8.3.1 Creating an Incubed Instance

creating always follow these steps:

```
#include <client/client.h> // the core client
#include <eth_full.h>      // the full ethereum verifier containing the EVM
#include <in3_curl.h>      // transport implementation

// register verifiers, in this case a full verifier allowing eth_call
// this needs to be called only once.
in3_register_eth_full();

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

(continues on next page)

(continued from previous page)

```
// create new client
in3_t* client = in3_new();

// ready to use ...
```

8.3.2 Calling a Function

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

// copy the hexcoded string into this address
hex2byte_arr("0x845E484b505443814B992Bf0319A5e8F5e407879", -1, contract, 20);

// ask for the number of servers registered
json_ctx_t* response = eth_call_fn(client, contract, "totalServers():uint256");

// handle response
if (!response) {
  printf("Could not get the response: %s", eth_last_error());
  return;
}

// convert the result to a integer
int number_of_servers = d_int(response->result);

// don't forget to free the response!
free_json(response);

// out put result
printf("Found %i servers registered : \n", number_of_servers);

// now we call a function with a complex result...
for (int i = 0; i < number_of_servers; i++) {

  // get all the details for one server.
  response = eth_call_fn(c, contract, "servers(uint256):(string,address,uint,uint,
↪uint,address)", to_uint256(i));

  // handle error
  if (!response) {
    printf("Could not get the response: %s", eth_last_error());
    return;
  }

  // decode data
  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)

  // print values
  printf("Server %i : %s owner = ", i, url);
```

(continues on next page)

(continued from previous page)

```
  ba_print(owner->data, owner->len);
  printf(", deposit = %" PRIu64 "\n", deposit);

  // clean up
  free_json(response);
}
```

8.4 Module api/eth1

8.4.1 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.

Location: 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 ()

```
#define BLKNUM_PENDING () ((eth_blknum_t){.def = BLK_PENDING, .is_u64 = false})
```

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 price per gas in wei.

arguments:

*in3_t **	**in3**

returns: uint64_t

eth_gasPrice

```
uint64_t eth_gasPrice(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_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 free_json()!

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 free_json()!

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, bytes32_t hash, 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**
bytes32_t	**hash**
`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**
uint256_tOPTIONAL_T(,)	**value**
bytes_tOPTIONAL_T(,)	**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 free_tx_receipt()

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_last_error

```
char* eth_last_error();
```

The current error or null if all is ok.

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)

free_log

```
void free_log(eth_log_t *log);
```

Frees a eth_log_t object.

arguments:

eth_log_t *	**log**

free_tx_receipt

```
void free_tx_receipt(eth_tx_receipt_t *txr);
```

Frees a eth_tx_receipt_t object.

arguments:

*eth_tx_receipt_t **	**txr**

8.5 Module api/usn

8.5.1 usn_api.h

USN API.

This header-file defines easy to use function, which are verifying USN-Messages.

Location: 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)

8.6 Module cmd/in3

8.6.1 in3_storage.h

storage handler storing cache in the home-dir/.in3

Location: src/cmd/in3/in3_storage.h

storage_get_item

```
bytes_t* storage_get_item(void *cptr, char *key);
```

arguments:

`void *`	**cptr**
`char *`	**key**

returns: *bytes_t **

storage_set_item

```
void storage_set_item(void *cptr, char *key, bytes_t *content);
```

arguments:

void *	**cptr**
char *	**key**
bytes_t *	**content**

8.7 Module core

8.7.1 cache.h

handles caching and storage.

storing nodelists and other caches with the storage handler as specified in the client. If no storage handler is specified nothing will be cached.

Location: src/core/client/cache.h

in3_cache_update_nodelist

```
in3_ret_t in3_cache_update_nodelist(in3_t *c, in3_chain_t *chain);
```

reads the nodelist from cache.

This function is usually called internally to fill the weights and nodelist from the the cache. If you call `in3_cache_init` there is no need to call this explicitly.

arguments:

in3_t *	**c**	the incubed client
in3_chain_t *	**chain**	chain to configure

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_cache_store_nodelist

```
in3_ret_t in3_cache_store_nodelist(in3_ctx_t *ctx, in3_chain_t *chain);
```

stores the nodelist to thes cache.

It will automaticly called if the nodelist has changed and read from the nodes or the wirght of a node changed.

arguments:

in3_ctx_t *	**ctx**	the current incubed context
in3_chain_t *	**chain**	the chain upating to cache

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.7.2 client.h

incubed main client file.

This includes the definition of the client and used enum values.

Location: src/core/client/client.h

IN3_PROTO_VER

```
#define IN3_PROTO_VER 0x2
```

ETH_CHAIN_ID_MAINNET

```
#define ETH_CHAIN_ID_MAINNET 0x01L
```

ETH_CHAIN_ID_KOVAN

```
#define ETH_CHAIN_ID_KOVAN 0x2aL
```

ETH_CHAIN_ID_TOBALABA

```
#define ETH_CHAIN_ID_TOBALABA 0x44dL
```

ETH_CHAIN_ID_GOERLI

```
#define ETH_CHAIN_ID_GOERLI 0x5L
```

ETH_CHAIN_ID_EVAN

```
#define ETH_CHAIN_ID_EVAN 0x4b1L
```

ETH_CHAIN_ID_IPFS

```
#define ETH_CHAIN_ID_IPFS 0x7d0
```

ETH_CHAIN_ID_VOLTA

```
#define ETH_CHAIN_ID_VOLTA 0x12046
```

ETH_CHAIN_ID_LOCAL

```
#define ETH_CHAIN_ID_LOCAL 0xFFFFL
```

IN3_SIGN_ERR_REJECTED

return value used by the signer if the the signature-request was rejected.

```
#define IN3_SIGN_ERR_REJECTED -1
```

IN3_SIGN_ERR_ACCOUNT_NOT_FOUND

return value used by the signer if the requested account was not found.

```
#define IN3_SIGN_ERR_ACCOUNT_NOT_FOUND -2
```

IN3_SIGN_ERR_INVALID_MESSAGE

return value used by the signer if the message was invalid.

```
#define IN3_SIGN_ERR_INVALID_MESSAGE -3
```

IN3_SIGN_ERR_GENERAL_ERROR

return value used by the signer for unspecified errors.

```
#define IN3_SIGN_ERR_GENERAL_ERROR -4
```

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_IOTA	4	IOTA chain.
CHAIN_GENERIC	5	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_verification_t

verification as delivered by the server.

This will be part of the in3-request and will be generated based on the prooftype.

The enum type contains the following values:

VERIFICATION_NEVER	0	No Verifacation.
VERIFICATION_PROOF	1	Includes the proof of the data.
VERIFICATION_PROOF_WITH_SIGNATURE	2	Proof + Signatures.

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_filter_type_t

The enum type contains the following values:

FILTER_EVENT	0	Event filter.
FILTER_BLOCK	1	Block filter.
FILTER_PENDING	2	Pending filter (Unsupported)

in3_request_config_t

the configuration as part of each incubed request.

This will be generated for each request based on the client-configuration. the verifier may access this during verification in order to check against the request.

The stuct contains following fields:

uint64_t	**chainId**	the chain to be used. this is holding the integer-value of the hexstring.
uint8_t	**includeCode**	if true the code needed will always be devlivered.
uint8_t	**useFullProof**	this flaqg is set, if the proof is set to "PROOF_FULL"
uint8_t	**useBinary**	this flaqg is set, the client should use binary-format
bytes_t *	**verifiedHashes**	a list of blockhashes already verified. The Server will not send any proof for them again .
uint16_t	**verifiedHashesCount**	number of verified blockhashes
uint16_t	**latestBlock**	the last blocknumber the nodelistz changed
uint16_t	**finality**	number of signatures(in percent) needed in order to reach finality.
in3_verification_t	**verification**	Verification-type.
bytes_t *	**clientSignature**	the signature of the client with the client key
bytes_t *	**signatures**	the addresses of servers requested to sign the blockhash
uint8_t	**signaturesCount**	number or addresses

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:

uint32_t	**index**	index within the nodelist, also used in the contract as key
bytes_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
uint32_t	**capacity**	the maximal capacity able to handle
uint64_t	**props**	a bit set used to identify the cabalilities of the server.
char *	**url**	the url of the node

in3_node_weight_t

Weight or reputation of a node.

Based on the past performance of the node a weight is calulcated given faster nodes a heigher 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:

float	**weight**	current weight
uint32_t	**response_count**	counter for responses
uint32_t	**total_response_time**	total of all response times
uint64_t	**blacklistedUntil**	if >0 this node is blacklisted until k. k is a unix timestamp

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:

uint64_t	**chainId**	chainId, which could be a free or based on the public ethereum networkId
in3_chain_type_t	**type**	chaintype
uint64_t	**lastBlock**	last blocknumber the nodeList was updated, which is used to detect changed in the nodelist
bool	**needsUpdate**	if true the nodelist should be updated and will trigger a *in3_nodeList*-request before the next request is send.
int	**nodeListLength**	number of nodes in the nodeList
in3_node_t *	**nodeList**	array of nodes
in3_node_weight_t *	**weights**	stats and weights recorded for each node
bytes_t **	**initAddresses**	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
uint8_t	**version**	version of the chain
json_ctx_t *	**spec**	optional chain specification, defining the transaitions and forks

in3_storage_get_item

storage handler function for reading from cache.

```
typedef bytes_t*(* in3_storage_get_item) (void *cptr, 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, char *key, bytes_t *value)
```

in3_storage_handler_t

storage handler to handle cache.

The stuct contains following fields:

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.
void *	**cptr**	custom pointer which will will be passed to functions

in3_sign

signing function.

signs the given data and write the signature to dst. the return value must be the number of bytes written to dst. In case of an error a negativ value must be returned. It should be one of the IN3_SIGN_ERR... values.

```
typedef in3_ret_t(* in3_sign) (void *wallet, d_signature_type_t type, bytes_t message,
↪ bytes_t account, uint8_t *dst)
```

returns: *in3_ret_t (** the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_signer_t

The stuct contains following fields:

in3_sign	**sign**
void *	**wallet**

in3_response_t

response-object.

if the error has a length>0 the response will be rejected

The stuct contains following fields:

sb_t	**error**	a stringbuilder to add any errors!
sb_t	**result**	a stringbuilder to add the result

in3_transport_send

the transport function to be implemented by the transport provider.

