Incubed Documentation

Release 1.2

Simon Jentzsch

Jun 24, 2019

Concept:

CHAPTER 1

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 underminded 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.

1.1 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.

1.2 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 have compute units not 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.

1.3 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.

1.4 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.

1.4.1 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.

1.4.2 Smart Bike Lock

Description

An 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.

1.4.3 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.

1.4.4 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.

1.4.5 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.

1.4.6 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.

1.5 Architecture

1.5.1 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.

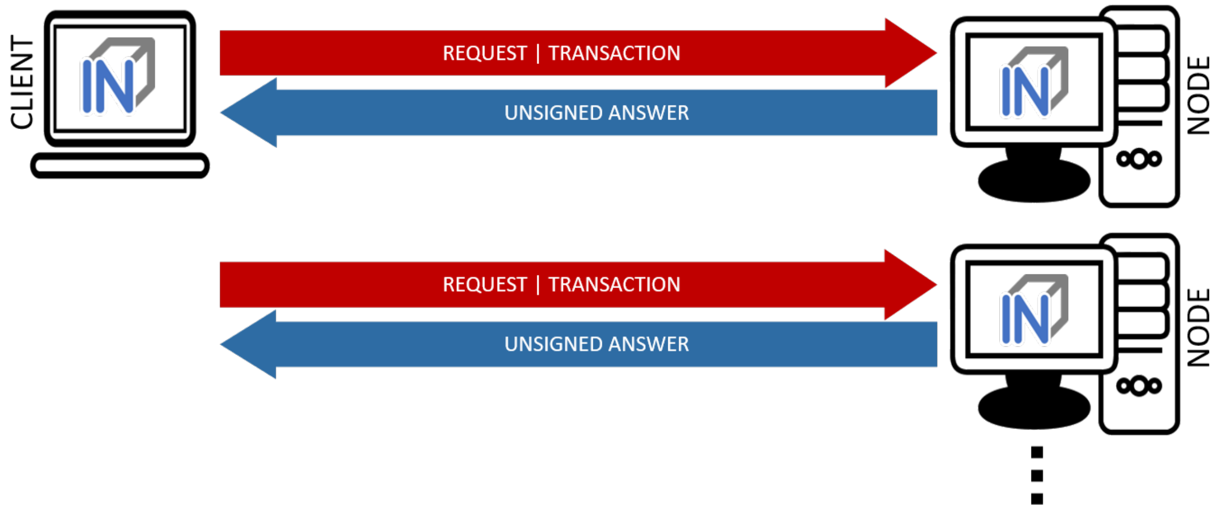
The selection of the nodes to be queried must be made at random. In particular, successive queries should always be sent to different nodes. This way it is not possible, or at least only very difficult, for a possibly misbehaving node to send specific incorrect answers to a certain client, since it cannot 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.

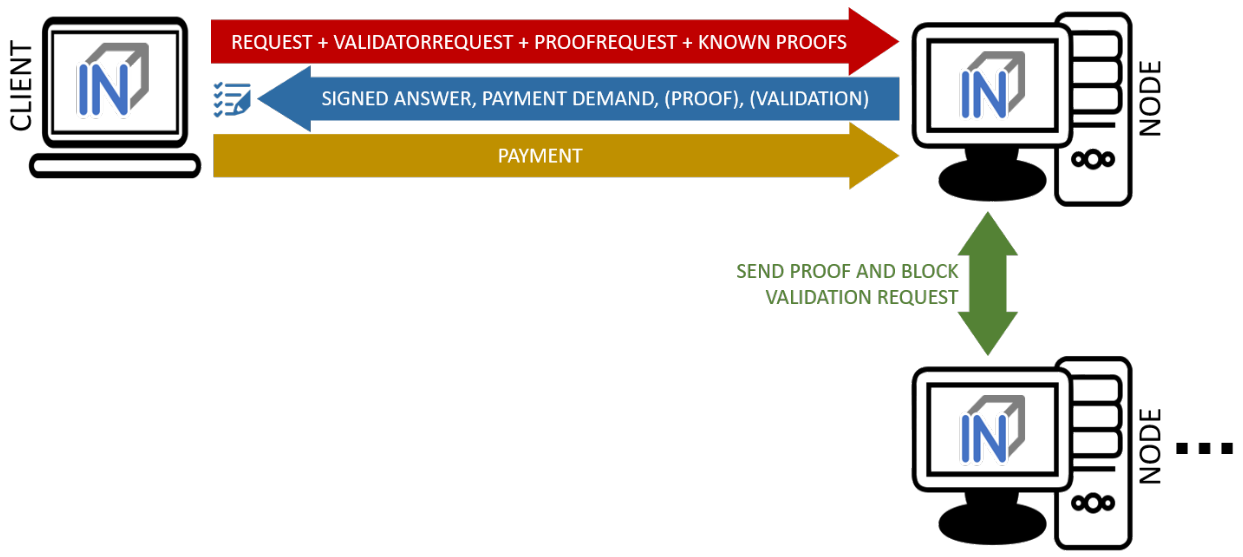
There are two strong incentives for the node to provide the correct response with high performance since it loses its deposit when a validator (wrong block hash) detects misbehavior and is eliminated from the registry, and receives a reward for this if it provides a correct response.

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.


CLIENT
REQUEST + VALIDATORREQUEST + PROOFREQUEST + KNOWN PROOFS
SIGNED ANSWER, PAYMENT DEMAND, (PROOF), (VALIDATION)
PAYMENT
NODE
SEND PROOF AND BLOCK VALIDATION REQUEST
NODE

1.5.2 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.

1.5.3 Netservice-Node

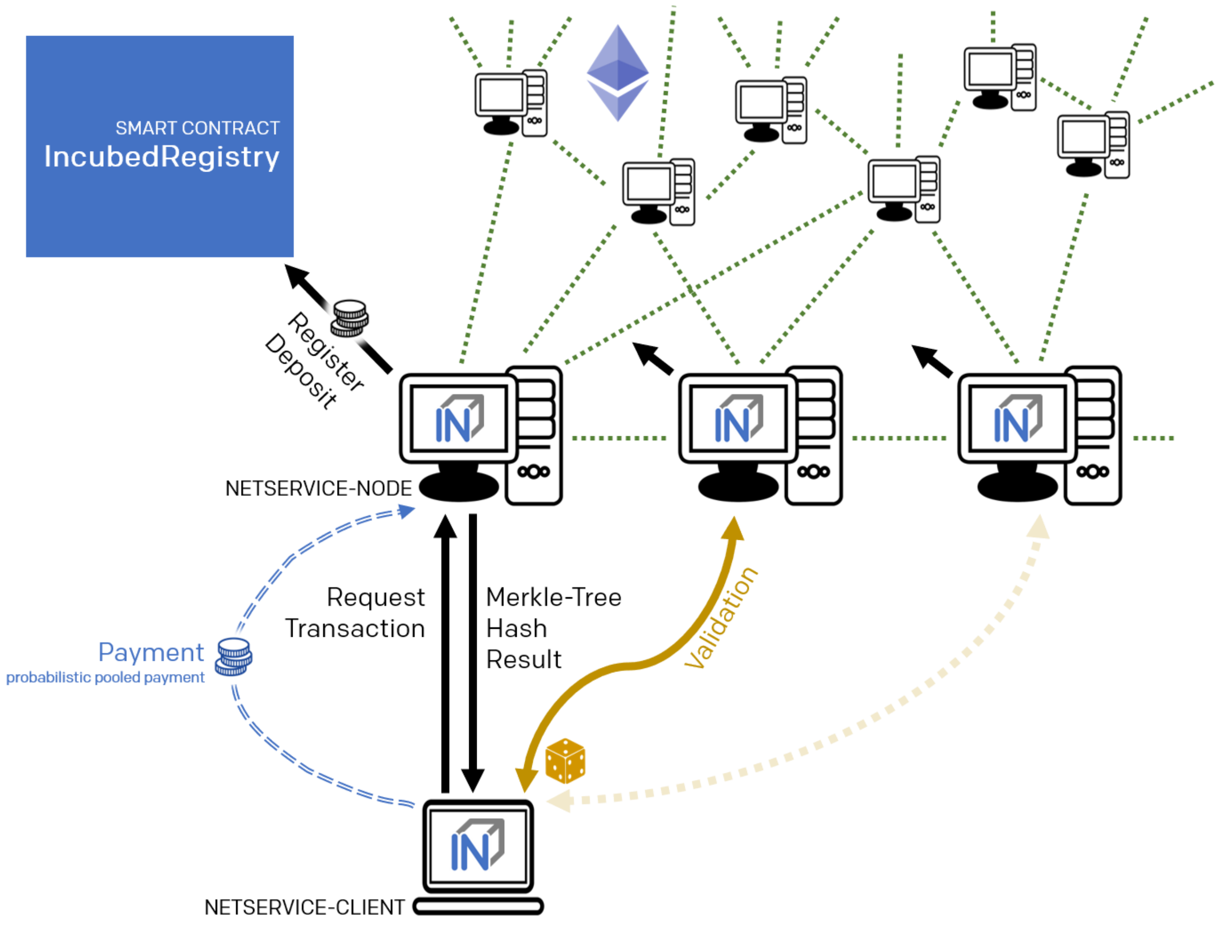
The net service node (short: node) is the communication interface for the client to the blockchain client. It can be implemented as a separate application or as an integrated module of a blockchain client (such as Geth or Parity).

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.

1.5.4 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.

1.5.5 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.

1.6 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.

1.6.1 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.

1.6.2 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 2

Getting Started

INCUBED can be used in different ways.

table

Stack	Size	Code Base	Use Case
TS/JS	2.7MB (browserified)	TypeScript	WebApplication (Client in the Browser) or Mobile Applications
C/C++	200kB	C	IoT-Devices, can be integrated nicely on many micro controllers (like [zephyr-supported boards] (https://docs.zephyrproject.org/latest/boards/index.html)) or anny other C/C++ -Application
Java	205kB	C	Java-Implementation of a native-wrapper
Docker	74MB	TypeScript	For replacing existing clients with this docker and connect to incubed via localhost:8545 without the need to change the architecture
bash	200kB	C	the commandline utils can be used directly as executable within bash-script or on the shell

other Languages will be supported soon (or can simply use the shared library directly).

