hyperchain Documentation

Hyperchain Corp.

Aug 31, 2021

Get Started:

CHAPTER 1

Pre-requisites

1.1 OS Recommendations

The charts below show how Hyperchain's requirements map onto various platforms.

Platforms

Distro	Release	Arch
RHEL	6 or later	amd64, 386
CentOS	6 or later	amd64, 386
SLES	11SP3 or later	amd64, 386
Ubuntu	14.04 or later	amd64, 386
macOS	10.8 or later	amd64, 386

1.2 Install Go

Hyperchain uses the Go programming language for its components, thus we need to install Go for developing.

1.2.1 Download Go

Go provides binary distributions for Mac OS X, Linux, and Windows. If you are using a different OS, you can download the Go source code and install from source.

Download the latest version of Go for your platform here: Downloads - version `1.7.x or above`

1.2.2 Install Go

Follow the instructions for your platform to install the Go tools: Install the Go tools. It is recommended to use the default installation settings.

- On Mac OS X and Linux, by default Go is installed to directory `/usr/local/go/`, and the `GOROOT` environment variable is set to `/usr/local/go`.

```
export GOROOT=/usr/local/go
```

- Also set the `GOROOT/bin` variable, which is used to run Go command.

```
export PATH=$PATH:$GOROOT/bin
```

1.2.3 Set GOPATH

Your Go working directory (`GOPATH`) is where you store your Go code. It can be any path you choose but must be separate from your Go installation directory (`GOROOT`).

The following instructions describe how to set your `GOPATH`. Refer to the official Go documentation for more details: https://golang.org/doc/code.html.

- On Mac OS X and Linux Set the `GOPATH` environment variable for your workspace:

```
export GOPATH=$HOME/go
```

- Also set the `GOPATH/bin` variable, which is used to run compiled Go programs.

```
export PATH=$PATH:$GOPATH/bin
```

- Since we'll be doing a bunch of coding in Go, you might want to add the following to your`~/.bashrc`:

```
export GOROOT=/usr/local/go
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin:$GOROOT/bin
```

1.2.4 Test Go Installation

Create and run the hello.go application described here: https://golang.org/doc/install#testing.

If you set up your Go environment properly, you should be able to run "hello" from any directory and see the program execute successfully.

1.3 Install Go vendor tool

Go vendor is a tool for managing Go packages and their dependencies. This tool will copy the dependent packages to the project's `vendor` directory, and record their versions in a file named `vendor.json`.

1.3.1 Installation

```
go get -u github.com/kardianos/govendor
```

1.3.2 Test Go vendor Installation

To verify you setup govendor properly, please make sure govendor version information displays correctly.

At the command prompt, type the following command and make sure you see govendor version information:

```
$ govendor --version
v1.0.9
```

1.3.3 More Details

You can goto the project's home page for more details. - Go vendor

1.4 Install Contract Compiler(Optional)

Hyperchain supports Smart Contract which written in Solidity and then compiled into bytecode to be uploaded on the blockchain.

Given that we are writing in Solidity, we need to be sure that we have installed contract compiler named `solc` for compiling.

We've provided some general installers for some platforms in our source code, you can use them to install `solc` quickly, you can also refer to the official site - Installing Solidity for installation.

CHAPTER 2

Quick Start

If you haven't completed your *Pre-requisites* Checklist, do that first. This Quick Start tells you how to build Hyperchain from source code, and how to start up a Hyperchain cluster.

2.1 Building from Source

2.1.1 Create Your Clone

Clone the repository to a directory of your `GOPATH` source path:

```
mkdir -p $GOPATH/src/github.com/hyperchain
cd $GOPATH/src/github.com/hyperchain
git clone https://github.com/hyperchain/hyperchain
```

2.1.2 Building

Please make sure you've installed Go tool properly, if you don't have it already, please see *Pre-requisites*

To build Hyperchain:

```
cd $GOPATH/src/github.com/hyperchain/hyperchain
govendor build
```

You can run `go build` as well.

2.2 Start up Hyperchain

Since a Hyperchain cluster needs at least 4 nodes to establish a BFT system, we recommend starting up Hyperchain nodes in these modes: - Local Mode - Local 4 Nodes - Distributed Mode - Distributed 4 Nodes

2.2.1 Local Mode - Local 4 Nodes

We've provided a script named `local.sh` which starts all Hyperchain nodes locally.

```
cd $GOPATH/src/github.com/hyperchain/hyperchain/scripts
./local.sh
```

You'll see these information if all Hyperchain nodes start up properly.

```
$./local.sh
...
...
start up node 1 ... done
start up node 2 ... done
start up node 3 ... done
start up node 4 ... done
```

2.2.2 Distributed Mode - Distribute 4 Nodes

Enable Password Less

Since `server.sh` script prompts for a password when executing ssh operations, we recommend generating SSH keys on the deploy node and distribute the public key to each Hyperchain node.

1. Generate the SSH keys, and leave the passphrase empty:

```
ssh-keygen

Generating public/private key pair.
Enter file in which to save the key (/home/hyperchain/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hyperchain/.ssh/id_rsa.
Your public key has been saved in /home/hyperchain/.ssh/id_rsa.pub.
```

2. Copy the key to each Hyperchain node, replacing `{username}` with the user name you created.

```
ssh-copy-id {username}@node1
ssh-copy-id {username}@node2
ssh-copy-id {username}@node3
ssh-copy-id {username}@node4
```

Distribute Hyperchain

We've provided a script named `server.sh` which distributes Hyperchain to all nodes and starts up them separately.

1. Put servers' IP addresses into a file named serverlist.txt which under hyperchain/scripts directory.

For instance:

```
cat $GOPATH/src/github.com/hyperchain/hyperchain/scripts/serverlist.txt
172.16.1.101
172.16.1.102
172.16.1.103
172.16.1.104
```

2. Start up Hyperchain with server.sh script.

```
cd $GOPATH/src/github.com/hyperchain/hyperchain/scripts
./server.sh
```

You'll see these information if all Hyperchain nodes start up properly.

```
$./server.sh
...
...
start up node 1 ... done
start up node 2 ... done
start up node 3 ... done
start up node 4 ... done
```

CHAPTER 3

Hyperchain Samples

In this section, we will introduce some simple examples to use Hyperchain.

3.1 HyperCli

We recommend using `HyperCli` for administration.

`HyperCli` is a CLI tool for Hyperchain administration, it has various functions. We'll introduce its contract and transaction related functions in the following steps.

To build HyperCli:

```
cd $GOPATH/src/github.com/hyperchain/hyperchain/hypercli
govendor build
```

You can run `go build` as well.

Note: By default, HyperCli sends message to localhost:8081, so we recommend you to run HyperCli on your Hyperchain node locally. Otherwise you need to specify HyperCli's `--host` and `--port` parameters with Hyperchain node IP and JSON-RPC port for remote execution.

3.2 Sample Contract 1 - Set/Get Hash

Here is a sample contract which implements setHash and getHash functionality.

```
contract Anchor{
    mapping(bytes32 => bytes32) hashMap;

    function setHash(bytes32 key,bytes32 value) returns(bool,bytes32){
```

(continues on next page)

(continued from previous page)

```
        if(hashMap[key] != 0x0){
            return (false,"the key exist");
        }
        hashMap[key] = value;
        return (true,"Success");
    }

    function getHash(bytes32 key) returns(bool,bytes32,bytes32){
        if(hashMap[key] == 0x0){
            return (false,"the key is not exist",0x0);
        }
        return (true,"Success",hashMap[key]);
    }
}
```

3.2.1 Compiling Contract

You can get contract's bytecode with a simple CLI command if you've installed `solc`. Meanwhile, you can use the following bytecode which is the compiled result of this contract if you've not installed the solidity compiler.

bytecode

```
0x606060405261015c806100126000396000f3606060405260e060020a60003504633cf5040a81146100295780631007
```

Assuming that your contract file named `sample1.sol`, you can get the bytecode with the following command:

```
solc --bin sample1.sol
```

3.2.2 Deploying Contract

HyperCli provides a 'contract deploy' function, here's its parameters:

```
$ ./hypercli contract deploy --help

NAME:
   hypercli contract deploy - Deploy a contract

USAGE:
   hypercli contract deploy [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --directory value, -d value  specify the contract file directory
```

You can specify the contract's bytecode as the value of '–payload option' to deploy this contract, for example:

```
./hypercli contract deploy --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x606060405261015c806100126000396000f3606060405260e060020a60003504633cf5040a81146100295780631007
```

This command means HyperCli deploys the contract from the account address 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd. Though HyperCli has some default values for its parameters, you can also specify them explicitly.

You'll see these information if HyperCli command executed properly.


```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xb1b7d4f083ac65679ddd31a9b864fc8ca1ec75eee2f7a46cca1b223eae94527c","vmType":"EVM",
↪"contractAddress":"0xbbe2b6412ccf633222374de8958f2acc76cda9c9","gasUsed":69660,"ret
↪":
↪"0x606060405260e060020a60003504633cf5040a8114610029578063d7fa10071461007b575b610002565b34610002576
↪","log":[]}}
```


You can get the contract address from the result, in this case, it's:

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

It will be used later to invoke contract's functions.

3.2.3 Invoking Contract

HyperCli also provides a 'contract invoke' function, here's its parameters:

```
$ ./hypercli contract invoke --help

NAME:
   hypercli contract invoke - Invoke a contract

USAGE:
   hypercli contract invoke [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --to value, -t value         specify the contract address
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --args value, -a value       specify the args of invoke contract
```

To invoke a contract function, you need to specify at least 2 parameters in this case:

- payload: payload of the function call
- to: the contract address which you got before

We've provided some function calls' payloads, you can use them directly for invoking.

setHash

Invoke setHash function to set key1 = value1, and here's its payload:


```
0xd7fa10076b65793100000000000000000000000000000000000000000000000000000000076616c75653100000000000000
```

Here's the contract address you got before:

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0xd7fa10076b65793100000000000000000000000000000000000000000000000000000000000076616c7565310000000000000
↪--to 0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xa28350777a964f5ab6f4ef355131c0c241388ac6e8548c191aa5b3b94af95571","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":20477,"ret
↪":
↪"0x000000000000000000000000000000000000000000000000000000000000001537563636573730000000000000000000
↪","log":[]}}
```

getHash

Invoke getHash function to get key1, and here's its payload:

```
0x3cf5040a6b65793100000000000000000000000000000000000000000000000000000000000000000
```

Here's the contract address you got before:

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x3cf5040a6b65793100000000000000000000000000000000000000000000000000000000000000000 --to␣
↪0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0x185dba5451ace5ffcf4c11d10968e1e4ed299eb78ca6ddda65539dfca2fc56df","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":523,"ret":
↪"0x000000000000000000000000000000000000000000000000000000000000001537563636573730000000000000000000
↪","log":[]}}
```

3.3 Sample Contract 2 - Simulate Bank

Here is another sample contract which implements asset administration.

```
contract SimulateBank{
    address owner;
    bytes32 bankName;
    uint bankNum;
    bool isInvalid;
```

(continues on next page)

(continued from previous page)

```
    mapping(address => uint) public accounts;
    function SimulateBank( bytes32 _bankName,uint _bankNum,bool _isInvalid){
        bankName = _bankName;
        bankNum = _bankNum;
        isInvalid = _isInvalid;
        owner = msg.sender;
    }
    function issue(address addr,uint number) returns (bool){
        if(msg.sender==owner){
            accounts[addr] = accounts[addr] + number;
            return true;
        }
        return false;
    }
    function transfer(address addr1,address addr2,uint amount) returns (bool){
        if(accounts[addr1] >= amount){
            accounts[addr1] = accounts[addr1] - amount;
            accounts[addr2] = accounts[addr2] + amount;
            return true;
        }
        return false;
    }
    function getAccountBalance(address addr) returns(uint){
        return accounts[addr];
    }
}
```

3.3.1 Compiling Contract

You can get contract's bytecode with a simple CLI command if you've installed `solc`. Meanwhile, you can use the following bytecode which is the compiled result of this contract if you've not installed the solidity compiler.

bytecode

```
0x6060604052604051606080610200083395060c0604052516080516a0516001839055600282905560038054f0100000000
```

Assuming that your contract file named `sample2.sol`, you can get the bytecode with the following command:

```
solc --bin sample2.sol
```

3.3.2 Deploying Contract

As mentioned, HyperCli provides a 'contract deploy' function, here's its parameters:

```
$ ./hypercli contract deploy --help

NAME:
   hypercli contract deploy - Deploy a contract

USAGE:
   hypercli contract deploy [command options] [arguments...]

OPTIONS:
```

(continues on next page)

(continued from previous page)

```
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --directory value, -d value  specify the contract file directory
```

Thus you can specify the contract's bytecode as the value of '–payload option', for example:

```
./hypercli contract deploy --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x60606040526040516060806102083395060c06040525160805160a05160018390556002829055600380547f0100000
```

This command means HyperCli deploys the contract from the account address 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd. Though HyperCli has some default values for its parameters, you can also specify them explicitly.

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0x6790126ca4c072f53d1684dff9e080098db931358d8eca04c833373ae580ed9e","vmType":"EVM",
↪"contractAddress":"0xbbe2b6412ccf633222374de8958f2acc76cda9c9","gasUsed":109363,"ret
↪":
↪"0x606060405260e060020a60003504635e5c06e2811461003f578063867904b41461005c57806393423e9c146100a8578
↪","log":[]}}
```

You can get the contract address from the result, in this case, it's:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

It will be used later to invoke contract's functions.

3.3.3 Invoking Contract

As mentioned, HyperCli provides a 'contract invoke' function, here's its parameters:

```
$ ./hypercli contract invoke --help

NAME:
   hypercli contract invoke - Invoke a contract

USAGE:
   hypercli contract invoke [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --to value, -t value         specify the contract address
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --args value, -a value       specify the args of invoke contract
```

To invoke a contract function, you need to specify at least 2 parameters in this case:

- payload: payload of the function call
- to: the contract address which you got before

We've provided some function calls' payloads, you can use them directly for invoking.

Issue Asset

Issue asset for user 0x1234567 with 1000000000, and here's its payload:

```
0x867904b4000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000000
```

Here's the contract address you got before:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x867904b4000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000000
↪--to 0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0x9f602fc0c4ac12d383cf765e2931978a0e5daaec666888e242485bd6314e17f9","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":20498,"ret
↪":"0x0000000000000000000000000000000000000000000000000000000000000001","log":[]}}
```

Transfer Asset

Transfer from user 0x1234567 to user 0x2345678 with 1 amount, and here's its payload:

```
0xbeabacc8000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000000
```

Here's the contract address you got before:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0xbeabacc8000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000000
↪--to 0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0x2d779df9db29541102e98ed9996263db42da6c13a2a6ac5c7ba9c606acecba28","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":25733,"ret
↪":"0x0000000000000000000000000000000000000000000000000000000000000001","log":[]}}
```

Get Balance

Get balance of user 0x1234567, and here's its payload:

```
0x93423e9c0000000000000000000000000000000000000000000000000000000001234567
```

Here's the contract address you got before:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x93423e9c0000000000000000000000000000000000000000000000000000000001234567 --to␣
↪0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xcca7b19dde84bf174243398de4107ee0b783b6e243176e162a2239bae1f475f7","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":353,"ret":
↪"0x0000000000000000000000000000000000000000000000003b9ac9ff","log":[]}}
```

Get balance of user 0x2345678, and here's its payload:

```
0x93423e9c0000000000000000000000000000000000000000000000000000000002345678
```

Here's the contract address you got before:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

Now you can run the command as follows:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --payload␣
↪0x93423e9c0000000000000000000000000000000000000000000000000000000002345678 --to␣
↪0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0x04b82a4fcdadcf102d559b4eb6a29030f7ef29195f40a5f9b986021b48b48552","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":353,"ret":
↪"0x0000000000000000000000000000000000000000000000000000000000000001","log":[]}}
```

3.4 Transaction Delivery

HyperCli also provides transaction related functions, and here's its parameters:

```
$ ./hypercli tx --help
NAME:
   hypercli tx - transaction related commands

USAGE:
```

(continues on next page)

(continued from previous page)

```
   hypercli tx command [command options] [arguments...]

COMMANDS:
     send     send normal transactions
     info     query the transaction info by hash
     receipt  query the transaction receipt by hash

OPTIONS:
   --help, -h  show help
```

There are 3 sub-commands for HyperCli tx command, we'll introduce them with some examples as follows.

3.4.1 Send Transaction

The sub-command 'send' is used for sending normal transactions, it has some parameters as follows:

```
$ ./hypercli tx send --help
NAME:
   hypercli tx send - send normal transactions

USAGE:
   hypercli tx send [command options] [arguments...]

OPTIONS:
   --count value, -c value      send how many transactions (default: 1)
   --namespace value, -n value  specify the namespace to send transactions to␣
↪(default: "global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --to value, -t value         specify the contract address
   --password value, -p value   specify the password used to generate signature␣
↪(default: "123")
   --amount value, -a value     specify the amount to transfer (default: 0)
   --extra value, -e value      specify the extra information
   --snapshot value, -s value   specify the snapshot ID
   --simulate                   simulate execute or not
```

For example, you can use this command to send transactions, it means send 10 transactions in a row:

```
./hypercli tx send -c 10
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xefbc9f9b5048337fbaf64047ad5eff03c40c1c76991b0364686ba3620e2c5ea3","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,"ret":
↪"0x0","log":[]}}

...
...

{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,"ret":
↪"0x0","log":[]}}
```

(continues on next page)

(continued from previous page)

Please select one txHash in the results, it will be used later to execute 'info' and 'receipt' sub-commands. In this case, we select this txHash:

```
0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

3.4.2 Transaction Info

The sub-command 'info' is used for getting a transaction's information, it has some parameters as follows:

```
./hypercli tx info --help
NAME:
   hypercli tx info - query the transaction info by hash

USAGE:
   hypercli tx info [command options] [arguments...]

OPTIONS:
   --hash value                 specify the tx hash used to query the detailed␣
↪information
   --namespace value, -n value  specify the namespace to query transaction␣
↪information (default: "global")
```

For example, you can use this command to get the transaction's information with the txHash you got before:

```
./hypercli tx info --hash␣
↪0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","hash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","blockNumber":
↪"0xf","blockHash":
↪"0x2592e0f3e1f156effe93325a60b1190533be2053aa102eb182bd64f95b28080a","txIndex":"0x0
↪","from":"0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd","to":
↪"0x6201cb0448964ac597faf6fdf1f472edf2a22b89","amount":"0x8","timestamp
↪":1512022032747286761,"nonce":5475154203949975728,"extra":"","executeTime":"0x5",
↪"payload":"0x0"}}
```

3.4.3 Transaction Receipt

The sub-command 'receipt' is used for getting a transaction's receipt, it has some parameters as follows:

```
$ ./hypercli tx receipt --help
NAME:
   hypercli tx receipt - query the transaction receipt by hash

USAGE:
   hypercli tx receipt [command options] [arguments...]

OPTIONS:
```

(continues on next page)

(continued from previous page)

```
   --hash value                specify the tx hash used to query the transaction␣
↪receipt
   --namespace value, -n value  specify the namespace to query transaction receipt␣
↪(default: "global")
```

For example, you can use this command to get the transaction's receipt with the txHash you got before:

```
./hypercli tx receipt --hash␣
↪0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

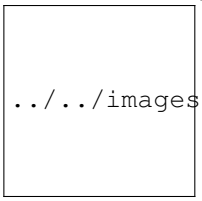
You'll see these information if HyperCli command executed properly.

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":{
↪"version":"1.3","txHash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","vmType":"EVM",
↪"contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,"ret":
↪"0x0","log":[]}}
```

CHAPTER 4

Transaction Flow

This document outlines the transactional mechanism from a transaction initialized by client to final updated into the blockchain ledger. The scenario includes the client that initiating the transaction and the consensus peer A (also called validate peer, VP) directly connected to it. The client interacts with the blockchain ledger by sending transactions through the SDK (Java supported) with the VP-A; the VP-A fully connected with other VP B, C and D... , while VP-A has two backup nodes a1 and a2 (also called non-validate peer, NVP) .



