Dipperin-Core Documentation

Release Master

Keychain Foundation Ltd.

Aug 21, 2019

Development


Dipperin

CHAPTER 1

Quick start

Welcome to use Dipperin. Follow this guide you can run a Dipperin node on Dipperin Testnet.

1.1 Prerequisites

Dipperin uses the Go Programming Language for many of its components.

- Go version 1.11.x is required.
- C compiler

1.2 Building the source

1.2.1 Mac & Linux

Get source code to your go path

```
$ mkdir -p ~/go/src/github.com/dipperin
$ cd ~/go/src/github.com/dipperin
$ git clone git@github.com:dipperin/dipperin-core.git
```

Run tests

```
$ cd ~/go/src/github.com/dipperin/dipperin-core
$ make test
# or
$ ./cs.sh
```

Build dipperin to `~/go/bin`

```
$ cd ~/go/src/github.com/dipperin/dipperin-core
$ make build
```

Build following softwares to ~/go/bin

1. dipperin
2. dipperincli
3. bootnode
4. miner
5. chain_checker

```
$ ./cs.sh install
```

1.2.2 Windows

The Chocolatey package manager provides an easy way to get the required build tools installed. If you don't have chocolatey yet, follow the instructions on https://chocolatey.org to install it first.

Then open an Administrator command prompt and install the build tools we need:

```
C:\Windows\system32> choco install git
C:\Windows\system32> choco install golang
C:\Windows\system32> choco install mingw
```

Use git shell run commands below, and copy source code to your go path

You can't run the tests if you don't put source code in your home folder.

($HOME means home folder, example C:\Users\qydev)

```
$ mkdir -p $HOME\go\src\github.com\dipperin
$ cd $HOME\go\src\github.com\dipperin
$ git clone git@github.com:dipperin/dipperin-core.git
```

Restart cmd and run tests

```
$ cd $HOME\go\src\github.com\dipperin\dipperin-core
$ go test -p 1 ./...
```

Build dipperin to User

```
$ cd $HOME\go\src\github.com\dipperin\dipperin-core\cmd\dipperin
$ go install
```

Build dipperincli to User

```
$ cd $HOME\go\src\github.com\dipperin\dipperin-core\cmd\dipperincli
$ go install
```

1.3 Executables

The dipperin-core project comes with several wrappers/executables found in the cmd directory.

- dipperin

Our chain CLI client. It is the entry point into the Dipperin network, capable of running as a full node. It can be used by other processes as a gateway into the Dipperin network via JSON RPC endpoints exposed on top of HTTP, WebSocket and/or IPC transports.

- dipperincli

Our chain CLI client with console. It has all features of `dipperin`, and provides a easy way to start the node. You can give commands to node in command line console, like starting mining `miner StartMine` or querying current block `chain CurrentBlock`

- bootnode

Stripped down version of our Dipperin client implementation that only takes part in the network node discovery protocol, but does not run any of the higher level application protocols. It can be used as a lightweight bootstrap node to aid in finding peers in private networks.

- miner

Mine block client, It must work with a `mine master` started by `dipperin` or `dipperincli`. `mine master` dispatch sharding works for every miner registered. So all miner do different works when mining a block. |

1.4 Running dipperin

1.4.1 Setting environment variables

- Mac & Linux

```
$ vi ~/.bashrc
```

Add command `export PATH=$PATH:~/go/bin` at bottom of the file, and `:wq` for save and quit the file.

- Windows

(`$HOME` means home folder, example `C:\Users\qydev`)

```
$ set PATH=%PATH%;$HOME\go\bin
```

Going through all the possible command line flags

```
$ dipperin -h
# or
$ dipperincli -h
```

1.4.2 Full node on the main Dipperin network

- Mac & Linux

```
$ boots_env=venus dipperincli
```

- Windows

```
$ set boots_env=venus
$ dipperincli
```

This command will:

- Guide you to setup your personal Dipperin start config, and will write these args to your `$HOME/.dipperin/start_conf.json`, you can change these start args in this file.
- Start sync Dipperin test-net data from other nodes.

1.5 Using command line

The following command is to start a node, which requires a wallet password.

If no wallet path is specified, the default system path is used: `$Home/.dipperin/`.

```
$ dipperincli -- node_type [type] -- soft_wallet_pwd [password]
```

Example:

Local startup miner:

```
$ dipperincli -- node_type 1 -- soft_wallet_pwd 123
```

Local startup miner(start mining):

```
$ dipperincli -- node_type 1 -- soft_wallet_pwd 123 -- is_start_mine 1
```

Local startup verifier:

```
$ dipperincli -- node_type 2 -- soft_wallet_pwd 123
```

1.5.1 Error

If dipperincli started in a wrong way, it may be that the local link data is not synchronized with the link state, and the local link data needs to be deleted:

- Mac & Linux

```
$ cd ~
$ rm .dipperin -fr
```

- Windows

```
$ rd /s /q $HOME\.dipperin
```

restart command line tool

See more details for Command Line Tool

CHAPTER 2

Key Consepts

2.1 Blockchain

2.1.1 What is blockchain?

Blockchain is a distributed system recording and storing transaction records. More specifically, blockchain is a shared, immutable record of peer-to-peer transactions built from linked transaction blocks and stored in a digital ledger.

Blockchain technology originated from Bitcoin. Bitcoin shows an electronic money trading system that does not rely on trustable third parties. This system uses a synchronized ledger distributed across all nodes instead of a single ledger on a traditional central server, replacing the subjective "trust" of intermediaries with an objective "consensus" mechanism, enabling reliable value transfer in a non-trusted environment.