2.1 TypeScript/JavaScript

Installing incubes is as easy as installing any other module:

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

2.1.1 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')
...
```

2.1.2 Direct 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
})

...
```

2.2 As Docker-Container

In order to start the incubed-client as a standalone client (allowing others none-js-application to connect to it), you can start the container as

```
docker run -d -p 8545:8545  slockit/in3:latest --chainId=mainnet
```

The application would then accept the following arguments:

–nodeLimit the limit of nodes to store in the client.

–keepIn3 if true, the in3-section of thr response will be kept. Otherwise it will be removed after validating the data. This is useful for debugging or if the proof should be used afterwards.

–format the format for sending the data to the client. Default is json, but using cbor means using only 30-40% of the payload since it is using binary encoding.

–autoConfig if true the config will be adjusted depending on the request

–retryWithoutProof if true the request may be handled without proof in case of an error. (use with care!)

–includeCode if true, the request should include the codes of all accounts. otherwise only the codeHash is returned. In this case the client may ask by calling eth_getCode() afterwards

–maxCodeCache number of max bytes used to cache the code in memory

–maxBlockCache number of number of blocks cached in memory

–proof 'none' for no verification, 'standard' for verifying all important fields, 'full' veryfying all fields even if this means a high payload.

–signatureCount number of signatures requested

–finality percenage of validators signed blockheaders - this is used for PoA (aura)

–minDeposit min stake of the server. Only nodes owning at least this amount will be chosen.

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

–requestCount the number of request send when getting a first answer

–timeout specifies the number of milliseconds before the request times out. increasing may be helpful if the device uses a slow connection.

–chainId servers to filter for the given chain. The chain-id based on EIP-155.

–chainRegistry main chain-registry contract

–mainChain main chain-id, where the chain registry is running.

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

–loggerUrl a url of RES-Endpoint, the client will log all errors to. The client will post to this endpoint JSON like { id?, level, message, meta? }

2.3 C - Implementation

The C-Implemetation will be released soon!

```c
#include <stdio.h>
#include <in3/client.h>  // the core client
#include <eth_full.h>    // the full ethereum verifier containing the EVM
#include <in3/eth_api.h> // wrapper for easier use
#include <in3_curl.h>    // transport implementation

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

(continues on next page)

(continued from previous page)

```
  // register a chain-verifier for full Ethereum-Support
  in3_register_eth_full();

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

  // set your config
  c->transport    = send_curl; // use curl to handle the requests
  c->requestCount = 1;         // number of requests to send
  c->chainId      = 0x1;       // use main chain

  // use a ethereum-api instead of pure JSON-RPC-Requests
  eth_block_t* block = eth_getBlockByNumber(c, atoi(argv[1]), true);
  if (!block)
    printf("Could not find the Block: %s", eth_last_error());
  else {
    printf("Number of verified transactions in block: %i", block->tx_count);
    free(block);
  }

  ...
}
```

More Details are comming soon. . .

2.4 Java

The Java-Implementation uses a wrapper of the C-Implemenation. That's why you need to make sure the libin3.so or in3.dll or libin3.dylib can be found in the java.library.path, like

java -Djava.library.path="path_to_in3;${env_var:PATH}" HelloIN3.class

```
import org.json.*;
import in3.IN3;

public class HelloIN3 {
   //
   public static void main(String[] args) {
       String blockNumber = args[0];
       IN3 in3 = new IN3();
       JSONObject result = new JSONObject(in3.sendRPC("eth_getBlockByNumber",{
↪blockNumberm ,true})));
       ....
   }
}
```

2.5 Commandline Tool

Based on the C-Implementation a Commandline-Util is build, 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=`cat mysecret_key.txt` in3 eth_sendTransaction '{"from":
↪"0x5338d77B5905CdEEa7c55a1F3A88d03559c36D73", "to":
↪"0xb5049E77a70c4ea06355E3bcbfcF8fDADa912481", "value":"0x10000"}'
```

2.6 Supported Chains

Currently incubed is deployed on the following chains:

2.6.1 Mainnet

Registry : 0x2736D225f85740f42D17987100dc8d58e9e16252

ChainId : 0x1 (alias `mainnet`)

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

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

2.6.2 Kovan

Registry : 0x27a37a1210df14f7e058393d026e2fb53b7cf8c1

ChainId : 0x2a (alias `kovan`)

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

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

2.6.3 Tobalaba

Registry : 0x845E484b505443814B992Bf0319A5e8F5e407879

ChainId : 0x44d (alias `tobalaba`)

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

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

2.6.4 Evan

Registry : 0x85613723dB1Bc29f332A37EeF10b61F8a4225c7e

ChainId : 0x4b1 (alias `evan`)

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

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

2.6.5 Görli

Registry : 0x85613723dB1Bc29f332A37EeF10b61F8a4225c7e

ChainId : 0x5 (alias `goerli`)

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

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

2.6.6 IPFS

Registry : 0xf0fb87f4757c77ea3416afe87f36acaa0496c7e9

ChainId : 0x7d0 (alias `ipfs`)

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

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

2.7 Registering a own in3-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",
```

(continues on next page)

(continued from previous page)

To run a 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 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 automaticly update the client
    - --pruning=archive
    - --pruning-memory=30000                                # limit storage
```

CHAPTER 3

Technical Background

3.1 Ethereum Verification

The Incubed is also often called Minimal Verifying Client because it may not sync, but still is able to verify all incoming data. This is possible since ethereum is based on a technology allowing to verify almost any value.

Our goal was to verify at least all standard `eth_...` rpc methods as described in the Specification.

In order to prove something, you always need a starting value. In our case this is the BlockHash. Why do we use the BlockHash? If you know the BlockHash of a block, you can easily verify the full BlockHeader. And since the BlockHeader contains the stateRoot, transationRoot and receiptRoot, these can be verified as well. And the rest will simply depend on them.

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

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

- ***Block Proof*** - verifies the content of the BlockHeader
- ***Transaction Proof*** - verifies the input data of a transaction
- ***Receipt Proof*** - verifies the outcome of a transaction
- ***Log Proof*** - verifies the response of `eth_getPastLogs`
- ***Account Proof*** - verifies the state of an account
- ***Call Proof*** - verifies the result of a `eth_call` - response

3.1.1 BlockProof

BlockProofs are used whenever you want to read data of a Block and verify them. This would be:

- eth_getBlockTransactionCountByHash

- eth_getBlockTransactionCountByNumber
- eth_getBlockByHash
- eth_getBlockByNumber

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

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

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

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

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

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

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

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

3.1.2 Transaction Proof

TransactionProofs are used for the following transaction-methods:

- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex

In order to verify we need :

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)

2. serialize the transaction-data

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

1. verify the merkleProof of the transaction with

```
verifyMerkleProof(
  blockHeader.transactionRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transaction /* expected value */
)
```

The Proof-Data will look like these:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "blockHash": "0xf1a2fd6a36f27950c78ce559b1dc4e991d46590683cb8cb84804fa672bca395b",
    "blockNumber": "0xca",
    "from": "0x7e5f4552091a69125d5dfcb7b8c2659029395bdf",
    "gas": "0x55f0",
    "gasPrice": "0x0",
    "hash": "0xe9c15c3b26342e3287bb069e433de48ac3fa4ddd32a31b48e426d19d761d7e9b",
    "input": "0x00",
    "value": "0x3e8"
    ...
  },
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block": "0xf901e6a040997a53895b48...", // serialized blockheader
      "merkleProof": [  /* serialized nodes starting with the root-node */

↪"f868822080b863f86136808255f0942b5ad5c4795c026514f8317c7a215e218dccd6cf8203e8001ca0dc967310342af50
↪"
      ],
      "txIndex": 0,
      "signatures": [...]
    }
  }
}
```

3.1.3 Receipt Proof

Proofs for the transactionReceipt are used for the following transaction-method:

- eth_getTransactionReceipt

In order to verify we need :

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)
2. serialize the transaction receipt

```
transactionReceipt = rlp.encode([
  uint( r.status || r.root ),
  uint( r.cumulativeGasUsed ),
  bytes256( r.logsBloom ),
  r.logs.map(l => [
    address( l.address ),
    l.topics.map( bytes32 ),
    bytes( l.data )
  ])
].slice(r.status === null && r.root === null ? 1 : 0))
```

1. verify the merkleProof of the transaction receipt with

```
verifyMerkleProof(
  blockHeader.transactionReceiptRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transactionReceipt /* expected value */
)
```

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

3.1.4 Log Proof

Proofs for logs are only for the one rpc-method:

- eth_getLogs

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

That's why this proof needs to provide

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

The Proof data structure will look like this:

```
  Proof {
    type: 'logProof',
    logProof: {
      [blockNr: string]: {  // the blockNumber in hex as key
        block : string  // serialized blockheader
        receipts: {
          [txHash: string]: {  // the transactionHash as key
            txIndex: number // transactionIndex within the block
```

(continues on next page)

(continued from previous page)

```
            txProof: string[] // the merkle Proof-Array for the transaction
            proof: string[] // the merkle Proof-Array for the receipts
          }
        }
      }
    }
  }
```

In order to verify we need :

1. deserialize each blockheader and compare the blockhash with the signed hashes. (See *BlockProof*)
2. for each blockheader we verify all receipts by using

```
verifyMerkleProof(
  blockHeader.transactionReceiptRoot, /* root */,
  keccak256(proof.txIndex), /* key or path */
  proof.merkleProof, /* serialized nodes starting with the root-node */
  transactionReceipt /* expected value */
)
```

1. The resulting values are the receipts. For each log-entry, we are comparing the verified values of the receipt with the returned logs to ensure that they are correct.

3.1.5 Account Proof

Prooving an account-value applies to these functions:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt

eth_getProof

For the Transaction or Block Proofs all needed data can be found in the block itself and retrieved through standard rpc calls, but if we want to approve the values of an account, we need the MerkleTree of the state, which is not accessable through the standard rpc. That's why we have created a EIP to add this function and also implemented this in geth and parity:

- parity (Status: pending pull request) - Docker
- geth (Status: pending pull request) - Docker

This function accepts 3 parameter :

1. `account` - the address of the account to proof
2. `storage` - a array of storage-keys to include in the proof.
3. `block` - integer block number, or the string "latest", "earliest" or "pending"