Assumption

This flow assumes that a channel is set up and running. The application user has registered and enrolled with the organization's certificate authority (CA) and received back necessary cryptographic material (SDKCert) , which is used to authenticate to the blockchain network.

The smart contract (including the initial state and all the related functions) has been deployed on the blockchain peers in the working namespace.

Client initiates a transaction

What's happening? - Client is sending a transaction request (calling one of the methods in the smart contract). The request targets VP-A, through which it goes into blockchain network.

The client initializes a *HyperchainAPI* object by calling the interface of the SDK. During initialization, the SDK requests the VP-A with the SDKCert and the public key to obtain the TCert needed for initiating the transaction. After that, client generates a transaction by calling the *Transaction* interface of the SDK. The SDK firstly signs the transaction with the private key specified by the client, then signs the message with the private key corresponding to the TCert after the transaction is encapsulated in the JSONRPC protocol. *HTTP/HTTPS* "short" connection and

../../images/get_tcert.png

WebSocket "long" connection is supported during the SDK and representative peer.

Peer accepts transaction & sends to the blockchain network

VP-A performs TCert authentication as soon as it receives the transaction. The peer only processes the request that passed the validation of TCert. Then the API module of the peer will do the following transaction verification:

(1) The transaction throughput has not exceed the configuration of rate limit;

(2) The transaction proposal is well formed (Verify the legality of the transaction field, like the legitimacy of the timestamp);

(3) It has not been submitted already in the past (replay-attack protection);

(4) The signature of transaction is valid (ECDH & SM2 supported).

After passing all the aforementioned verification, the transaction will be submitted to the consensus module, and the consensus module will broadcast it to all VPs in the entire network.

Consensus of transaction: ordering, validating, writing

The transaction goes through the three-phase protocol the Consensus Algorithm (RBFT):

(1) *Pre-Prepare* The primary peer (master peer among all VPs) orders tansactions chronologically by channel, and creates block of transactions in a certain period of time (or a certain number). And then, primary broadcasts the block to all VPs.

../../images/txs_to_block.png

(2) *Prepare* Replicas (another name of all VPs used in consensus) make confirmation and pre-execute the transactions in the block. Then they broadcast the result hash.

../../images/block_to_ledger.png

(3) *Commit* Replicas write the block, update in the blockchain ledger.

All the illegal transactions found after pre-executing will be recorded into the database of illegal transactions instead of blockchain ledger. The blockchain ledger is a ledger of blocks including all valid transactions.

Furthermore, all VPs will push the block to all the respectively connected NVPs after the block is successfully updated in blockchain ledger.

Transaction Receipt SDK implements timely query in the result of deploying transaction in the *Transaction* interface, that is, transaction receipt. The configuration of the number of transactions and packaging time (used in ordering) set in Blockchain network will affect the latency of transaction.

CHAPTER 5

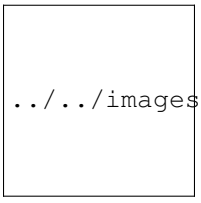
Consensus

5.1 1. Overview

Consensus mechanism is the foundation of blockchain consistency, it ensures that all consensus nodes (or say validating peer, VP) execute transactions in the same order and then write into exactly the same ledgers. Accounting nodes(or say non-validating peer, NVP) which connect to one or more VP(s) can only synchronize ledger information from its connected VP(s), so NVP don't participate in consensus while NVP can forward transactions to VP(s) received from client.

Hyperchain supports the pluggable consensus mechanism and provides different consensus algorithms for different scenarios of the blockchain. Current version has implemented the improved algorithm of PBFT: Robust Byzantine Fault Tolerance (RBFT), the idea of this algorithm comes from many classic papers (especially Aardvark). Hyperchain will continue to support other consensus algorithms such as RAFT later.

After received transcations from clients, the API layer parses out the transactions and forwards to the consensus module. Consensus module receives and stores the transactions into local transaction pool (TxPool). TxPool takes the role of caching transactions and packaging blocks so it is implemented as a sub-module of consensus module. In addition, consensus module needs to maintain a consensus database to store some variables required by the algorithm for autonomous recovery after the system is crashed. For example, the RBFT algorithm needs to maintain consensus information such as View, PrePrepare, Prepare, and Commit.



5.2 2. RBFT related parameters

In a consensus network of N nodes (N> = 4), RBFT can tolerate at most f Byzantine faults which:

$$f = \lfloor \frac{N-1}{3} \rfloor$$

The number of nodes that can guarantee consensus is:

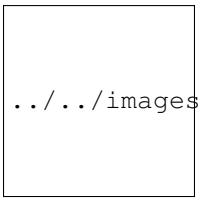
$$quorum = \lceil \frac{N+f+1}{2} \rceil$$

5.3 3. RBFT normal case

The normal case of RBFT ensures that each consensus node in blockchain processes the transactions from the client in the same order. RBFT requires at least 3f + 1 nodes to tolerate f Byzantine falult which is the same as PBFT. The figure below is the consensus flow under the minimum number of cluster nodes, where N = 4 and f = 1. Primary1 is the master node which is dynamically elected by the consensus node, and is responsible for sorting and packing the transactions sent from the client. Replica2, 3 and 4 are backup nodes. All Replica nodes execute the transaction with the same logic and are able to participate in the election of new primary node when the primary node fails.

5.3.1 Process of normal case

The consensus of RBFT retains PBFT's original three-phase submission flow (PrePrepare, Prepare, Commit) and inserts important transaction validation session which not only guarantees the consensus on the transaction execution sequence but also gurantees the consensus on block validation results.



RBFT inserts important transaction validation session into native PBFT normal case operations. Primary will validate block immidiately after packing the transactions, and then include the validation result into the PrePrepare message for the whole network broadcast, so PrePrepare message contains both the ordered Transaction information and block validation result. After receiving a PrePrepare message from primary, the backup nodes check the legitimacy of the message. After passing the legality check, backup node broadcasts Prepare message which indicates that the backup node agrees with primary's sorting result. The backup node starts to validate the batch after receiving (quorum-1) Prepare messages and compares the validation result with the validation result of primary's. If consistent, backup broadcasts Commit message which indicates that the backup node agrees with the validation result of primary's, otherwise, directly triggers ViewChange which indicates that the current node discovers primary's abnormal behavior. RBFT normal case operation is divided into the following steps:

1. **Transaction forward:** Client sen transactions to any node (consensus nodes or accounting nodes) in the blockchain. Accounting nodes need to forward the transactions received from clients to its connected consensus nodes and the consensus nodes broadcast transactions received from clients or accounting nodes to all other consensus nodes, so the transaction pool for all consensus nodes maintains a complete list of transactions;
2. **PrePrepare**: Primary packs transactions according to the policies below: User can customize the batch timeout and the packed batch size according to demand. Primary triggers the package event when it collects more than batchsize transactions during the batch timeout or it dosen't collect batchsize transactions when batch timeout happens. Primary packs the transactions into blocks according to the received chronological order,

then validates and computes the execution result, and finally writes the ordered transaction information together with the validation result into the PrePrepare message to be broadcast to all consensus nodes which starts the three-phase processing flow;

3. **Prepare:** After receiving the PrePrepare message from the primary, backup node first checks the legitimacy of the message(such as current view and block number information). If the check passed, backup nodes broadcast Prepare message to all consensus node;
4. **Commit:** After receiving the (quorum-1) Prepare message and the corresponding PrePrepare message, the backup node validates the batch and compares the validation result with the validation result of primary which is written in the PrePrepare message. If consistent, the backup node broadcasts Commit message which indicates that backup node agrees with the validation result of primary, otherwise, directly triggers the ViewChange event which indicates that the current node discovers primary's abnormal behavior;
5. **Write Block:** All consensus nodes write the execution result to the local ledger after receiving quorum Commit messages.

By adding a validation mechanism in the consensus module, Hyperchain ensures that every backup node participates in checking all primay's ordering results, so backup can discovery primary's Byzantine behavior as soon as possible which improves the stability of the system.

5.3.2 Checkpoint

Consensus nodes need to periodically clean up some useless message caches in order to prevent unlimit message caching during operation. RBFT collectes garbage by introducing checkpoint mechanism in the PBFT algorithm and fixedly set the checkpoint size K to 10. The node reaches a checkpoint after writing into an integer multiple of K and broadcasts the checkpoint information. After receiving the same checkpoint information from other quorum-1 nodes, replica reaches a stable checkpoint, then replica can clean up some of the message cache whose message number is less than checkpoint index.

5.3.3 Transaction pool(txpool)

Transaction pool(txpool) is the transaction cache place of consensus node. The existence of txpool on the one hand limits the client's sending frequence, on the other hand reduces the bandwidth pressure of primary. Firstly, by limiting the size of the transaction pool, consensus node can refuse transactions from clients after the transaction pool reaches its limit size, so users can maximize utilization without abnormalities by setting the transaction cache size for a reasonable assessment of machine performance. Secondly, the consensus node stores the transactions received from the client into its own transaction pool and then broadcasts the transactions to other consensus nodes to ensure that all consensus nodes maintain a complete transaction list. After primary packed transactions, it only needs to put the transaction hash list into the PrePrepare message for broadcasting instead of put the complete transaction list into PrePrepare for broadcasting, which greatly reduces the pressure of the egress bandwidth of primary. If the backup node finds that some transactions are missing before validation, it needs only fetch the missing entries from primary rather than fetching all the transactions in the block.

5.4 4. RBFT ViewChange

The ViewChange mechanism of RBFT solves the problem that the primary node may become a Byzantine node. In the RBFT algorithm, nodes participating in consensus can be divided into Primary node and Replica nodes according to roles. The most important function of the Primary node is to package the received transactions according to a specific strategy, order the transactions, and have all the nodes execute in this order. However, if the Primary node crashes, goes wrong, or is hacked (that is, it becomes a Byzantine node), the Replica nodes need to discover the abnormality of the Primary node in time and elect a new Primary node. This is a problem that all BFT algorithms must solve in order to achieve stability.

5.4.1 view

In RBFT, the concept of view has been introduced as same as PBFT. The view is changed each time a new Primary node is elected. At present, RBFT chooses the Primary node by rotation, and the view increases monotonically from zero. The current view and the total number of nodes N determines the Primary node id:

$$PrimaryId = (view + 1) \bmod N$$

5.4.2 Byzantine behavior that can be detected

Currently, there are mainly two types of Primary's Byzantine behavior that RBFT can detect:

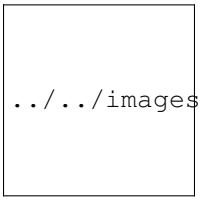
1. The Primary node stops working and sending no message;
2. The Primary node sends wrong messages.

For scenario 1, being detected could be guaranteed by the nullRequest mechanism. A properly behaved Primary node will send nullRequest messages to all Replica nodes periodically to maintain normal connection when no transaction occurs. If the Replica node does not receive a nullRequest messages within the specified time, the ViewChange process is triggered to elect a new Primary node.

For scenario 2, Replica nodes would check the messages sent from the Primary node, such as the verification result contained in the PrePrepare message, which is mentioned in the previous section. The Replica node will directly initiate the ViewChange process to elect a new Primary node if the messages fail to pass the verification.

In addition, RBFT provides a configurable option called ViewChangePeriod. Users can set this option according to their needs. Each time a certain number of blocks are written, the network would take a proactive ViewChange process to rotate the Primary node. This can alleviate the additional pressure on the primary node as a package node. And secondly, all the nodes participating in the consensus can take some packaging work to ensure fairness.

5.4.3 Process of ViewChange



In the above figure, Primary 1 is a Byzantine node and the network need to take ViewChange process. The ViewChange process in RBFT is as follows

1. Replica nodes broadcast a ViewChange message to the entire network after detecting an abnormal behavior of Primary node (without receiving a nullRequest message on time) or after receiving a ViewChange message from other f + 1 nodes, and change their view from v to v + 1;
2. In the new view, after receiving N-f ViewChange messages, Primary node calculates the checkpoint where Primary node would start executing from in the new view and the transactions to be processed next, according to the received ViewChange message, then encapsulates them into the NewView message and broadcasts the message. Finally Primary node initiates the VcReset;
3. After receiving the NewView message, Replica nodes validate the message. If the message passes validation, Replica nodes initiate the VcReset. If it does not pass validation, Replica nodes send ViewChange message to start another round of ViewChange;
4. After finishing VcReset, all nodes broadcast FinishVcReset to the whole network;

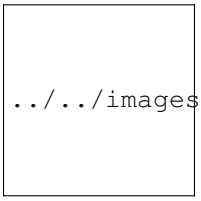
5. After each node receives N-f FinishVcReset messages, it starts to process the transactions after the determined checkpoint and finishes the entire ViewChange process.

Because the communication between the consensus module and the execution module is asynchronous and execution module may have some useless validation cache after ViewChange, the consensus module needs to inform the execution module to clear this useless cache before the end of ViewChange. The RBFT proactively notifies the execution module through the VcReset event to clear the cache. The node can finish ViewChange only after clearing the cache.

5.5 5. RBFT Recovery

During the operation of the blockchain network, the execution speed of some nodes may lag behind that of most nodes due to network jittering, sudden power failure, disk failure and the like. In this scenario, these nodes need to be able to recover automatically to continue participating in subsequent consensus processes. In order to solve this kind of data recovery problem, the RBFT algorithm provides a mechanism for automatic recovery of dynamic data (recovery). Node updates its storage status by actively retrieving information such as the view of all nodes in the existing consensus network and the latest block information, and finally synchronizes to the latest status of the entire system. When the node is going to start up, restart or the node falls behind, it will automatically enter recovery and synchronize to the latest state of the entire system.

5.5.1 Process of Recovery



In the above figure, replica 4 is a backward node and needs to be recovered. This node's automatic recovery process in RBFT is as follows:

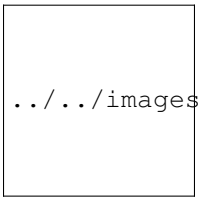
1. At the beginning, replica 4 broadcasts the NegotiateView message to retrieve the current view of the active nodes;
2. The remaining three nodes send NegotiateViewResponse back to replica 4, returning the current view;
3. Replica 4 updates its own view after it receives quorum NegotiateViewResponse messages;
4. Replica 4 broadcasts the RecoveryInit message to the remaining nodes to notify them that replica 4 needs to be recovered, and requests the checkpoint information and the latest block information of the remaining nodes;
5. After receiving the RecoveryInit message, the active node sends a RecoveryResponse to return its own checkpoint information and latest block information to the Replica 4;
6. After Replica 4 receives quorum RecoveryResponse messages, it tries to find the highest checkpoint among these responses, and then updates its status to this checkpoint point;
7. Replica 4 requests PQC data after the checkpoint from the active node , and finally synchronize to the latest status of the entire network

5.6 6. RBFT Node Management

In consortium blockchain, the dynamic addition and deletion of members is required due to the expansion of the consortium or the withdrawal of some members, but the traditional PBFT algorithm does not support it. To make it

easier to control addition and deletion of members, RBFT adds the function to dynamically add and remove nodes without shutting down the cluster.

5.6.1 Process of Adding nodes



In the above figure, replica 5 is the node to be added. The process of dynamically adding this node is as follows:

1. Newly added node replica 5 initiates connections to all existing nodes by reading the configuration file information. After confirming that all nodes are connected successfully, replica 5 updates its own routing table and initiates recovery;
2. After receiving a connection request from replica 5, the existing node(including node 1, node 2, node 3 and node 4) confirms that replica 5 is allowed to join, and then broadcasts an AddNode message to the entire network, indicating that it agrees replica 5 to join the consensus network;
3. When an existing node receives N AddNode messages (N is the total number of nodes in the current blockchain consensus network), it updates its own routing table and then starts to respond to the replica 5's consensus message request (before this, All the consensus message from replica 5 would not be processed);
4. After Replica 5 finishs recovery, it broadcasts ReadyForN requests to existing nodes across the network;
5. After receiving the ReadyForN request, the existing node recalculates N and view after replica 5 joins, and then encapsulates the PQC message into AgreeUpdateN message and broadcast it to the whole network;
6. There would be a new primary node after Replica 5 joins, and now it's still node 1. After receiving N-f AgreeUpdateN messages, node 1 sends the UpdateN message as the new primary node;
7. All nodes in the network check the correctness of the UpdateN message after receiving it, and proceed to VCReset if there is no problem;
8. After completing VCReset, each node broadcasts FinishUpdate message to the whole network;
9. After receiving N-f FinishUpdate messages, all nodes process the subsequent requests and complete the adding node process.

CHAPTER 6

Ledger

6.1 1. Overview

The ledger is an important module in the hyperchain platform and is responsible for the maintenance and organization of the blockchain ledger data. Ledger data can be divided into two parts.

- Blockchain data
- Account data

Among them, the blockchain data include:

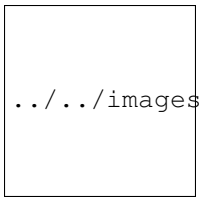
- block,
- transaction
- receipt
- other data.

This part is what we call the blockchain in the traditional sense. The latter refers to the collection of all account states on the blockchain, collectively referred to as world states. Because hyperchain supports smart contracts, as with Ethereum, it discards Bitcoin's UTXO model and uses an account model to organize the data, so this part of the data is called account data.

Blockchain data is mainly concatenated in blocks. All blocks are chained sequentially from back to front in a chain, with each block pointing to its parent block. Block contains a number of transactions, the consensus module is responsible for unified packaging and sequencing. After receiving a block, the block chain node executes the transaction in turn based on the original account status, and during this period reads / writes the status data of the relevant account. The execution of a transaction means that the state of the blockchain has undergone a transition.

Each transaction, in the hyperchain will have a transaction receipt or illegal transaction records to indicate the final execution result. If the transaction is a valid transaction, the execution result of the transaction will be recorded in the transaction receipt after the execution is completed. Conversely, the cause of the error is recorded in an illegal transaction record.

The logical relationship between the various parts of the ledger can be as follows.



6.2 2. Blockchain data

In this chapter, we describe the relationship between the following data structures:

- Block
- Transaction
- Receipt
- Chain
- Invalid Transaction Record

The first two types of data structures form "blockchain data" in the blockchain network, which is the data that needs to be "consensus" in the blockchain network. The latter three types of data structures are maintained locally by each node. The above five data structures make up all the blockchain data in one node.

6.2.1 Block

The block structure can be divided into two parts:

- Header
- Body

The block header mainly contains some blockchain metadata, including: (1) block height (2) block hash (3) parent block hash (4) world state hash (5) transaction set hash (6) receipt set hash (7) timestamp (8) log filtering data.

Block body contains all the transaction data.

The main function of the block is to encapsulate the transaction data and record the blockchain meta data.

6.2.2 Transaction

The transaction is initiated by an external user and records the user-specified call information in the transaction.