The birth of the blockchain provides a new type of social trust mechanism for human beings, marking the beginning of a truly trustworthy Internet for human beings and forms the foundation of value Internet.

2.1.2 How does blockchain work?

Blockchain is a shared ledgers that records transactions in a peer-to-peer network.

Transactions are collected and recorded into a data structure called block. All the confirmed and validated transaction blocks are linked and chained from the beginning of the chain to the most current block, hence the name blockchain.

A distributed network

The ledger is distributed to all member nodes in the network. Each node maintains their own copy of the ledger.

The decentralized peer-to-peer blockchain network prevents any single participant or group of participants from controlling the underlying infrastructure or undermining the entire system. Participants in the network are all equal, adhering to the same protocols. They can be individuals, state actors, organizations, or a combination of all these types of participants.

Consensus

Anyone in the blockchain network can submit transactions to be stored onto a blockchain and therefore it is important that there is review and confirmation, in the form of a consensus about whether to add those transactions.

Consensus ensures that the shared ledger are exact copies, and lowers the risk of fraudulent transactions. It makes nodes on the blockchain network agree on the same state of blockchain.

2.1.3 What is Dipperin?

Dipperin is a public blockchain founded by the Keychain Foundation Ltd. Dipperin builds a native multi-chain system which supports multi-industry (digital asset, supply) and multi-form (built-in paradigm on the chain, cloud server, applet, side chain, etc.) decentralized applications through tiered smart contract design, high-frequency transaction paradigm and unlicensed business innovation support, combined with multiple technologies. Dipperin has technical features:

- **Original deterministic proof of work consensus mechanism**

  We proposed a deterministic proof of work consensus mechanism. Our protocol separates the block production and verification. It keeps the decentralization level of the bitcoin, shortening the time for reaching consensus, and providing the performance to meet the tens of millions of daily activities (DAU). For more information please refer consensus.

- **Verifiable Random Function(VRF) based verifier sortition mechanism**

  Dipperin has applied a cryptographic sortition for verifiers based on VRF(Verifiable Random Function). This sortition mechanism ensures that only a small percentage of users are selected, and the chance of being selected is propotional to its weight. Random results cannot be predicted in advance and cannot be manipulated by any malicious adversory. Dipperin's sortition mechanism provides objective security, that is, the whole process is objective, and decisions are made entirely through calculations. Human intervention cannot affect this process.

- **Layered smart contract architecture**

  The layered intelligent contract design separates the business logic from the state consensus, and standardizes the cross-chain interface to achieve cross-chain communication at the telecommunications network level while standardizing the burden on the main chain. Dipperin follows the stacking principle, and each layer is designed as Plug-and-Play (plug-and-play), in which the core modules are replaceable, making Dipperin a simpler and more extensive service.