```
typedef in3_ret_t(* in3_transport_send) (char **urls, int urls_len, char *payload,␣
↪in3_response_t *results)
```

returns: *in3_ret_t (** the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_filter_t

The stuct contains following fields:

in3_filter_type_t	**type**	filter type: (event, block or pending)
char *	**options**	associated filter options
uint64_t	**last_block**	block no. when filter was created OR eth_getFilterChanges was called
void(*	**release**	method to release owned resources

in3_filter_handler_t

The stuct contains following fields:

in3_filter_t **	**array**	
size_t	**count**	array of filters

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:

uint32_t	**cacheTimeout**	number of seconds requests can be cached.
uint16_t	**nodeLimit**	the limit of nodes to store in the client.
bytes_t *	**key**	the client key to sign requests
uint32_t	**maxCodeCache**	number of max bytes used to cache the code in memory
uint32_t	**maxBlockCache**	number of number of blocks cached in memory
in3_proof_t	**proof**	the type of proof used
uint8_t	**requestCount**	the number of request send when getting a first answer
uint8_t	**signatureCount**	the number of signatures used to proof the blockhash.
uint64_t	**minDeposit**	min stake of the server. Only nodes owning at least this amount will be chosen.
uint16_t	**replaceLatestBlock**	if specified, the blocknumber *latest* will be replaced by blockNumber- specified value
uint16_t	**finality**	the number of signatures in percent required for the request
uint16_t	**max_attempts**	the max number of attempts before giving up
uint32_t	**timeout**	specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection.
uint64_t	**chainId**	servers to filter for the given chain. The chain-id based on EIP-155.
uint8_t	**autoUpdateList**	if true the nodelist will be automaticly updated if the lastBlock is newer
in3_storage_handler_t *	**cacheStorage**	a cache handler offering 2 functions (setItem(string,string), getItem(string))
in3_signer_t *	**signer**	signer-struct managing a wallet
in3_transport_send	**transport**	the transporthandler sending requests
uint8_t	**includeCode**	includes the code when sending eth_call-requests
uint8_t	**use_binary**	if true the client will use binary format
uint8_t	**use_http**	if true the client will try to use http instead of https
in3_chain_t *	**chains**	chain spec and nodeList definitions
uint16_t	**chainsCount**	number of configured chains
in3_filter_handler_t *	**filters**	filter handler

in3_new

```
in3_t* in3_new();
```

creates a new Incubes configuration and returns the pointer.

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 storage...
in3_storage_handler_t storage_handler;
storage_handler.get_item = storage_get_item;
storage_handler.set_item = storage_set_item;

// configure transport
client->transport    = send_curl;

// configure storage
client->cacheStorage = &storage_handler;

// init cache
in3_cache_init(client);

// ready to use ...
```

returns: *in3_t ** : the incubed instance.

in3_client_rpc

```
in3_ret_t in3_client_rpc(in3_t *c, char *method, 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.
char *	**method**	the name of the rpc-funcgtion to call.
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_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_register_chain

```
in3_ret_t in3_client_register_chain(in3_t *client, uint64_t chain_id, in3_chain_type_
↪t type, address_t contract, bytes32_t registry_id, uint8_t version, json_ctx_t␣
↪*spec);
```

registers a new chain or replaces a existing (but keeps the nodelist)

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
uint64_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.
*json_ctx_t **	**spec**	chainspec or NULL.

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

```
in3_ret_t in3_client_add_node(in3_t *client, uint64_t chain_id, char *url, uint64_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.
uint64_t	**chain_id**	the chain id.
char *	**url**	url of the nodes.
uint64_t	**props**	properties of the node.
address_t	**address**	

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_remove_node

```
in3_ret_t in3_client_remove_node(in3_t *client, uint64_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.
uint64_t	**chain_id**	the chain id.
address_t	**address**	

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_clear_nodes

```
in3_ret_t in3_client_clear_nodes(in3_t *client, uint64_t chain_id);
```

removes all nodes from the nodelist

[in] the chain id.

arguments:

*in3_t **	**client**	the pointer to the incubed client config.
uint64_t	**chain_id**	

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_free

```
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.

in3_cache_init

```
in3_ret_t in3_cache_init(in3_t *c);
```

inits the cache.

arguments:

*in3_t **	**c**	the incubed client

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_configure

```
in3_ret_t in3_configure(in3_t *c, 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

arguments:

*in3_t **	**c**
`char *`	**config**

returns: *`in3_ret_t`* the *result-status* of the function.

Please make sure you check if it was successfull (`==IN3_OK`)

in3_set_default_transport

```
void in3_set_default_transport(in3_transport_send transport);
```

defines a default transport which is used when creating a new client.

arguments:

in3_transport_send	**transport**

in3_set_default_storage

```
void in3_set_default_storage(in3_storage_handler_t *cacheStorage);
```

defines a default storage handler which is used when creating a new client.

arguments:

*in3_storage_handler_t **	**cacheStorage**

in3_set_default_signer

```
void in3_set_default_signer(in3_signer_t *signer);
```

defines a default signer which is used when creating a new client.

arguments:

*in3_signer_t **	**signer**

8.7.3 context.h

Request Context.

This is used for each request holding request and response-pointers.

Location: src/core/client/context.h

node_weight_t

the weight of a ceertain node as linked list

The stuct contains following fields:

*in3_node_t **	**node**	the node definition including the url
*in3_node_weight_t **	**weight**	the current weight and blacklisting-stats
`float`	**s**	The starting value.
`float`	**w**	weight value
*weightstruct , **	**next**	next in the linkedlistt or NULL if this is the last element

new_ctx

```
in3_ctx_t* new_ctx(in3_t *client, char *req_data);
```

creates a new context.

the request data will be parsed and represented in the context.

arguments:

*in3_t **	**client**
`char *`	**req_data**

returns: *in3_ctx_t **

ctx_parse_response

```
in3_ret_t ctx_parse_response(in3_ctx_t *ctx, char *response_data, int len);
```

arguments:

*in3_ctx_t **	**ctx**
`char *`	**response_data**
`int`	**len**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

free_ctx

```
void free_ctx(in3_ctx_t *ctx);
```

arguments:

*in3_ctx_t **	**ctx**

ctx_create_payload

```
in3_ret_t ctx_create_payload(in3_ctx_t *c, sb_t *sb);
```

arguments:

*in3_ctx_t **	**c**
*sb_t **	**sb**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

ctx_set_error

```
in3_ret_t ctx_set_error(in3_ctx_t *c, char *msg, in3_ret_t errnumber);
```

arguments:

*in3_ctx_t **	**c**
`char *`	**msg**
in3_ret_t	**errnumber**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

ctx_get_error

```
in3_ret_t ctx_get_error(in3_ctx_t *ctx, int id);
```

arguments:

*in3_ctx_t **	**ctx**
`int`	**id**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_client_rpc_ctx

```
in3_ctx_t* in3_client_rpc_ctx(in3_t *c, char *method, char *params);
```

sends a request and returns a context used to access the result or errors.

This context *MUST* be freed with free_ctx(ctx) after usage to release the resources.

arguments:

*in3_t **	**c**
`char *`	**method**
`char *`	**params**

returns: *in3_ctx_t **

free_ctx_nodes

```
void free_ctx_nodes(node_weight_t *c);
```

arguments:

*node_weight_t **	**c**

ctx_nodes_len

```
int ctx_nodes_len(node_weight_t *root);
```

arguments:

*node_weight_t **	**root**

returns: `int`

8.7.4 nodelist.h

handles nodelists.

Location: src/core/client/nodelist.h

in3_nodelist_clear

```
void in3_nodelist_clear(in3_chain_t *chain);
```

removes all nodes and their weights from the nodelist

arguments:

*in3_chain_t **	**chain**

in3_node_list_get

```
in3_ret_t in3_node_list_get(in3_ctx_t *ctx, uint64_t chain_id, bool update, in3_node_
↪t **nodeList, int *nodeListLength, in3_node_weight_t **weights);
```

check if the nodelist is up to date.

if not it will fetch a new version first (if the needs_update-flag is set).

arguments:

in3_ctx_t *	**ctx**
uint64_t	**chain_id**
bool	**update**
in3_node_t **	**nodeList**
int *	**nodeListLength**
in3_node_weight_t **	**weights**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_node_list_fill_weight

```
node_weight_t* in3_node_list_fill_weight(in3_t *c, in3_node_t *all_nodes, in3_node_
↪weight_t *weights, int len, _time_t now, float *total_weight, int *total_found);
```

filters and fills the weights on a returned linked list.

arguments:

in3_t *	**c**
in3_node_t *	**all_nodes**
in3_node_weight_t *	**weights**
int	**len**
_time_t	**now**
float *	**total_weight**
int *	**total_found**

returns: *node_weight_t **

in3_node_list_pick_nodes

```
in3_ret_t in3_node_list_pick_nodes(in3_ctx_t *ctx, node_weight_t **nodes);
```

picks (based on the config) a random number of nodes and returns them as weightslist.

arguments:

in3_ctx_t *	**ctx**
node_weight_t **	**nodes**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.7.5 send.h

handles caching and storage.

handles the request.

Location: src/core/client/send.h

in3_send_ctx

```
in3_ret_t in3_send_ctx(in3_ctx_t *ctx);
```

executes a request context by picking nodes and sending it.

arguments:

*in3_ctx_t **	**ctx**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.7.6 verifier.h

Verification Context.

This context is passed to the verifier.

Location: src/core/client/verifier.h

in3_verify

function to verify the result.

```
typedef in3_ret_t(* in3_verify) (in3_vctx_t *c)
```

returns: *in3_ret_t (** the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_pre_handle

```
typedef in3_ret_t(* in3_pre_handle) (in3_ctx_t *ctx, in3_response_t **response)
```

returns: *in3_ret_t (** the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_verifier_t

The stuct contains following fields:

in3_verify	**verify**
in3_pre_handle	**pre_handle**
in3_chain_type_t	**type**
*verifierstruct , **	**next**

in3_get_verifier

```
in3_verifier_t* in3_get_verifier(in3_chain_type_t type);
```

returns the verifier for the given chainType

arguments:

in3_chain_type_t	**type**

returns: *in3_verifier_t **

in3_register_verifier

```
void in3_register_verifier(in3_verifier_t *verifier);
```

arguments:

*in3_verifier_t **	**verifier**

vc_err

```
in3_ret_t vc_err(in3_vctx_t *vc, char *msg);
```

arguments:

*in3_vctx_t **	**vc**
char *	**msg**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.7.7 bytes.h

util helper on byte arrays.

Location: src/core/util/bytes.h

bb_new ()

```
#define bb_new () bb_new1(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:

uint32_t	**len**	the length of the array ion bytes
uint8_t *	**data**	the byte-data

b_new

```
bytes_t* b_new(char *data, int len);
```

allocates a new byte array with 0 filled

arguments:

char *	**data**
int	**len**

returns: *bytes_t **

b_print

```
void b_print(bytes_t *a);
```

prints a the bytes as hex to stdout

arguments:

*bytes_t **	**a**

ba_print

```
void ba_print(uint8_t *a, size_t l);
```

prints a the bytes as hex to stdout

arguments:

uint8_t *	**a**
size_t	**l**

b_cmp

```
int b_cmp(bytes_t *a, bytes_t *b);
```

compares 2 byte arrays and returns 1 for equal and 0 for not equal

arguments:

*bytes_t **	**a**
*bytes_t **	**b**

returns: `int`

bytes_cmp

```
int bytes_cmp(bytes_t a, bytes_t b);
```

compares 2 byte arrays and returns 1 for equal and 0 for not equal

arguments:

bytes_t	**a**
bytes_t	**b**

returns: `int`

b_free

```
void b_free(bytes_t *a);
```

frees the data

arguments:

*bytes_t **	**a**

b_dup

```
bytes_t* b_dup(bytes_t *a);
```

clones a byte array

arguments:

*bytes_t **	**a**

returns: *bytes_t **

b_read_byte

```
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: uint8_t

b_read_short

```
uint16_t b_read_short(bytes_t *b, size_t *pos);
```

reads a short on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
size_t *	**pos**

returns: uint16_t

b_read_int

```
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: uint32_t

b_read_int_be

```
uint32_t b_read_int_be(bytes_t *b, size_t *pos, size_t len);
```

reads a unsigned integer as bigendian on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
size_t *	**pos**
size_t	**len**

returns: uint32_t

b_read_long

```
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: uint64_t

b_new_chars

```
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: char *

b_new_dyn_bytes

```
bytes_t* b_new_dyn_bytes(bytes_t *b, size_t *pos);
```

reads bytesn (which have the length stored as prefix) on the current position and updates the pos afterwards.

arguments:

*bytes_t **	**b**
size_t *	**pos**

returns: *bytes_t **

b_new_fixed_bytes

```
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_t **

bb_newl

```
bytes_builder_t* bb_newl(size_t l);
```

arguments:

size_t	**l**

returns: *bytes_builder_t **

bb_free

```
void bb_free(bytes_builder_t *bb);
```

frees a bytebuilder and its content.

arguments:

*bytes_builder_t **	**bb**

bb_check_size

```
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: int

bb_write_chars

```
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**

bb_write_dyn_bytes

```
void bb_write_dyn_bytes(bytes_builder_t *bb, bytes_t *src);
```

writes bytes to the builder with a prefixed length.

arguments:

*bytes_builder_t **	**bb**
*bytes_t **	**src**

bb_write_fixed_bytes

```
void bb_write_fixed_bytes(bytes_builder_t *bb, bytes_t *src);
```

writes fixed bytes to the builder.

arguments:

*bytes_builder_t **	**bb**
*bytes_t **	**src**

bb_write_int

```
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**

bb_write_long

```
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**

bb_write_long_be

```
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**

bb_write_byte

```
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**

bb_write_short

```
void bb_write_short(bytes_builder_t *bb, uint16_t val);
```

writes a short to the builder.

arguments:

bytes_builder_t *	**bb**
uint16_t	**val**

bb_write_raw_bytes

```
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**

bb_clear

```
void bb_clear(bytes_builder_t *bb);
```

resets the content of the builder.

arguments:

bytes_builder_t *	**bb**

bb_replace

```
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**

bb_move_to_bytes

```
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_t **

bb_push

```
void bb_push(bytes_builder_t *bb, uint8_t *data, uint8_t len);
```

arguments:

bytes_builder_t *	**bb**
uint8_t *	**data**
uint8_t	**len**

bb_read_long

```
uint64_t bb_read_long(bytes_builder_t *bb, size_t *i);
```

arguments:

bytes_builder_t *	**bb**
size_t *	**i**

returns: uint64_t

bb_read_int

```
uint32_t bb_read_int(bytes_builder_t *bb, size_t *i);
```

arguments:

bytes_builder_t *	**bb**
size_t *	**i**

returns: uint32_t

bytes

```
static bytes_t bytes(uint8_t *a, uint32_t len);
```

arguments:

uint8_t *	**a**
uint32_t	**len**

returns: *bytes_t*

cloned_bytes

```
bytes_t cloned_bytes(bytes_t data);
```

arguments:

bytes_t	**data**

returns: *bytes_t*

b_optimize_len

```
static void b_optimize_len(bytes_t *b);
```

arguments:

bytes_t *	**b**

8.7.8 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.

Location: 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:

`uint32_t`	**len**	the length of the content (or number of properties) depending + type.
`uint8_t *`	**data**	the byte or string-data
`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.
`size_t`	**allocated**	
`size_t`	**len**	amount of tokens allocated result
`size_t`	**depth**	number of tokens in result
`char *`	**c**	max depth of tokens in result

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:

`int`	**left**	number of result left
d_token_t *	**token**	current token

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

```
uint32_t d_int(const d_token_t *item);
```

returns the value as integer.

only if type is integer

arguments:

*d_token_tconst , **	**item**

returns: `uint32_t`

d_intd

```
uint32_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: `uint32_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

```
d_key_t keyn(const char *c, const int len);
```

generates the keyhash for the given stringrange as defined by len

arguments:

`const char *`	**c**
`const int`	**len**

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_prev

```
d_token_t* d_prev(d_token_t *item);
```

returns the prev sibling of an array or object

arguments:

*d_token_t **	**item**

returns: *d_token_t **

d_serialize_binary

```
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**

parse_binary

```
json_ctx_t* parse_binary(bytes_t *data);
```

parses the data and returns the context with the token, which needs to be freed after usage!

arguments:

*bytes_t **	**data**

returns: *json_ctx_t **

parse_binary_str

```
json_ctx_t* parse_binary_str(char *data, int len);
```

parses the data and returns the context with the token, which needs to be freed after usage!

arguments:

`char *`	**data**
`int`	**len**

returns: *json_ctx_t **

parse_json

```
json_ctx_t* parse_json(char *js);
```

parses json-data, which needs to be freed after usage!

arguments:

`char *`	**js**

returns: *json_ctx_t **

free_json

```
void free_json(json_ctx_t *parser_ctx);
```

frees the parse-context after usage

arguments:

*json_ctx_t **	**parser_ctx**

d_to_json

```
str_range_t d_to_json(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_t **	**item**

returns: *str_range_t*

d_create_json

```
char* d_create_json(d_token_t *item);
```

creates a json-string.

It does not work for objects if the parsed data were binary!

arguments:

*d_token_t **	**item**

returns: `char *`

json_create

```
json_ctx_t* json_create();
```

returns: *json_ctx_t **

json_create_null

```
d_token_t* json_create_null(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_t **

json_create_bool

```
d_token_t* json_create_bool(json_ctx_t *jp, bool value);
```

arguments:

*json_ctx_t **	**jp**
`bool`	**value**

returns: *d_token_t **

json_create_int

```
d_token_t* json_create_int(json_ctx_t *jp, uint64_t value);
```

arguments:

*json_ctx_t **	**jp**
`uint64_t`	**value**

returns: *d_token_t **

json_create_string

```
d_token_t* json_create_string(json_ctx_t *jp, char *value);
```

arguments:

*json_ctx_t **	**jp**
`char *`	**value**

returns: *d_token_t **

json_create_bytes

```
d_token_t* json_create_bytes(json_ctx_t *jp, bytes_t value);
```

arguments:

*json_ctx_t **	**jp**
bytes_t	**value**

returns: *d_token_t **

json_create_object

```
d_token_t* json_create_object(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_t **

json_create_array

```
d_token_t* json_create_array(json_ctx_t *jp);
```

arguments:

*json_ctx_t **	**jp**

returns: *d_token_t **

json_object_add_prop

```
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_t **

json_array_add_value

```
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_t **

json_get_int_value

```
int json_get_int_value(char *js, char *prop);
```

parses the json and return the value as int.

arguments:

char *	**js**
char *	**prop**

returns: int

json_get_str_value

```
void json_get_str_value(char *js, char *prop, char *dst);
```

parses the json and return the value as string.

arguments:

char *	**js**
char *	**prop**
char *	**dst**

json_get_json_value

```
char* json_get_json_value(char *js, char *prop);
```

parses the json and return the value as json-string.

arguments:

char *	**js**
char *	**prop**

returns: char *

d_get_keystr

```
char* d_get_keystr(d_key_t k);
```

returns the string for a key.

This only works track_keynames was activated before!

arguments:

d_key_t	**k**

returns: char *

d_track_keynames

```
void d_track_keynames(uint8_t v);
```

activates the keyname-cache, which stores the string for the keys when parsing.

arguments:

uint8_t	**v**

d_clear_keynames

```
void d_clear_keynames();
```

delete the cached keynames

key

```
static d_key_t key(const char *c);
```

arguments:

const char *	**c**

returns: 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 uint32_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: `uint32_t`

d_get_intkd

```
static uint32_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: `uint32_t`

d_get_int

```
static uint32_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: uint32_t

d_get_int_at

```
static uint32_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: uint32_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

```
static 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`

8.7.9 debug.h

logs debug data only if the DEBUG-flag is set.

Location: src/core/util/debug.h

dbg_log (msg,...)

dbg_log_raw (msg,...)

msg_dump

```
void msg_dump(const char *s, unsigned char *data, unsigned len);
```

arguments:

`const char *`	**s**
`unsigned char *`	**data**
`unsigned`	**len**

8.7.10 error.h

defines the return-values of a function call.

Location: src/core/util/error.h

OPTIONAL_T (t)

```
#define OPTIONAL_T (t) opt_##t
```

DEFINE_OPTIONAL_T (t)

```
#define DEFINE_OPTIONAL_T (t) typedef struct {           \
    t    value;              \
    bool defined;            \
  } OPTIONAL_T(t)
```

OPTIONAL_T_UNDEFINED (t)

```
#define OPTIONAL_T_UNDEFINED (t) ((OPTIONAL_T(t)){.defined = false})
```

OPTIONAL_T_VALUE (t,v)

```
#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.

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.

8.7.11 scache.h

util helper on byte arrays.

Location: src/core/util/scache.h

cache_entry_t

The stuct contains following fields:

bytes_t	**key**
bytes_t	**value**
`uint8_t`	**must_free**
`uint8_t`	**buffer**
cache_entrystruct , *	**next**

in3_cache_get_entry

```
bytes_t* in3_cache_get_entry(cache_entry_t *cache, bytes_t *key);
```

arguments:

cache_entry_t *	**cache**
bytes_t *	**key**

returns: *bytes_t **

in3_cache_add_entry

```
cache_entry_t* in3_cache_add_entry(cache_entry_t *cache, bytes_t key, bytes_t value);
```

arguments:

cache_entry_t *	**cache**
bytes_t	**key**
bytes_t	**value**

returns: *cache_entry_t **

in3_cache_free

```
void in3_cache_free(cache_entry_t *cache);
```

arguments:

cache_entry_t *	**cache**

8.7.12 stringbuilder.h

simple string buffer used to dynamicly add content.