Transactions can be divided into two categories based on whether smart contracts are executed:

- normal transaction;
- contract transaction;
- contract deployment
- contract invocation

The former means that the execution of the transaction does not perform the operation of the smart contract, only the hyperchain token transfer.

The latter that the will trigger smart contract code running.

The latter can be divided to two categories:

(1) contract deployment transaction

(2) contract invocation transaction.

Transaction includes these fields：

- **Version**： Indicating the version of the transaction data structure, for backward compatibility;
- **Transaction initiator**： Identification of the initiator of the transaction, 20 bytes in length;
- **Transaction receiver**： Identification of the recipient of the transaction, 20 bytes in length.
- If the transaction type is contract invocation, the field is the address of the contract to be invoked;
- If the field is empty, it indicates that the transaction type is contract deployment;
- **Calling information**:
- If the transaction is a normal transaction, specify the number of tokens that need to be transferred in the calling information;
- If the transaction is a contract invocation transaction, specify the function to be called and the calling parameter in the calling information;
- If the transaction is a contract deployment transaction, You need to specify the contract's binary code in the call information;
- **Random value**： Random uint64;
- **Transaction Signature**： The user uses his private key to sign the content of five fields of (1) transaction initiator (2) transaction receiver (3) call information (4) timestamp (5) random value, and the generated signature content is filled in the field to prevent the contents of the transaction been tampered;
- **Transaction Hash**： Hash the above (1) - (5) fields together with the transaction signature to obtain a hash value indicating the transaction;

6.2.3 Receipt

Each legitimate transaction, the results of its execution will be packaged into a transaction receipt stored in the blockchain. Transaction receipt includes:

- **Version**： Indicates the version information defined by the receipt data structure for backward compatibility;
- **Transaction Hash**： Transaction hash associated with this receipt;
- **Contract Address**： If the transaction is a contract for deployment contract, the newly deployed contract address is placed in the field, otherwise the field is empty;
- **Execution Result**： If the transaction is a contract invocation transaction, the result of the execution is placed in the field, otherwise the field is empty;
- **Contract Logs**： During a smart contract execution, a series of logs may be generated and the log data is placed in this field;
- **Contract Type**： Contract type is placed in this field, EVM(ethereum virtual machine), JVM or something else.

6.2.4 Invalid Transaction Record

Each illegal transaction, the error message will be packaged into an illegal transaction record, stored in the local node.

Except transaction data related to the illegal record, the specific causes of the error will also been recorded, for example: (1) the balance is insufficient (2) the parameters of the contract invocation illegal (3) call permission is not enough and so on.

6.2.5 Chain

A local node maintains some blockchain metadata for quick query, so in the hyperchain there is a data structure named chain that records this data, including：

- Latest parent block hash;
- Latest block hash;
- Latest block number;
- Genesis block number: default genesis block number is 0, but can be affected by *data archieve/restore*;
- Transaction amount;
- Extra;

6.2.6 Consensus comparison

After executing all the transactions in a block, local node needs to compare the "results" with other nodes in the network, and only when "enough(quorum)" nodes have same result with local node, these results will be submitted to the database.

The "execution result" of a block consists of the following contents：

- **World state hash**: During the execution of the transaction, the world state data will be changed. When all transactions in a block are executed, the bucket tree is used to perform a hash calculation on the world state, and the calculation result is the world state hash.
- **Transaction set hash**: Using the *important field* of each transaction in the block as input to the sha256 algorithm, hash result represents the entire transaction set. The important fields are: (1) transaction initiator (2) transaction receiver (3) call information (4) timestamp (5) random value；
- **Receipt set hash**: Using the *important field* for each receipt in a block as input to the sha256 algorithm, hash result represents the entire set of receipts. The important fields are as follows: (1) VM execution counters (2) Execution results (3) VM execution logs

6.3 3. World state

The blockchain data mentioned above can in fact be summarized as a water-flow collection of contract invocation information. The smart contract needs to read / write the contract status data during the execution. Now introduce the structure of this part of the data。

Because hyperchain needs to be compatible with EVM (Ethereum Virtual Machine), and EVM has strong coupling with Ethereum's account system, hyperchain's state is based on Ethereum, and a series of modifications and optimization have been made.

6.3.1 Account type

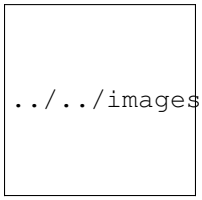
Like Ethereum, accounts in hyperchain can be divided into two categories:

- **External Account**: The private key of external accounts are controlled by the users themselves; this type account and can initiate transaction. Besides, such account does not contain smart contract codes;
- **Contract Account**: The contract account contains an executable smart contract code and has its own storage space for storing its own state variables. The operation of the smart contract can be triggered by initiating a transaction with an external account or by another contract.

Although the two types of accounts differ in logic, but share the same definition:

The metadata for an account includes the following fields:

- **Account address**: 20 bytes, generated by the hash function according to certain input, regardless of hash conflicts, there will be no two accounts with the same address;
- **Balance**: The balance indicates the number token owned by the account. Such tokens can be manipulated through smart contracts or can be traded by initiating normal transaction transfers;
- **State variables hash**: A contract account need to store all of its state variables, a hash value used to represent these state variables is stored in the field;
- **Code hash**: a hash to represents contract code;
- **Status**: the status of contract, normal or frozen;
- **Birthday**: If the account is a contract account, this block number when this contract been deployed will be placed in this field;
- **Creator**: If the account is a contract account, the creator address will be placed in this field;
- **Deployed list**: If the account is a external account, all contract address deployed by itself will been recorded in this list;

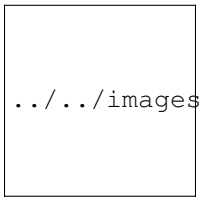


Except these "simply" data placed in the account metadata, there are (1) contract source code (2) state variables which require a lot of storage space data are stored directly to database. Only the hash value is stored in account metadata.

In fact, contract state variables are a series of key value pairs. In hyperchain, there is a bucket tree for each contract account to compute the status hash of the contract state variables.

Each time a transaction is executed and a series of state variables are modified, these changes can just as input to the bucket tree in order to quickly calculate the "new" state variables hash。

6.3.2 Account set



hyperchain serializes the metadata of an account, using the serialized binary as the content of an account. All account data can eventually be converted into a series of kv pairs, the key is the address of the account, and the value is the metadata serialized content.

For the account set, there will be a global bucket tree for the **world state** hash calculation as shown in the figure.

Each account is serialized as a record in the bucket tree, the hash value of the entire world state is uniformly calculated by that bucket tree. This hash value, as a state of the world state, is not only one of the bases for comparison of the consensus stage but will be recorded later in the block header.

6.3.3 Atomicity

Hyperchain uses the batch tool provided by the underlying database leveldb to ensure the atomicity of the ledger. Hyperchain uses rbft as a consensus algorithm, so the entire process is split into 3 phases. During the execution phase, all changes to the ledger will be pre-stored in a leveldb batch. When the result of this execution passes the consensus comparison between nodes, the batch will be removed from the cache and all changes will be placed on the disk.

CHAPTER 7

Bucket tree

7.1 Overview

In hyperchain, the ledger data can be divided into two parts:

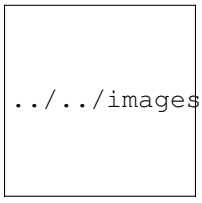
- blockchain data
- account data

The blockchain data includes: block, transaction, receipt and other data. This part is what we call the blockchain in the traditional sense. All blocks are chained sequentially from back to front in this chain, with each block pointing to its parent block.

Block contains a number of transactions, the consensus module will be responsible for the deciding order of the received transactions and packaged into a block for distribution. After receiving a block, the block chain node executes the transaction in sequence on the basis of the original state, and reads/writes status data of the relevant account during this period. When the execution ends, all account data are written and changed atomically. The execution of each transaction means that the blockchain has undergone a state transition.

The state of the blockchain refers to a collection of all the account states on the blockchain, called **world state**. As support smart contracts, like Ethereum, hyperchain abandoned the Bitcoin UTXO model and used an account model to organize the data, so this part of the data is called account data.

Therefore, hyperchain's ledger can be broadly divided into the above two parts, the structure diagram is as follows.



In this article we will not start the discussion of the structure of the ledger, but to discuss a tree structure used to *quickly calculate the world state hash*.

In hyperchain, after each block executed, each node needs to compare whether the results of the execution is consistent in the third phase of BFT. In other words, the account data needs to be consistent in each node. Therefore, the hyperchain uses a bucket tree structure to hash the account data. The nodes only need to compare the *tree hash values* to determine the consistency of the account data.

Note: Bucket tree does not organize and maintain the account data directly, but only calculates the hash.

Bucket tree has the following characteristics：

- Provides a mechanism to quickly calculate the hash of account data;
- Provides a mechanism for ledger rollback;

The theory of bucket tree in hyperchain is the earliest learned from the *fabric project*, a series of reconstruction and optimization, making the final performance in line with production needs. In the following, the structure of this tree, the core operations, and the final performance are described in detail.

7.2 Structural analysis

Bucket tree is actually a combination of two different data structures, the two data structures are：

- merkle tree
- hash table

Therefore, before introducing the structure of bucket tree, we first briefly introduce these two data structures。

7.2.1 merkle tree

The Merkle tree was introduced many years ago by computer scientist Ralph Merkle and named with his own name. This data structure is used in bitcoin networks for verification of data correctness.

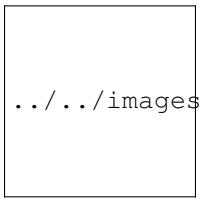
In bitcoin networks, the merkle tree is used to summarize all transactions in a block and to generate a digital fingerprint of the entire transaction set. In addition, due to the existence of the merkle tree, it becomes possible to extend a "light node" for simple payment verification in the case of a Bitcoin chain.

Features

- Merkle tree is a kind of tree, most of them are binary tree, also can be multi-tree. It has all the characteristics of tree structure;
- Merkle tree leaf node value is the content of the data entry, or the data entry hash value;
- The value of a non-leaf node is calculated by Hash according to the information of its child node;

Principle

In bitcoin networks, the merkle tree is built from the bottom up. In the example below, we first hash the four data entry of ENTRY1-ENTRY4 and then store the hash value to the corresponding leaf node. These nodes are Hash0-0, Hash0-1, Hash1-0, Hash1-1



Combine two adjacent node's hash into a single string, and then calculate the hash of this string.

The result is the parent node's hash value of these two nodes.

If the number of tree nodes in this layer is a single number, then this case directly performs hashing on the last remaining tree node, and the hash of its parent node is the hash of its hash value (for the singular There are different ways to deal with leaf nodes, and you can copy the last leaf node to get even number of leaf nodes). Loop repeat the calculation process, get the last node as the root node, the hash of the root node is the hash of the entire tree.

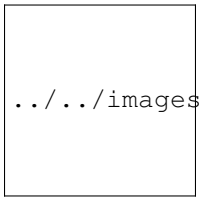
If the two trees have the same root hash, the contents of the two trees must be the same.

The advantage of using the merkle tree is that when the content of a node changes, it only needs to recalculate the hash of *all the tree nodes in the path from the node to the root node* to obtain a hash that can represent the status of the whole tree value.

It is also because of this feature of the merkle tree that the bucket tree avoids many unnecessary computational overheads and has the ability to quickly compute the world state hash.

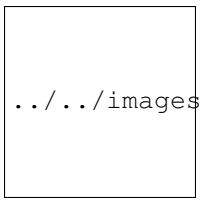
7.2.2 Hash table

Hash table, also known as hash table, is a very familiar data structure, is based on the key (key) and direct access to the memory storage location. In other words, it speeds up lookup by calculating a function on the key that maps the data of the desired query to a location in the table to access the record. This mapping function is called a hash function, the record array is called a hash table.



The description about the hash is not repeat here. In the bucket tree, use the hash table to maintain the original data。

7.2.3 bucket tree



Bucket tree consists of two parts: the bottom hash table and the upper merkel tree. Bucket tree is actually a merkle tree built on the hash table.

A hash table consists of a series of buckets, each of which contains a number of entries that have been hashed into the bucket, all of which are arranged in sequence. Each bucket has a hash value to represent the state of the entire hash bucket, which is hashed according to the contents of all the data entries in the bucket.

Except the underlying hashtable, the upper level is a series of merkle tree nodes. A merkle tree node corresponds to n hash buckets or merkle tree nodes in the lower level. This n is also called the degree of aggregation of the merkle tree. The merkle tree node maintains the hash values of the n child nodes, and the hash value of the merkle tree node is calculated according to the hash values of the n child nodes.

So continuous iteration, the ultimate tree node is the root node of the entire tree, the node's hash value represents the hash of the entire tree.

The purpose of this design is:

- Using the characteristics of the merkle tree, the computational cost of re-hashing is minimized every time the tree state changes;
- The use of hash table for the maintenance of the underlying data, making the data entries evenly distributed;

For example, in the figure above, a new data entry `entry5` is inserted, and the data entry is hashed to a bucket at POS 2. The bucket, as well as all the nodes from the bucket to the root node are dirty nodes marked in pink. Recalculation only with these dirty nodes can gives you a new hash value to represent the new tree state.

Because the bucket tree is a fixed-size tree (that is, the underlying hash table can not be changed after the tree has been initialized). Using hash table to distribute data entries evenly can avoid data aggregation occurs.

In addition, bucket tree has two important tunable parameters:

- capacity
- aggreation

The former indicates the **capacity of the hash table**. The larger capacity, more number of data entries that the entire tree can accommodate. Under the condition of the same degree of aggregation, the larger capacity, the higher tree height, and more number of tree nodes in the path from the leaf node to the root node, the number of hash calculation increases.

The latter represents **the number of child nodes corresponding to a parent node**. The larger aggreation, the faster the tree converges. Under the premise of the same hash table capacity, the tree height is lower, the number of tree nodes in the path from the leaf node to the root node is less, and the hash calculation times are reduced. However, the size of each Merkle tree node is larger, which increases the database IO overhead.

7.2.4 Bucket

The definition of a hash bucket is made up of a series of data entries, note that these data entries are sorted by key's lexicographical order, each of which represents a user data (which can be optimized to store only the hash of user data).

```
type Bucket []*DataEntry
```

7.2.5 merkle node

The definition of merkle node is as follows. The main field is the list of children nodes related to it. Each element in this list is a hash value of a child node.

```
// MerkleNode merkleNode represents a tree node except the lowest level's hash bucket.
// Each node contains a list of children's hash. It's hash is derived from children's content.
// If the aggreation is larger(children number is increased), the size of a merkle node will increase too.
type MerkleNode struct {
    pos      *Position
    children [][]byte
    dirty    []bool
    deleted  bool
    lock     sync.RWMutex
```

(continues on next page)

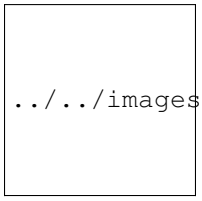
(continued from previous page)

```
    log      *logging.Logger
}
```

7.3 Core operation

The calculation procedure of bucket tree can be divided to four parts:

1. Initialize
2. Prepare
3. Process
4. Commit



7.3.1 Initialize

In the initialization phase, building tree shape construction, cache initialization and historical data recovery (from the db read the latest root node hash)

The tree structure uses user configuration capacity and aggreation two parameters to construct . The build function is as follows:

```
var (
        curlevel  int
        curSize   int = cap
        levelInfo     = make(map[int]int)
    )
    levelInfo[curlevel] = curSize
    for curSize > 1 {
        parSize := curSize / aggr
        if curSize%aggr != 0 {
            parSize++
        }
        curSize = parSize
        curlevel++
        levelInfo[curlevel] = curSize
    }
    conf.lowest = curlevel
    for k, v := range levelInfo {
        conf.levelInfo[conf.lowest-k] = v
    }
```

In addition, bucket tree in order to

1. increase the efficiency of reading
2. to prevent write loss

uses two caches for cache hash bucket and merkle node data.

These two cache are achieved by LRUCache, each update will be synchronized to update the contents of the cache. So, the during next hash calculation, bucket tree can hit hot data from the cache and try to avoid the disk read.

As for the prevention of write loss, since **validation** and **commit** are two separate asynchronous processes in hyperchain, validation of block 101 may be performed immediately based on the state the validation process of block 100 proceeds. At this moment, the modification of the ledger in the execution of the block 100 has not yet been submitted to the database, so in order to "prevent write loss", the contents need to hit from the cache.

If at this moment there is a situation where the capacity of the cache is **too small** and the content that which not submitted is driven out. The result of the validation of the block 101 is inconsistent with other nodes (usually the primary node). When this happen, this node will enter `recovery` procedure and `rbft` will promise the corecctness.

The bucket cache is used to store the hash bucket data, each hash bucket is a cache data item. The problem is that a hash bucket itself consists of a number of data entries, with the running time increasing, the size of a hash bucket will be larger and larger, resulting in increasing memory usage.

The merkle node cache is used to store all merkle node data except for the highest layer. The number of merkle nodes is fixed, and the size of each node is also capped. Therefore, merkle node cache does not have a problem that content occupancy is increased.

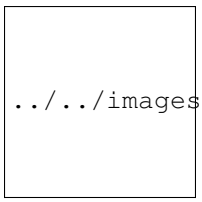
7.3.2 Prepare

During the preparation phase, the bucket tree receives the modified set passed in by the user and constructs a dirty set of hash buckets using the contents of the modified set. Note that in the returned hash bucket, the internal data items are arranged in ascending lexicographical order.

```
func newBuckets(prefix string, entries Entries) *Buckets {
    buckets := &Buckets{make(map[Position]Bucket)}
    for key, value := range entries {
        buckets.add(prefix, key, value)
    }
    for _, bucket := range buckets.data {
        sort.Sort(bucket)
    }
    return buckets
}
```

7.3.3 Process

Process is the hash recalculation phase, can be divided into two parts (1) dirty hash bucket re-calculation (2) dirty merkle node hash re-calculation.



Dirty hash bucket re-calculation

As shown in the figure above, two new entry data items entry5 and entry6 are inserted in the bucket tree. The hash address obtained by entry5 is Pos2, and the hash address obtained by entry6 is Pos5.

There is a merge operation in Pos2 to insert the new data and merge with historical data with the fixed sorting algorithm to reorder, and ultimately get a new hash bucket, contains all new and old data in order.

The hash value of each hash bucket is *a result of hashing the data in the whole bucket*.

As shown in the figure, Pos2 and Pos5 are two dirty hash buckets. After the calculation is completed, the corresponding child hash value is set in the parent node.

Dirty merkle node re-calculation

When the hash bucket calculation is completed, the hash calculation of the merkle node can be performed. In this step, only the dirty merkle nodes are hash calculated.

Note that the hash calculation of the merkle node is done hierarchically.

Each merkle node maintains the hash value of its child node, and if the child node hash value of the lower layer changes, the latest hash value is placed in the parent node in the previous calculation. For no change child node, bucket tree can directly use the history value.

The hash value of each merkle node is *a result of hashing all its child node hashes*.

7.3.4 Commit

After the calculation is completed, the latest hash bucket data and merkle node data needs to be persisted.

In addition, all the hash bucket data, merkle node data will be stored in the cache as hot data, both to improve data search efficiency, but also to avoid data write loss.

CHAPTER 8

Smart Contract

8.1 1. Introduction

Note: A smart contract is a piece of computer program deployed on the blockchain that automatically executes the terms. Smart contract can automatically execute predefined protocols based on outside input and complete the transfer of relevant status within the blockchain.