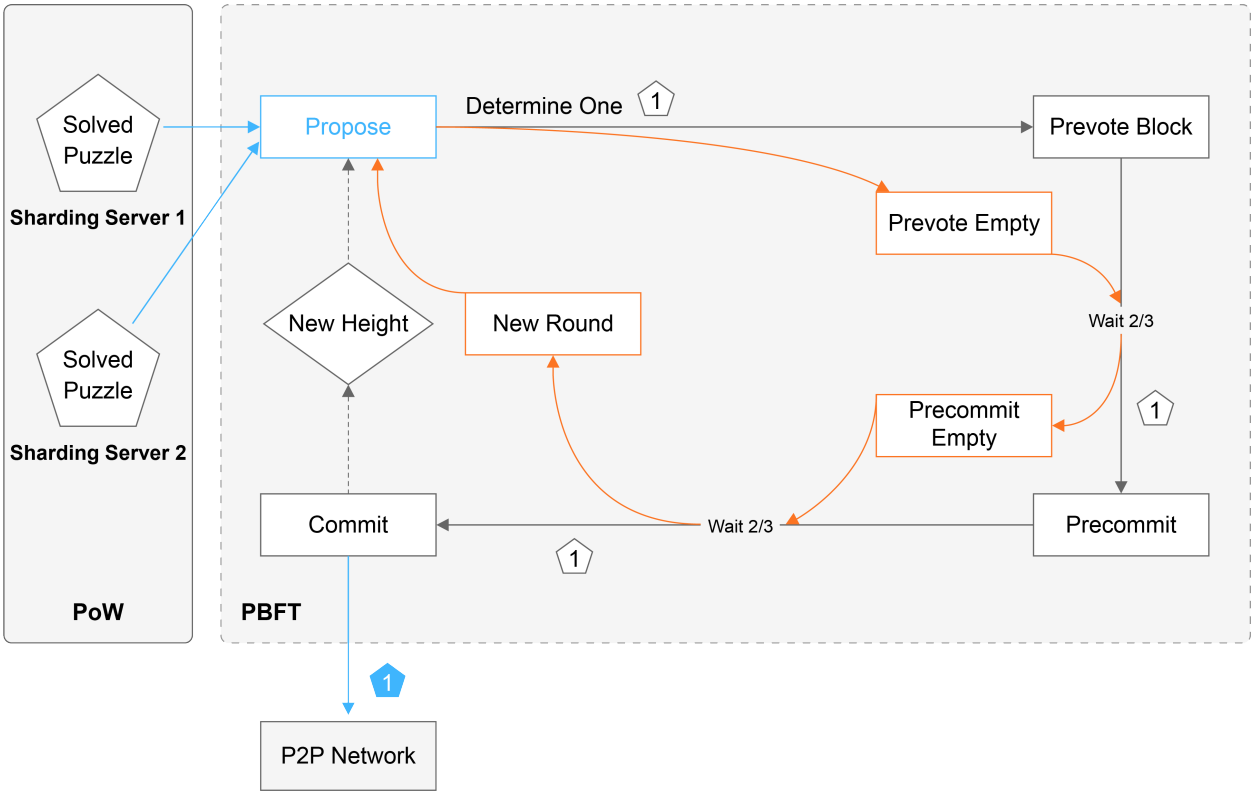
- **Native multi-chain system**

  Dipperin's native multi-chain architecture provides a model for the main chain and sidechain division of labor. Its ultra-light wallet payment verification (CSPV) technology provides strong support for mobile users. CSPV technology allows ultra-light nodes to synchronize data with minimal amounts and does not grow with blockchain growth, and any mobile client or even smart contract code area can support it.

2.2 Consensus

2.2.1 Deterministic Proof of Work

We implemented a new consensus protocol called Deterministic Proof of Work(DPoW) that guarantees the deterministic finality of transactions.



DPoW is a hybrid consensus protocol, it introduces a Map-reduce PoW mining mechanism to work alongside Byzantine Fault Tolerance(BFT) verification. It combines the advantages of both PoW and BFT, allows transactions to be confirmed immediately while keep highly decentralized. It assure strong consistency and security against a multitude of attacks largely because the protocol does not allow forking.

2.2.2 Verifiable Random Function(VRF) based verifier sortition mechanism

Of all consensus algorithms, BFT is particularly characterized by high consistency and efficiency. Nonetheless, at least 2/3 of all nodes in the system should be honest to ensure the safety and liveness of the system. Therefore, it is essential to select honest nodes as verifiers from so vast candidates.

Weighted Users

Reputation is very important in business. Our system quantifies the reputation of users and measures the weight of users by reputation. The verifier is selected by lottery, and the candidate with higher reputation has more chance of being selected. Under hypothesis that 2/3 of the network's reputation is good, the chain's security can be guaranteed. We believe that reputation-based weights are more fair than weighting method based on computing power or based on stocks.

```
Reputation :  Reputation = F(Stake, Performance Nonce)
```

There are three factors for measuring reputation, stake, performance and nonce.

Verifier Sortition

The role of verifier sortition is to select candidates as block proposer or verifier whose identity can be verified by all other users.

The implementation of cryptographic sortition uses VRF: In the VRF algorithm, for a common input, the user can generate a hash value and a proof using its private key SK and a hash function. Other users can use this hash value and

the proof, combined with the user's public key PK, to verify whether the hash value is generated by the user for that input. The hash value determines who would be selected.

In the process, the user's private key is not leaked at all from beginning to end. The user is authenticated in this way, and other users can believe his role as a verifier for a certain period of time. In this way, a set of users can be randomly selected through a common input and their identity can be verified by others.

```
Procedure Sortition(Stake, PerformanceNonce, Seed)
------------------------------------------------------------------------
reputation = Reputation(Stake, PerformanceNonce)

<hash, proof> = VRF(PrivateKey,Seed)

priority = Priority(hash,reputation)

return (priority, proof)
```

The purpose of our introduction of reputation is to make high-credit users more likely to be selected. Whether a certain user can be selected is not a deterministic event. The randomness in the sortition algorithm comes from a publicly known random seed.

Seed

Dipperin produce a public known seed for each height. This seed cannot be controlled by attackers or be predicted in advance; otherwise, an adversary may be able to choose a seed that favors selection of corrupted users.

The seed is produced by miners when pack new blocks. Once the network reaches agreement on the block for height r-1, miners can computer the next seed as `Seed_r, proof := VRF(SK_r-1 proposer , Seed_r-1)`. Each random output of the VRF is unpredictable by any miner until it becomes available to everyone. The winner block of height r would determines the seedr.

Dipperin verifier committee members are replaced each 220 blocks. We call each 220 blocks a "slot". The seed in the last block of Slot `d` (e.g. #299 block of Slot 3), determints the verifier committee members of Slot `d + 2` (e.g. Slot 5).

2.3 Transactions

Transactions are proposals to update the ledger and are collected into blocks.

2.3.1 Transaction data structure

Transaction is created by user and broadcasted into the blockchain network. When the transaction was included in a block and accepted by full nodes, the transactions is confirmed. The data structure of transaction is as below:

```
type transaction struct {
    AccountNonce uint64 // The nonce value of the account who has launched this␣
↪transaction
    Recipient []byte // the counterparty of this transaction which has a length of 22␣
↪bytes where the first 2 bytes are used to mark the type of the transaction and the␣
↪last 20 are the address of the counterparty
    Amount []byte // transaction amount
    Fee []byte // the trasaction fee for the miner
    ExtraData []byte // additional remarks for the transaction, if it is a contract␣
↪transaction,then it's accompanied by the data of the contract operation
    Witness []byte // the signature of the originator
}
```