```
{
  "jsonrpc": "2.0",
  "id": 1,
```

(continues on next page)

(continued from previous page)

```
  "method": "eth_getProof",
  "params": [
    "0x7F0d15C7FAae65896648C8273B6d7E43f58Fa842",
    [  "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421" ],
    "latest"
  ]
}
```

The result will look like this:

```
{
  "jsonrpc": "2.0",
  "result": {
    "accountProof": [
      "0xf90211a...0701bc80",
      "0xf90211a...0d832380",
      "0xf90211a...5fb20c80",
      "0xf90211a...0675b80",
      "0xf90151a0...ca08080"
    ],
    "address": "0x7f0d15c7faae65896648c8273b6d7e43f58fa842",
    "balance": "0x0",
    "codeHash": "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
    "nonce": "0x0",
    "storageHash": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
↪",
    "storageProof": [
      {
        "key": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
        "proof": [
          "0xf90211a...0701bc80",
          "0xf90211a...0d832380"
        ],
        "value": "0x1"
      }
    ]
  },
  "id": 1
}
```

In order to run the verification the blockheader is needed as well.

The Verification of such a proof is done in the following steps:

1. serialize the blockheader and compare the blockhash with the signed hash as well as with the blockHash and number of the transaction. (See *BlockProof*)
2. Serialize the account, which holds the 4 values:

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

1. verify the merkle Proof for the account using the stateRoot of the blockHeader:

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

In case the account does exist yet, (which is the case if `none == startNonce` and `codeHash == '0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470'`), the proof may end with one of these nodes:

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

Both would prove, that this key does not exist.

1. Verify each merkle Proof for the storage using the storageHash of the account:

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

3.1.6 Call Proof

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

- eth_call

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

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

For Verifying you need to follow these steps:

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

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

  // fill in values
  for (const adr of Object.keys(accounts)) {
```

(continues on next page)

(continued from previous page)