In the wide sense, smart contracts also include smart contract programming languages, compilers, virtual machines, events, state machines, fault tolerance mechanisms, and more. The most import components of smart contract is the smart contract programming language and its execution engine.

Smart contract virtual machines are generally sandboxed for security reasons, and the entire execution environment is completely isolated.Smart contracts executed inside virtual machines are not allowed to access to system resources such as network, file system, process threads, and so on.

Different smart contracts own different levels of security and richness of expression. The HyperVM, which is developed independently by the Hyperchain team, is a general-purpose intelligent contract engine designed to allow access by many different smart contract engines. Currently, HyperEVM is compatible with Ethereum's Solidity language and HyperJVM, a Java-enabled smart contract execution engine.

8.2 2. Smart contract execution engine HyperVM

HyperVM developed by Hyperchain is a pluggable smart contract engine generic framework that allows different smart contract execution engine embed.

As shown in the following figure is the schematic diagram of HyperVM architecture, HyperVM architecture provides smart contract compiler, interpreter, executor and state management components and other related major components. Among them, Compiler provides smart contract compilation related functions, Interpreter and Executor provide smart contract interpretation and execution related functions, and State components help contract to operate blockchain ledger. Guard module provides smart contract security-related mechanisms.

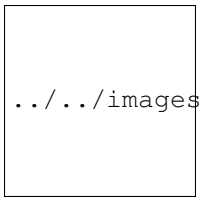


Fig. 1: hypervm-architecture

8.2.1 2.1 HyperEVM

To maximize the open source community's research and accumulation in smart contract technologies, enhance the reusability and compatibility of smart contracts. HyperEVM implementation uses a fully compliant Ethereum smart contract specification using Solidity as the smart contract development language and adopt the optimized Ethereum virtual machine EVM as the default backend execution engine.

The following figure shows the HyperEVM smart contract execution flow:

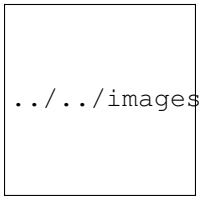


Fig. 2: hyperevm-flow

HyperEVM executes a transaction, it returns an execution result. The system stores it in a variable called a transaction receipt, and the platform client can query the transaction result according to the current transaction hash.

This process runs as follows:

1. HyperEVM received the upper transmission of the transaction, and do preliminary verification;
2. Determine the type of transaction, if it is the disposition of the contract, go to step 3, otherwise go to step 4;
3. HyperEVM create a new contract account to store the contract address and the compiled code;
4. HyperEVM verify transaction parameters and signature information, and call its execution engine to execute the corresponding smart contract byte code;
5. After the instruction is executed, HyperVM will determine if it is down normaly, skip Step 2 if not, otherwise go to Step 6;
6. Determine whether the shutdown state of HyperVM is normal, and if it is normal, stop execution; otherwise, go to Step 7;
7. Undo operation, the state should also rollback.

The instruction set execution module is the core of HyperEVM execution component.The execution of the instruction module has two implementations, namely bytecode-based execution and more complex and efficient Just-in-time compilation.

Bytecode implementation is relatively simple, HyperEVM virtual machine have an instruction execution unit. The instruction execution unit will always try to execute the instruction set. When the designated time is not completed, the virtual machine will interrupt the calculation logic and return a timeout error message to prevent the execution of malicious code in the smart contract.

JIT execution is relatively more complex, instant compilation also known as timely compilation, real-time compilation, is a form of dynamic compilation is a way to improve the efficiency of the program. In general, there are two ways to run a program: static compilation and dynamic transliteration. Statically compiled programs are all translated into machine code before execution, while literal translation is performed while translation is performed. The real-time compiler mixes both, compiling the source code sentence by sentence, but caches translated code to reduce performance penalties. Compare to the static compiled code, compiled code on the fly can handle delayed binding and enhance security. JIT execution model mainly includes the following steps:

1. All the information related to the smart contract is encapsulated in the contract object, and then find out whether the contract object stored and compiled by the hash of the code.There are four common status of the contract object, namely: unknown, compiled, ready to execute through JIT, error.
2. If the contract status is ready for execution through the JIT, HyperEVM selects the JIT executor to execute the contract. During execution, the virtual machine will further compile the compiled smart contract into a machine code and optimize the push and jump instructions.
3. If the contract status is unknown, HyperEVM first needs to check if the virtual machine is forcing the JIT to execute, and if so, it will be compiled sequentially and executed by the JTI instruction. Otherwise, open a separate thread to compile, the current program is still compiled by ordinary bytecode. Next time the virtual machine encounters the same encoded contract again during execution, the virtual machine directly selects the optimized contract. As a result of the optimization of the instruction set of such a contract, the efficiency of the execution and deployment of the contract can be greatly improved.

8.3 3. Usage of Smart contract

8.3.1 3.1 Smart Contract Based on the Solidity

8.3.2 Write a contract

The Solidity-based smart contract is similar to a JavaScript program and consists of a series of variables and related functions. Below is a smart contract that simulates simple accumulator functionality. We use this as an example to briefly introduce the basic components of a Solidity smart contract.

```
contract Accumulator{
    uint32 sum = 0;
    function increment(){
        sum = sum + 1;
    }

    function getSum() returns(uint32){
        return sum;
    }

    function add(uint32 num1,uint32 num2) {
        sum = sum+num1+num2;
    }
}
```

Accumulator contract description:

- Solidity-based smart contracts begin with the keyword `contract`, similar to the keyword `class` in Java and other languages;
- The contract can have variables and functions inside, the sum is a simple variable uint32 type, Solidity smart contract also supports map and other complex collection types;

- The contract allows the definition of the implementation of the function `function` keyword definition;

Reference to [Solidiy official website] (https://solidity.readthedocs.io/en/develop/) for detailed specification of smart contracts based on the Solidity language.

8.3.3 Compile the contract

Hyperchain's smart contracts can be compiled either with the official Solidity compiler or using the smart contract JSON-RPC interface provided by Hyperchain (this scenario requires installing the Solidity compiler `sloc` on the host where Hyperchain installed).

The command which call Hyperchain to compile the solidity smart contract as follows:

```
curl -X POST --data
'{
    "jsonrpc":"2.0",
    "namespace":"global",
    "method":"contract_compileContract",
    "params":["contract_code"],
    "id":1
}'
```

The contract compilation call returned as follows:

```
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "abi": [
      "[{\"constant\":false,\"inputs\":[{\"name\":\"num1\",\"type\":\"uint32\"},{\
↪"name\":\"num2\",\"type\":\"uint32\"}],\"name\":\"add\",\"outputs\":[],\"payable\
↪":false,\"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"getSum\
↪",\"outputs\":[{\"name\":\"\",\"type\":\"uint32\"}],\"payable\":false,\"type\":\
↪"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"increment\",\"outputs\
↪":[],\"payable\":false,\"type\":\"function\"}]"
    ],
    "bin": [

↪"0x6060604052600805463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633ad14a
↪"
    ],
    "types": [
      "Accumulator"
    ]
  }
}
```

The content corresponding to the field bin is the bytecode representation of the contract, and the bin will be used for subsequent deployment.

8.3.4 Deploy the contract

Hyperchain deploy solidity contract command is as follows:

```
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global",  "method":
↪"contract_deployContract", "params":[{
"from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc ",
"nonce":5373500328656597,
"payload":
↪"0x6060604052600080546 3ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633ad14a
↪",
"signature":
↪"0x388ad7cb71b1281eb5a0746fa8fe6fda006bd28571cbe69947ff0115ff8f3cd00bdf2f45748e0068e49803428999280c
↪",
"timestamp":1487771157166000000
}],"id":"1"}'
```

Deploy contract execution is returned as follows, where the result field is the address of the contract in the blockchain, and the contract address needs to be specified for subsequent execution call to the contract.

```
{
    "jsonrpc": "2.0",
    "namespace":"global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "0x406f89cb205e136411fd7f5befbf8383bbfdec5f6e8bcfe50b16dcff037d1d8a"
}
```

8.3.5 Call contract

The Hyperchain invocation command is as follows, where `payload` is the encoding result of the function in the invocation contract and its parameter value, and `to` is the address of the invoked contract.

```
curl localhost:8081 --data

'{
    "jsonrpc":"2.0",
    "namespace":"global",
    "method": "contract_invokeContract",
    "params":[{
            "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
            "nonce":5019420501875693,
            "payload":
↪"0x3ad14af3000000000000000000000000000000000000000000000000000000000000001000000000000000000000000
↪",
            "signature":
↪"0xde467ec4c0bd9033bdc3b6faa43a8d3c5dcf393ed9f34ec1c1310b0859a0ecba15c5be4480a9ad2aaaea8416324cb54e
↪",
            "simulate":false,
            "timestamp":1487773188814000000,
            "to":"0x313bbf563991dc4c1be9d98a058a26108adfcf81"
            }],
    "id":"1"
}'
```

The contract call will immediately return the hash value of the transaction to the client, and later query the execution result of the specific transaction according to the hash value of the transaction.

```
{
    "jsonrpc":"2.0",
    "namespace":"global",
    "id":1,
    "code":0,
    "message":"SUCCESS",
    "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

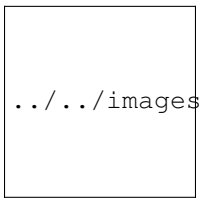

The other contract operation methods and specifications of methods parameters are detailed in: [Hyperchain API Documentation] ()

CHAPTER 9

P2P

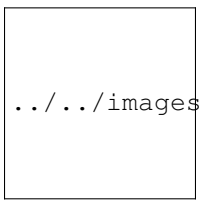
9.1 1. Overview

The P2P module is the underlying Hyperchain network communication module. It guarantees the data transmission security of the communication link. The user can configure whether to enable TLS security and enable data transmission encryption (or replace the data encryption algorithm) through the configuration file. In this module, the physical connection (Network layer) is separated from the logical connection. The overall architecture of the module is as follows:



9.2 2. Hypernet

As the underlying communication infrastructure of Hyperchain, Hypernet provides network communication services to the upper layer by registering slots. Its main functions include establishment of communication links, data transmission, link security, and link activity control. It has two important members `Server` and `Client`. The overall structure is as follows:



9.2.1 Server

In Hypernet, the Server is responsible for registering network slots, listening for services, and distributing various types of messages from the Client.

Network Slot

As the main mechanism of Hypernet to communicate with the upper layer, the slot is actually a multi-dimensional thread-safe map, which maps the relevant methods to different message processors for processing.

Slots as server members, with a set of slot that correspond to different namespaces, process messages from different namespaces, respectively.

9.2.2 Client

Client corresponds with the Server, mainly used to handle different message sending requests. Usually a Client corresponds to several different remote Server (because a node will be connected to multiple remote peer), and communicates with different servers which will distribute message to different namespace slot to deal with.

9.2.3 TLS

Transport layer security of Hyperchain, default enable, which USES the TLSCA certificate issued by the Hyperchain to carry out secure communication, which guarantees the security of information and communications from the transport layer. Further, this option is optional.

TLS guarantees the security of information transmission at the transport layer. It is the most common implementation standard for network transmission and is adopted in almost all network security transmission.

9.3 3. P2PManager

`P2PManager` is used to allocate `PeerManager` in different namespaces, it has only one global instance.

9.3.1 PeerManager

`PeerManager` is mainly responsible for the following sections:

- Provide different external message sending service interfaces;
- Post message to Hyperchain message middleware`eventhub`;
- Use `PeerManagerEventHub` message middleware distributes and manages the control messages of `PeerManager` to maintain the state of the entire logical network and process more complicated message logic.

In the entire network, the node can be called `Node` or `Peer`, the following look at the difference between the two.

Peer

All logical remote nodes are called peer, a remote node corresponds to a peer, the peer is mainly used for processing the logically message sending request. Its main function is to encrypt the message, and then call the corresponding message sending method of Hypernet Client, and the message needs to be attached namespace information.

Node

Node that is the local node, which is also a logical server, is mainly responsible for logically message processing of node, decryption of the message received from the network and then posts decrypted message to Hyperchain message middleware `eventhub`. Finally, the message middleware identifies this type of message should be post to which module to process, such as consensus module, executor module.

9.3.2 Data transmission encryption

Data Transfer Encryption refers to the encryption of transaction information and communication messages transmitted over the network. According to user requirements, all information transmitted on Hyperchain network can be encrypted. The encryption scheme is similar to TLS, firstly the node use the ECDH algorithm to negotiate the corresponding session key, and then the session key is used to encrypt the service information which is decrypted by the peer. All the communication between nodes will be encrypted with different session keys. This is a supplement to the security of the transport layer. Currently, hyperchain messages can use Symmetric Encryption through configuration, and can be handled in this way if there is a more sophisticated message encryption requirement.

CHAPTER 10

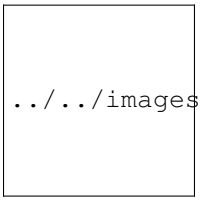
Digital Certificate

10.1 1.Overview

Hyperchain is a consortium blockchain service platform with granular granularity of permission control.It requires multi-level CA certificates for permission control. Access control is divided into two aspects:

- Node access control
- Trading authority control

First of all, we need to know is that the permissions control is the Namespace level, that is, each Namespace will have a corresponding CaManager for CA certificate management and Namespace level permission control. The following is our PKI system (certificate system) map:



- root.ca (root certification authority): It represents the trust anchor in the PKI architecture. Verification of digital certificates follows the chain of trust. The root CA is the top-level CA in the PKI hierarchy and is used to issue enrollment certificate authorities and role certificate authorities.
- eca.ca (enrollment certificate authority): Used to issue nodes node Enrollment certificates (ecert) and sdk certificates (sdkcert).
- rca.ca (role certificate authority): Used to issue a role certificate (rcert) to a node.
- ecert.cert (enrollment certificate): An enrollment certificate is a long-term certificate issued to a node for access to the node. If no enrollment certificate is available, the node can not join the Namespace and the enrollment certificate is also used as a transaction The issuer of the certificate is used to issue the transaction certificate.
- rcert.cert (role certificate): A role certificate is a long-term certificate that is issued to a node for authentication of the role of the node. If there is no role certificate, this node is an NVP and can not participate in the consensus.

Otherwise, it is a VP node.

- sdkcert.cert (sdk certificate): The SDK certificate is issued to the SDK to determine the basis of the (Tcert) used to authenticate the SDK and obtain the transaction certificate when receiving the SDK certificate.
- tcert.cert (transaction certificate): The SDK needs to send the transaction with the transaction certificate. If there is no transaction certificate or the certificate verification fails, the transaction is abandoned.

10.2 2.Certificate Introduction

Hyperchian blockchain platform certificates are in line with ITU-T X.509 international standards, it only contains the public key information without private key information, is publicly available, so X.509 certificate object generally do not need to be encrypted. The format of the X.509 certificate is usually as follows:

```
---BEGIN CERTIFICATE---
......PEM encoded X.509 certificate content (omitted)......
---END CERTIFICATE---
```

The full name of PEM encoding is Privacy Enhanced Mail, which is a coding standard for confidential email. In general, the process of encoding information is basically as follows：

- The information is converted to ASCII or other encoding, such as using DER encoding.
- Use symmetric encryption algorithm to encrypt the encoded information.
- Use BASE64 to encode the encrypted information.
- The use of some header definitions to encapsulate the information, mainly contains the necessary information for the correct decoding.

In addition, the Hyperchain blockchain platform certificate specifically includes the following information:

1. X.509 version number: Pointed out that the certificate which version of the X.509 standard version number will affect some specific information in the certificate. The current version is 3.
2. Certificate holder's public key: includes the certificate holder's public key, the identifier of the algorithm (indicating which cryptographic system the key belongs to) and other relevant key parameters.
3. the serial number of the certificate: given by the CA assigned to each certificate a unique number, when the certificate is canceled, the certificate is actually the serial number of the certificate issued by the CA CRL (Certificate Revocation List certificate revocation list, Or certificate black list). This is also the only reason for the serial number.
4. Topic information: The unique identifier of the certificate holder (or DN-distinguished name) This name should be unique on the Internet. The DN consists of many parts that look like this:

```
CN=Bob Allen, OU=Total Network Security Division
O=Network Associates, Inc.
C=US
```

The information indicates the common name of the subject, the name of the organizational unit, organization and country or certificate holder, and the location of the service.

5. the validity of the certificate: the certificate start date and time and the date and time of termination; specified certificate valid for these two periods.
6. Certification body: The certificate issuer, which is the X.509 name of the only CA that issued the certificate. Using this certificate means trusting the entity that issued the certificate. (Note: In some cases, such as a root or top-level CA certificate, the publisher itself issues a certificate)

7. Digital signature of the publisher: This is a signature generated using the publisher's private key to ensure that the certificate has not been redirected since it was released.
8. Signature algorithm identifier: used to specify the signature algorithm used by the CA to sign the certificate. The algorithm identifier is used to specify the public key algorithm and hash algorithm used by the CA when issuing a certificate.

10.3 3.CA Configuration

CA configuration required in the namespace.toml configuration file, the specific parameters are as follows:

```
[encryption]
[encryption.ecert]
 eca    = "config/certs/eca.cert"
 ecert  = "config/certs/ecert.cert"
 priv   = "config/certs/ecert.priv"

[encryption.rcert]
 #if you do not have rcert, leave this item blank
 rca    = "config/certs/rca.cert"
 rcert  = "config/certs/rcert.cert"
 priv   = "config/certs/rcert.priv"

[encryption.tcert]
 #Tcert whitelist configuration.
 whiteList = false
 listDir  = "config/certs/tcerts"