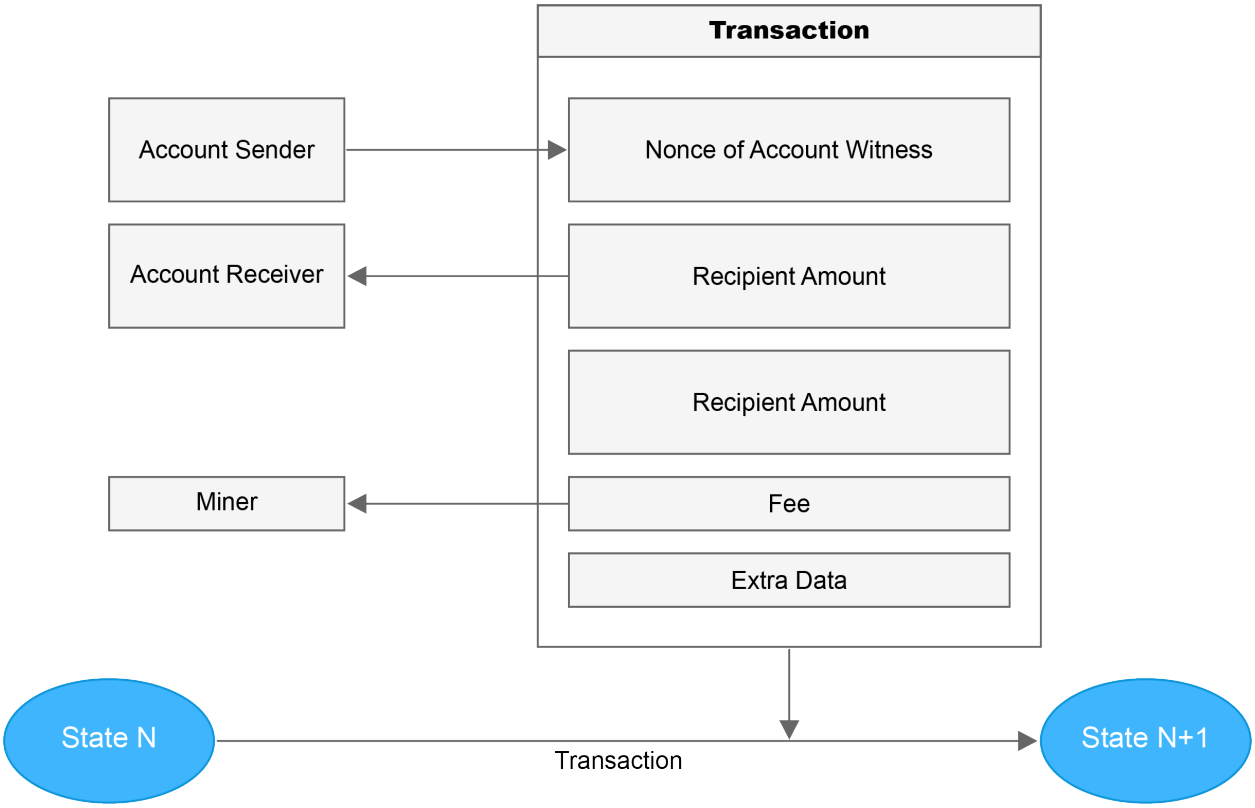
The "AccountNonce" field can avoid a transaction be broadcasted duplicated. And transaction fee would be charged for the usage of computing power and disk space.

2.3.2 Committing transactions

The process of committing a transaction is:

1. User signs off on a transaction from their wallet application to send a certain crypto or token from them to someone else.
2. The transaction is broadcasted by the wallet application.
3. Some miner packed this transaction and submitted to the verifier group.
4. Verifiers consensus on the block, broadcast the result to the rest of the network.
5. Full nodes received the verified block from verifiers, update their ledger.

A transaction is a proposal to update the ledger.



The state of ledger can be transfer from one state to another state by processing a transaction.

State

State of the ledger is the whole states of all accounts.

```
type account struct {
    Nonce uint64 // the total number of transactions originated by this account
    Balance []byte // the balance of this account address
    Stake []byte // the amount of the deposit in this account
    CommitNum uint64 // the total number of messages committed by this account
    Performance uint64 // the performance as verifier of the account, which is an␣
↪important factor of the reputation
```

(continues on next page)

(continued from previous page)

```
    VerifyNum uint64 // the number of blocks that this account should verify in␣
↪total
    LastElect uint64 // the height of last elect transaction. The stake cannot be␣
↪retrieved within 4 periods after the height
    ContractRoot []byte // the root hash of the contract trie created by this␣
↪account
    DataRoot []byte // the root hash of the tree structure of the data generated by␣
↪the application of the account executed off the chain
}
```

There are two kinds of account:

- Normal account
- Contract account

The fields "Stake", "CommitNumber", "Performance", "VerifyNum" and "LastElect" are used by normal accounts to record their verifier status.

The fields "ContractRoot" and "DataRoot" are used by contract accounts.

For example, there is a committed transaction that alice transfer 5 coins to Bob, with 0.05 transaction fee. To process this transaction, balance of Alice would minus 5.05, balance of bob add 5. The rest 0.05 would be distributed to miner and verifiers.