Location: src/core/util/stringbuilder.h

sb_add_hexuint (sb,i)

```
#define sb_add_hexuint (sb,i) sb_add_hexuint_l(sb, i, sizeof(i))
```

sb_t

The stuct contains following fields:

char *	**data**
size_t	**allocted**
size_t	**len**

sb_new

```
sb_t* sb_new(char *chars);
```

arguments:

char *	**chars**

returns: *sb_t **

sb_init

```
sb_t* sb_init(sb_t *sb);
```

arguments:

*sb_t **	**sb**

returns: *sb_t **

sb_free

```
void sb_free(sb_t *sb);
```

arguments:

*sb_t **	**sb**

sb_add_char

```
sb_t* sb_add_char(sb_t *sb, char c);
```

arguments:

*sb_t **	**sb**
char	**c**

returns: *sb_t **

sb_add_chars

```
sb_t* sb_add_chars(sb_t *sb, char *chars);
```

arguments:

*sb_t **	**sb**
char *	**chars**

returns: *sb_t **

sb_add_range

```
sb_t* sb_add_range(sb_t *sb, const char *chars, int start, int len);
```

arguments:

*sb_t **	**sb**
const char *	**chars**
int	**start**
int	**len**

returns: *sb_t **

sb_add_key_value

```
sb_t* sb_add_key_value(sb_t *sb, char *key, char *value, int lv, bool as_string);
```

arguments:

*sb_t **	**sb**
char *	**key**
char *	**value**
int	**lv**
bool	**as_string**

returns: *sb_t **

sb_add_bytes

```
sb_t* sb_add_bytes(sb_t *sb, char *prefix, bytes_t *bytes, int len, bool as_array);
```

arguments:

*sb_t **	**sb**
`char *`	**prefix**
*bytes_t **	**bytes**
`int`	**len**
`bool`	**as_array**

returns: *sb_t **

sb_add_hexuint_l

```
sb_t* sb_add_hexuint_l(sb_t *sb, uintmax_t uint, size_t l);
```

Other types not supported

arguments:

*sb_t **	**sb**
`uintmax_t`	**uint**
`size_t`	**l**

returns: *sb_t **

8.7.13 utils.h

utility functions.

Location: src/core/util/utils.h

SWAP (a,b)

```
#define SWAP (a,b) {                \
    void* p = a;    \
    a       = b;    \
    b       = p;    \
  }
```

min (a,b)

```
#define min (a,b) ((a) < (b) ? (a) : (b))
```

max (a,b)

```
#define max (a,b) ((a) > (b) ? (a) : (b))
```

IS_APPROX (n1,n2,err)

```
#define IS_APPROX (n1,n2,err) ((n1 > n2) ? ((n1 - n2) <= err) : ((n2 - n1) <= err))
```

optimize_len (a,l)

```
#define optimize_len (a,l) while (l > 1 && *a == 0) { \
    l--;                      \
    a++;                      \
  }
```

TRY (exp)

```
#define TRY (exp) {                       \
    int _r = (exp);         \
    if (_r < 0) return _r; \
  }
```

TRY_SET (var,exp)

```
#define TRY_SET (var,exp) {                         \
    var = (exp);              \
    if (var < 0) return var; \
  }
```

TRY_GOTO (exp)

```
#define TRY_GOTO (exp) {                           \
    res = (exp);              \
    if (res < 0) goto clean; \
  }
```

pb_size_t

```
typedef uint32_t pb_size_t
```

pb_byte_t

```
typedef uint_least8_t pb_byte_t
```

bytes_to_long

```
uint64_t bytes_to_long(uint8_t *data, int len);
```

converts the bytes to a unsigned long (at least the last max len bytes)

arguments:

uint8_t *	**data**
int	**len**

returns: uint64_t

bytes_to_int

```
static uint32_t bytes_to_int(uint8_t *data, int len);
```

converts the bytes to a unsigned int (at least the last max len bytes)

arguments:

uint8_t *	**data**
int	**len**

returns: uint32_t

c_to_long

```
uint64_t c_to_long(char *a, int l);
```

converts a character into a uint64_t

arguments:

char *	**a**
int	**l**

returns: uint64_t

size_of_bytes

```
int size_of_bytes(int str_len);
```

the number of bytes used for a conerting a hex into bytes.

arguments:

int	**str_len**

returns: int

strtohex

```
uint8_t strtohex(char c);
```

converts a hexchar to byte (4bit)

arguments:

char	**c**

returns: uint8_t

u64tostr

```
const unsigned char* u64tostr(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 unsigned char *

hex2byte_arr

```
int hex2byte_arr(char *buf, int len, uint8_t *out, int outbuf_size);
```

convert a c string to a byte array storing it into a existing buffer

arguments:

char *	**buf**
int	**len**
uint8_t *	**out**
int	**outbuf_size**

returns: int

hex2long

```
uint64_t hex2long(char *buf);
```

convert hex to long

arguments:

char *	**buf**

returns: `uint64_t`

hex2byte_new_bytes

```
bytes_t* hex2byte_new_bytes(char *buf, int len);
```

convert a c string to a byte array creating a new buffer

arguments:

`char *`	**buf**
`int`	**len**

returns: *bytes_t **

bytes_to_hex

```
int bytes_to_hex(uint8_t *buffer, int len, char *out);
```

convefrts a bytes into hex

arguments:

`uint8_t *`	**buffer**
`int`	**len**
`char *`	**out**

returns: `int`

sha3

```
bytes_t* sha3(bytes_t *data);
```

hashes the bytes and creates a new bytes_t

arguments:

*bytes_t **	**data**

returns: *bytes_t **

sha3_to

```
int sha3_to(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**

hash_cmp

```
int hash_cmp(uint8_t *a, uint8_t *b);
```

compares 32 bytes and returns 0 if equal

arguments:

`uint8_t *`	**a**
`uint8_t *`	**b**

returns: `int`

_strdupn

```
char* _strdupn(char *src, int len);
```

duplicate the string

arguments:

`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(uint8_t *src, wlen_t src_len, uint8_t dst[32]);
```

sets a variable value to 32byte word.

arguments:

uint8_t *	**src**
wlen_t	**src_len**
uint8_t	**dst**

str_replace

```
char* str_replace(char *orig, char *rep, char *with);
```

arguments:

char *	**orig**
char *	**rep**
char *	**with**

returns: char *

str_replace_pos

```
char* str_replace_pos(char *orig, size_t pos, size_t len, const char *rep);
```

arguments:

char *	**orig**
size_t	**pos**
size_t	**len**
const char *	**rep**

returns: char *

str_find

```
char* str_find(char *haystack, const char *needle);
```

arguments:

`char *`	**haystack**
`const char *`	**needle**

returns: `char *`

8.8 Module transport/curl

8.8.1 in3_curl.h

transport-handler using libcurl.

Location: src/transport/curl/in3_curl.h

send_curl

```
in3_ret_t send_curl(char **urls, int urls_len, char *payload, in3_response_t *result);
```

the transport function using curl.

arguments:

`char **`	**urls**
`int`	**urls_len**
`char *`	**payload**
in3_response_t *	**result**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

in3_register_curl

```
void in3_register_curl();
```

registers curl as a default transport.

8.9 Module transport/http

8.9.1 in3_http.h

transport-handler using simple http.

Location: src/transport/http/in3_http.h

send_http

```
in3_ret_t send_http(char **urls, int urls_len, char *payload, in3_response_t *result);
```

arguments:

char **	**urls**
int	**urls_len**
char *	**payload**
in3_response_t *	**result**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.10 Module verifier/eth1/basic

8.10.1 eth_basic.h

Ethereum Nanon verification.

Location: src/verifier/eth1/basic/eth_basic.h

in3_verify_eth_basic

```
in3_ret_t in3_verify_eth_basic(in3_vctx_t *v);
```

entry-function to execute the verification context.

arguments:

in3_vctx_t *	**v**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

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_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)

in3_register_eth_basic

```
void in3_register_eth_basic();
```

this function should only be called once and will register the eth-nano verifier.

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_handle_intern

```
in3_ret_t eth_handle_intern(in3_ctx_t *ctx, in3_response_t **response);
```

this is called before a request is send

arguments:

*in3_ctx_t **	**ctx**
*in3_response_t ***	**response**

returns: *in3_ret_t* the *result-status* of the function.

Please make sure you check if it was successfull (==IN3_OK)

8.10.2 signer.h

Ethereum Nano verification.

Location: src/verifier/eth1/basic/signer.h

eth_set_pk_signer

```
in3_ret_t eth_set_pk_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)

8.10.3 trie.h

Patricia Merkle Tree Imnpl.

Location: 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.
uint8_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**

8.11 Module verifier/eth1/evm

8.11.1 big.h

Ethereum Nanon verification.

Location: 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

8.11.2 code.h

code cache.

Location: src/verifier/eth1/evm/code.h

in3_get_code

```
cache_entry_t* in3_get_code(in3_vctx_t *vc, uint8_t *address);
```

arguments:

*in3_vctx_t **	**vc**
`uint8_t *`	**address**

returns: *cache_entry_t **

8.11.3 evm.h

main evm-file.

Location: src/verifier/eth1/evm/evm.h

gas_options

EVM_ERROR_EMPTY_STACK

the no more elements on the stack

```
#define EVM_ERROR_EMPTY_STACK -1
```

EVM_ERROR_INVALID_OPCODE

the opcode is not supported

```
#define EVM_ERROR_INVALID_OPCODE -2
```

EVM_ERROR_BUFFER_TOO_SMALL

reading data from a position, which is not initialized

```
#define EVM_ERROR_BUFFER_TOO_SMALL -3
```

EVM_ERROR_ILLEGAL_MEMORY_ACCESS

the memory-offset does not exist

```
#define EVM_ERROR_ILLEGAL_MEMORY_ACCESS -4
```

EVM_ERROR_INVALID_JUMPDEST

the jump destination is not marked as valid destination

```
#define EVM_ERROR_INVALID_JUMPDEST -5
```

EVM_ERROR_INVALID_PUSH

the push data is empy

```
#define EVM_ERROR_INVALID_PUSH -6
```

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 -7
```

EVM_ERROR_TIMEOUT

the evm ran into a loop

```
#define EVM_ERROR_TIMEOUT -8
```

EVM_ERROR_INVALID_ENV

the enviroment could not deliver the data

```
#define EVM_ERROR_INVALID_ENV -9
```

EVM_ERROR_OUT_OF_GAS

not enough gas to exewcute the opcode

```
#define EVM_ERROR_OUT_OF_GAS -10
```

EVM_ERROR_BALANCE_TOO_LOW

not enough funds to transfer the requested value.

```
#define EVM_ERROR_BALANCE_TOO_LOW -11
```

EVM_ERROR_STACK_LIMIT

stack limit reached

```
#define EVM_ERROR_STACK_LIMIT -12
```

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_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 (...) 0
```

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 stuct contains following fields:

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**	
`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	

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_stack_peek_len

```
int evm_stack_peek_len(evm_t *evm);
```

arguments:

*evm_t **	**evm**

returns: `int`

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, 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**
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_run_precompiled

```
int evm_run_precompiled(evm_t *evm, uint8_t address[20]);
```

arguments:

*evm_t **	**evm**
uint8_t	**address**

returns: int

evm_is_precompiled

```
uint8_t evm_is_precompiled(evm_t *evm, uint8_t address[20]);
```

arguments:

*evm_t **	**evm**
uint8_t	**address**

returns: uint8_t

uint256_set

```
void uint256_set(uint8_t *src, wlen_t src_len, uint8_t dst[32]);
```

sets a variable value to 32byte word.

arguments:

uint8_t *	**src**
wlen_t	**src_len**
uint8_t	**dst**

evm_execute

```
int evm_execute(evm_t *evm);
```

arguments:

evm_t *	**evm**

returns: int

8.11.4 gas.h

evm gas defines.