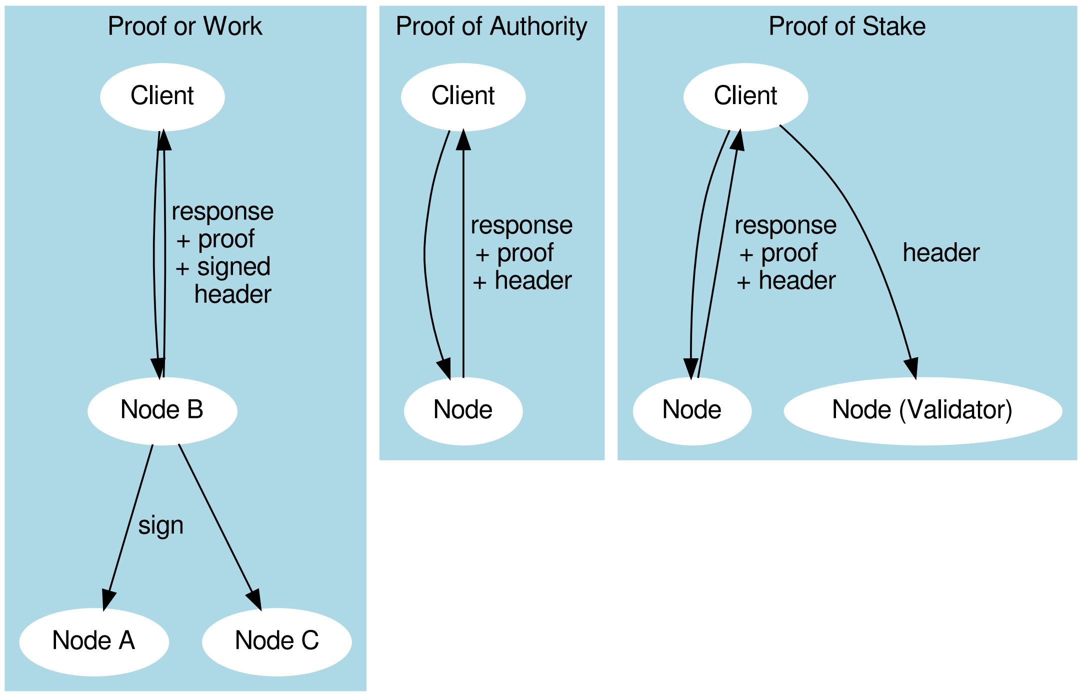
```
    const ac = accounts[adr]

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

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

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

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

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

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

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

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

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

CHAPTER 4

Verifying Blockheaders

Since all proofs always include the blockheader, it is crucial to verify the correctness of these data as well. But verification depends on the consensys of the underlying blockchain. (for Details See *Ethereum Verification and MerkleProof*)



4.1 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 this 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. That's 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 chooses one node and let this node run a subrequest to get the signatures. This means less requests for the clients, but also at least one node checks the signatures and convicts the other if they lied.

4.2 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 can do not need an additional blockhash signed. The only critical information we rely on is the list of validators.

Currently there are 2 Consensys algorithms:

4.2.1 Aura

Aura is used by parity only and there are 2 ways to configure such:

- **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 which offers a 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. That's why the proof for such a contract depends on the rules layed out in the contract and is chain-specific.

4.2.2 Clique

Clique is a protocol developed by the geth-team and is now also supported by parity (see görli-testnet). Instead of relying on a contract, clique defines a protocol of how validator nodes may change. All votes are done directly in the blockheader. This makes it easier to prove, since it does not rely on any contract.

The Incubed nodes will check all the blocks for votes and create a `validatorlist` which 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 he verified each blockheader and vote. Even though malicious nodes cannot forge the signatures of a validator, that may skip votes in the validatorlist. That is why a validatorlist update should always be done by running multiple requests and merging them together.

CHAPTER 5

Incentivization

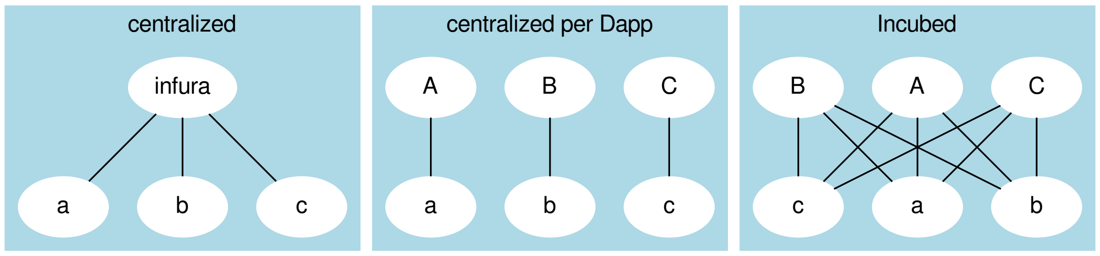
Important: This concept is still in development and discussion and not yet fully implemented.

The original idea of blockchain is a permissionless peer-to-peer network in which anybody can participate if he only runs a node and syncs with other peers. Since this is still true, we know that such a node won't run on a small IoT-device.

5.1 Decentralizing Access

This is why a lot of users try remote-nodes to server their devices. But 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.



5.2 Incentivization for nodes

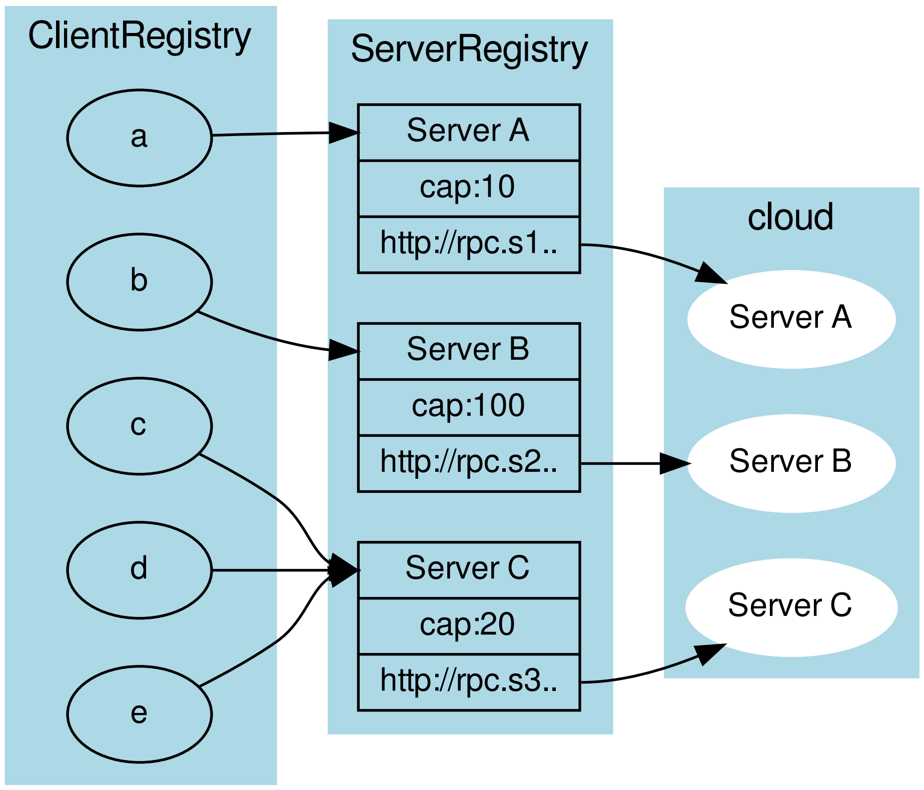
In order to incentivize a node to serve requests to clients, there must be something to gain (payment) or to lose (access to other nodes for its clients).

5.3 Connecting Clients and Server

As a simple rule we can define:

> **The Incubed network will serve your client requests if you also run an honest node.**

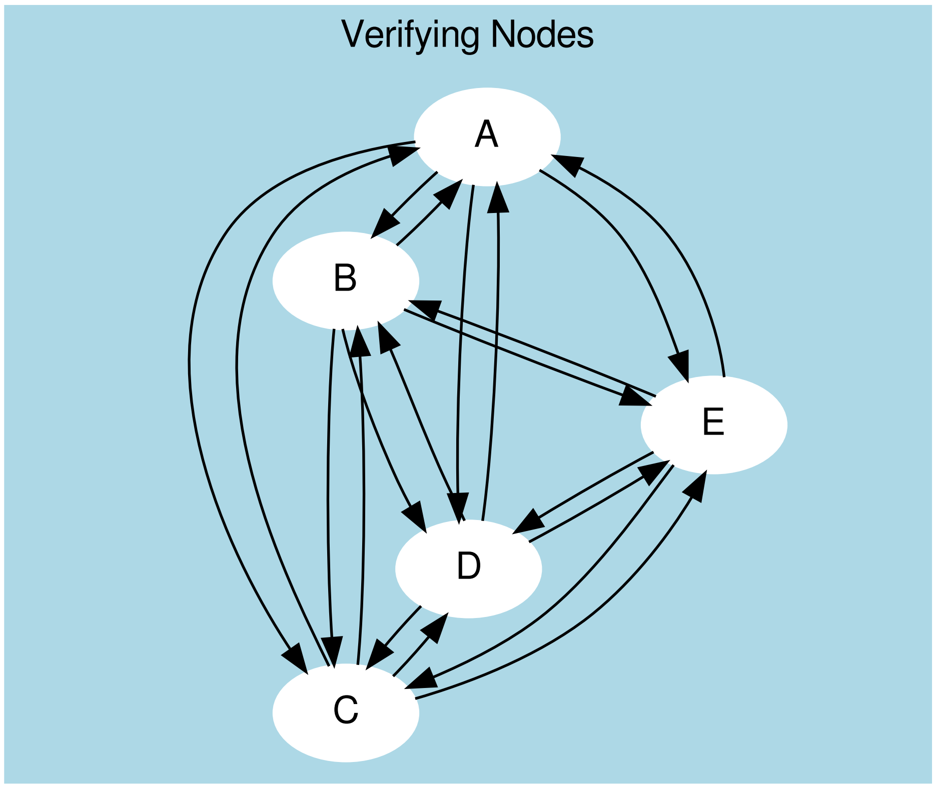
This requires to connect a client key (used to sign his requests) with a registered server. Clients are able to share keys as long as the owner of the node is able to ensure their security. This makes it possible to use one key for the same mobile app or device. The owner may also register as many keys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



5.4 Ensuring Client Access

Connecting a client key to a server does not mean he relies on it, but his requests are simply served in the same quality as the connected node serves other clients. This creates a very strong incentive to deliver all clients, because if a server node were offline or refused to deliver, eventually other nodes would also 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 (like every hour in order to make sure a malicious server will not respond to test requests only, these requests may be sent through an anonymous network like tor)

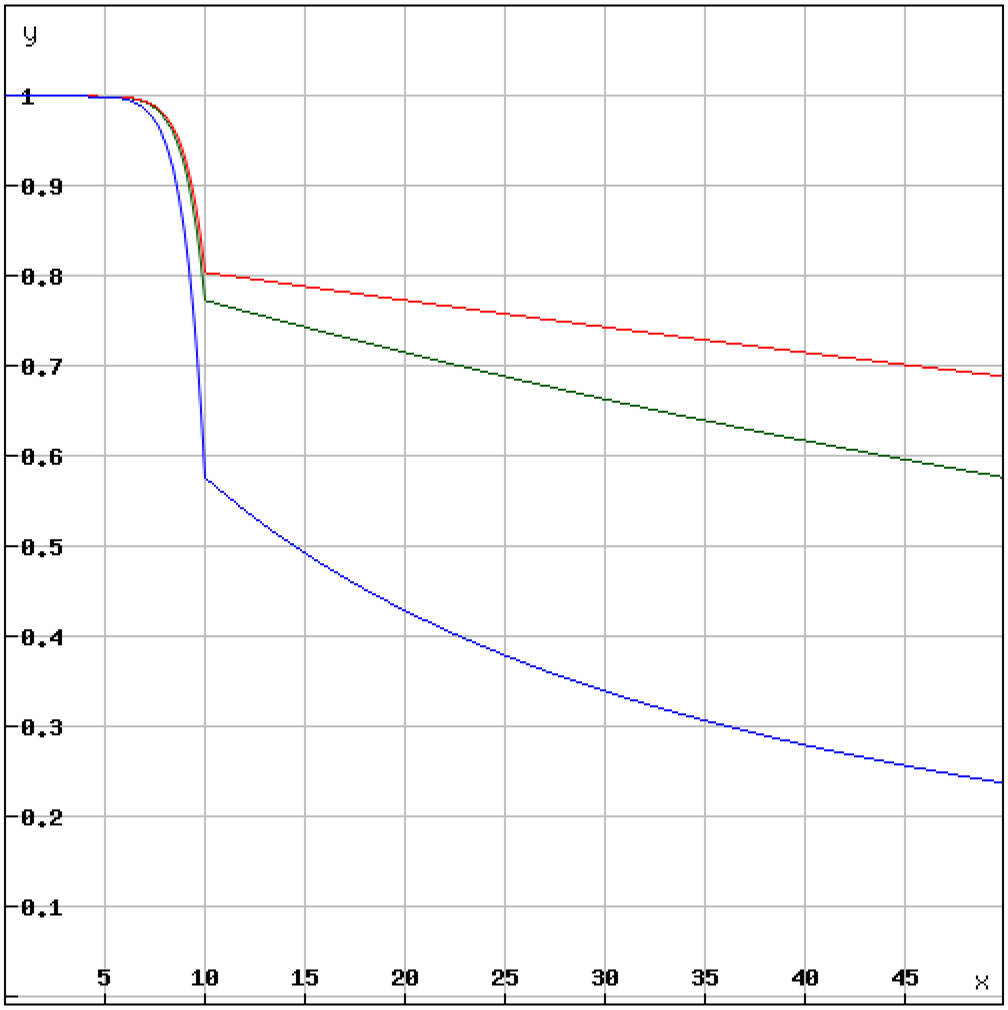
Based on the long-term (>1 day) and short-term (<1 day) availibility the score is calculated:

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

In order to balance long-term availability and short-term issues, each node measures both and calculates a factor for the score. This factor should ensure that a short-term issues will not drop the score immediately, but keep it up for a while and then drop. Also, long-term availibility must be rewarded by dropping more slowly.

$$A = 1 - (1 - \frac{A_{long} + 5 \cdot A_{short}}{6})^{10}$$

- A_{long} - the ratio between valid request received and sent within the last month
- A_{short} - the ratio between valid request received and sent within the last 24h



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 servers clients (See LoadBalancing)
- C_{max} - the maximal Number of open or parallel Requests the own 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. check the signature
2. calculate the score based on the score of the node it is connected with.
3. accept or reject 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 I have a low score and the load in the network is high, my clients may get rejected often and so have to wait longer for responses. And if I have a score of 0, they are blacklisted even.

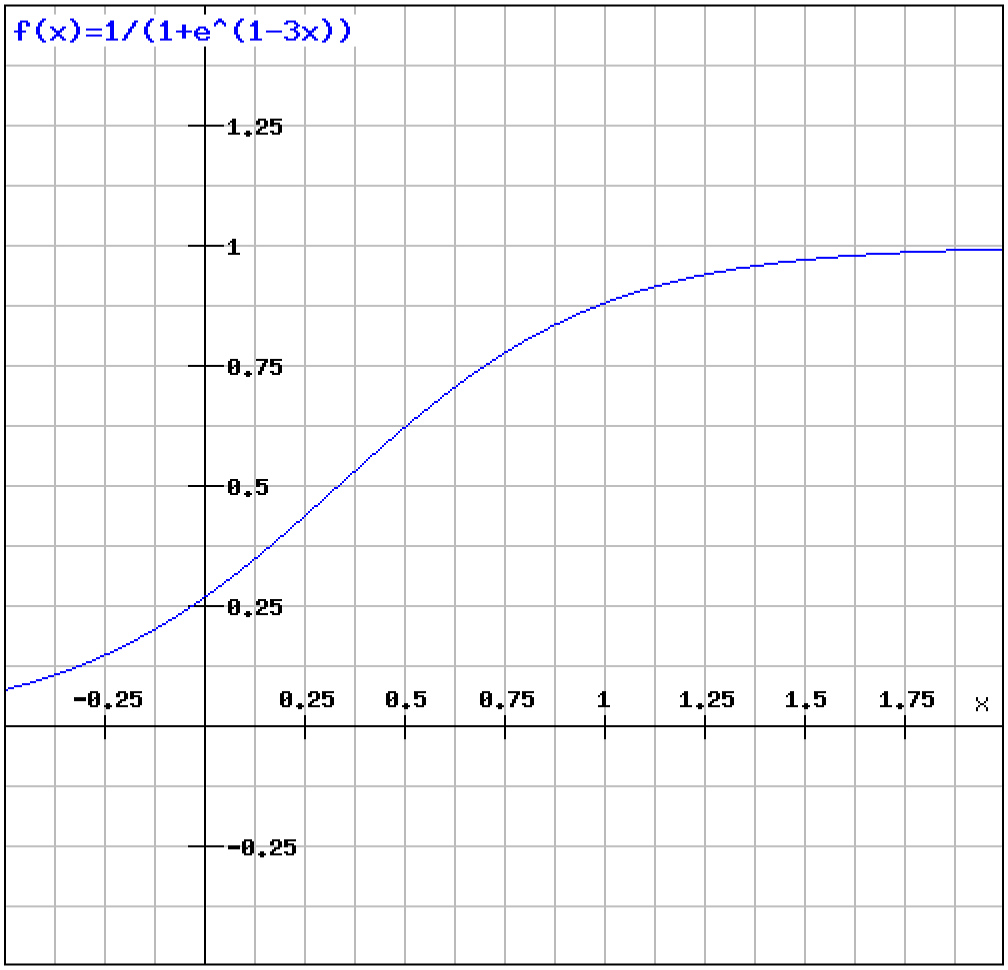
5.5 Deposit

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 so get only 26.8% of the max cap while any node with an average deposit gets 88% and above and quickly reaches 99%



5.6 LoadBalancing

In an optimal network, each server would handle the same amount as the servers 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, while the node only delivers 10% of requests in the network, we need to decrease the score for clients sending more requests than their shares. So 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 - number of request serverd to one of the clients connected to the node
- $\sum_{i=0}^{n} R_i$ - total number of request serverd
- $\sum_{i=0}^{n} C_i$ - total number of capacities of the registered servers
- C_n - Capacity of the registered node

Each node will update the $score$ and the $weight$ for the other nodes after each check and this way prioritize incoming requests.

The capacity of a node is the maximal number of parallel request it can handle and is stored in the ServerRegistry. This way all client know the cap and will weigh the nodes accordingly, which leads to more load to stronger servers. A node declaring a high capacity will gain a higher score and so its clients will get more reliable responses. On the other hand, if you cannot deliver the load, you may lose your availability and so you score.

5.7 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 server requests only if the current load or the open requests are less then this number, which would mean that getting a response from the network without signing may take very long because this client would send a lot of requests until he is lucky enough to get a response if the load is high. Chances are a lot higher if the load is very low.

5.8 Convict

Even though servers are allowed to register without a deposit, convicting is still a hard punishment. Because in this case the server is not part of the registry anymore and all his connected clients are treated as without signature. In this case, his devices or app will probably stop working or be extremely slow. (depending on the freeScore configured in all the nodes)

5.9 Handling conflicts

In case of a conflict, each client has now at least one server he knows he 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.

CHAPTER 6

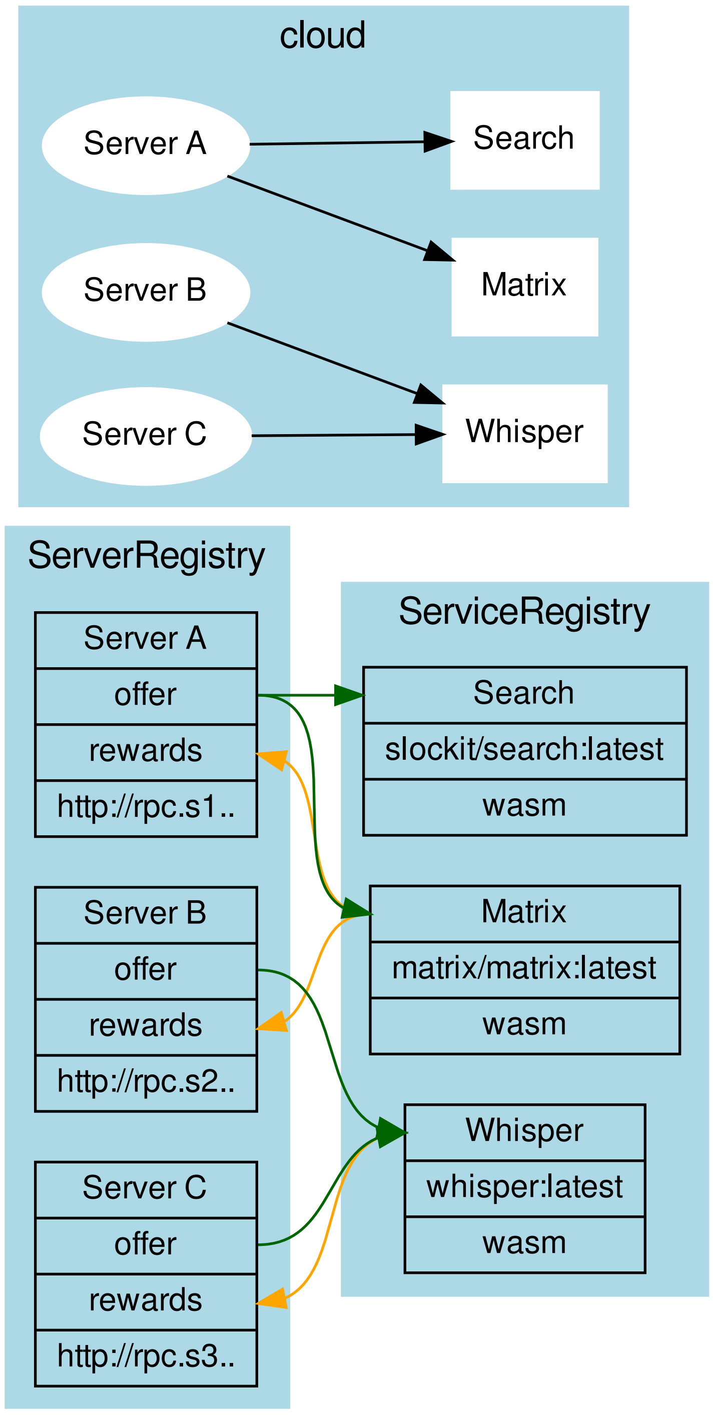
Decentralizing central services

Important: This concept is still in early development and discussion and not implemented yet, but planned in future milestones.

Many DAPPs still need some offchain-services, like search-services running on a server, which of course can be seen as single point of failure. In order to dectralize these even dapp-specific services they must fullfill these criteria:

1. **stateless** - since requests may be send to different servers they cannot hold a users 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
Server B
Server C
Search
Matrix
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

6.1 Incentivication

Each Server can define

- a list of services as offer
- a list of services to reward

The main idea is simply:

> **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, he will also run at least as many rewareded services of the other node.

6.2 Verification

Each Service specify a Verifier, which is a wasm-module (specified through a ipfs-hash). This wasm offers 2 function:

```
function minRequests():number

function verify(request:RPCRequest[], responses:RPCResponse[])
```

A minimal version could simply asure running at least 2 requests and comparing them. In case they differ they can

- check with the "home"-server
- convict nodes

6.2.1 convicting

As a generic service convicting on chain can not be done, but each server is able to verify the result and if false downgrade the score.

CHAPTER 7

Threat Model for Incubed

7.1 Registry Issues

7.1.1 Long Time Attack

Status: open

A client is offline for a long time and did not update the nodelist. During this time a Server has now been 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: Yes. I think often the fallback is "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 can not withdraw their funds. Client will most likely look first for those, especially those who only need data from the chain occasionally.
>
> SIM: Yes this could help, but only protects from regular unregistering. If you convict a server, then this timeout does not help.
>
> In order to remove this issue completely you would need a trusted authority where you can 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 blocknumber when the nodeList changed, the client will find out, that he needs to update the nodeList and by having multiple Requests in parallel he reduces the risk of relying on a manipulated nodeList. Because the malicious Server may return a correct nodeList for an older block when this server was still valid and even get signatures for this, but not for a newer BlockNumber, which can only be found out by asking as many servers an needed.
>
> Another point is, that as long as the signature does not come from the same server, the Data-Provider 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. An in order to use this attack both (The DataProvider and BlockHashSigner) must work together in order 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 source (ideally from bootstrap nodes similar to light clients and ask for the current list). Actually, we could define the same bootstrap nodes as are currently hard coded in parity and geth

7.1.2 Inactive Server Spam Attack

Status: solved

Everyone can register a lot of servers that don't even exist or are not running. He may even put in a decent deposit. Of course the client would try and find out that these nodes are inactive. If an attacker is able to onboard enough inactive servers, the chances for an in3 client to find a working server decreases.

Solutions:

In order 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 4,3mio ether when trying to register 2 servers within 1 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 on possible spam attacks (e.g. through voting).

In addition, the smart contract provides a voting function for removing inactive servers: In order to vote, a server has to sign a message with a current block and the owner of the server he wants 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 votingPower gets capped at 1 year and won't increase further. The server being voted out will also gets an oppositional voting power that is capped at 2 years. In order for the server to be voted out, the combined votingPower of all the servers has to be greater then the oppositional voting power and also more 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 maximal amount of signatures at 24: This means to vote out an server that has been active for more then 2 years, 24 in3-servers with a lifetime of 1 month are required to vote. This number decreases when more older servers are voting. This mechanic will prevent the rapid onboarding of many malicious in3-servers that would vote all regular servers and take control of the in3-nodelist. In addition, we do not allow all servers to vote. Instead, we chose up to 24 servers randomly with the blockhash as seed. For the vote to succeed, they have to sign on the same blockhash and have enough voting power.

In order to "punish" the a server-owner for having an inactive server, after a successful vote he will lose 1% of his deposit while the rest gets locked until his deposit timeout expires, ensuring possible liabilities. A part of this 1% deposit will be used to reimburse the transaction costs, the rest will be burned. To make sure that the transaction always will be payed, a minimum deposit of 10 finney (= 0.01 ether) is also enforced.

7.1.3 Self Convict Attack

Status: solved

A User may register a mailcious Server and even store a deposit, but as soon as he signes a wrong blockhash use a 2nd Account and convict himself in order to get the deposit before somebody else can.

Solutions:

> SIM: In this case cas the Attacker would lose 50% of his deposit, because this will be burned. But this also means he would get the other half and so the price he would have to pay for lying is up to 50% of his deposit. This should be considered by the clients when picking nodes for signatures.

7.1.4 Convict Frontrunner Attack

Status: solved

Servers are acting as watchdogs and automaticly call convict if they receive a wrong blockhash. This will cost them some gas to send the transaction. Especially 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 he is incentiviced to do so, because after successfully convicting he gets a reward (50% of the deposit). A miner or other Attacker could now simply wait for a pending transaction for convict and simply use the data and send the same transaction with a high gasprice, which eventually mean, his transaction would be mined first and the server, after putting so much work and costs into preparing the convict, will get nothing.

Solution: Convicting a server requires two steps: The 1st one is calling the `convict` function with the blocknumber of the wrongly signed block and `keccak256(_blockhash, sender, v, r, s)`. Both the real blockhash and the provided hash will be stored in the smart contract. In a 2nd step the function `revealConvict` function has to be called. The missing information are revealed there, but only the previous sender is able to reproduce the provided hash of the 1st transaction, thus being able to convict a server.

7.2 Network-Attacks

7.2.1 Blacklist Attack

Status: open

If the client has no direct internet connection and must rely on a proxy or the phone to do the requests, this would give the intermedier 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 he may prepare a wrong answer, but he cannot fake the signatures of the blockhash. But instead of sending back any signed hashes, he may return no signatures, which indicates to the client, that those were not willing to sign them. Then the client will 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 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, he will not convict himself and so prepare a proof for a wrong response.

Solutions:

> First we may consider signing the response of the Data provider-Node, even if this signature can not be used to convict, but then the client knows that this response came from the client he requested and als was checked by him. This would reduce the chances of this attack, since this would mean, the client picked 2 random Servers, that both are acting malicious together. If the client blacklisted more than 50% of the nodes, he should stop. The only issues here is, that the client does not know, whether this is a 'Inactive Server Spam Attack' or not. In case of the a '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) which needs to sign in the case the response is no signature at all. No signature at all should default to 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 he forwards the signed reponse, he should become trusted again.

7.2.2 DDOS-Attacks

Since the URLS of the Network are known, they may be targets for DDOS-Attacks.

Solution:

> Each node is reponsible for protecting with services like Cloudflare. Also the nodes should have a upper limit of concurrent requests it 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

7.2.3 None Verifying Data Provider

A Data Provider should always check the signatures of blockhash he received from the signers and of course he incentivised to do so, because then he can get their deposit, but after getting the deposit he is not incentivised to report this to the client. There are 2 szenariose :

1. The Data Provider is getting the signature, but not checking it. In this case at least the verification inside the client will fail since the provided blockheader does not match.
2. The Data Provider works together with Signer. In this case he would prepare a wrong blockheader that fits to the wrong blockhash and would pass the verification inside the client.

Solutions:

> SIM: In this case only a higher number of signatures could increase security.

7.3 Privacy

7.3.1 Private Keys as API-Keys

For the scoring-model we are using private keys. The perfect security-model would be registering each client, which is almost impossible on mainnet, if you have a lot of devices. So Using shared keys will very likely to happen, but this a nightmare for security experts.

Solution:

1. limit the power of such a key, so the worst thing that can happen, is a leaked key could be used by other client which will then be able to use the score of the server the key is assigned to.
2. keep the private key secretly and manage the connection to the server only offchain.

7.3.2 Filtering of Nodes

All nodes are known with its URL in the ServerRegistry-contract. For countries trying to filter blockchain-requests this makes it easy to add these URLs on blacklists of firewalls, which would stop the incubed network.

Solution:

> Support Onion-URLs, dynamic IPs, LORA, BLE or other protocols.

7.3.3 Inspecting Data in Relays or Proxies

For Devices, like BLE a Relay like a phone is used to connect to the internet. Since relay is able to read the content it is possible to read the data or even pretend the server not responding. (See Blacklist-Attack)

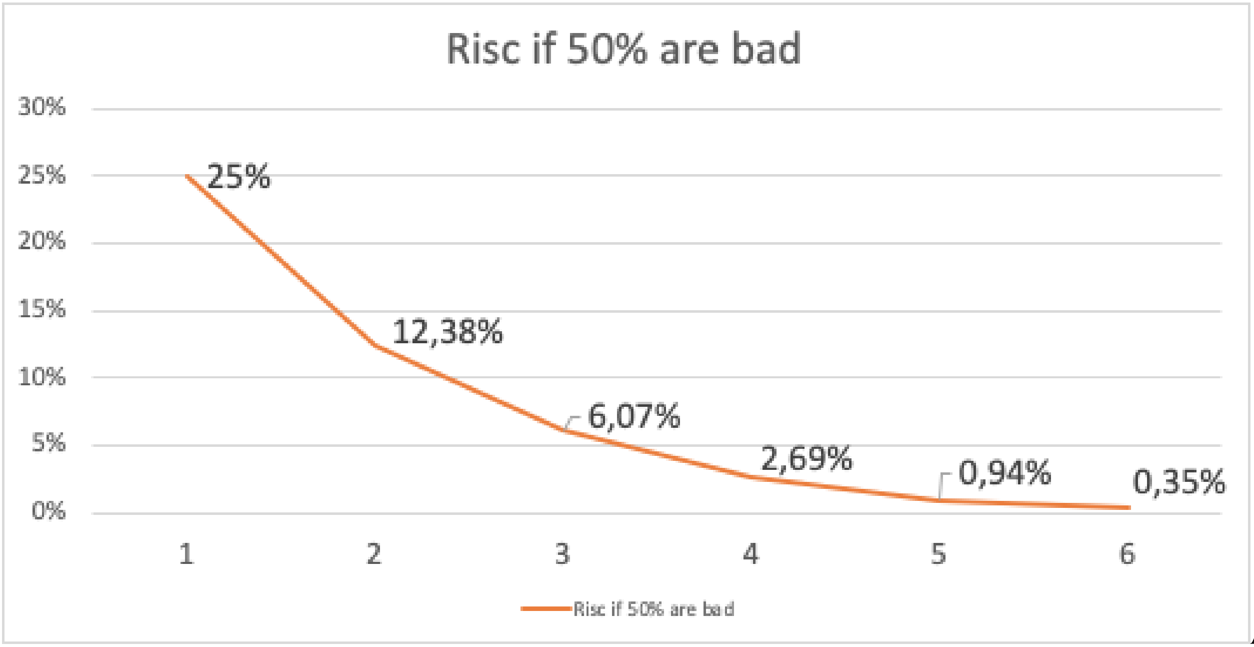
Solution:

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

7.4 Risc Calculation

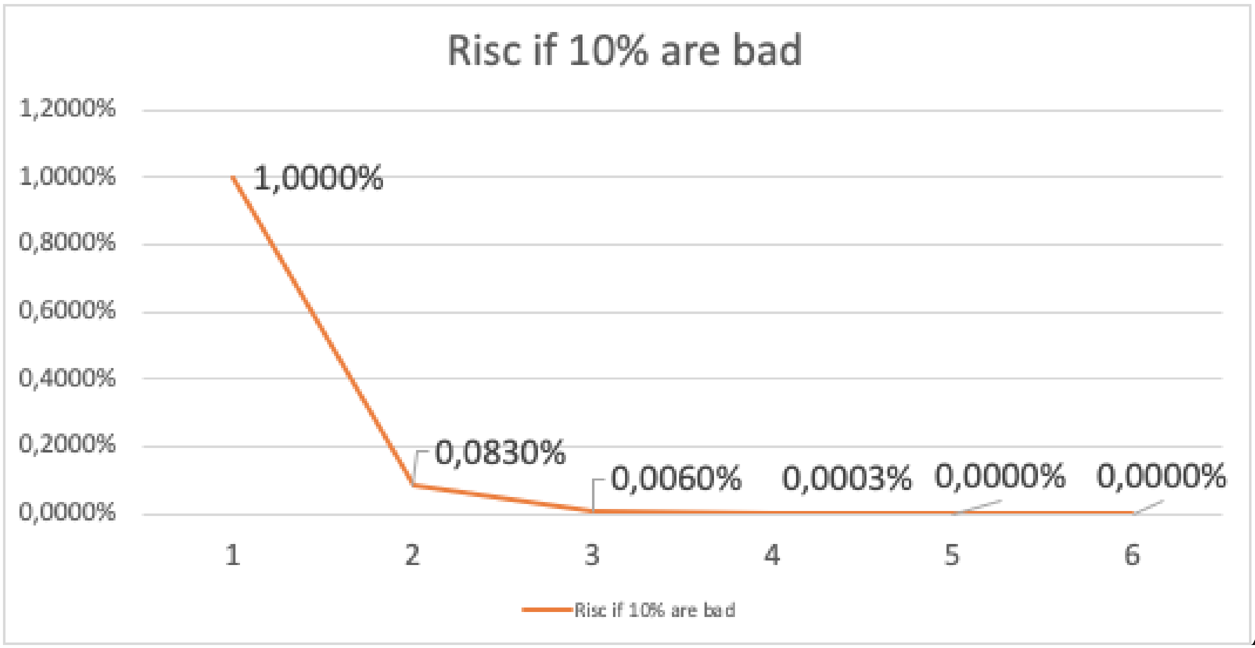
Just like the light client there is no 100% Security to protect from malicious Servers. The only way to reach this, would be to trust special authority nodes for signing the blockhash. For all other nodes we must always assume they are trying to find ways to cheat. The risc of losing the deposit is significantly lower, if the DataProvider Node and the Signing Nodes are run by the same Attacker. In this case he will not only skip checks, but also prepare the data and the proof and also a blockhash that matches the blockheader. If this is the only request and the client would have no other anchor, he would accept a malicious response.

Depending on how many malicious Nodes have registered themselves and are working together, the risc can be calculated. If 10% of all registered Nodes would be run by a Attacker (with the same deposit as the rest) , the risk of getting a malicious Response would be: 1% with only 1 signature and go down to 0,006% with 3 Signatures:



Alt text

In case of a 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 takes more than 4 Signatures to reduce this under 1%.



Alt text

Solution:

> The risk can be reduced by sending 2 requests in parallel. This way the attacker can not be sure that his 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 server as possible, where these server then can check the blockhash and may convict the malicious server.

CHAPTER 8

Roadmap

Incubed implements 2 Versions.

- **Typescript / Javascript** , which is optimized for dapps, webapps or mobile apps.
- **embedded C** optimized for microcontrollers and all other use cases.

8.1 V1.2 Stable - Q3 2019

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 no payment or incentivisation included yet.

- **Failsafe Connection** - The Incubed client will connect to any ethereum-blockchain (providing in3-servers) by randomly selecting nodes within the Incubed-network and automatically retry with different nodes, if the node cannot be reached or delivers verifiable responses.
- **Reputation Management** - Nodes which are not available will be automatically temporarily blacklisted and loose repuatation. 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 onchain and will trigger events if the nodelist changes. Each request will always return the blocknumber of the last event, so the client knows when to update its nodelist.
- **Partial Nodelist** - In order to support small devices, the nodelist can be limited and still be fully verfied 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 JSPN-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 in order to confirm the result.

- **IPFS-Support** - Incubed is able to write and read IPFS-content and verify the data by hashing and creating the the multihash.
- **Caching Support** - an optional cache allows to store the result of rpc-requests and automatically use it again within a configurable time span or if the client if offline. This also includes RPC-Requests, blocks, code and nodelists)
- **Custom Configuration** - The client is highly customizable. For each single request a configuration can be explicitly passed or by adjusting it through events (`client.on('beforeRequest',...)`). This allows to optimize proof-level or number of requests to be sent depending on the context.
- **Proof-Levels** Incubed supports different proof-levels: `none` - for no verifiaction, `standard` - for verifying only relevant properties and `full` - for complete vertification including uncle blocks or previous Transaction (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** - For PoA-Chains (using Aura) 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 already support different transport formats (json/cbor/in3)
- **Replace Latest-Blocks** - Since most applications per default always ask request for the latest block, which cannot be considered as final in a PoW-Chain, a configuration allows to automatically use a certain blockheight to run the request. (like 6 blocks)
- **Light Ethereum API** - Incubed comes with a typesafe simple 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 argument.
- **TypeScript Support** - as Incubed is written 100% in typescript, you get all the advantages of a typesafe tollchain.
- **Integrations** - Incubed has been succesfully tested in all major browsers, nodejs and even react-native.

8.2 V1.2 Incentivisation - Q3 2019

This release will introduce the Incentivisation-Layer, which should help provide more nodes to create the decentralized Network.

- **PoA Clique** - Support Clique PoA to verify Blockheader.
- **Signed Requests** - Incubed supports the Incentivisation-Layer which requires signed requests in order to assign client requests to certain nodes.
- **Network-Balancing** - Nodes will balance the Network based on Load and Reputation.

8.3 V1.3 eWasm - Q1 2020

For `eth_call`-Verification the client and server must be able to execute the code. This release adds the ability to also

- **eWasm** - Support eWasm Contracts in eth_casll.

8.4 V1.4 Substrate - Q3 2020

Supporting Polkadot or any Substrate-based chains.

- **Substrate** - Framework support
- **Runtime-Optimization** - Using pre-compiled Runtimes.

8.5 V1.5 Services - Q1 2021

Generic Interface for any deterministic Service (as docker-container) to be decentralized and verified.

CHAPTER 9

IN3-Specification

This document describes the communication between a incubed client and a incubed node. This communication is based on requests which use extended JSON-RPC-Format. Especially for ethereum-based requests this means each node also accepts all standard requests as at https://github.com/ethereum/wiki/wiki/JSON-RPC, which also includes handling Bulk-requests.

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

9.1 Incubed Requests

Requests without a `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 following:

- **chainId** `string<hex>` - the requested *chainId*. This property is optinal, 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 a undefined behavior since the client can not 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 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 allows to client to skip requiring signed blockhashes for blocks already verified.
- **latestBlock** `integer` - if specified, the blocknumber `latest` will be replaced by blockNumber- specified value. This allows the incubed client to define finality for PoW-Chains, which is important, since the `latest`-block can not 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 refered if occuring 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 temp 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 finaliy. 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 forMust 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 blockheader, but without any signed blockhashes
  - `'proofWithSignature'` : The returned proof will also includ 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"]
  }
}
```