2.3.3 Transaction validity

Transactions would be validated before add into the blockchain. There are two main rules:

- Transaction is digitally signed by the required parties
- Output state of the transaction is valid

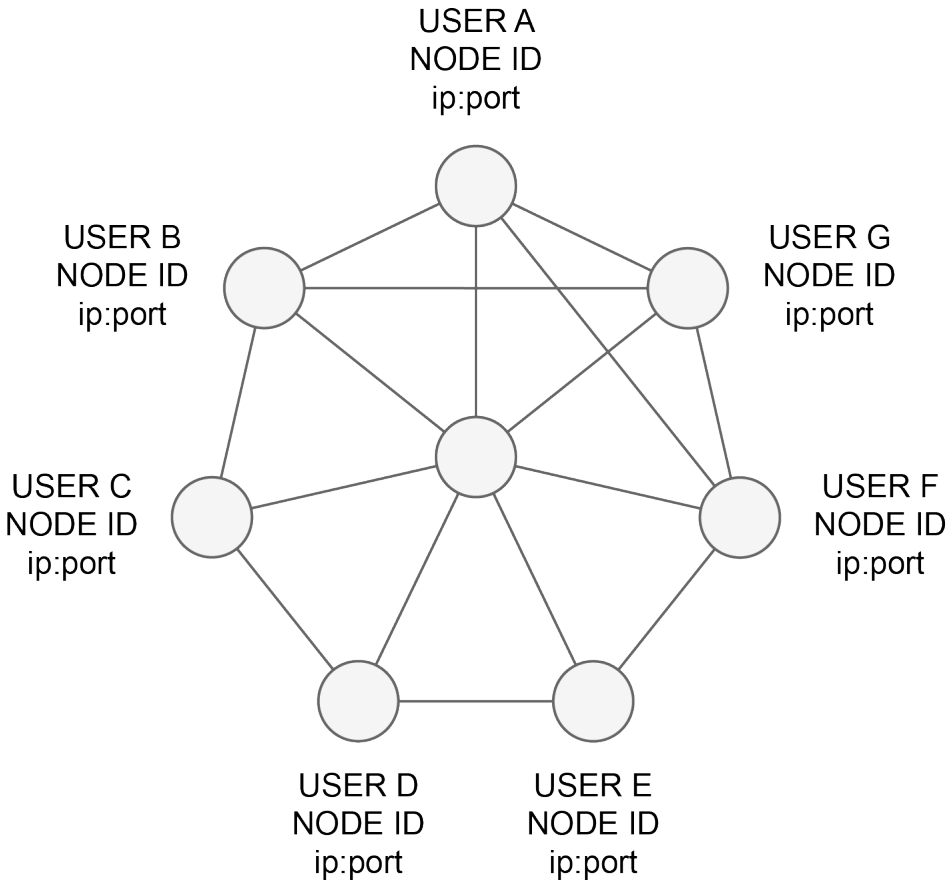
A block is valid when block header is valid, all transactions and verifiers' votes included in the block are valid.

2.4 The Network

Dipperin uses DHT(Distributed Hash Table) at P2P network structure, to improve searching effectiveness between nodes and the P2P network capability that defend DOS(Denial of Service) attack. In this case even if a whole batch of nodes in the network were attacked, the availability of the network would not be significantly affected. Dipperin uses Kademlia algorithm to realize DHT.

2.4.1 Network structure

A Dipperin network is a peer-to-peer decentralized network of nodes. Each node runs the Dipperin software.

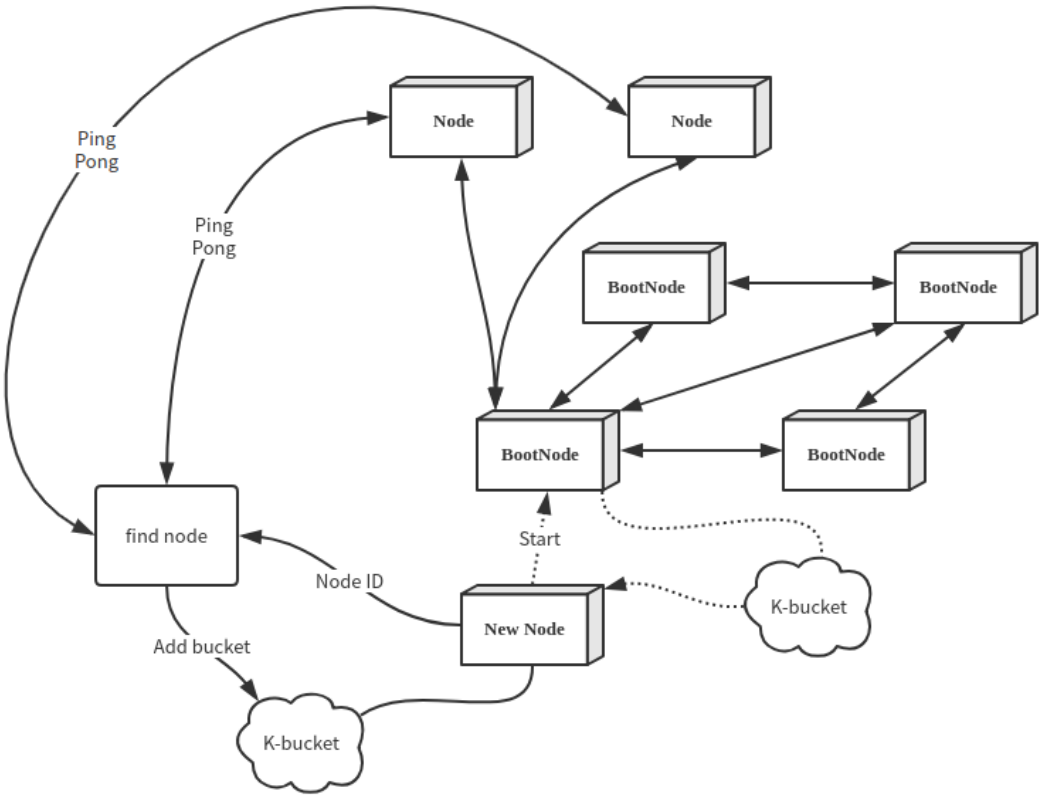


All communication between nodes is point-to-point and encrypted using transport-layer security.