Location: 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
```

SELFDESTRUCT_GAS (evm,g)

```
#define SELFDESTRUCT_GAS (evm,g) EVM_ERROR_UNSUPPORTED_CALL_OPCODE
```

UPDATE_ACCOUNT_CODE (...)

8.12 Module verifier/eth1/full

8.12.1 eth_full.h

Ethereum Nanon verification.

Location: src/verifier/eth1/full/eth_full.h

in3_verify_eth_full

```
int in3_verify_eth_full(in3_vctx_t *v);
```

entry-function to execute the verification context.

arguments:

*in3_vctx_t **	**v**

returns: `int`

in3_register_eth_full

```
void in3_register_eth_full();
```

this function should only be called once and will register the eth-full verifier.

8.13 Module verifier/eth1/nano

8.13.1 chainspec.h

Ethereum chain specification.

Location: 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(d_token_t *data);
```

arguments:

*d_token_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(uint64_t chain_id);
```

arguments:

uint64_t	**chain_id**

returns: *chainspec_t **

chainspec_put

```
void chainspec_put(uint64_t chain_id, chainspec_t *spec);
```

arguments:

uint64_t	**chain_id**
*chainspec_t **	**spec**

8.13.2 eth_nano.h

Ethereum Nanon verification.

Location: src/verifier/eth1/nano/eth_nano.h

in3_verify_eth_nano

```
in3_ret_t in3_verify_eth_nano(in3_vctx_t *v);
```

entry-function to execute the verification context.

arguments:

*in3_vctx_t **	**v**

returns: *in3_ret_t* 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

```
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: `int`

ecrecover_signature

```
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_t **

eth_verify_eth_getTransactionReceipt

```
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_t* 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)

in3_register_eth_nano

```
void in3_register_eth_nano();
```

this function should only be called once and will register the eth-nano verifier.

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 **

8.13.3 merkle.h

Merkle Proof Verification.

Location: 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

```
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: `uint8_t *` : the resulting bytes represent a 4bit-number each and are terminated with a 0xFF.

trie_matching_nibbles

```
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: `int`

trie_free_proof

```
void trie_free_proof(bytes_t **proof);
```

used to free the NULL-terminated proof-array.

arguments:

bytes_t **	**proof**

8.13.4 rlp.h

RLP-En/Decoding as described in the Ethereum RLP-Spec.

This decoding works without allocating new memory.

Location: 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_decode_item_len

```
int rlp_decode_item_len(bytes_t *b, int index);
```

returns the number of bytes of the element specified by index.

arguments:

*bytes_t **	**b**
int	**index**

returns: int : the number of bytes or 0 if not found.

rlp_decode_item_type

```
int rlp_decode_item_type(bytes_t *b, int index);
```

returns the type of the element specified by index.

arguments:

*bytes_t **	**b**
int	**index**

returns: `int` : - 0 : means item out of range

- 1 : item found
- 2 : list found (you can then decode the same bytes again)

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**

8.13.5 serialize.h

serialization of ETH-Objects.

This incoming tokens will represent their values as properties based on JSON-RPC.

Location: 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.

CHAPTER 9

API Reference Java

9.1 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
```

9.1.1 Downloading

Just download the latest jar-file here.

9.1.2 Building

For building the shared library you need to enable java by using the `-DJAVA=true` flag:

```
git clone git@github.com:slockit/in3-core.git
mkdir -p in3-core/build
cd in3-core/build
cmake -DJAVA=true .. && make
```

You will find the `in3.jar` in the build/lib - folder.

9.1.3 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
  core
  verifier/eth1/nano
  verifier/eth1/evm
  verifier/eth1/basic
  verifier/eth1/full
  bindings/java
  third-party/crypto
  third-party/tommath
  api/eth1)
        file(MAKE_DIRECTORY in3-core/src/${module}/outputs)
        add_subdirectory( in3-core/src/${module} in3-core/src/${module}/outputs )
endforeach()
```

Step 2: clone in3-core into the `app`-folder or use this script to clone and update incubed:

```
#!/usr/bin/env sh

#github-url for in3-core
IN3_SRC=git@github.com:SlockItEarlyAccess/in3-core.git

cd app

# if it exists we only call git pull
if [ -d in3-core ]; then
    cd in3-core
    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-core/src/bindings/java/in3 src/main/java/
# but not the native libs, since these will be build
rm -rf src/main/java/in3/native
```

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/SlockItEarlyAccess/in3-android-example

9.2 Examples

9.2.1 Using Incubed directly

```
import in3.IN3;

public class HelloIN3 {
   //
   public static void main(String[] args) {
       String blockNumber = args[0];

       // create incubed
       IN3 in3 = new IN3();

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

       // execute the request
       String jsonResult = in3.sendRPC("eth_getBlockByNumber",new Object[]{␣
↪blockNumber ,true});

       ....
   }
}
```

9.2.2 Using the API

Incubed also offers a API for getting Information directly in a structured way.

Reading Blocks

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

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
```

(continues on next page)

(continued from previous page)

```
        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");
    }

}
```

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

       // configure
       in3.setChainId(0x1);  // 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]);


       ....
   }
}
```

Sending Transactions

In order to send, you need a Signer. The SimpleWallet class is a basic implementation which can be used.

```
package in3;

```

(continues on next page)

(continued from previous page)

```
import java.io.IOException;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

import in3.*;
import in3.eth1.*;

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

        // configure
        in3.setChainId(0x1); // 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.from = account;
        tx.to = "0x1234567890123456789012345678901234567890";
        tx.function = "transfer(address,uint256)";
        tx.params = new Object[] { receipient, value };

        String txHash = in3.getEth1API().sendTransaction(tx);

        System.out.println("Transaction sent with hash = " + txHash);

    }
}
```

9.3 Package in3

9.3.1 class IN3

This is the main class creating the incubed client.

The client can then be configured.

getCacheTimeout

number of seconds requests can be cached.

public `native int` getCacheTimeout();

setCacheTimeout

sets number of seconds requests can be cached.

public `native void` setCacheTimeout(*`int`* val);

arguments:

`int`	**val**

getNodeLimit

the limit of nodes to store in the client.

public `native int` getNodeLimit();

setNodeLimit

sets the limit of nodes to store in the client.

public `native void` setNodeLimit(*`int`* val);

arguments:

`int`	**val**

getKey

the client key to sign requests

public `native byte []` getKey();

setKey

sets the client key to sign requests

public native void setKey(*byte[]* val);

arguments:

byte []	**val**

setKey

sets the client key as hexstring to sign requests

public void setKey(*String* val);

arguments:

String	**val**

getMaxCodeCache

number of max bytes used to cache the code in memory

public native int getMaxCodeCache();

setMaxCodeCache

sets number of max bytes used to cache the code in memory

public native void setMaxCodeCache(*int* val);

arguments:

int	**val**

getMaxBlockCache

number of blocks cached in memory

public native int getMaxBlockCache();

setMaxBlockCache

sets the number of blocks cached in memory

public native void setMaxBlockCache(*int* val);

arguments:

int	**val**

getProof

the type of proof used

public *Proofnative* getProof();

setProof

sets the type of proof used

public native void setProof(*Proof* val);

arguments:

Proof	**val**

getRequestCount

the number of request send when getting a first answer

public native int getRequestCount();

setRequestCount

sets the number of requests send when getting a first answer

public native void setRequestCount(*int* val);

arguments:

int	**val**

getSignatureCount

the number of signatures used to proof the blockhash.

public native int getSignatureCount();

setSignatureCount

sets the number of signatures used to proof the blockhash.

public native void setSignatureCount(*int* val);

arguments:

int	**val**

getMinDeposit

min stake of the server.

Only nodes owning at least this amount will be chosen.

public `native long` getMinDeposit();

setMinDeposit

sets min stake of the server.

Only nodes owning at least this amount will be chosen.

public `native void` setMinDeposit(*`long`* val);

arguments:

`long`	**val**

getReplaceLatestBlock

if specified, the blocknumber *latest* will be replaced by blockNumber- specified value

public `native int` getReplaceLatestBlock();

setReplaceLatestBlock

replaces the *latest* with blockNumber- specified value

public `native void` setReplaceLatestBlock(*`int`* val);

arguments:

`int`	**val**

getFinality

the number of signatures in percent required for the request

public `native int` getFinality();

setFinality

sets the number of signatures in percent required for the request

public `native void` setFinality(*`int`* val);

arguments:

`int`	**val**

getMaxAttempts

the max number of attempts before giving up

public `native int` getMaxAttempts();

setMaxAttempts

sets the max number of attempts before giving up

public `native void` setMaxAttempts(*`int`* val);

arguments:

`int`	**val**

getSigner

returns the signer or wallet.

public *`Signer`* getSigner();

getEth1API

gets the ethereum-api

public *`in3.eth1.API`* getEth1API();

setSigner

sets the signer or wallet.

public `void` setSigner(*`Signer`* signer);

arguments:

Signer	**signer**

getTimeout

specifies the number of milliseconds before the request times out.

increasing may be helpful if the device uses a slow connection.

public `native int` getTimeout();

setTimeout

specifies the number of milliseconds before the request times out.

increasing may be helpful if the device uses a slow connection.

public `native void` setTimeout(*`int`* val);

arguments:

int	**val**

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**

isAutoUpdateList

if true the nodelist will be automaticly updated if the lastBlock is newer

public native boolean isAutoUpdateList();

setAutoUpdateList

activates the auto update.if true the nodelist will be automaticly updated if the lastBlock is newer

public native void setAutoUpdateList(*boolean* val);

arguments:

boolean	**val**

getStorageProvider

provides the ability to cache content

public *StorageProvider* getStorageProvider();

setStorageProvider

provides the ability to cache content like nodelists, contract codes and validatorlists

public void setStorageProvider(*StorageProvider* val);

arguments:

StorageProvider	**val**

send

send a request.

The request must a valid json-string with method and params

public `native 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 `native 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

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**

IN3

constrcutor.

creates a new Incubed client.

public IN3();

main

public static `void` main(*`String[]`* args);

arguments:

`String []`	**args**

9.3.2 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

asStringArray

public `String []` asStringArray(*`Object`* o);

arguments:

`Object`	**o**

toString

public `String` toString();

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**

9.3.3 class Loader

loadLibrary

public static `void` loadLibrary();

9.3.4 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**

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**
`byte []`	**content**

9.3.5 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.

9.3.6 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**

hasAccount

returns true if the account is supported (or unlocked)

public `boolean` hasAccount(*String* address);

arguments:

`String`	**address**

sign

signing of the raw data.

public `String` sign(*String* data, *String* address);

arguments:

`String`	**data**
`String`	**address**

9.3.7 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

9.4 Package in3.eth1

9.4.1 class API

a Wrapper for the incubed client offering Type-safe Access and additional helper functions.

API

creates a 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 *`Block []`* 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 `long` 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

sendTransaction

sends a Transaction as desribed 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.