[encryption.check]
 enable     = true  #enable ERCert
 enableT    = false  #enable TCert
```

First of all, the first six parameters are related to the configuration change Namespace certificate path, namely eca, ecert and ecert corresponding private key, rcar, rcert and rcert corresponding private key.

And the TCert configuration, the platform supports the TCert whitelist policy, that is, when the `whiteList = true`, the TCert whitelist policy is enabled, and the TCert certificate under the listDir parameter configuration is an immediately available transaction certificate. On the other hand, when the `whiteList` is false, the whitelist policy is not enabled. Only when the validity of the transaction certificate is verified and the transaction certificate is determined to be the node and the transaction certificate promulgated under the Namespace can the verification be fully verified.

The last two parameters are configured switch parameters, `enable` is used to open the enrollment certificate and the role of the certificate check switch. `EnableT` is uesd to open the transaction certificate verification switch configuration, only when the parameter is set to true Before the transaction certificate verification. The dynamic switch configuration also makes the block chain more flexible.

10.4 4.Certificate acquisition and verification process

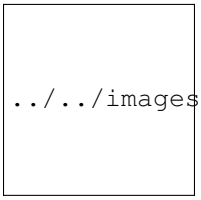
10.4.1 4.1 ECert and RCert

4.1.1 Acquisition

Enrollment certificates and role certificates are mainly issued under the control of the line, a Certgen certificate issuance tool for certificate generation.

4.1.2 Verification

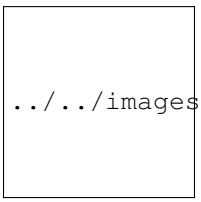
If the ECert and RCert check switches are turned on, the specific verification flow is as follows:



ECert and RCert exchanged certificates and authenticated the certificate when the node handshake the connection for the first time to determine whether the node is allowed to enter the chain and the role information of the connected node.

10.4.2 4.2 TCert

TCert acquisition and verification of the flow chart as shown below:



4.2.1 Acquisition

First, the SDK or the external application needs to send a GetTcert message to the connected node. The message needs to carry the SDKCert to authenticate the SDK or the external application. After the authentication is passed, the TCert certificate is generated and promulgated.

4.2.2 Verification

If the SDK or the external application acquires the TCert successfully, the following transaction needs to carry the relevant transaction certificate to the relevant node for verification. Only after the transaction certificate is verified, the next transaction execution will be performed.

CHAPTER 11

Namespace

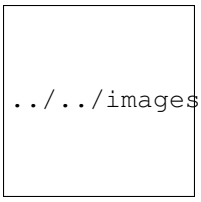
11.1 1.Overview

Hyperchain achieves a partitioned consensus on internal transactions in blockchain network by the Namespace mechanism. Users of the Hyperchain platform can differentiate business transactions by namespace. Nodes in the same Hyperchain consortium blockchain network form sub-blockchain networks in namespace-size based on the services they participate in. Namespaces achieve business-level privacy protection through physical level isolation of business transaction consensus, distribution and storage .

11.2 2.Cluster Architecture

Namespaces can be dynamically created, and individual Hyperchain nodes can participate in one or more namespaces according to their business needs. The following figure shows the overall cluster architecture of the namespace mechanism: six nodes participate in two namespaces. Node1, node2, node4 and node5 form namespace1, and node2, node3, node5 and node6 form namespace2. Node1 and node4 only participate in namespace1, node3 and node6 only participate in namespace2, while node2 and node4 participate in two namespaces at the same time. The namespace controls the dynamic joining and exiting of nodes through CA authentication, and each node is allowed to participate in one or more namesapces.

The verification, consensus, storage and transmission of transactions with specific namespace information are performed only among nodes that participate in the corresponding namespace. Transactions between different namespaces can be executed in parallel. As shown in the following figure, Node1 can only participate in the verification of the transactions in namespace1 and the corresponding ledger maintenance. Node2 can simultaneously participate in transaction execution and account maintenance of namespace1 and namespace2. However, the ledgers of namespace1 and namespace2 in node2 are not accessible to each other.



11.3 3.Node Architecture

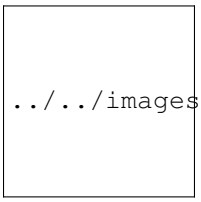
Hyperchain's single node with the partition consensus mechanism will include a NamespaceManager object. NamespaceManager is the key management component of the partition consensus mechanism, which is responsible for a series of life-cycle state operations such as registering, starting, stopping and de-registering a specific namespace.

NamespaceManager contains multiple namespaces, in addition to JvmManager and BloomFilter.

In particular:

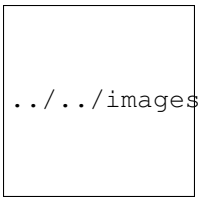
- JvmManager is responsible for managing the jvm executor, and starting JvmManager or not should be configured in the configuration file;
- BloomFilter is a bloomFilter for transactions, which is mainly responsible for detecting duplicate transactions and preventing replay attacks.

One of the partitions is called a namespace, and each namespace is isolated from each other, including the execution space and data storage space. Each node joins the namespace named global by default. Each namespace contains key components such as consenter, executor, eventHub, peerManager, caManager, requestProcessor, etc.. And these key components implement consensus service, transaction execution and storage, asynchronous interaction among modules, inter-node communication, identity authentication, transaction processing for the respective namespace blockchain.



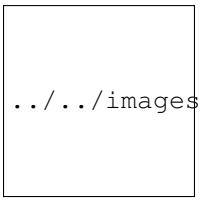
To be specific:

- consenter provides consensus services and currently supports the RBFT algorithm, which is responsible for sequencing the transactions to ensure the consistency of the hyperchain nodes in the same namespace;
- executor is responsible for the invocation of the smart contract in the namespace and the maintenance of the ledger;
- eventHub is an event bus, which is a message transit center for asynchronous interaction between various key components in the namespace;
- peerManager provides node communication management, is responsible for network communication between namespace members;
- caManager is a certificate authority, which is responsible for the identity authentication on the Internet;
- requestProcessor is a request processing component, which is responsible for processing JSON-RPC messages and eventually invokes the corresponding api via reflection.



11.4 4.Transaction Flow

After the introduction of namespace, hyperchain node's transaction execution flow has been changed. The client can send the transaction to the corresponding namespace. After the transaction is passed to the hyperchain platform, the json rpc service will first conduct preliminary work on parameter checking and signature verifying. After that, the json rpc service forwards the request to the NamespaceManager. The NamespaceManager will send the transaction to the specific namespace according to the namespace field in the request. Transactions between different namespaces can execute concurrently. The corresponding namespace will call requestProcessor to process transactions, and firstly check the parameters of the request. If there is no problem, the corresponding processing function is called by reflection and the result could be returned.



CHAPTER 12

Cryptography Algorithm

12.1 1.Overview

Hyperchain blockchain platform to support multi-level cryptography encryption to ensure data security, the use of the following cryptographic algorithms to ensure data security issues:

- Elliptic curve digital signature:secp256k1,secp256r1
- Symmetric encryption algorithm:3DES,AES
- Key exchange:ECDH
- Hash:Keccak-256

12.2 2. Elliptic curve digital signature

Elliptic Curve Digital Signature Algorithm (ECDSA) is a simulation of Digital Signature Algorithm (DSA) using [Elliptic Curve Cryptography (ECC). ECDSA became ANSI standard in 1999 and became the IEEE and NIST standards in 2000. It was accepted by the ISO in 1998 and some of the other standards that include it are also under ISO's consideration. Unlike the discrete logarithm problem (DLP) and the integer factorization problem (IFP), the elliptic curve discrete logarithm problem (ECDLP) has no solution to the subexponential time. Therefore, the unit bit strength of elliptic curve cryptography is higher than that of other public key systems. Elliptic curve graphics as shown below:

The Hyperchain blockchain uses the secp256k1 curve to sign and verify the platform transaction to ensure the correctness and completeness of the transaction. At the same time, the platform supports the use of secp256r1 curve to sign messages between nodes to verify the integrity and correctness of message communication between nodes. Node message signature is pluggable, you can open it through the configuration file.the configuration file in the namespace.toml:

```
[encryption.check]
sign = true  #enable Signature
```

That is, when `sign = true`, you need to verify the signature of messages between nodes, otherwise you do not need to verify.

12.3 3.Symmetric encryption algorithm

The Hyperchain blockchain platform supports both AES and 3DES symmetric encryption algorithms to ensure ciphertext transfer between nodes.

3DES, also known as Triple DES, is a mode of the DES encryption algorithm that uses three 56-bit keys to encrypt 3DES data three times. The Data Encryption Standard (DES) is a well-established encryption standard in the United States that uses symmetric key cryptography. DES uses a 56-bit key and cryptographic block method, whereas in the cryptographic block method, the text is divided into 64 Bit-sized text blocks are then encrypted. 3DES is safer than the original DES. Encryption algorithm, which is specifically implemented as follows: Let Ek () and Dk () represent the encryption and decryption process of the DES algorithm, K represents the key used by the DES algorithm, M represents the plaintext, and C represents the ciphertext:

```
3DES Encryption Process: C=Ek3(Dk2(Ek1(M)))
3DES Decryption Process: M=Dk1(EK2(Dk3(C)))
```

AES, also known as Advanced Encryption Standard, is a block encryption standard used by the U.S. federal government. The AES algorithm is based on permutation and permutation operations, and AES uses several different methods to perform permutation and permutation operations. AES is an iterative, symmetric-key-group password that uses 128, 192, and 256-bit keys and encrypts and decrypts data with 128-bit (16-byte) packets. Unlike public key password use key pairs, symmetric key passwords use the same key to encrypt and decrypt data. The encrypted data returned by the block password has the same number of bits as the input data. Iterative encryption uses a loop structure in which the input data is repeatedly replaced and replaced.

In addition,Hyperchain blockchain platform supports symmetric encryption algorithm configuration option to encrypt node messages, configured under namespace.toml:

```
[encryption.security]
algo = "3des"    # Selective symmetric encryption algorithm (pure,3des or aes)
```

Support pure, 3des and aes three parameters:

- pure:do not perform any encryption, in plain text
- 3des:3des encryption and decryption
- aes:aes encryption and decryption

12.4 4.Key exchange algorithm

ECDH is a combination of ECC and DH algorithms for key negotiation. The exchange parties can negotiate a key without sharing any secrets. ECC is a cryptosystem based on the discrete logarithm problem of elliptic curve. Given a point P and an integer k on the elliptic curve, it is easy to solve `Q = kP`. Given a point P, Q, we know that `Q = kP` , Find the integer k is indeed a problem. ECDH is built on this math puzzle.

In addition, Hyperchain exchanges keys when the node handshake for the first time and generates shared keys for each other. This key is the symmetric encryption key between the nodes.

12.5 5.Hash algorithm

Hash algorithm, Hyperchain platform using Keccak256 algorithm for Hash calculation, the results of the settlement for the signature of the summary, but also can be used for address calculation.

CHAPTER 13

JSON-RPC API

13.1 1. JSON-RPC Overview

JSON-RPC is a stateless, light-weight remote procedure call(RPC) protocol. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses JSON (RFC 4627) as data format. A rpc call is represented by sending a Request object to a Server. The Request object has the following members:

jsonrpc A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

method A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.

params A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null and Numbers SHOULD NOT contain fractional parts.

When a rpc call is made, the Server MUST reply with a Response object. The Response object has the following members:

jsonrpc A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

result This member is REQUIRED on success. This member MUST NOT exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server.

error This member is REQUIRED on error. This member MUST NOT exist if there was no error triggered during invocation. The value for this member MUST be an Object including `code` field and `message` field.

id This member is REQUIRED. It MUST be the same as the value of the id member in the Request Object. If there was an error in detecting the id in the Request object (e.g. Parse error/Invalid Request), it MUST be Null.

13.2 2. Hyperchain JSON-RPC API Design

Hyperchain JSON-RPC API consists of seven services:

1. Transaction service, the method name prefix is `"tx"`.
2. Contract service, the method name prefix is `"contract"`.
3. Block service, the method name prefix is `"block"`.
4. Archive service, the method name prefix is `"archive"`.
5. Event subscription service, the method name prefix is `"sub"`.
6. Node service, the method name prefix is `"node"`.
7. Cert service, the method name prefix is `"cert"`.

Hyperchain JSON-RPC API design bases on JSON-RPC 2.0 specification, all the requests are POST. The Request object has the following members:

- `jsonrpc`: A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".
- `namespace`: A String specifying namespace name. This request will be sent to this namespace to handle.
- `method`: A String containing the name of the method to be invoked. The format is: [service prefix]_[method name].
- `params`: A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.
- `id`: An identifier established by the Client that MUST contain a String, Number.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"block_latestBlock","namespace":"global",
↪"params":[],"id":1}' localhost:8081
```

The Response object has the following members:

- `jsonrpc`: A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".
- `namespace`: A String specifying which namespace this response is sent from.
- `code`: Status code. If successful, the value is 0, others status code see the form below。
- `message`: Status message. If successful, the value is SUCCESS, otherwise, the value is error detail message.
- `result`: The data returned by the method invoked on the Server. This member doen't exist if there was an error invoking the method.
- `id`: The same as the value of the id member in the Request Object.

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.4",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
```

(continues on next page)

(continued from previous page)

```
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c
↪",
    "transactions": [
      {
        "version": "1.4",
        "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
        "blockNumber": "0x3",
        "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
        "txIndex": "0x0",
        "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
        "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
        "amount": "0x0",
        "timestamp": 1481778652973000000,
        "nonce": 3573634504790373,
        "extra": "",
        "executeTime": "0x2",
        "payload":
↪"0x81053a7000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪"
      }
    ]
  }
}
```

If the method call succeeds, the members returned including: `jsonrpc`, `namespace`, `id`, `code`, `message`, `result`. And the `code` value is 0, `message` value is SUCCESS, so the client can determine the success of the method call through the values of these two fields.

If the method call failed, the members returned including: `jsonrpc`, `namespace`, `id`, `code`, `message`. And the `code` is non-zero value, `message` value is error message. Status code definition:

code	implication
0	Request successfully
-32700	Parse error, invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.
-32600	Invalid request, the JSON sent is not a valid Request object.
-32601	The method does not exist / is not available.
-32602	Invalid method parameter(s).
-32603	Internal JSON-RPC error.
-32000	Internal Hyperchain error or the node does not install solidity environment.
-32001	The data for the query does not exist.
-32002	The account is out of balance.
-32003	Invalid transaction signature.
-32004	Contract deploy failed.
-32005	Contract invoke failed.
-32006	System is too busy.
-32007	duplicate transaction was received by the server.
-32008	Not enough permission to operate contract.
-32009	The (contract) account does not exist.
-32010	The namespace does not exist.
-32011	No block generated on the chain. It may return When querying latest block.
-32012	The event client subscribed does not exist. (*Reserved code*)
-32013	It returns when making snapshot or data archive happen unexpected error.
-32098	Invalid TCert or missing TCert when sending request to node.
-32099	Failed to get TCert.

13.3 3. JSON-RPC Methods

13.3.1 Transaction

- *tx_getTransactions*

- *tx_getDiscardTransactions*
- *tx_getTransactionByHash*
- *tx_getTransactionByBlockHashAndIndex*
- *tx_getTransactionByBlockNumberAndIndex*
- *tx_getTransactionsCount*
- *tx_getTxAvgTimeByBlockNumber*
- *tx_getTransactionReceipt*
- *tx_getBlockTransactionCountByHash*
- *tx_getBlockTransactionCountByNumber*
- *tx_getSignHash*
- *tx_getTransactionsByTime*
- *tx_getDiscardTransactionsByTime*
- *tx_getBatchTransactions*
- *tx_getBatchReceipt*

13.3.2 Contract

- *contract_compileContract*
- *contract_deployContract*
- *contract_invokeContract*
- *contract_getCode*
- *contract_getContractCountByAddr*
- *contarct_maintainContract*
- *contract_getStatus*
- *contract_getCreator*
- *contract_getCreateTime*
- *contract_getDeployedList*

13.3.3 Block

- *block_latestBlock*
- *block_getBlocks*
- *block_getBlockByHash*
- *block_getBlockByNumber*
- *block_getAvgGenerateTimeByBlockNumber*
- *block_getBlocksByTime*
- *block_getGenesisBlock*
- *block_getChainHeight*

- *block_getBatchBlocksByHash*
- *block_getBatchBlocksByNumber*

13.3.4 Subscription

- *sub_newBlockSubscription*
- *sub_newEventSubscription*
- *sub_getLogs*
- *sub_newSystemStatusSubscription*
- *sub_getSubscriptionChanges*
- *sub_unSubscription*

13.3.5 Node

- *node_getNodes*
- *node_getNodeHash*
- *node_deleteVP*
- *node_deleteNVP*

13.3.6 Certificate

- *cert_getTCert*

13.4 4. JSON-RPC API Reference

13.4.1 tx_getTransactions

Returns a list of transactions in blocks from start block number to end block number.

Parameters

1. `<Object>`

- `from`: `<blockNumber>` - Start block number.
- `to`: `<blockNumber>` - End block number.

`from` must be less than or equal `to`, otherwise returns error.

Type `<blockNumber>` can be:

- Decimal integer.
- Hex string.
- The string `"latest"` for the latest block.

Returns

1. `[<Transaction>]` - the valid Transaction object has the following members:

- `version`: `<string>` - Platform version number.
- `hash`: `<string>`, 32 Bytes - Hash of the transaction.
- `blockNumber`: `<string>` - Block number where this transaction was in.
- `blockHash`: `<string>`, 32 Bytes - Hash of the block where this transaction was in.
- `txIndex`: `<string>` - Transaction index in the block.
- `from`: `<string>`, 20 Bytes - Address of the sender.
- `to`: `<string>`, 20 Bytes - Address of the receiver.
- `amount`: `<string>` - Transfer amount.
- `timestamp`: `<number>` - The unix timestamp for when the transaction was generated.
- `nonce`: `<number>` - 16-bit random number.
- `extra`: `<string>` - Extra information of this transaction.
- `executeTime`: `<string>` - The time it takes to execute this transaction (ms).
- `payload`: `<string>` - The data send along with deploying contract, invoking contract and upgrading contract.

Example1: Nomal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactions", "params": [{"from": 1, "to": 2}], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "result": [
    {
      "version": "1.0",
      "hash": "0x88d5b325dc9042ff92a9fa26ed8c943719bb049ac7022abd09bb85da36f531e4",
      "blockNumber": "0x2",
      "blockHash": "0xc6418753c28ad6d744cb4bbe689521696ba65ad010ce24056b6f8def9fc5cdd5
↪",
      "txIndex": "0x0",
      "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1486994814684628715,
      "nonce": 7948317390228704,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x60606040526000805463ffffffff19168155609e908190601e90396000f3606060405260e060020a60003504633ad14a
↪"
    },
```

(continues on next page)

(continued from previous page)


```
    {
      "version": "1.0",
      "hash": "0xf7149a8349f1853d8d713a15935e5059e6f55c2827f0c88f8414dd0402d6760b",
      "blockNumber": "0x1",
      "blockHash": "0x4bab3f9297e737eb197d666a2f08219f94460ace08a8e1ecad87e6e52183bcd5
↪",
      "txIndex": "0x0",
      "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1486994799163184948,
      "nonce": 2099818402815731,
      "extra": "",
      "executeTime": "0x7",
      "payload":
↪"0x6060604052600080546ffffffff19168155609e908190601e90396000f3606060405260e060020a60003504633ad14a
↪"
    }
  ]
}
```


Example2: Block does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactions", "params": [{"from": 1, "to": 2}], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32602,
  "message": "block number 1 is out of range, and now latest block number is 0"
}
```

13.4.2 tx_getDiscardTransactions

Returns all the invalid transactions.

Parameters

none

Returns

1. `[<Transaction>]` - the invalid Transaction object has the following members:

- `version`: `<string>` - Platform version number.
- `hash`: `<string>`, 32 Bytes - Hash of the transaction.
- `from`: `<string>`, 20 Bytes - Address of the sender.

- `to`: `<string>`, 20 Bytes - Address of the receiver.
- `amount`: `<string>` - Transfer amount.
- `timestamp`: `<number>` - The unix timestamp for when the transaction was generated.
- `nonce`: `<number>` - 16-bit random number.
- `extra`: `<string>` - Extra information of this transaction.
- `payload`: `<string>` - The data send along with deploying contract, invoking contract and upgrading contract.
- `invalid`: `<boolean>` - The flag of invalid transaction.
- `invalidMsg`: `<string>` - Invalid message of this transaction.

For invalid transaction, `invalid` value is true, and `invalidMsg` has following situations:

- **DEPLOY_CONTRACT_FAILED** - Contract deploy failed;
- **INVOKE_CONTRACT_FAILED** - Contract invoke failed;
- **SIGFAILED** - The transaction signature is invalid;
- **OUTOFBALANCE** - Transfer of account is out of balance;
- **INVALID_PERMISSION** - Not enough permission to operate this contract;

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getDiscardTransactions", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "",
      "hash": "0x100ff931204d149f88c0778f6e7b8d4b11ba3c8c720f0cc3e204b46999954ed4",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1482405417011000000,
      "nonce": 6848885244669098,
"extra": "",
      "payload": "0x6060604052600260005600256",
      "invalid": true,
      "invalidMsg": "DEPLOY_CONTRACT_FAILED"
}
  ]
}
```