2.4.2 Admission to the network

Dipperin networks are public. To join a network, Node need connect the Dipperin Boot Nodes.

Before the chain release, Dipperin deploys some start nodes (BootNode), hard-coded in the program. In this way, when these nodes are started for the first time, they will connect automatically the bootnodes, exchange the K-bucket between the nodes, and obtain more nodes ID to make connections, thus joining the entire Dipperin network.

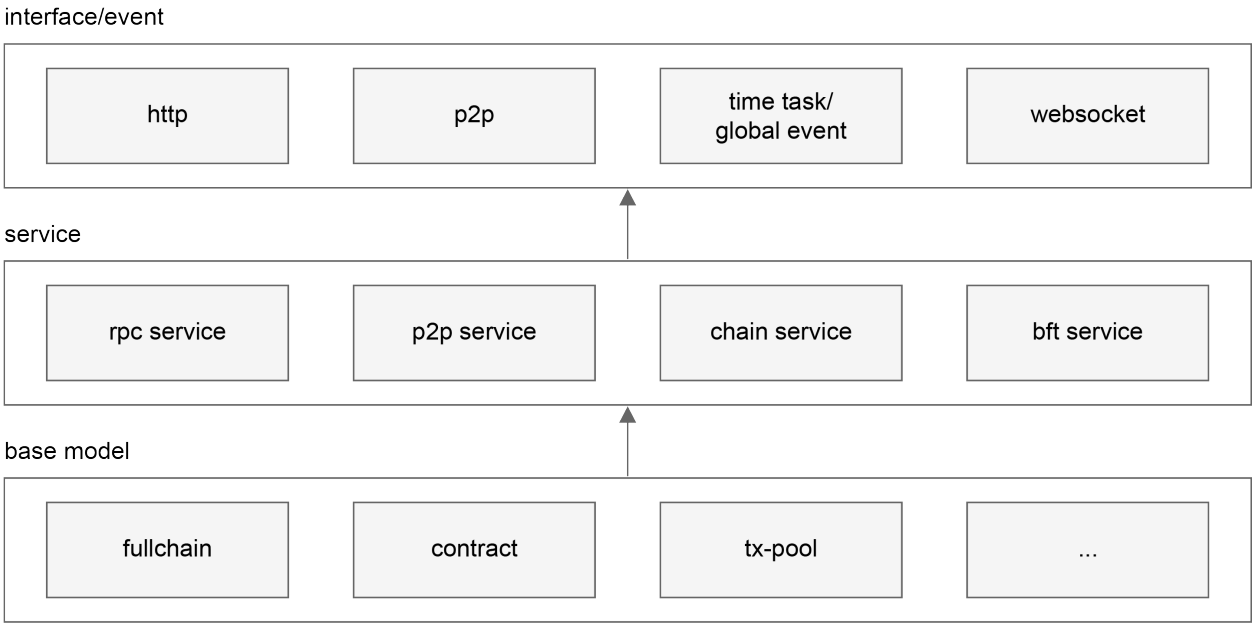


The first time a node connects to the network it uses one of the predefined boot nodes. Through these boot nodes a node can join the network and find other nodes.

2.5 Nodes

2.5.1 Node architecture

Dipperin written in Golang. We can visualize the node's internal architecture as follows:



Dipperin node's internal is a three-tier architecture, isolates parts that have clearly different functions.

2.5.2 interface/event

This layer is the interface that triggers the node response Users can interact with other nodes through http, websocket and p2p.

2.5.3 service

This layer is to process the business logic, encapsulate the logical methods of each business module, provide an interface upwards for the upper layer to call.

2.5.4 base model

This layer is mainly to achieve the underlying functions required by the node, like transaction-pool, contract base module.

2.6 Smart Contract

2.6.1 What is a smart contract?

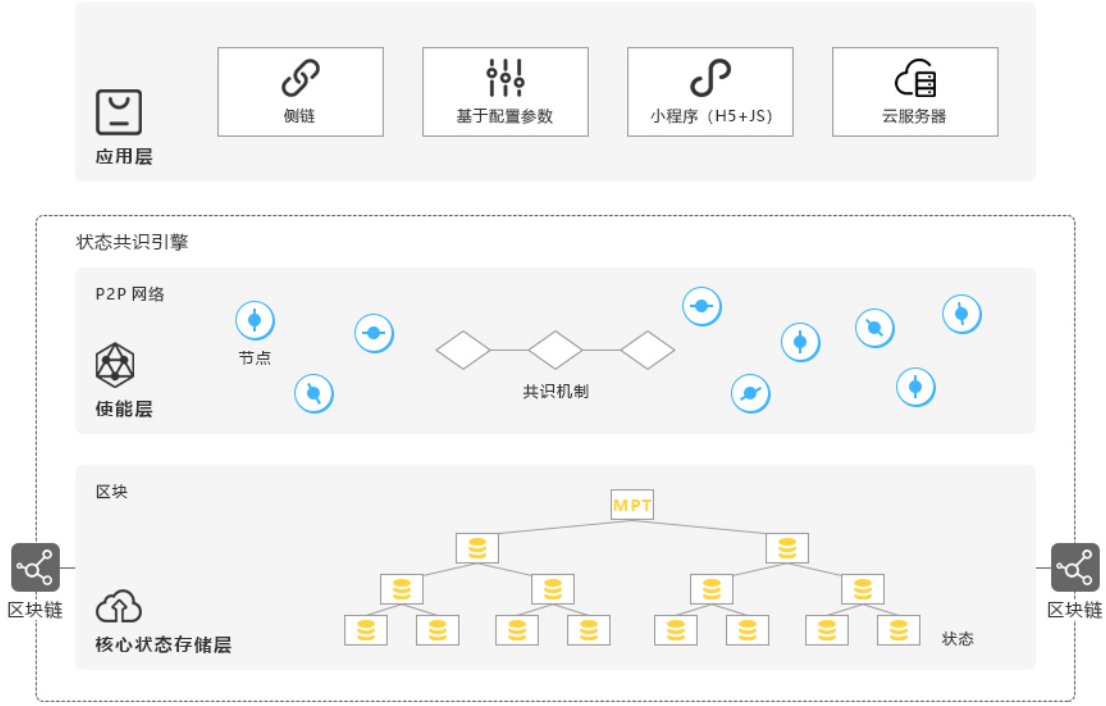
The blockchain concept has been extended over last 9 years, for use not only with crypto currency but other types of records, as well as smart contracts and other decentralized applications. Code is run on the blockchain through the use of smart contracts.