9.4.2 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();

9.4.3 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();

9.4.4 class LogFilter

Log configuration for search logs.

toString

creates a JSON-String.

public `String` toString();

9.4.5 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**

hasAccount

returns true if the account is supported (or unlocked)

public `boolean` hasAccount(*`String`* address);

arguments:

`String`	**address**

sign

signing of the raw data.

public `String` sign(*`String`* data, *`String`* address);

arguments:

`String`	**data**
`String`	**address**

9.4.6 class Transaction

represents a Transaction 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();

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();

9.4.7 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();

9.4.8 class TransactionRequest

represents a Transaction Request which should be send or called.

from

the from address

Type: `String`

to

the recipients address

Type: `String`

data

the data

Type: `String`

value

the value of the transaction

Type: `BigInteger`

nonce

the nonce (transactionCount of the sender)

Type: `long`

gas

the gas to use

Type: `long`

gasPrice

the gas price to use

Type: `long`

function

the signature for the function to call

Type: `String`

params

the params to use for encoding in the data

Type: `Object []`

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**

CHAPTER 10

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.

10.1 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.
-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.

10.2 Install

10.2.1 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.

10.2.2 From Package Managers

We currently support

10.3 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
```

10.4 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
```

10.4.1 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-core.git

# create build-folder
cd in3-core
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 .

10.4.2 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
```

10.5 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.

10.6 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`
```

10.7 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
```

10.8 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.

10.9 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`
```

(continues on next page)

(continued from previous page)

```
# 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
```

10.10 Autocompletion

If you want autocompletion, simply add these lines to your *.bashrc* or *.bash_profile*:

```
_IN3_WORDS=`in3 autocompletelist`
complete -W "$_IN3_WORDS" in3
```

10.11 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`

10.12 Examples

10.12.1 Getting the Current Block

```
# On a command line:
in3 eth_blockNumber
> 8035324

# For a different chain:
in3 -c kovan eth_blockNumber
> 11834906

# Getting it as hex:
in3 -c kovan -hex eth_blockNumber
```

(continues on next page)

(continued from previous page)

```
> 0xb49625

# As part of shell script:
BLOCK_NUMBER=`in3 eth_blockNumber`
```

10.12.2 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"
  }
```

10.12.3 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
 > https://in3.slock.it/mainnet/nd-1
 > 0x784bfa9eb182c3a02dbeb5285e3dba92d717e07a
 > 65535
```

(continues on next page)

(continued from previous page)

```
 > 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"]
```

10.12.4 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
```

10.12.5 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 11

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.

11.1 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.
--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.
--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).

11.2 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
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
8. Consider using tools such as AWS Shield to protect your server from DOS attacks

11.3 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 .. code-block:: yaml

version: '2' services: incubed-parity:

image: parity:latest # Parity image with the proof function implemented. command: - –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: .. code-block:: 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.

CHAPTER 12

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.

12.1 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.

12.2 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.

12.3 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.

12.4 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.

12.4.1 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.

12.4.2 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.

12.4.3 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.

12.4.4 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.

12.4.5 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.

12.4.6 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.

12.5 Architecture

12.5.1 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

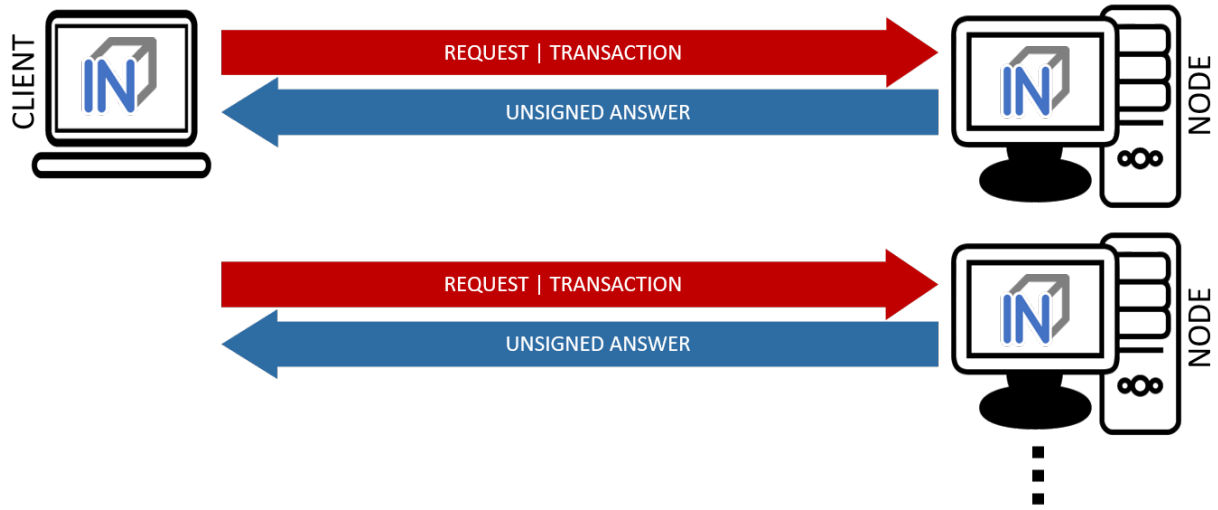
also send a follow-up query to the same node, for example, and thus the danger is high that the misbehavior will be uncovered.

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.

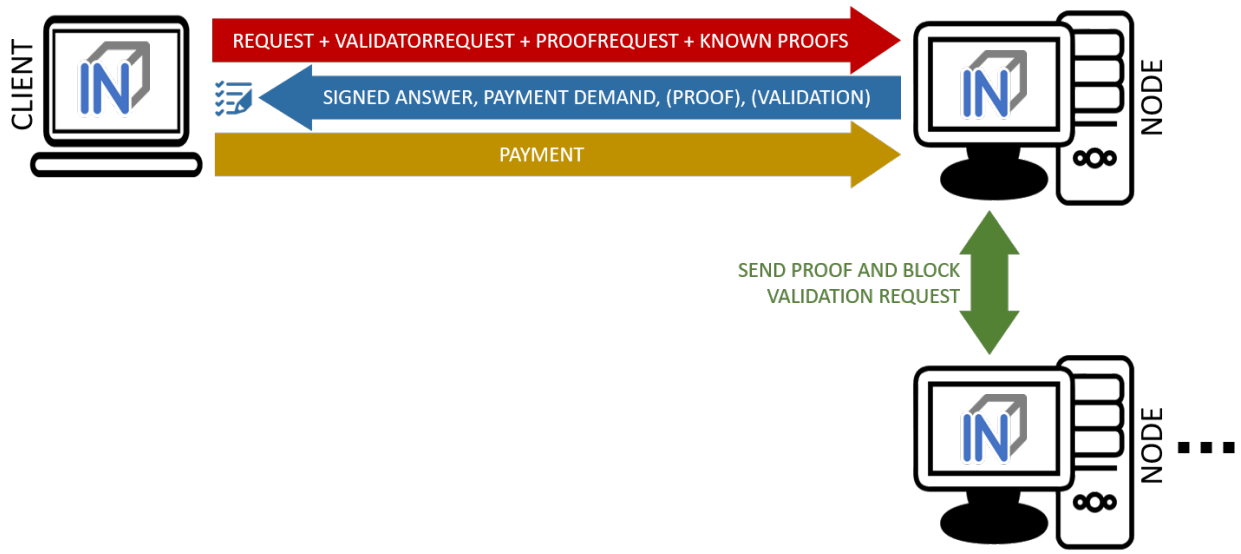
There are two strong incentives for the node to provide the correct response with high performance since it loses its deposit when a validator (wrong block hash) detects misbehavior and is eliminated from the registry, and receives a reward for this if it provides a correct response.

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.



12.5.2 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.

12.5.3 Netservice-Node

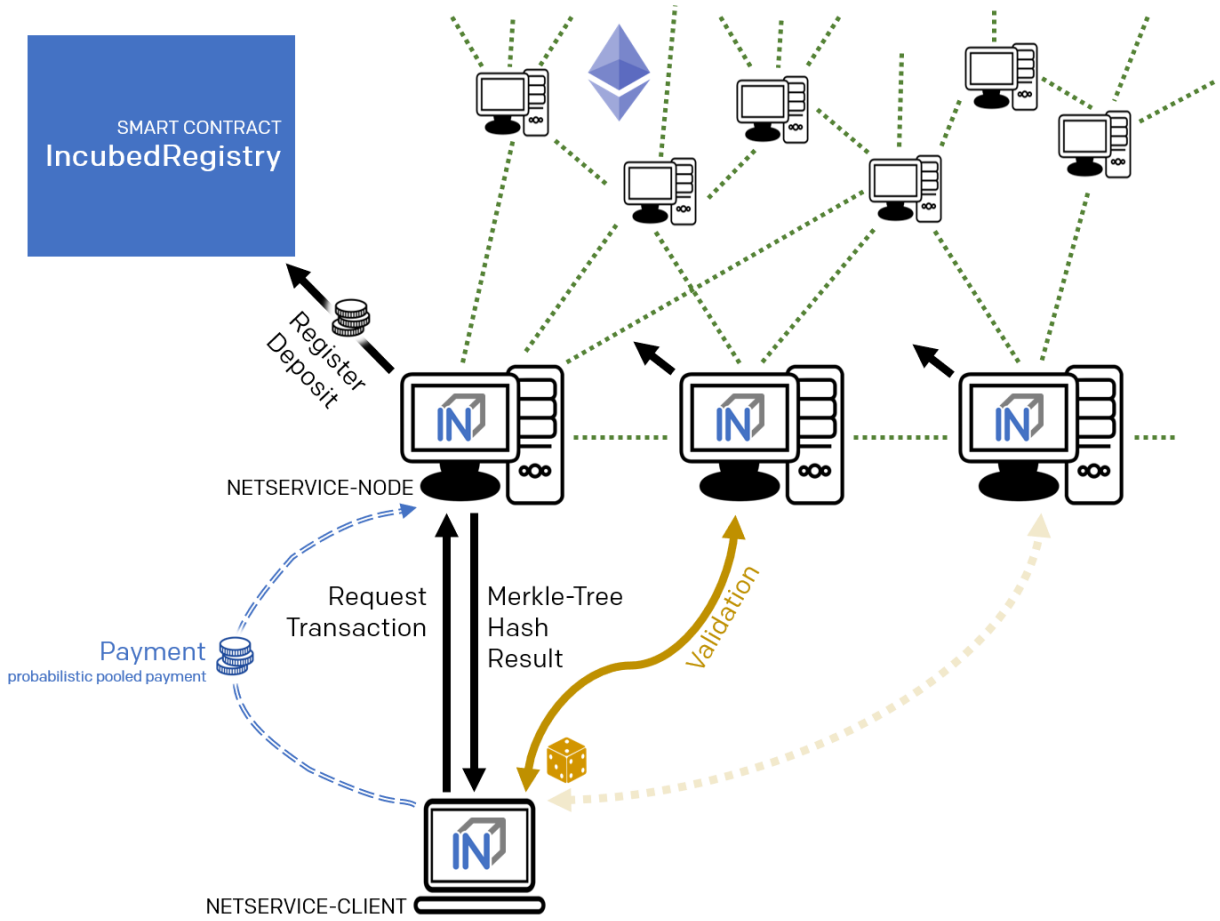
The net service node (short: node) is the communication interface for the client to the blockchain client. It can be implemented as a separate application or as an integrated module of a blockchain client (such as Geth or Parity).

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.

12.5.4 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.

12.5.5 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.

12.6 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.

12.6.1 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.

12.6.2 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.

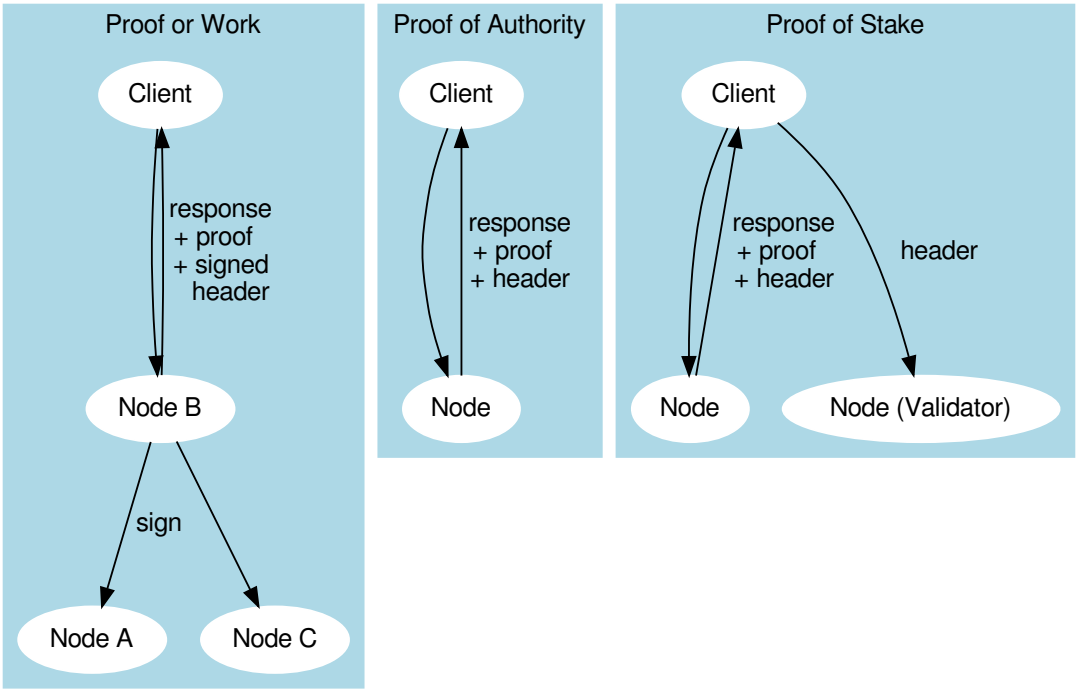
INCUBED also provides a platform for scaling by allowing multiple blockchains to be accessed in parallel from the same client. Although INCUBED is designed in the first instance for the Ethereum network (and other chains using the Ethereum protocol), in principle other networks and blockchains can also be integrated, as long as it is possible to realize a node that can work as information provider (incl. proof) and validator.

CHAPTER 13

Blockheader Verification

13.1 Ethereum

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.)



13.1.1 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.

13.1.2 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:

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.

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.

13.2 Bitcoin

Bitcoin may be a complete different chain, but there are ways to verify a Bitcoin Blockheader within a Ethereum Smart Contract. This requires a little bit more effort but you can use all the features of Incubed.

13.2.1 Block Proof

The data we want to verify are mainly Blocks and Transactions. Usually, if we want to get the BlockHeader or the complete block we already know the blockhash. And if we know that this hash is correct, verifying the rest of the block is easy.

1. We take the first 80 Bytes of the Blockdata, which is the blockHeader and hash it twice with sha256. Since Bitcoin stores the hashes in little endian, we then have to reverse the byteorder.