Example2: There is no invalid transactions

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getDiscardTransactions", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": -32001,
  "message": "Not found discard transactions "
}
```

13.4.3 tx_getTransactionByHash

Returns the information about a transaction by transaction hash.

Parameters

1. `<string>`, 32 Bytes - Hash of a transaction.

Returns

1. `[<Transaction>]` - the Transaction object has the following members:

- `version`: `<string>` - Platform version number.
- `hash`: `<string>`, 32 Bytes - Hash of the transaction.
- `blockNumber`: `<string>` - Block number where this transaction was in.
- `blockHash`: `<string>`, 32 Bytes - Hash of the block where this transaction was in.
- `txIndex`: `<string>` - Transaction index in the block.
- `from`: `<string>`, 20 Bytes - Address of the sender.
- `to`: `<string>`, 20 Bytes - Address of the receiver.
- `amount`: `<string>` - Transfer amount.
- `timestamp`: `<number>` - The unix timestamp for when the transaction was generated.
- `nonce`: `<number>` - 16-bit random number.
- `extra`: `<string>` - Extra information of this transaction.
- `executeTime`: `<string>` - The time it takes to execute this transaction (ms).
- `payload`: `<string>` - The data send along with deploying contract, invoking contract and upgrading contract.
- `invalid`: `<boolean>` - The flag of invalid transaction.
- `invalidMsg`: `<string>` - Invalid message of this transaction.

For invalid transaction, `invalid` value is true, and `invalidMsg` has following situations:

- **DEPLOY_CONTRACT_FAILED** - Contract deploy failed;
- **INVOKE_CONTRACT_FAILED** - Contract invoke failed;
- **SIGFAILED** - The transaction signature is invalid;
- **OUTOFBALANCE** - Transfer of account is out of balance;
- **INVALID_PERMISSION** - Not enough permission to operate this contract;

Example1: Query valid transaction

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getTransactionByHash", "params":[
↪"0xe652e25e617c5f193b240c0d8ff1941a8cfb1d15434eb3830892b7a8389730aa"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe652e25e617c5f193b240c0d8ff1941a8cfb1d15434eb3830892b7a8389730aa",
    "blockNumber": "0x4",
    "blockHash": "0x6ea0c80c1532c273c124511e364fc0a9225e0d129e53249f8e26752ee7d7d989",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x0000000000000000000000000000000000000000",
    "amount": "0x0",
    "timestamp": 1482405601747000000,
    "nonce": 6788653222523786,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6060604052600080556002d8060146000396000f3606060405260e060020a6000350463be1c766b8114601c575b60025
↪"
  }
}
```

Example2: Query invalid transaction

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getTransactionByHash","params":[
↪"0x1f6dc4c744ce5e8a39e6a19f19dc27c99d7efd8e38061e80550bf5e7ab1060e1"],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
```

(continues on next page)

(continued from previous page)


```
  "result": {
    "version": "1.3",
    "hash": "0x1f6dc4c744ce5e8a39e6a19f19dc27c99d7efd8e38061e80550bf5e7ab1060e1",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x0000000000000000000000000000000000000000",
    "amount": "0x0",
    "timestamp": 1509448178302000000,
    "nonce": 1166705097783423,
    "extra": "",
    "payload":
↪"0x6060604052600080553415601257600080fd5b5b6002600090815580fd5b5b5b60918061002d6000396000f300606060
↪",
    "invalid": true,
    "invalidMsg": "DEPLOY_CONTRACT_FAILED"
  }
}
```


Example3: The transaction requested does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getTransactionByHash", "params":["␣
↪0x0e707231fd779779ce25a06f51aec60faed8bf6907e6d74fb11a3fd585831a7e"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": -32001,
  "message": "Not found transaction␣
↪0x0e707231fd779779ce25a06f51aec60faed8bf6907e6d74fb11a3fd585831a7e"
}
```

13.4.4 tx_getTransactionByBlockHashAndIndex

Returns information about a transaction by block hash and transaction index position.

Parameters

1. `<string>`, 32 Bytes - Hash of a block.
2. `<number>` - Transaction index position. This value can be decimal integer or hex string.

Returns

1. `<Transaction>` - the members of Transaction object see *Valid Transaction*.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsByBlockHashAndIndex","params": [
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9", 0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe81d39df11779c7f83e6073cc659c7ee85708c135b6557d318e765b9f938c02f",
    "blockNumber": "0x2",
    "blockHash": "0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "amount": "0x0",
    "timestamp": 1481767474717000000,
    "nonce": 8054165127693853,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000007b0000000000000000000000000
↪"
  }
}
```

Example2: The block requested does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsByBlockHashAndIndex","params": [
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9", 0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32001,
  "message": "Not found block␣
↪0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9"
}
```

13.4.5 tx_getTransactionByBlockNumberAndIndex

Returns information about a transaction by block number and transaction index position.

Parameters

1. <blockNumber> - Block number. See type *Block Number*.
2. <number> - Transaction index position. This value can be decimal integer or hex string.

Returns

1. <Transaction> - the members of Transaction object see *Valid Transaction*.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":   "tx_
↪getTransactionByBlockNumberAndIndex", "params": [2,0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe81d39df11779c7f83e6073cc659c7ee85708c135b6557d318e765b9f938c02f",
    "blockNumber": "0x2",
    "blockHash": "0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "amount": "0x0",
    "timestamp": 1481767474717000000,
    "nonce": 8054165127693853,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000007b0000000000000000000000000
↪"
  }
}
```

Example2: The block requested does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":   "tx_
↪getTransactionsByBlockNumberAndIndex", "params": [2,0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32602,
```

(continues on next page)

(continued from previous page)

```
  "message": "block number 2 is out of range, and now latest block number is 0"
}
```

13.4.6 tx_getTransactionsCount

Returns the number of transactions on the chain.

Parameters

none

Returns

1. <Object>

- count: <string> - The number of transactions.
- timestamp: <number> - The unix timestamp for response (ns).

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsCount", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 71,
  "code": 0,
  "message": "SUCCESS":,
  "result": {
    "count": "0x9",
    "timestamp": 1480069870678091862
   }
}
```

13.4.7 tx_getTxAvgTimeByBlockNumber

Returns the average execution time of all transactions in the given block number.

Parameters

1. <Object>

- from: <blockNumber> - Start block number. See type *Block Number*.
- to: <blockNumber> - End block number. See type *Block Number*.

from must be less than or equal to, otherwise returns error.

Returns

1. `<string>` - the average execution time of all transactions (ms).

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTxAvgTimeByBlockNumber", "params": [{"from":10, "to":19}], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xa9"
}
```

13.4.8 tx_getTransactionReceipt

Returns the receipt of a transaction by transaction hash.

Parameters

1. `<string>`, 32 Bytes - Hash of a transaction.

Returns

1. `<Receipt>` - the Receipt object has the following members:

- `version`: `<string>` - Platform version number.
- `txHash`: `<string>`, 32 Bytes - Hash of the transaction.
- `vmType`: `<string>` - The execution engine type used by this transaction execution, this value is `EVM` OR `JVM`.
- `contractAddress`: `<string>`, 20 Bytes - The contract address created, if the transaction was a contract creation, otherwise `0x0000000000000000000000000000000000000000`.
- `gasUsed`: `<number>`- The amount of gas used by this specific transaction alone.
- `ret`: `<string>` - Contract compling code OR contract execution results.
- `log`: `[<Log>]` - Array of Log. The Log object has following members:
  - `address`: `<string>`, 20 Bytes - Contract address by which this event log is generated.
  - `topics`: `[<string>]` - Array of 32 Bytes string. Topics are order-dependent. Each topic can also be an array of string with "or" options. The first topic is the unique identity of the event.
  - `data`: `<string>` - Log message or data.
  - `blockNumber`: `<number>` - Block number where this transaction was in.

  - `blockHash`: `<string>`, 32 Bytes - Hash of the block where this transaction was in.
  - `txIndex`: `<number>` - Transaction index position in the block.
  - `index`: `<number>` - Event log index position in all logs generated in this transaction.

If the transaction requested has not been confirmed, the error code returned is **-32001**. If an error occurs during the transaction processing, the error may be:

- **OUTOFBALANCE** - Transfer of account is out of balance, code is **-32002**;
- **SIGFAILED** - The transaction signature is invalid, code is **-32003**;
- **DEPLOY_CONTRACT_FAILED** - Contract deploy failed, code is **-32004**;
- **INVOKE_CONTRACT_FAILED** - Contract invoke failed, code is **-32005**;
- **INVALID_PERMISSION** - Not enough permission to operate this contract, code is **-32008**;

Example1: The transaction has not been confirmed

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x0e0758305cde33c53f8c2b852e75bc9b670c14c547dd785d93cb48f661a2b36a "],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": -32001,
  "message": "Not found receipt by␣
↪0x0e0758305cde33c53f8c2b852e75bc9b670c14c547dd785d93cb48f661a2b36a"
}
```

Example2: Deploying contract failed

For this example, we use the following contract to recreate the situation:

```
contract TestContractor{
    int length = 0;

    modifier justForTest(){
        length = 2;
        throw;
        _;
    }
    function TestContractor()justForTest{
    }

    function getLength() returns(int){
        return length;
    }
}
```

We use contract compiled code as the value of `payload` in *contract_deployContract* , then the deploying contract request is as follows:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_deployContract", "params":[{
"from":"17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"nonce":7021040367249265,
"payload":
↪"0x6060604052600060005534600057 5b6002600055600056 5b5b5b603f8061002660003960 00f3606060405260e060020a
↪",
"timestamp":1487042279126000000,
"signature":
↪"0xfc1cb1986dd4ee4a5f8d8238e2f7bac1866aad235d587eb641d76270bf686418310ab7d42dc0f2575aa858a88ae7732
↪"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x33aef7e6bad2ae27c23a8ab44f56aef87042f1f0b02e1b0ee5e8a304705292a6"
}
```

Next, trying to get information about receipt of this transaction by transaction hash, an error will be returned:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x33aef7e6bad2ae27c23a8ab44f56aef87042f1f0b02e1b0ee5e8a304705292a6"],"id":1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":-32004,
  "message":"DEPLOY_CONTRACT_FAILED"
}
```

Example3: Invoking contract failed

For this example, we use the following contract to recreate the situation:

```
contract TestContractor{
    int length = 0;

    modifier justForTest(){
        length = 2;
        throw;
        _;
    }
    function TestContractor(){
    }

    function getLength()justForTest returns(int){
        return length;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

We invoke the method `getLength()`, then the invoking contract request is as follows:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params": [{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"to":"0xaeccd2fd1118334402c5de1cb014a9c192c498df",
"timestamp":1487042517534000000,
"nonce":2472630987523856,
"payload":"0xbe1c766b",
"signature":
↪"0x8c56f025610dd9cb3f4ac346d35978639a536505527b7593d87f3b45c35328637280995ed32f6a6809069da915740b3
↪"}],"id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x5233d18f46e9c1ed49dbdeb4273c1c1e0eb176efcedf6edb6d9fa59d33d02fee "
}
```

Next, trying to get information about receipt of this transaction by transaction hash, an error will be returned:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x5233d18f46e9c1ed49dbdeb4273c1c1e0eb176efcedf6edb6d9fa59d33d02fee"],"id":1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":-32005,
  "message":"INVOKE_CONTRACT_FAILED"
}
```

Example4: Invalid transaction signature

In this example, we use Example3 request example, but change the last letter "c" of param `from` to "0" in order to simulate the situation of illegal signature, then the invoking contract request is as follows:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params": [{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bff0",
"to":"0xaeccd2fd1118334402c5de1cb014a9c192c498df",
"timestamp":1481872888621000000,
"nonce":9206467481004664,
```

(continues on next page)

(continued from previous page)

```
"payload":"0xbe1c766b",
"signature":
↪"0x57dfa7f2c2d8c762c9c0e5ef7b1c4dda84b584f36799ab751891c8dc553862145f64d1991441c9460481af4e4231db3
↪"}],"id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "id": "SUCCESS",
  "result":"0x621d09cd9d5e9027d9b82c5e1fd911ac31297775dbb0c4dab6c6fcd64310fe23"
}
```

Next, trying to get information about receipt of this transaction by transaction hash, an error will be returned:

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x621d09cd9d5e9027d9b82c5e1fd911ac31297775dbb0c4dab6c6fcd64310fe23"],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": -32003,
  "message": " SIGFAILED "
}
```

Example5

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionReceipt", "params":[
↪"0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.3",
    "txHash": "0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059",
    "vmType": "EVM",
    "contractAddress": "0x0000000000000000000000000000000000000000",
    "ret": "0x0",
    "log": [
      {
        "address": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "topics": [
```

(continues on next page)

(continued from previous page)

```
↪"0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
        ],
    "data": "0000000000000000000000000000000000000000000000000000000000000064",
        "blockNumber": 2,
        "blockHash":
↪"0x0c14a89b9611f7f268f26d4ce552de966bebba4aab6aaea988022f3b6817f61b",
        "txHash": "0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059
↪",
        "txIndex": 0,
    "index": 0
      }
    ]
  }
}
```

13.4.9 tx_getBlockTransactionCountByHash

Returns the number of transactions in a block from a block matching the given block hash.

Parameters

1. <string>, 32 Bytes - Hash of a block.

Returns

1. <string> - The number of transactions in a block.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getBlockTransactionCountByHash", "params":  [
↪"0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xaf5"
}
```

Example2: The block requested does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getBlockTransactionCountByHash", "params":  [
↪"0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32001,
  "message":"Not found block␣
↪0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"
}
```

13.4.10 tx_getBlockTransactionCountByNumber

Returns the number of transactions in a block from a block matching the given block number.

Parameters

1. `<blcokNumber>` - Block number. See type *Block Number*.

Returns

1. `<string>` - The number of transactions in a block.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":" tx_
↪getBlockTransactionCountByNumber", "params":  ["0x2"], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xaf5"
}
```

Example2: The block requested does not exist

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":" tx_
↪getBlockTransactionCountByNumber", "params":  ["0x2"], "id": 71}'

# Response
```

(continues on next page)

(continued from previous page)

```
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32602,
  "message": "block number 0x2 is out of range, and now latest block number is 0"
}
```

13.4.11 tx_getSignHash

Returns transaction content hash string used to sign a transaction by client.

Parameters

1. <Object>

- from: <string>, 20 Bytes - Address of the sender.
- to: <string>, 20 Bytes - [optional] Address of the receiver(account address OR contract address). If it's a contract deployment transaction, needn't specify this member.
- nonce: <number> - 16-bit random number.
- extra: <string> - [optional] Extra information of this transaction.
- value: <string> - [optional] Transfer amount. Value can not be empty if it's a normal transfer transaction, otherwise, you needn't specify this member.
- payload: <string> - [optional] Payload can not be empty if it's a contract deployment transaction(See *contract_deployContract*), a contract invoke transaction(See *contract_invokeContract*) or a contract upgrade transaction(See *contract_maintainContract*), otherwise, you needn't specify this member.
- timestamp: <number> - The unix timestamp for when the transaction was generated.

Returns

1. <string> - A hash string.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getSignHash", "params":[{
    "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "nonce":5373500328656597,
    "payload":
↪"0x6060604052600080546
↪",
    "timestamp":1487771157166000000 }],"id":"1"}'

# Response
{
  "jsonrpc": "2.0",
```

(continues on next page)

(continued from previous page)

```
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x2e6a644a4ca6a9daba4444995dc0dda039208e642df11db35438d18e7c3b13c3"
}
```

13.4.12 tx_getTransactionsByTime

Returns a list of valid transactions generated at specific time periods.

Parameters

1. <Object>

- startTime: <number> - The start unix timestamp.
- endTime: <number> - The end unix timestamp.

Returns

1. [<Transaction>] - the members of Transaction object see *Valid Transaction*.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionsByTime","params":[{"startTime":1, "endTime":1581776001230590326}],"id
↪":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [{
    "version": "1.0",
    "hash": "0xbd441c7234e3b83a05c89ed5d548c3d1877306975e271a08e7354d74e45431bc",
    "blockNumber": "0x1",
    "blockHash": "0xa6a4b2df16c7bdeb578aa7de7b05f9b54d96202bdc8414196741842834156ebd",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x0000000000000000000000000000000000000000",
    "amount": "0x0",
    "timestamp": 1481767468349000000,
    "nonce": 1775845467490815,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6060604052346100005575b6101e18061001860003960000f3606060405260e060020a60003504636fd7cc16811461002
↪"
```

(continues on next page)

(continued from previous page)

```
  }]
}
```

Example2: There is no data

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getTransactionsByTime","params":[{
↪"startTime":1681776001230590326, "endTime":1681776001230590326}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": []
}
```

13.4.13 tx_getDiscardTransactionsByTime

Returns a list of invalid transactions generated at specific time periods.

Parameters

1. `<Object>`

- `startTime`: `<number>` - The start unix timestamp.
- `endTime`: `<number>` - The end unix timestamp.

Returns

1. `[<Transaction>]` - the members of invalid Transaction object see *Invalid Transaction*.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" tx_
↪getDiscardTransactionsByTime","params":[{"startTime":1, "endTime
↪":1581776001230590326}],"id":1}'

# Result
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
```

(continues on next page)

(continued from previous page)


```
    {
      "version": "1.3",
      "hash": "0x4e468969d94b92622e385246779d05981ef43869b17c8afedc7e6b5b138ae807",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
      "amount": "0x1",
      "timestamp": 1501586411342000000,
      "nonce": 4563214039387098,
      "extra": "",
      "payload": "0x0",
      "invalid": true,
      "invalidMsg": "OUTOFBALANCE"
    }
  ]
}
```


13.4.14 tx_getBatchTransactions

Returns a list of transactions by a list of specific transaction hash.

Parameters

1. `<Object>`

- `hashs`: `[<string>]` - Array of 32 Bytes string, a list of transaction hash.

Returns

1. `[<Transaction>]` - Array of Transaction object, the members of Transaction object see *Valid Transaction*.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getBatchTransactions","params":[{
    "hashes":["0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602"]
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "hash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
      "blockNumber": "0x2",
      "blockHash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b
↪",
```

(continues on next page)

(continued from previous page)


```
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "amount": "0x0",
      "timestamp": 1509440823410000000,
      "nonce": 8291834415403909,
      "extra": "",
      "executeTime": "0x6",
      "payload": "0x0a9ae69d"
    },
    {
      "version": "1.3",
      "hash": "0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602",
      "blockNumber": "0x1",
      "blockHash": "0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0
↪",
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1509440820498000000,
      "nonce": 5098902950712745,
      "extra": "",
      "executeTime": "0x11",
      "payload":
↪"0x60606040523415610000f57600080fd5b6040516040806101608339810160405280805191906020018051915050505b600
↪"
    }
  ]
}
```


13.4.15 tx_getBatchReceipt

Returns a list of receipt of transactions by a list of specific transaction hash.

Parameters

1. `<Object>`

- `hashs`: `[<string>]` - Array of 32 Bytes string, a list of transaction hash.

Returns

1. `[<Receipt>]` - Array of Receipt object, the Receipt object see type *Receipt*.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"tx_getBatchReceipt","params":[{
    "hashes":["0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602"]
}],"id":1}'
```

(continues on next page)

(continued from previous page)

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "txHash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
      "vmType": "EVM",
      "contractAddress": "0x0000000000000000000000000000000000000000",
      "ret": "0x0000000000000000000000000000000000000000000000000000000000000005",
      "log": []
    },
    {
      "version": "1.3",
      "txHash": "0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602",
      "vmType": "EVM",
      "contractAddress": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "ret":
↪"0x606060405263ffffffff7c0100000000000000000000000000000000000000000000000000000000600350416630a9a
↪",
      "log": []
    }
  ]
}
```

13.4.16 contract_compileContract

Compiles contract and returns compiled solidity code and abi definition.