Smart contracts are computer protocols that facilitate, verify, or force the negotiation or performance of contract. Terms of smart contract are recorded in a form of computer code instead of legal language. It can be half-automatically executed by a computer system. In other words, smart contract is computer program that directly controls some digital assets. Smart contracts allows to sign and witness transactions between any two or more parties without the necessity of third party.

2.6.2 Dipperin smart contract

Dipperin supports smart contract. It follows a structured design philosophy, stripping business logic and state consensus. Is provides a broader and easier-to-use smart contract development support while reducing the load on the main chain.

Dipperin's smart contract architecture consists of the following three layers:



- **Core state storage layer**:

  Based on MPT technology, it provides an efficient distributed application state storage, while implementing a chain-limited finite state machine. By applying state constraints on the chain of smart contracts and applying boundary definitions to their contracts, the security and performance energy levels of the application and the underlying blockchain are leapfrogged.

- **Enabling layer**:

  Consensus communication layer uses P2P communication technology and plugable and replaceable consensus algorithm, and integrates high frequency contract paradigm primitives such as ERC20/ERC721, etc. This layer also provides side chain bidirectional anchoring and cross The basic capabilities of chain communication.

- **Application layer**:

  change the state of the application through the interface provided by the state consensus engine. The implementation forms include the chain-based high-frequency, paradigm-based dApp based on configuration parameters, and the cloud server interacting with the consensus engine in the wallet sandbox environment. Run the applet (H5+JS), a custom sidechain generated by one-click provided by Dipperin.

CHAPTER 3

Dapps

3.1 Dipperin Dapp Development

3.1.1 How to develop a Dapp with Dipperin Wallet Extension

Dipperin Wallet Extension supply a set of interfaces for Dapp developer, which makes developing a Dapp more easily.

How Dipperin Wallet Extension works?

If you have already installed Dipperin Wallet Extension in your Chromeit will inject all Dipperin supplied interfaces into all web pages in your browser. By this way, Dapp can interact with Dipperin network. Developers can get these interface by following ways.

```
window.dipperinEx
```

Interfaces

DipperinEx supplied 5 interfacesthey have functions as follow

```
window.dipperinEx.isApproved; // Get the authorization state of Dapp
window.dipperinEx.approve; // Authorize the Dapp
window.dipperinEx.send; // Send transactions
window.dipperinEx.getActiveAccount; // Get accounts of users.
window.dipperinEx.on; // Listen for the message from the wallet extension.
```

dipperinEx.isApproved

dipperinEx.isApproved supply the Dapp authorization state.

```
const dapp_name = "Your Dapp's name";
/**
 * @param {string} dappName
 * @returns {Promise<{isApproved: boolean}>}
 */
window.dipperinEx
  .isApproved(dappName)
  .then(res => console.log(res)) // { isApproved: true }
  .catch(e => console.log(e));
```

If the value isApproved is true, it represent that Dapp is authorized

dipperinEx.approve

`dipperinEx.approve` is used for Dipperin Wallet Extension to authorize Dapp. Function can be called as follow.

```
/**
 * @interface ApproveRes
 */
interface ApproveRes {
  popupExist: boolean;
  isHaveWallet: boolean;
  isUnlock: boolean;
}
/**
 * @param {string} dappName
 * @returns {Promise<ApproveRes>}
 * @throws {ApproveRes}
 */
window.dipperinEx
  .approve(dappName)
  .then(res => console.log(res)) // {popupExist: false, isHaveWallet: true, isUnlock:␣
↪true}
  .catch(e => console.log(e)); // {popupExist: false, isHaveWallet: true, isUnlock:␣
↪false}
```

After call the function, there will shown dialog to request for user's authorization.