9.2 Incubed Responses

Each incubed node responses is based on JSON-RPC, but also adds then `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 he should update the nodeList.
- **lastValidatorChange** `number` - only for PoA-chains. the blocknumber of the last change of the validatorList. If the client has a smaller number he 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":
↪"0xa9059cbb000000000000000000000000290648fc6f2cb27a2a81dc35a429090872991b920000000000000000000000000
↪",
    "nonce": "0xa8",
    "publicKey":
↪"0x6b30c392dda89d58866bf2c1bedf8229d12c6ae3589d82d0f52ae588838a475aacda64775b7a1b376935d732bb80226
↪",
    "r": "0x4666976b528fc7802edd9330b935c7d48fce0144ce97ade8236da29878c1aa96",
    "raw":
↪"0xf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb000000000000
↪",
    "s": "0x5089dca7ecf7b061bec3cca7726aab1fcb4c8beb51517886f91c9b0ca710b09d",
    "standardV": "0x0",
    "to": "0xd3ebdaea9aeac98de723f640bce4aa07e2e44192",
    "transactionIndex": "0x3e",
    "v": "0x25",
    "value": "0x0"
  },
  "id": 2,
  "in3": {
    "proof": {
      "type": "transactionProof",
      "block":
↪"0xf90219a03d050deecd980b16cad9752133333ccdface463cc69e784f32dd981e2e751e34a01dcc4de8dec75d7aab85b
↪",
      "merkleProof": [

↪"0xf90131a00150ff50e29f3df34b89870f183c85a82a73f21722d7e6c787e663159f165010a0b8c56f207a223067c7ae5
↪",

↪"0xf90211a0f4a5e4a1197190f910e4a026f50bd6a169716b52be42c99ddb043ad9b4da6117a09ad1def70dd1d991331d0
↪",

↪"0xf8b020b8adf8ab81a88504a817c800830186a094d3ebdaea9aeac98de723f640bce4aa07e2e4419280b844a9059cbb0
↪"
      ],
      "txIndex": 62,
      "signatures": [
        {
          "blockHash":
↪"0x2dbbac3abe47a1d0a7843d378fe3b8701ca7892f530fd1d2b13a46b202af4297",
          "block": 7994038,
          "r": "0xef73a527ae8d38b595437e6436bd4fa037d50550bf3840ad0cd3c6ca641a951e",
          "s": "0x6a5815db16c12b890347d42c014d19b60e1605d2e8e64b729f89e662f9ce706b",
          "v": 27,
          "msgHash":
↪"0xa8fc6e2564e496efc5fd7db8e70f03fd50af53e092f47c98329c84c96026fdff"
        }
      ]
    },
```

(continues on next page)

(continued from previous page)

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

9.3 ChainId

Incubed support multiple chains and a client may even run request 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 then definition of a chain in the context of incubed is simply a distributed data domain offering verifieable api-functions implemented in a 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 hold in the client as they specify the rules how responses may be validated.

9.4 Registry

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

This contract serves different purposes. Primary it serves to manage all the incubed nodes, i.e. it manages both the onboarding and also unregistering process. In order to do so, it also has to 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 servers leaves the in3-netwerk.

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

9.4.1 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 withitn the nodeList.
- **timeout** `uint64` - timeout after which the owner is allowed to receive his stored deposit. This information is also important for the client, since a invalid blockhash-signature can only convicted as long as the server is registered. A long timout may give a 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 signes 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 server pure Ethereum JSON/RPC, thus also simple remote nodes may 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 server requests on standardn 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 a unregister. For the client nodes with a `unregisterTime` set have a less trust, since he 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 withtin the incentivication layer to calculate the score.

The following functions are offered within the registry:

9.4.2 NodeRegistry functions

//TODO add interface for new contract.

9.4.3 BlockHashRegistry functions

9.5 Binary Format

Since Incubed is optimized for embedded devices, server may not only support JSON, but a special binary-format. This binary-format is highly optimized for small devices and will reduce the payload to about 30%. This is achieved with 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%.
- 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 easy 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;
}
```

The binary format is based on JSON-structure, but uses a RLP-encoding aproach. Each node or value is represented by a these 4 values:

- **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.
- **key** `uint16_t` - the key hash of the property. This value will only passed, if the structure is a property of a JSON-Object.
- **value** `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 ther size of the data. so the last 5 bit of the first bytes are interpreted as following:

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

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

Depending on these type 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 copying memory in RAM.
- `0x2` : **array** - represents a 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 comming next. In this case the properties following this element will have a `key` specified.
- `0x4` : **boolean** - boolean value where len must be either `0x1`= `true` or `0x0` = `false`.
- `0x5` : **integer** - a 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.

9.6 Communication

9.7 Proofs

9.8 RPC-Methods Ethereum

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

9.8.1 web3_clientVersion

Returns the underlying client version.

See web3_clientversion for spec. No Proof or verifiaction possible.

9.8.2 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.

9.8.3 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.

9.8.4 eth_blockNumber

Returns the number of 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 other. With the `blockTime` from the chainspec including a tolerance the cuurrent blocknumber may be checked if in the proposed range.

9.8.5 eth_getBalance

Returns the balance of the account of given address.

See eth_getBalance for spec.

A AccountProof, since there is none, but the client should verify the result by comparing it to the current blocks returned from other. With the `blockTime` from the chainspec including a tolerance the cuurrent blocknumber may be checked if in the proposed range.

9.9 PoA Validations

CHAPTER 10

API Reference TS

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

- ***AccountProof*** : `interface` - the Proof-for a single Account
- ***AuraValidatoryProof*** : `interface` - a Object holding proofs for validator logs. The key is the blockNumber as hex
- ***BlockData*** : `interface` - Block as returned by eth_getBlockByNumber
- ***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
- ***LogData*** : `interface` - LogData as part of the TransactionReceipt
- ***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
- ***ReceiptData*** : `interface` - TransactionReceipt as returned by eth_getTransactionReceipt
- ***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.
- ***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.
- **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`
- **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`[]
- ***eth*** : `class`
- **header**
  - ***AuthSpec*** :`interface` - Authority specification for proof of authority chains
  - **checkBlockSignatures**(blockHeaders :`string`|*Buffer*|*Block*|*BlockData*[], getChainSpec :) :`Promise<number>` - verify a Blockheader and returns the percentage of finality
  - **getChainSpec**(b :*Block*, ctx :*ChainContext*) :*Promise<AuthSpec>*
  - **getSigner**(data :*Block*) :*Buffer*
- **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*) :*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
- **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.

- **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**
  - **checkForError**(res :*`T`*) :*`T`* - check a RPC-Response for errors and rejects the promise if found
  - **getAddress**(pk :`string`) :`string` - returns a address from a private key
  - **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`
- **validate**(ob :`any`, def :`any`) :`void`

10.1 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

10.2 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

10.3 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)*

10.4 Type ChainSpec

describes the chainspecific consensus params

Source: types/types.ts

- **engine** :`string` *(optional)* - the engine type (like Ethhash, authorityRound, . . .)
- **validatorContract** :`string` *(optional)* - the aura contract to get the validators
- **validatorList** :`any`[] *(optional)* - the list of validators

10.5 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`*
- **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.

10.6 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.

10.7 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
- **timeout** :`number` *(optional)* - the time (in seconds) until an owner is able to receive his deposit back after he unregisters himself example: 3600
- **url** :`string` - the endpoint to post to example: https://in3.slock.it

10.8 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

10.9 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

10.10 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)

10.11 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.