Parameters

1. `<string>` - The source code.

Returns

1. `<Object>`

- `abi`: `[<string>]` - The contract abi definition.
- `bin`: `[<string>]` - The compiled solidity code.
- `types`: `[<string>]` - The contract name.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"contract_
↪compileContract", "params":["contract Accumulator{    uint32 sum = 0;   function␣
↪increment(){         sum = sum + 1;     }      function getSum() returns(uint32){  ␣
↪      return sum;     }   function add(uint32 num1,uint32 num2) {         sum =␣
↪sum+num1+num2;     } }"],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "abi": [
          "[{\"constant\":false,\"inputs\":[{\"name\":\"num1\",\"type\":\"uint32\"},{\
↪"name\":\"num2\",\"type\":\"uint32\"}],\"name\":\"add\",\"outputs\":[],\"payable\
↪":false,\"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"getSum\
↪",\"outputs\":[{\"name\":\"\",\"type\":\"uint32\"}],\"payable\":false,\"type\":\
↪"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"increment\",\"outputs\
↪":[],\"payable\":false,\"type\":\"function\"}]"
          ],
    "bin": [

↪"0x6060604052600080546463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633ad14a
↪"
          ],
    "types": [
          "Accumulator"
          ]
  }
}
```

13.4.17 contract_deployContract

Returns a transaction hash after deploying contract.

Parameters

1. `<Object>`

- `from`: `<string>`, 20 Bytes - Address of the creator.
- `nonce`: `<number>` - 16-bit ramdom value.
- `extra`: `<string>` - [optional] Extra information of this transaction.
- `timestamp`: `<number>` - The unix timestamp for when the transaction was generated.
- `payload`: `<string>` - For **solidity** contract, if the constructor of contract need parameters, `payload` value is the complied solidity code and encoded parameters. For **java** contract, `payload` value is the byte stream after the class file and the configuration file are compressed.
- `signature`: `<string>` - The signature of transaction.
- `type`: `<string>` - [optional, default `"EVM"`] The execution engine type used by this transaction execution, this value is `EVM` OR `JVM`.

Returns

1. <string>, 32 Bytes - Hash of the transaction.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global",  "method":
↪"contract_deployContract", "params":[{
"from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"nonce":5373500328656597,   "payload":
↪"0x60606040526000805463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633ad14a
↪",
"signature":
↪"0x388ad7cb71b1281eb5a0746fa8fe6fda006bd28571cbe69947ff0115ff8f3cd00bdf2f45748e0068e49803428999280c
↪",
"timestamp":1487771157166000000
}],"id":"1"}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x406f89cb205e136411fd7f5befbf8383bbfdec5f6e8bcfe50b16dcff037d1d8a"
}
```

13.4.18 contract_invokeContract

Returns a transaction hash after invoking contract.

Parameters

1. <Object>

- from: <string>, 20 Bytes - Address of the account invoked the contract.
- to: <string>, 20 Bytes - Address of the contract. You can get contract address through *tx_getTransactionReceipt* after deploying contract.
- nonce: <number> - 16-bit random value.
- extra: <string> - [optional] Extra information of this transaction.
- timestamp: <number> - The unix timestamp for when the transaction was generated.
- payload: <string> - The hash of the invoked method signature and encoded parameters.
- signature: <string> - The signature of transaction.
- simulate: <boolean> - [optional, default false] Determines whether the transaction requires consensus or not, if true, no consensus.
- type: <string> - [optional, default "EVM"] The execution engine type used by this transaction execution, this value is EVM OR JVM.

Returns

1. <string>, 32 Bytes - Hash of the transaction.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params":[{
"from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"nonce":5019420501875693,
"payload":
↪"0x3ad14af3000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000
↪",
"signature":
↪"0xde467ec4c0bd9033bdc3b6faa43a8d3c5dcf393ed9f34ec1c1310b0859a0ecba15c5be4480a9ad2aaaea8416324cb54e
↪",
"simulate":false,
"timestamp":1487773188814000000,
"to":"0x313bbf563991dc4c1be9d98a058a26108adfcf81"
}],"id":"1"}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

13.4.19 contract_getCode

Returns the compiled contract code by the given contract address.

Parameters

1. <string> - The address of contract.

Returns

1. <string> - contract code.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"contract_
↪getCode","params": ["0xaeccd2fd1118334402c5de1cb014a9c192c498df"],"id": 1}'

# Response
```

(continues on next page)

(continued from previous page)

```
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":
↪"0x6060604052361561005657 60e060020a600035046301000dd7811461005b5780638e739461146100e55780638f24d79
↪"
}
```

13.4.20 contract_getContractCountByAddr

Returns the number of contract that has been deployed by given account address.

Parameters

1. <string>, 20 Bytes - The address of account.

Returns

1. <string> - The number of contract.

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "contract_
↪getContractCountByAddr", "params": ["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b"],
↪"id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x3"
}
```

13.4.21 contarct_maintainContract

Upgrade contract, freeze contract and unfreeze contract.

Note: Only contract deployers have the authority to upgrade contract, freeze contract and unfreeze contract.

Parameters

1. `<Object>`

- `from`: `<string>`, 20 Bytes - Address of the account.
- `to`: `<string>`, 20 Bytes - Address of the contract. You can get contract address through *tx_getTransactionReceipt* after deploying contract.
- `nonce`: `<number>` - 16-bit random value.
- `extra`: `<string>` - [optional] Extra information of this transaction.
- `timestamp`: `<number>` - The unix timestamp for when the transaction was generated.
- `payload`: `<string>` - [optional] *Only upgrade contract operation need to specify this member.* Represents the new contract compiled code.
- `signature`: `<string>` - The signature of transaction.
- `type`: `<string>` - [optional, default `"EVM"`] The execution engine type used by this transaction execution, this value is `EVM` OR `JVM`.
- `opcode`: This value may be:
  - `1`: Upgrade contract.
  - `2`: Freeze contract.
  - `3`: Unfreeze contract.

Returns

1. `<string>`, 32 Bytes - Hash of the transaction.

Example1: Upgrade contract

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
"timestamp":1481767474717000000,
"nonce": 8054165127693853,
"payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000007b000000000000000000000000
↪",
"signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fca63
↪",
"opcode": 1}],"id": 1}'


# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
```

(continues on next page)

(continued from previous page)

```
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

Example2: Freeze contract

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
    "from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "timestamp":1481767474717000000,
    "nonce": 8054165127693853,
    "signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fca639
↪",
    "opcode": 2}],
"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

Example3: Unfreeze contract

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
"timestamp":1481767474717000000,
"nonce": 8054165127693853,
"signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fca639
↪",
"opcode": 3}],"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

13.4.22 contract_getStatus

Returns status of a contract by contract address.

Parameters

1. <string>, 20 Bytes - The address of contract.

Returns

1. <string> - Status of the contract, this value may be:
   - normal: Normal status.
   - frozen: The contract has been frozen.
   - non-contract: The given address is not a contract address. It may be a normal account address.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getStatus","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"],"id":␣
↪1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":" normal"
}
```

13.4.23 contract_getCreator

Returns the address of contract creator.

Parameters

1. <string>, 20 Bytes - The address of contract.

Returns

1. <string>, 20 Bytes - The address of contract creator.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getCreator","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"],"id
↪": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":" 0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd "
}
```

13.4.24 contract_getCreateTime

Returns the date and time a contract was created.

Parameters

1. <string>, 20 Bytes - The address of contract.

Returns

1. <string> - The date and time of contract created.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getCreateTime","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"],
↪"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"2017-04-07 12:37:06.152111325 +0800 CST"
}
```

13.4.25 contract_getDeployedList

Returns a list of deployed contract address by account address.

Parameters

1. <string>, 20 Bytes - The address of contract creator.

Returns

1. [<string>] - a list of deployed contract address.

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getDeployedList","params": ["0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd"],
↪"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"]
}
```

13.4.26 block_latestBlock

Returns information about the latest block.

Parameters

none

Returns

1. <Block> - The Block object has the following members:

- version: <string> - Platform version number.
- number: <string> - The block number.
- hash: <string>, 32 Bytes - Hash of the block.
- parentHash: <string>, 32 Bytes - Hash of the parent block.
- writeTime: <number> - The unix timestamp for when the block was written.
- avgTime: <string> - The average time it takes to execute transactions in the block (ms).
- txCounts: <string> - The number of transactions in the block.
- merkleRoot: <string> - Merkle tree root hash.
- transactions: [<Transaction>] - The list of transactions in the block. The Transaction object see *Valid Transaction*.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪latestBlock","params":[],"id":71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c
↪",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a7000000000000000000000000000000000000000000000000000000000000000004000000000000000000000000
↪"
      }
    ]
  }
}
```

Example2: There is no block on the chain

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪latestBlock","params":[],"id":71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
```

(continues on next page)

(continued from previous page)

```
  "id": 1,
  "code": -32602,
  "message": "There is no block generated!"
}
```

13.4.27 block_getBlocks

Returns a list of blocks from start block number to end block number.

Parameters

1. `<Object>`

- `from`: `<blockNumber>` - Start block number. See type *Block Number*.
- `to`: `<blockNumber>` - End block number. See type *Block Number*.
- `isPlain`: `<boolean>` - [optional, default `false`] If `true` it returns block excluding transactions, if `false` it returns block including transactions.

`from` must be less than or equal `to`, otherwise returns error.

Returns

1. `[<Block>]` - array of Block, the members of Block object see *Block*.

Example1: Returns block including transactions

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlocks", "params": [{"from":2,"to":3}], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.0",
      "number": "0x3",
      "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
      "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "writeTime": 1481778653997475900,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
      "transactions": [
```

(continues on next page)

(continued from previous page)


```
        {
      "version": "1.0",
      "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
      "blockNumber": "0x3",
      "blockHash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914
↪",
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "amount": "0x0",
      "timestamp": 1481778652973000000,
      "nonce": 3573634504790373,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x81053a70000000000000000000000000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪"
    }
      ]
    },
    {
      "version": "1.0",
      "number": "0x2",
      "hash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "parentHash":
↪"0xe287c62aae77462aa772bd68da9f1a1ba21a0d044e2cc47f742409c20643e50c",
      "writeTime": 1481778642328872960,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
      "transactions": [
        {
      "version": "1.0",
      "hash": "0x07d606a25d1eab009f5374950383e9c0697599e6c35999337b969ba356800168",
      "blockNumber": "0x2",
      "blockHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "amount": "0x0",
      "timestamp": 1481778641306000000,
      "nonce": 1628827117185765,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000000000000303a0000000000000000000000000
↪"
    }
      ]
    }
  ]
}
```

Example2: Returns block excluding transactions

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlocks", "params": [{"from":2,"to":3,"isPlain":true}], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.0",
      "number": "0x3",
      "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
      "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "writeTime": 1481778653997475900,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
    },
    {
      "version": "1.0",
      "number": "0x2",
      "hash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "parentHash":
↪"0xe287c62aae77462aa772bd68da9f1a1ba21a0d044e2cc47f742409c20643e50c",
      "writeTime": 1481778642328872960,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
    }
  ]
}
```

13.4.28 block_getBlockByHash

Returns information about a block by hash.

Parameters

1. <string>, 32 Bytes - Hash of a block.
2. <boolean> - If true it returns block excluding transactions, if false it returns block including transactions.

Returns

1. <Block> - the members of Block object see *Block*.

Example1: Returns block including transactions

```
# Request
curl -X POST -data  '{"jsonrpc":"2.0","namespace":"global","method":"block_
↪getBlockByHash","params":[
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914", false],"id":1}
↪'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0"
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c
↪",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a700000000000000000000000000000000000000000000000000000000000000004000000000000000000000000
↪"
      }
    ]
  }
}
```

Example2: Returns block excluding transactions

```
# Request
curl -X POST -data  '{"jsonrpc":"2.0","namespace":"global","method":"block_
↪getBlockByHash","params":[
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914", true],"id":1}'

# Response
```

(continues on next page)

(continued from previous page)

```
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
  }
}
```

13.4.29 block_getBlockByNumber

Returns information about a block by number.

Parameters

1. `<blockNumber>` - The block number. See type *Block Number*.
2. `<boolean>` - If `true` it returns block excluding transactions, if `false` it returns block including transactions.

Returns

1. `<Block>` - the members of Block object see *Block*.

Example1: Returns block including transactions

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlockByNumber", "params": ["0x3", false], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
```

(continues on next page)

(continued from previous page)


```
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c
↪",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a7000000000000000000000000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪"
      }
    ]
  }
}
```


Example2: Returns block excluding transactions

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlockByNumber", "params": ["0x3", true], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9
↪",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
  }
}
```

13.4.30 block_getAvgGenerateTimeByBlockNumber

Returns the average generation time of all blocks for the given block number.

Parameters

1. <Object>

- from: <blockNumber> - Start block number. See type *Block Number*.
- to: <blockNumber> - End block number. See type *Block Number*.

Returns

1. <string> - the average generation time of all blocks (ms).

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪getAvgGenerateTimeByBlockNumber","params": [{"from": 10, "to": 19}],"id":71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0x32"
}
```

13.4.31 block_getBlocksByTime

Returns the number of blocks generated at specific time periods.

Parameters

1. <Object>

- startTime: <number> - The start unix timestamp.
- endTime: <number> - The end unix timestamp.

Returns

1. <Object>

- sumOfBlocks: <string> - The number of blocks.
- startBlock: <string> - The start block number.
- endBlock: <string> - The end block number.

Example1: Normal request

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"block_
↪getBlocksByTime","params":[{"startTime":1481778635567920177, "endTime
↪":1481778653997475900}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "sumOfBlocks": "0x3",
    "startBlock": "0x1",
    "endBlock": "0x3"
  }
}
```

Example2: Start unix timestamp and end unix timestamp are both more than written unix timestamp of the latest block.

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"block_
↪getBlocksByTime","params":[{"startTime":1481778635567920177, "endTime
↪":1481778653997475900}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "sumOfBlocks": "0x0",
    "startBlock": null,
    "endBlock": null
  }
}
```

13.4.32 block_getGenesisBlock

Returns current genesis block number.

Parameters

none

Returns

1. <string> - The genesis block number.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getGenesisBlock","params":[],
↪"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x8"
}
```

13.4.33 block_getChainHeight

Returns the current chain height.

Parameters

none

Returns

1. <string> - The latest block number.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getChainHeight","params":[],"id
↪":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x11"
}
```

13.4.34 block_getBatchBlocksByHash

Returns a list of blocks by a list of specific block hash.

Parameters

1. `<Object>`

- `hashes`: `[<string>]` - Array of block hash.
- `isPlain`: `<boolean>` - If `true` it returns block excluding transactions, if `false` it returns block including transactions.

Returns

1. `[<Block>]` - Array of Block object, the members of Block object see *Block*.

Example1: Returns block including transactions

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByHash","params
↪":[{
    "hashes":["0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b"]
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x3",
      "hash": "0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
      "parentHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "writeTime": 1509448178829111592,
      "avgTime": "0x0",
      "txcounts": "0x0",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
```

(continues on next page)

(continued from previous page)

```
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
      "version": "1.3",
      "hash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
          "blockNumber": "0x2",
          "blockHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
          "amount": "0x0",
          "timestamp": 1509440823410000000,
          "nonce": 8291834415403909,
          "extra": "",
          "executeTime": "0x6",
          "payload": "0x0a9ae69d"
        }
      ]
    }
  ]
}
```

Example2: Returns block excluding transactions

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByHash","params
↪":[{
    "hashes":["0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b"],
    "isPlain": true
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x3",
      "hash": "0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
      "parentHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "writeTime": 1509448178829111592,
      "avgTime": "0x0",
      "txcounts": "0x0",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    }
  ]
}
```

13.4.35 block_getBatchBlocksByNumber

Returns a list of blocks by a list of specific block number.

Parameters

1. `<Object>`

- `numbers`: `[<blockNumber>]` - Array of block number. See type *Block Number*.
- `isPlain`: `<boolean>` - If `true` it returns block excluding transactions, if `false` it returns block including transactions.

Returns

1. `[<Block>]` - Array of Block object, the members of Block object see *Block*.

Example1: Returns block including transactions

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByNumber","params
↪":[{
    "numbers": ["1","2"]
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x1",
      "hash": "0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
```

(continues on next page)

(continued from previous page)


```
      "writeTime": 1509440821032039312,
      "avgTime": "0x11",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
          "version": "1.3",
          "hash": "0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602
↪",
          "blockNumber": "0x1",
          "blockHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0x0000000000000000000000000000000000000000",
          "amount": "0x0",
          "timestamp": 1509440820498000000,
          "nonce": 5098902950712745,
          "extra": "",
          "executeTime": "0x11",
          "payload":
↪"0x60606040523415610000f57600080fd5b60405160408061016083398101604052808051919060200180519150505b600
↪"
        }
      ]
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
          "version": "1.3",
          "hash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4
↪",
          "blockNumber": "0x2",
          "blockHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
          "amount": "0x0",
          "timestamp": 1509440823410000000,
          "nonce": 8291834415403909,
          "extra": "",
      "executeTime": "0x6",
      "payload": "0x0a9ae69d"
        }
      ]
```


(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

Example2: Returns block excluding transactions

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByNumber","params
↪":[{
    "numbers": ["1","2"],
    "isPlain": true
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x1",
      "hash": "0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
      "writeTime": 1509440821032039312,
      "avgTime": "0x11",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    }
  ]
}
```

13.4.36 sub_newBlockSubscription

To subscribe a new block event and create filter to notify client. The information of new block will be cached in filter when a new block is generated.

Parameters

1. <boolean> - If true, the filter will return the full object *Block* , if false it will return block hash.

Returns

1. <string> - Subscription ID.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪newBlockSubscription","params":[false],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.37 sub_newEventSubscription

To subscribe a new VM event and create filter to notify client. The VM event log will be cached when a VM event is triggered.

Parameters

1. <Object>

- fromBlock: <number> - [optional, default no limit] Start block number. This block number should be more than or equal to current latest block number.
- toBlock: <number> - [optional, default no limit] End block number. This block number is the future block number that is more than the start block number.
- addresses: [<string>] - [optional, default no limit] Array of 20 Bytes string, indicates that listen to event generated by specified address of contract.
- topics: [<string>][<string>] - [optional, default no limit] two-dimensional array of string, topics which the incoming message's topics should match. You can use the following combinations:
  - [A, B] = A && B
  - [A, [B, C]] = A && (B || C)
  - [null, A, B] = ANYTHING && A && B null works as a wildcard

Returns

1. <string> - Subscription ID.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪newEventSubscription","params":[{
    "fromBlock":100,
    "addresses": ["000f1a7a08ccc48e5d30f80850cf1cf283aa3abd"]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.38 sub_getLogs

Returns the eligible VM event log.

Parameters

1. `<Object>`

- `fromBlock`: `<number>` - [optional, default `0`] Start block number. This block number shouldn't be less than the genesis block number.
- `toBlock`: `<number>` - [optional, default `the latest block number`] End block number. This block number shouldn't be more than current latest block number.
- `addresses`: `[<string>]` - [optional, default no limit] Array of 20 Bytes string, indicates that listen to event generated by specified address of contract.
- `topics`: `[<string>][<string>]` - [optional, default no limit] two-dimensional array of string, topics which the incoming message's topics should match. You can use the following combinations:
  - `[A, B] = A && B`
  - `[A, [B, C]] = A && (B || C)`
  - `[null, A, B] = ANYTHING && A && B` `null` works as a wildcard

Returns

1. `[<Log>]` - the Log object has the following members:
   - `address`: `<string>`, 20 Bytes - Contract address by which this event log is generated.
   - `topics`: `[<string>]` - Array of 32 Bytes string. Topics are order-dependent. Each topic can also be an array of string with "or" options. The first topic is the unique identity of the event.
   - `data`: `<string>` - Log message or data.
   - `blockNumber`: `<number>` - Block number where this transaction was in.

- `blockHash`: `<string>`, 32 Bytes - Hash of the block where this transaction was in.
- `txHash`: `<string>`, 32 Bytes - Hash of the transaction where this log was in.
- `txIndex`: `<number>` - Transaction index position in the block.
- `index`: `<number>` - Event log index position in all logs generated in this transaction.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_getLogs",
↪"params":[{
    "addresses": ["0x313bbf563991dc4c1be9d98a058a26108adfcf81"]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":[
    {
      "address":"0x313bbf563991dc4c1be9d98a058a26108adfcf81",
      "topics":["0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"],
      "data":"0000000000000000000000000000000000000000000000000000000000000064",
      "blockNumber":4,
      "blockHash":"0xee93a66e170f2b20689cc05df27e290613da411c42a7bdfa951481c08fdefb16
↪",
      "txHash":"0xa676673a23f33a95a1a5960849ad780c5048dff76df961e9f78329b201670ae2",
      "txIndex":0,
      "index":0
    }
  ]
}
```

13.4.39 sub_newSystemStatusSubscription

To subscribe a new system status event and create filter to notify client. The information of system status will be cached when a new system status event is thrown.