```
// 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 BlockHeader, 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')
```

Note : In order to verify that the target is correct, we can :

- take the target from a different blockheader in the same 2016 blocks epoch
- if we don't have one, we should ask for multiple nodes to make sure we have a correct target.

3. If we want to know if this is final, the Node needs to provide us with additional BlockHeaders on top of the current Block (FinalityHeaders).

   These header need to be verified the same way. But additionaly we need to check the parentHash:

```
if (!parentHash.reverse().equals( blockData.slice(4,36) ))
  throw new Error('wrong parentHash!')
```

4. In order to verify the Transactions (only if we have the complete Block, not only the BlockHeader), we need to read them, hash each one and put them in a merkle tree. If the root of the tree matches the merkleRoot, the transactions are correct.

```
// we take each Transactiondata, hash them and put the transactionhashes into a␣
↪merkle tree
const merkleRoot = createMerkleRoot ( readTransactions(blockData).map(_=>hash(_).
↪reverse()) )

// compare the root with merkleRoot of the header starting at offset 36
if (!merkleRoot.equals(blockData.slice(36,68).reverse()))
  throw new Error('Invalid MerkleRoot!')
```

13.2.2 Transaction Proof

In order to Verify a Transaction, we need a Merkle Proof. So the Incubed Server will have create a complete Merkle-Tree and then pass the other part of the pair as Proof.

Verifying means we start by hashing the transaction and then keep on hashing this result with the next hash from the proof. The last hash must match the merkleRoot.

13.2.3 Convicting For wrong Blockhashes in the NodeRegistry

Just as the Incubed Client can ask for signed blockhashes 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 1-3 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 14

Incentivization

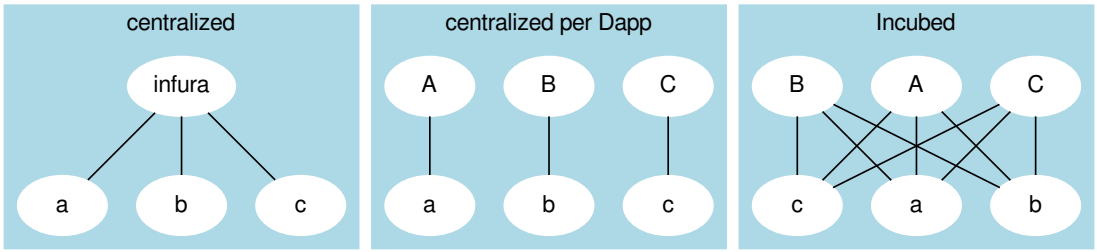
Important: This concept is still in development and discussion and is not yet fully implemented.