10.12 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

10.13 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`

10.14 Type LogProof

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

Source: types/types.ts

10.15 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:

10.16 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

10.17 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

10.18 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)*

10.19 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
- **totalServers** :`number` *(optional)* - number of servers

10.20 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: 0x72804cfa0179d648ccbe6a65b01a6463a8f1ebb14f3aff6b19cb91acf
- **s** :`string` - Positive non-zero Integer signature.s example: 0x6d17b34aeaf95fee98c0437b4ac839d8a2ece1b18166da704b86d8f4
- **v** :`number` - Calculated curve point, or identity element O. example: 28

10.21 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`

10.22 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)

10.23 Type AxiosTransport

Default Transport impl sending http-requests.

Source: util/transport.ts

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

10.24 Type API

Source: modules/eth/api.ts

- `constructor` **constructor**(client :*Client*) :*API*
- **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.

10.25 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

10.26 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

10.27 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`* - describes the chainspecific consensus params
- **client** :*`Client`* - Client for N3.
- **genericCache**
- **lastValidatorChange** :`number`

- **module** :*Module*
- **clearCache**(prefix :`string`) :`void`
- **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`

10.28 Type AccountData

Account-Object

Source: modules/eth/serialize.ts

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

10.29 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*

10.30 Type Receipt

Buffer[] of the Receipt

Source: modules/eth/serialize.ts

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

10.31 Type Account

Buffer[] of the Account

Source: modules/eth/serialize.ts

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

10.32 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)

10.33 Type BlockType

Source: modules/eth/api.ts

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

10.34 Type Address

Source: modules/eth/api.ts

- **Hex** :`string`

10.35 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'`

10.36 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`*

10.37 Type BN

Source: util/util.ts

10.38 Type Hash

Source: modules/eth/api.ts

- **Hex** :`string`

10.39 Type Quantity

Source: modules/eth/api.ts

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

10.40 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.

10.41 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`*

10.42 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`*

10.43 Type Data

Source: modules/eth/api.ts

- **Hex** :`string`

10.44 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`*

10.45 Type Hex

Source: modules/eth/api.ts

- **Hex** :`string`

10.46 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.

10.47 Type ABIField

Source: modules/eth/api.ts

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

CHAPTER 11

API Reference C

11.1 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 which should be combined. This allowes us to only generate the code needed and so reduce the requirements for flash and memory.

This is why the incubed is combind of different modules. While the core-module is always required additional functions will be prepared by differen modules:

11.1.1 Verifier

Incubed is a minimal verifaction client, which means, each response needs to be verifable. Depending on the expected requests and responses you need to carefully choose which verifier you may need to register. For ethereum we have developed 3 Modules:

- *nano* : a Minimal module only able to verify transaction receipts (`eth_getTransactionReceipt`)
- *basic* : module able to verify almost all other standard rpc-function. (except `eth_call`)
- *full* : module able to verify standard rpc-function. It also implements a full EVM in order 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()*.

11.1.2 Transport

In order to verify responses, you need to able to send requests. The way to handle them depend 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 a 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 execxute them. That's why the core client only defines a function pointer *in3_transport_send* which must handle the requests.

At the moment we offer these modules, other implementation by supported inside different hardware-modules.

- *curl* : module with a dependency to curl which executes these requests with curl, also supporting HTTPS. This modules is supposed to run an standard os with curl installed.

11.1.3 API

While incubed operates on JSON-RPC-Level, as a developer you might want to use a better structed API preparing these requests for you. These APIs are optional but make life easier:

- ***eth*** : This module offers all standard RPC-Functions as descriped in the Ethereum JSON-RPC Specification. In addition it allows you to sign and encode/decode calls and transactions.
- ***usn*** : This module offers basic USN-function like renting or event-handling and message-verifaction.

11.2 Module core

main incubed module defining the interfaces for transport, verifier and storage.

This module does not have any dependencies and cannot be used without additional modules providing verification and transport.

11.2.1 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_init

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

inits the client.

This is done by checking the cache and updating the local storage. This function should be called after creating a new incubed instance.

example

```
// 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;

```

(continues on next page)

(continued from previous page)

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

// init cache
in3_cache_init(client);

// ready to use ...
```

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_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`)

11.2.2 client.h

incubed main client file.

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

Location: src/core/client/client.h

IN3_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_DEBUG

flag used in the EVM (or the `evm_flags`) to turn on debug output.

```
#define IN3_DEBUG 65536
```

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
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
`uint32_t`	**evm_flags**	flags for the evm (EIPs)
*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_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.

11.2.3 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`

11.2.4 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)

11.2.5 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)

11.2.6 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)

11.2.7 bytes.h

util helper on byte arrays.

Location: src/core/util/bytes.h

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_new

```
bytes_builder_t* bb_new();
```

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**

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**

11.2.8 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

```
#define DATA_DEPTH_MAX 11
```

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_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`

11.2.9 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,. . .)

11.2.10 error.h

defines the return-values of a function call.

Location: src/core/util/error.h

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.

11.2.11 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**

11.2.12 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, char *chars, int start, int len);
```

arguments:

*sb_t **	**sb**
`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 **

11.2.13 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))
```

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`

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`

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`

11.3 Module eth_api

static lib

11.3.1 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/eth_api/eth_api.h

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

a 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_getStorageAt

```
uint256_t eth_getStorageAt(in3_t *in3, address_t account, bytes32_t key, uint64_t␣
↪block);
```

returns the storage value of a given address.

arguments:

in3_t *	**in3**
address_t	**account**
bytes32_t	**key**
uint64_t	**block**

returns: *uint256_t*

eth_getCode

```
bytes_t eth_getCode(in3_t *in3, address_t account, uint64_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**
uint64_t	**block**

returns: *bytes_t*

eth_getBalance

```
uint256_t eth_getBalance(in3_t *in3, address_t account, uint64_t block);
```

returns the balance of the account of given address.

arguments:

in3_t *	**in3**
address_t	**account**
uint64_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

arguments:

*in3_t **	**in3**

returns: `uint64_t`

eth_getBlockByNumber

```
eth_block_t* eth_getBlockByNumber(in3_t *in3, uint64_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 ,! otherwise make sure to free the result after using it!

arguments:

*in3_t **	**in3**
`uint64_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 ,! 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 ,! 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_call_fn

```
json_ctx_t* eth_call_fn(in3_t *in3, address_t contract, char *fn_sig,...);
```

returns the result of a function_call.

If result is null, check ,! otherwise make sure to free the result after using it with ,!

arguments:

in3_t *	**in3**
address_t	**contract**
`char *`	**fn_sig**
`...`	

returns: `json_ctx_t *`

eth_wait_for_receipt

```
char* eth_wait_for_receipt(in3_t *in3, bytes32_t tx_hash);
```

arguments:

in3_t *	**in3**
bytes32_t	**tx_hash**

returns: `char *`

eth_newFilter

```
in3_ret_t eth_newFilter(in3_t *in3, json_ctx_t *options);
```

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 filer)

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_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 , in a long double.

Important: since a long double stores max 16 byte, there is no garantee to have the full precision.

converts a , in a long double.

arguments:

uint256_t	**d**

returns: `long double`

as_long

```
uint64_t as_long(uint256_t d);
```

converts a , in a long .

Important: since a long double stores 8 byte, this will only use the last 8 byte of the value.

converts a , in a long .

arguments:

uint256_t	**d**

returns: `uint64_t`

to_uint256

```
uint256_t to_uint256(uint64_t value);
```

arguments:

uint64_t	**value**

returns: *uint256_t*

decrypt_key

```
in3_ret_t decrypt_key(d_token_t *key_data, char *password, bytes32_t dst);
```

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);
```

arguments:

*eth_log_t **	**log**

11.4 Module eth_basic

static lib

11.4.1 eth_basic.h

Ethereum Nanon verification.

Location: src/eth_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)

11.4.2 signer.h

Ethereum Nanon verification.

Location: src/eth_basic/signer.h

eth_sign

```
in3_ret_t eth_sign(void *pk, d_signature_type_t type, bytes_t message, bytes_t␣
↪account, uint8_t *dst);
```

signs the given data

arguments:

`void *`	**pk**
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)

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)

sign_tx

```
bytes_t sign_tx(d_token_t *tx, in3_ctx_t *ctx);
```

arguments:

*d_token_t **	**tx**
*in3_ctx_t **	**ctx**

returns: `bytes_t`

11.4.3 trie.h

Patricia Merkle Tree Imnpl.

Location: src/eth_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**

11.5 Module eth_full

tommath/bn_error.c

11.5.1 big.h

Ethereum Nanon verification.

Location: src/eth_full/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`

11.5.2 code.h

code cache.

Location: src/eth_full/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 *`*

11.5.3 eth_full.h

Ethereum Nanon verification.

Location: src/eth_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.

11.5.4 evm.h

main evm-file.

Location: src/eth_full/evm.h

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_DEBUG

```
#define EVM_PROP_DEBUG 65536
```

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
```

EVM_CALL_MODE_STATIC

```
#define EVM_CALL_MODE_STATIC 1
```

EVM_CALL_MODE_DELEGATE

```
#define EVM_CALL_MODE_DELEGATE 2
```

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

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);
```

arguments:

*evm_t **	**evm**

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(in3_vctx_t *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:

*in3_vctx_t **	**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]);
```

arguments:

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

11.5.5 gas.h

evm gas defines.

Location: src/eth_full/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
```

11.6 Module eth_nano

static lib

11.6.1 eth_nano.h

Ethereum Nanon verification.

Location: src/eth_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 *`

11.6.2 merkle.h

Merkle Proof Verification.

Location: src/eth_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**

11.6.3 rlp.h

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

This decoding works without allocating new memory.

Location: src/eth_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**

11.6.4 serialize.h

serialization of ETH-Objects.

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

Location: src/eth_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.

11.7 Module libin3

add the executablex

11.7.1 in3.h

the entry-points for the shares library.

Location: src/libin3/in3.h

in3_create

```
in3_t* in3_create();
```

creates a new client

returns: *in3_t **

in3_send

```
int in3_send(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**
`char *`	**method**
`char *`	**params**
`char **`	**result**
`char **`	**error**

returns: `int`

in3_dispose

```
void in3_dispose(in3_t *a);
```

frees the references of the client

arguments:

*in3_t **	**a**

11.8 Module transport_curl

add a option

11.8.1 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);
```

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`)

11.8.2 in3_storage.h

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

Location: src/transport_curl/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**

11.9 Module usn_api

static lib

11.9.1 usn_api.h

USN API.

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

Location: src/usn_api/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`)

Index