Parameters

1. `<Object>`

- `modules`: `[<string>]` - [optional, default no limit] Array of string, modules of system status which the client wants to subscribe. For example, **p2p**, **consensus**, **executor** and so on.
- `modules_exclude`: `[<string>]` - [optional, default no limit] Array of string, modules of system status which the client doesn't want to subscribe.
- `subtypes`: `[<string>]` - [optional, default no limit] Array of string, which type of system status information under the module the client wants to subscribe. For example, **viewchange**.

- `subtypes_exclude: [<string>]` - [optional, default no limit] Array of string, which type of system status information under the module the client doesn't want to subscribe.
- `error_codes: [<number>]` - [optional, default no limit] Array of number, specific status information under the module the client wants to subscribe.
- `error_codes_exclude: [<number>]` - [optional, default no limit] Array of number, specific status information under the module the client doesn't want to subscribe.

Returns

1. `<string>` - Subscription ID.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪newSystemStatusSubscription","params":[{
    "modules":["executor", "consensus"],
    "subtypes": ["viewchange"],
    "error_codes_exclude": [-1, -2]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.40 sub_getSubscriptionChanges

Polling method for filters. Returns new messages since the last call of this method.

Parameters

1. `<string>` - Subscription ID.

Returns

1. `<Array>` - Array of messages received since last poll.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪getSubscriptionChanges","params":["0x7e533eb0647ecbe473ae610ebdd1bba6"], "id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":{}
}
```

13.4.41 sub_unSubscription

To unsubscribe a subscription with given id. This method should be called when watch is no longer needed.

Parameters

1. `<string>` - Subscription ID.

Returns

1. `<boolean>` - `true` if the subscription was successfully unsubscribe, otherwise `false`.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪unsubscription","params":["0x7e533eb0647ecbe473ae610ebdd1bba6"], "id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":true,
}
```

13.4.42 node_getNodes

Returns information of all nodes.

Parameters

none

Returns

1. `[<PeerInfo>]` - Array of PeerInfo object, the PeerInfo object has following members:

- `id`: `<number>` - The node ID.
- `ip`: `<string>` - IP address of the node.
- `port`: `<number>` - GRPC port of the node.
- `namespace`: `<string>` - Which namespace the node is in.
- `hash`: `<string>` - Hash of the node.
- `hostname`: `<string>` - Hostname of the node.
- `isPrimary`: `<bool>` - If `true` represents the node is a primary node, otherwise not.
- `isvp`: `<bool>` - If `true` represents the node is a VP node, otherwise not.
- `status`: `<number>` - Status of the node. This value may be:
  - `0`: Alive status.
  - `1`: Pending status.
  - `2`: Stop status.
- `delay`: `<number>` - Represents the delay time(ns) between this node and requested node. If this value is `0`, it represents this PeerInfo is the information of local node.

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "node_getNodes
↪", "params": [],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "id": 1,
      "ip": "127.0.0.1",
      "port": "50011",
      "namespace": "global",
      "hash": "fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a",
      "hostname": "node1",
      "isPrimary": true,
      "isvp": true,
      "status": 0,
      "delay": 0
    },
    {
      "id": 2,
      "ip": "127.0.0.1",
      "port": "50012",
```

(continues on next page)

(continued from previous page)


```
      "namespace": "global",
      "hash": "c82a71a88c58540c62fc119e78306e7fdbe114d9b840c47ab564767cb1c706e2",
      "hostname": "node2",
      "isPrimary": false,
      "isvp": true,
      "status": 0,
      "delay": 347529
    },
    {
      "id": 3,
      "ip": "127.0.0.1",
      "port": "50013",
      "namespace": "global",
      "hash": "0c89dc7d8bdf45d1fed89fdbac27463d9f144875d3d73795f64f35dc204480fd",
      "hostname": "node3",
      "isPrimary": false,
      "isvp": true,
      "status": 0,
      "delay": 369554
    },
    {
      "id": 4,
      "ip": "127.0.0.1",
      "port": "50014",
      "namespace": "global",
      "hash": "34d299742260716bab353995fe98727004b5c27bde52489f61de093176e82088",
      "hostname": "node4",
      "isPrimary": false,
      "isvp": true,
      "status": 0,
      "delay": 430356
    }
  ]
}
```


13.4.43 node_getNodeHash

Return hash of the requested node.

Parameters

none

Returns

1. `<string>` - hash of the node.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"node_
↪getNodeHash","params":[],"id":1}'
```

(continues on next page)

(continued from previous page)

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"
}
```

13.4.44 node_deleteVP

To disconnect from a connected VP peer.

Parameters

1. <Object>

- nodehash: <string> - Hash of the VP peer to disconnect.

Returns

1. <string> - A message indicates whether the request is sent successfully or not.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"node_deleteVP
↪","params":[{"nodehash":
↪"c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete vp node, hash␣
↪c82a71a88c58540c62fc119e78306e7fdbe114d9b840c47ab564767cb1c706e2"
}
```

13.4.45 node_deleteNVP

VP node disconnects from connected NVP peer by hash of NVP peer.

Parameters

1. <Object>

- nodehash: <string> - Hash of NVP peer to disconnect.

Returns

1. <string> - A message indicates whether the request is sent successfully or not.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","namespace":"global", "method":"node_deleteNVP
↪","params":[{"nodehash":
↪"c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete nvp node, hash␣
↪050ba6f3a19a4aca46adf51f0d46cf822b0b97f50014207a2b8d8535f5da7aa8"
}
```

13.4.46 cert_getTCert

Returns the tcert certificate issued by the node to the client.

Parameters

1. <Object>

- pubkey: <string> - Public key(pem format).

Returns

1. <Object>

- tcert: <string> - tcert certificate.

Example: Getting tcert certificate failed

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"cert_getTCert
↪","params":[{
"pubkey":
↪"2d2d2d2d2d424547494e20454344534341205055424c4943204b45592d2d2d2d2d0a424a4b73413554414d2b5763446c7935
↪"}],"id":"1"}'
```

(continues on next page)

(continued from previous page)

```
# Response
{
   "jsonrpc": "2.0",
   "namespace":"global",
   "id": "1",
   "code": -32099,
   "message": "signed tcert failed"
}
```

CHAPTER 14

Node Operation

14.1 1. Add Node

Adding nodes requires the configuration of the following three files:

- `peerconfig.toml`: Peer profile, used to configure connection information of logical peer.
- `hosts.toml`: Host profile, used to configure information of physical peer, including the mapping between the hostname of node and the IP address.
- `addr.toml`: Address profile, used to configure physical interconnection address information, including the mapping of domain name and IP address. This file configuration is required when the peer connects in reverse to local node.

14.1.1 Example1: On the basis of four VP nodes, add the fifth VP node

The new VP node needs to be configured with the following configuration files before it starts:

1. `peerconfig.toml`

```
[[nodes]]
  hostname = "node1"
  id = 1
  static = true

[[nodes]]
  hostname = "node2"
  id = 2
  static = true

[[nodes]]
  hostname = "node3"
  id = 3
  static = true
```

(continues on next page)

(continued from previous page)

```
[[nodes]]
  hostname = "node4"
  id = 4
  static = true

[[nodes]]
  hostname = "node5"
  id = 5
  static = true

[self]
  caconf = "config/namespace.toml"
  hostname = "node5"
  id = 5         # The node ID
  n = 5          # The number of VP to connect to
  new = true     # Whether this is a new node to attend or not
  org = false    # Whether this is a original node or not
  rec = false    # Whether this node needs to be reconnected
  vp = true      # Whether this is a primary node or not
```

2. `hosts.toml`

This file needs to configure **the physical addresses of all the nodes to be connected**. The hyperchain nodes communicate with each other through the hostname, so the hostname can be configured as an arbitrary node address.

```
hosts = [
"node1 127.0.0.1:50011",
"node2 127.0.0.1:50012",
"node3 127.0.0.1:50013",
"node4 127.0.0.1:50014",
"node5 127.0.0.1:50015"
]
```

3. `addr.toml`

addr is declared in the form of a domain. For example, if two nodes with hostname of `node1` and `node5` belong to `domain1` and all other nodes belong to different domains, the configuration file should be configured as follows (this configuration file is `addr.toml` of `node5`):

```
addrs = [
"domain1 127.0.0.1:50015",
"domain2 192.168.100.20:50015",
"domain3 202.110.20.13:50015",
"domain4 127.0.0.1:50015",
]
domain = "domain1"
```

Note: To reiterate, `addr.toml` is a configuration to let other peer know the node domain and nodes in different domains are interconnected by different network address, which allows nodes to connect between complex network segments.

14.1.2 Example2: On the basis of four VP nodes, add a NVP node

The new NVP node needs to be configured with the following configuration files before it starts:

1. peerconfig.toml

```
[[nodes]]
  hostname = "node1"
  id = 1
  static = true

[[nodes]]
  hostname = "node2"
  id = 2
  static = true

[[nodes]]
  hostname = "node3"
  id = 3
  static = true

[[nodes]]
  hostname = "node4"
  id = 4
  static = true

[self]
  caconf = "config/namespace.toml"
  hostname = "node5"
  id = 0          # This value must be 0 for NVP node
  n = 4          # The number of VP to connect to
  new = true     # Whether this is a new node to attend or not
  org = false    # Whether this is a original node or not
  rec = false    # Whether this node needs to be reconnected
  vp = false     # Whether this is a primary node or not
```

2. hosts.toml

```
hosts = [
"node1 127.0.0.1:50011",
"node2 127.0.0.1:50012",
"node3 127.0.0.1:50013",
"node4 127.0.0.1:50014",
"node5 127.0.0.1:50015"
]
```

3. addr.toml

```
addrs = [
"domain1 127.0.0.1:50015",
"domain2 127.0.0.1:50015",
"domain3 127.0.0.1:50015",
"domain4 127.0.0.1:50015",
"domain5 127.0.0.1:50015"
]
domain = "domain5"
```

14.2 2. Delete Node

We divide deleting nodes into three cases:

1. VP disconnects from one VP;
2. VP proactively disconnects from one NVP;
3. NVP proactively disconnects from one VP;

The second and third results are the same, so that the NVP can no longer synchronize the VP data and the NVP no longer forwards the transaction to the VP.

In the following example, we assume that each node's JSON-RPC API service port mapping is as follows:

- Node 1: `8081`
- Node 2: `8082`
- Node 3: `8083`
- Node 4: `8084`
- Node 5: `8085`

14.2.1 Example1: VP disconnects from one VP

For example, there are currently five VP nodes and now delete VP node 5.

Firstly, get the hash of the VP node 5 to be deleted.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"
}
```

Then, the request of deleting node is sent to VP node 1, 2, 3, 4 and 5 respectively.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteVP","params":[{"nodehash":
↪"55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"}],"id":1,
↪"namespace":"global"}' localhost:8081/8082/8083/8084/8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
```

(continues on next page)

(continued from previous page)

```
  "result": "successful request to delete vp node, hash␣
↪55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"
}
```

When you see the following logs on the terminal, the VP node is deleted successfully.

```
global::p2p 12:33:17.709 DELETE NODE␣
↪55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc
global::p2p 12:33:17.709 delete validate peer 5
```

14.2.2 Example2: VP disconnects from a NVP

For example, there are currently four VP nodes and one NVP node which is connected to VP node 1.

Firstly, get the hash of the NVP node.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"
}
```

Then, send a request of deleting NVP to VP node 1.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteNVP","params":[{"nodehash":
↪"4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"}],"id":1,
↪"namespace":"global"}' localhost:8081

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete nvp node, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"
}
```

The following logs are seen at the terminal of the VP node 1,

```
global::p2p 13:28:02.857 delete NVP peer, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad, vp pool size(4)␣
↪nvp pool size(0)
```

At the same time, the NVP node also prints the following log notes that the VP node 1 has been disconnected.

```
global::p2p 13:28:02.858 peers_pool.go:244 delete validate peer 1
```

The NVP node is deleted successfully.

14.2.3 Example3: NVP disconnects from a VP

The situation is similar to that of Example2, except that we send a request for deleting a node to the NVP node this time.

For example, there are currently four VP nodes and one NVP node which is connected to VP node 1.

Firstly, get the hash of the VP node 1.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8081

# Response
{
    "jsonrpc": "2.0",
    "namespace": "global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"
}
```

Then, send a request of deleting VP to NVP node.

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteVP","params":[{"nodehash":
↪"fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"}],"id":1,
↪"namespace":"global"}' localhost:8085

# Response
{
    "jsonrpc": "2.0",
    "namespace": "global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "successful request to delete vp node, hash␣
↪fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"
}
```

The following logs are seen at the terminal of the NVP node,

```
global::p2p 13:47:17.744 delete validate peer 1
```

At the same time, the VP node 1 also prints the following log,

```
global::p2p 13:47:17.744 delete NVP peer, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad, vp pool size(4)␣
↪nvp pool size(0)
```

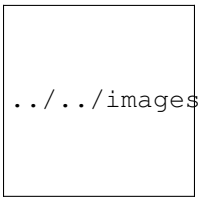
The VP node is deleted successfully.

CHAPTER 15

Developers' Guide

This document describes how to contribute to hyperchain project. It is an entry point for developers who are interested in build, develop, create an issue or pull request to hyperchain.

15.1 Workflow



1. fork

visit https://github.com/hyperchain/hyperchain, click `fork` button

2. clone

We assume that you have `go` v1.8 installed, and `GOPATH` is set.

Create your clone:

```
git clone git@github.com:$user/hyperchain.git
$GOPATH/src/github.com/hyperchain/hyperchain
```

3. fetch && rebase

Get your local master up to date:

```
cd $GOPATH/src/github.com/hyperchain/hyperchain
git fetch upstream
git checkout master
git rebase upstream/master
```

4. branch

Branch from it:

```
git checkout -b myfeature
```

Then edit code on the `myfeature` branch.

5. commit

```
git commit
```

6. push

When ready to review (or just to establish an offsite backup or your work), push your branch to your fork on `github.com`:

```
git push -f ${your_remote_name} myfeature
```

7. create pull request

1. Visit your fork at https://github.com/\protect\T1\textdollaruser/hyperchain
2. Click the `Compare & Pull Request` button next to your `myfeature` branch.

15.2 Building and Testing

build

Run instructions below in the project root directory

```
go build -o hyperchain
```

testing

Run unit tests in the project root directory

```
go test -cpu 4 ./...
```

15.3 Contributing

Each contributor should follow the git workflow. If your use of git is not particularly clear, please read this `document <http://nvie.com/posts/a-successful-git-branching-model/>__` first.

If you would like to contribute to hyperchain, please fork, commit and send a pull request for maintainers to review code and merge to main code base.

Issue

If use want to create a issue, please follow this issue template.

```
#### Environment

hyperchain version: `hyperchain --version`
OS: Windows/Linux/OSX
Commit hash:
```

(continues on next page)

(continued from previous page)

```
#### What happened:

#### What you expected to happen:

#### How to reproduce it (as minimally and precisely as possible):

#### Backtrace

````
[backtrace]
````
```

**Pull request**

- Pull request should be based on hyperchain's `master` branch
- Commit messages should begin with the package name they modify, e.g.
- core/executor: use default parameters to initialize executor
- Please use `go fmt` tool to format the source code before submit changes

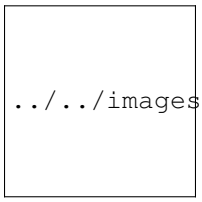
We encourage any form of contribution, no matter it's bug fixes, new features, or something else. Our team will do our best to make every valuable contribution to get merged as soon as possile.

CHAPTER 16

# Hyperchain Roadmap

Currently, hyperchain is available in the open source community with 1.4 stable version, will continue to introduce new features in the future.

Welcome to join the community of Hyperchain to participate in our development!




## 16.1 First community version

This is the first community version of Hyperchain that includes the complete components of the consortium blockchain platform:

1. Consensus engine based on RBFT algorithm;
2. Compatible with Ethereum smart contract virtual machine;
3. Data partition, isolate the business data physically ;
4. Rich event subscription interface, capture blockchain platform events in real-time;
5. Multi-level encryption mechanism, including asymmetric encryption, symmetric encryption, admittance mechanism based on digital certificate;
6. Available level of performance to meet the needs of most business scenarios;

## 16.2 Better smart contract

This version is planned to be released in February 2018.

The main feature is to support the smart contract development in the Java language, reducing the difficulty for blockchain application development.

In addition, there will have a easy-to-use hyperchain docker images for one-click cluster deployments.

## 16.3 Controllable data capacity

Currently, the existing blockchain data is accumulated, so the storage capacity of blockchain data will be a big problem.

We plan to release a version in April 2018 to support archiving of blockchain data and archiving of smart contract data.

## 16.4 Autonomous

The existing consortium blockchain admittance mechanism has the following problems:

1. There is a single point of failure;
2. The entire blockchain network is easy to be controlled by a single node;
3. Low degree of automation;

Therefore, we propose to release a version to support decentralized autonomous admittance mechanism in July 2018 that will enable automated member management, identity switching, version upgrades and more.

## 16.5 Protect your pricacy

Currently in the same partition, blockchain data is shared by all nodes, and there is no user privacy at all. In September 2018, we will release a version that support two advanced cryptographic features to protect user privacy.

The two cryptographic techniques are (1) zero knowledge proof (2) ring signature technique.

## 16.6 Run fast

Currently hyperchain's performance can meet the needs of most business scenarios, but for particularly high-frequency and complex scenarios still do nothing. As a result, we will make major adjustments to the existing architecture to transform hyperchain with microservices and cloud-based architectures so as to improve performance.

This version is scheduled to be released in December 2018.

## 16.7 What is your favourite

In June 2019, we will release a version that supports multiple consensus algorithms to support dynamic consensus engine switching.

Besides we plan to come up with a new consensus algorithm that will support larger node sizes in consortium blockchain.