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.

14.1 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.



14.2 Incentivization for Nodes

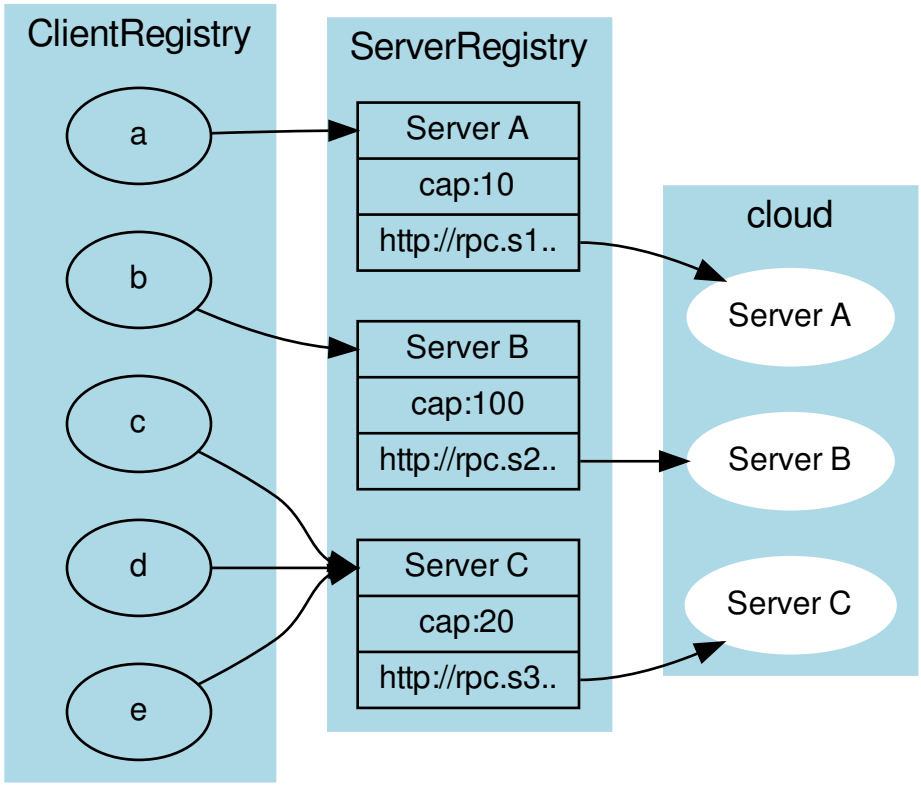
In order to incentivize a node to serve requests to clients, there must be something to gain (payment) or to lose (access to other nodes for its clients).

14.3 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.**

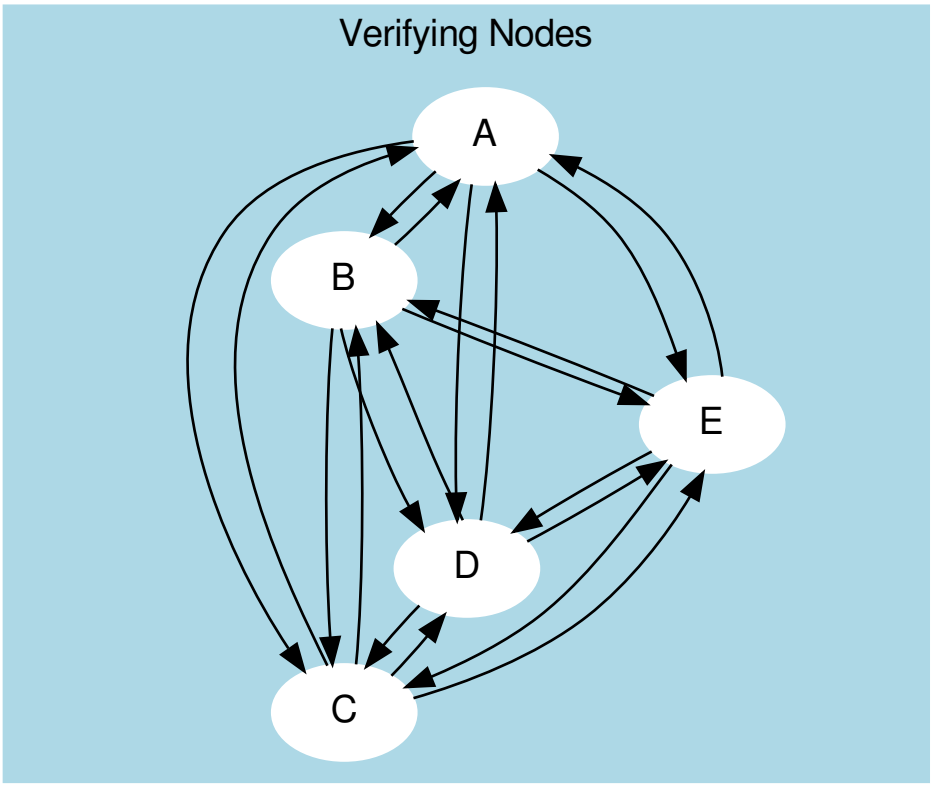
This requires a user to connect a client key (used to sign their requests) with a registered server. Clients are able to share keys as long as the owner of the node is able to ensure their security. This makes it possible to use one key for the same mobile app or device. The owner may also register as 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.



14.4 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.

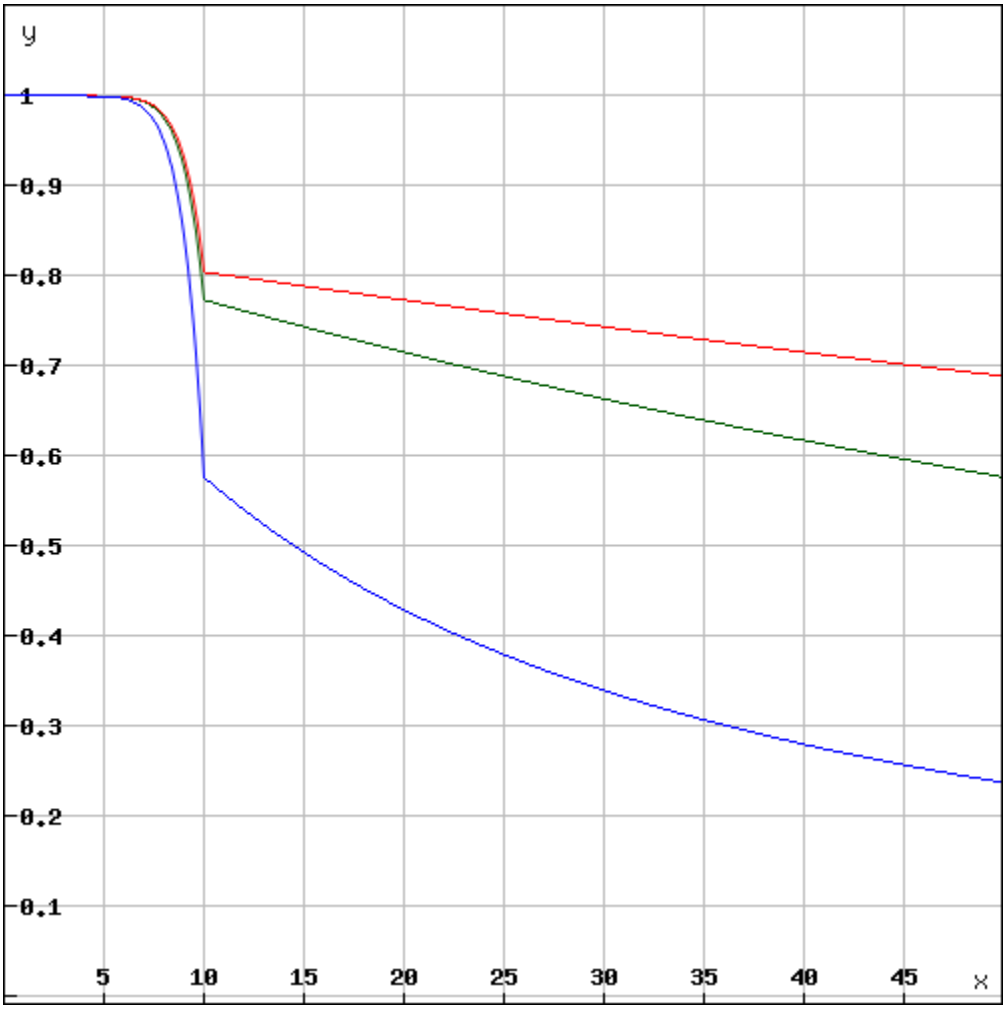
Based on the long-term (>1 day) and short-term (<1 day) availibility, the score is calculated as:

$$A = \frac{R_{received}}{R_{sent}}$$

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.

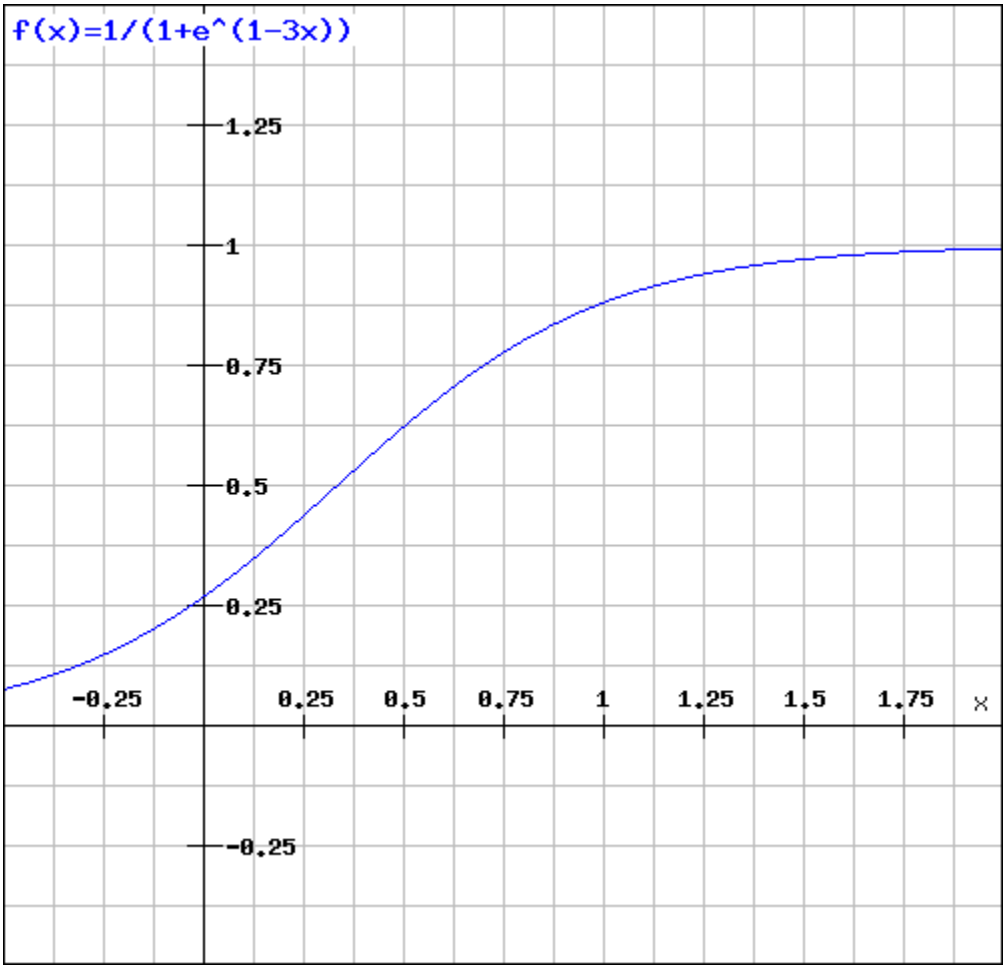
14.5 Deposit

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%.



14.6 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.

14.7 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.

14.8 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).

14.9 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.

14.10 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.

14.11 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 15

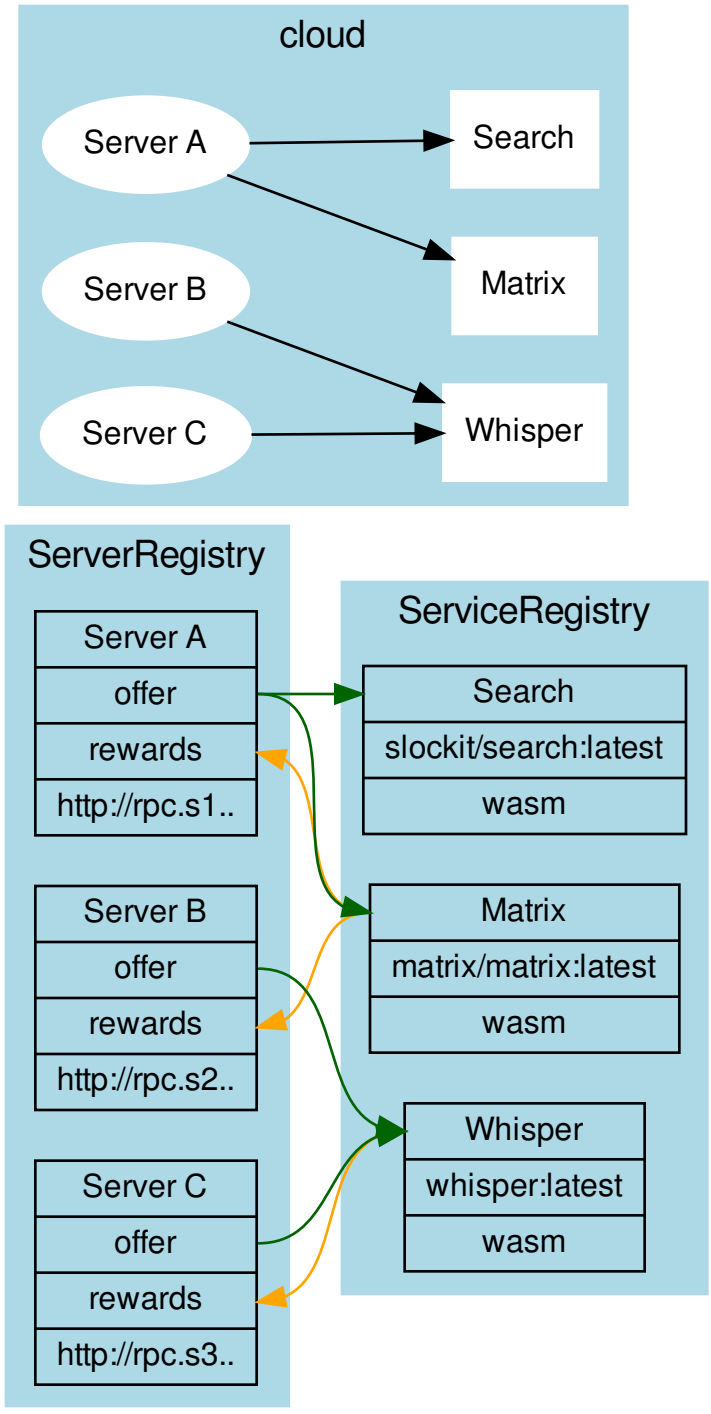
Decentralizing Central Services

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
Server B
Matrix
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

15.1 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.

15.2 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.

15.2.1 Convicting

Convicting on chain cannot be done, but each server is able to verify the result and, if false, downgrade the score.

- genindex

Index