The return value of isHaveWallet represents whether user have accounts in this extension. IsUnlocked means the wallet is unlocked. The result of user authorization can get by call `dipperinEx.isApproved`.

dipperinEx.send

Dipperin Wallet Extension supply `dipperinEx.send` for user to send transactions.

```
type Send = (
  name: string,
  to: string,
  value: string,
  extraData: string
) => Promise<SendResFailed | string>;
```

There are 4 input parameters, `name` for the name of the Dapp, `to` for the receiving address. `value` for the money to send, and `extraData` for extra data.

```
const address = "0x00003A9A328170b650E89F2C28F2E61364d2aEdC292e";
const amount = "1";
const extraData = "The message your dapp need";
/**
 * @interface SendResFailed
 */
interface SendResFailed {
  isApproved: boolean;
  isHaveWallet: boolean;
  isUnlock: boolean;
  info: string;
}

/**
 * @param {string} address
 * @param {string} amount
 * @param {string} extraData
 * @returns {Promise<string>}
 * @throws {ApproveRes}
 */
windox.dipperinEx
  .send(APP_NAME, DEAULT_ADDRESS, amount, extraData)
  .then(res => console.log(res)) //␣
↪0x8d303cb0b24fd332614a02c477605255e6a29afc3d477086603583f8aea5ddff
  .catch(e => console.log(e));   // {popupExist: false, isApproved: true,␣
↪isHaveWallet: true, isUnlock: true, info: "send tx failed"}
```

If success it will return a transaction hash, otherwise return an error message.

dipperinEx.getActiveAccount

`dipperinEx.getActiveAccount` can get authorized account address.

```
/**
 * @param {string} dappName
 * @returns {Promise<string>}
 */
window.dipperinEx.getActiveAddress(dappName);   //␣
↪0x00008522edBC22d9db52fa3AF05C2093dfFbFFF9DdBD
```

If success it will return an address, otherwise return an empty string.

dipperinEx.on

"dipperinEx.on" is used by Dapp to get Dipperin wallet extension messages.

```
window.dipperinEx.on("changeActiveAccount", () => {
  console.log("Have changed active account!");
});
```

CHAPTER 4

Tutorials

4.1 Build your first network

4.1.1 Install and start Dipperin Command Line Tool

If you don't know how to install or start Dipperin Command Line Tool, please take a look at the *Quick Start*

4.1.2 Init verifiers

After start verifier nodes, you'll see

```
t=2019-01-27T10:44:46+0800 lvl=info msg="setup default sign address"␣
↪addr=0x0000C82ADd56D1E464719D606bB873Ad52779c67F465
```

copy this `address` like `0x0000C82ADd56D1E464719D606bB873Ad52779c67F465` to your genesis verifiers.

```
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier1 --http_port 10001 --ws_port 10002 --p2p_listener 20001
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier2 --http_port 10003 --ws_port 10004 --p2p_listener 20002
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier3 --http_port 10005 --ws_port 10006 --p2p_listener 20003
...
```

This is done so that you can generate the default verifier wallet, which you can configure in `genesis.json`.

4.1.3 Setup genesis state

You need input content below into file `$HOME/softwares/dipperin_deploy/genesis.json`.

```
{
  "nonce": 11,
  "accounts": {
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af19": 1000,
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af29": 1000,
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af39": 1000
  },
  "timestamp": "1548554091989871000",
  "difficulty": "0x1e566611",
  "verifiers": [
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af19",
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af29",
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af39",
    "0x00005EE98a9d6776F4599f8cD9070843E6D03Ce6af49"
  ]
}
```

In the json, `accounts` is pre-fund some accounts for your private chain. `verifiers` is first round default verifiers for you private chain, this list must have `22` verifiers, you can change this number in `core/chain-config/config.go` at func `defaultChainConfig` -> `VerifierNumber`.

4.1.4 Start a bootnode

Generate bootnode private key file, and start it.

```
$ bootnode --genkey=boot.key
$ bootnode --nodekey=boot.key
```

You'll see the following code:

```
bootnode conn: enode://
↪958784048f7021c99b5ce82bd0078398037226ffd35c166b874fc8ff36d0c4e07e0a2a28eb02b6d993ec8b652f79a9bf79
↪0.0.1:30301
```

when bootnode started, copy this conn str to `core/chain-config/config.go` at func `initLocalBoots` -> `KBucketNodes`, and recompile your `dipperin`, your node will auto connect this bootnode when started. Or you can write this conn str to your node's `static_boot_nodes.json` file in `datadir`, it's content should like:

```
[
  "enode://
↪958784048f7021c99b5ce82bd0078398037226ffd35c166b874fc8ff36d0c4e07e0a2a28eb02b6d993ec8b652f79a9bf79
↪0.0.1:30301"
]
```

4.1.5 Start verifiers

You should remove `full_chain_data` in all `datadir` because of your genesis block has changed, and don't remove `CSWallet` in `datadir`. Then run commands below to started verifiers.

```
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier1 --http_port 10001 --ws_port 10002 --p2p_listener 20001
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier2 --http_port 10003 --ws_port 10004 --p2p_listener 20002
```

(continues on next page)

(continued from previous page)

```
$ dipperin --node_type 2 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/
↪verifier3 --http_port 10005 --ws_port 10006 --p2p_listener 20003
...
```

4.1.6 Start miner master(default have a miner)

```
$ dipperin --node_type 1 --soft_wallet_pwd 123 --data_dir /home/qydev/dipperin/mine_
↪master1 --http_port 10010 --ws_port 10011 --p2p_listener 20010
...
```

This command will start a `mine master` and start a `miner` in it, you'll see it is mining block and broadcast block to verifiers.

And your private chain block height is growing up.

4.2 Distribute your token

Dipperin supports formalized ERC20 token smart contract. ERC20 token can be deployed through three tools:

- Command line
- Wallet application
- Dipperin JavaScript API(dipperin.js)

4.2.1 Deploy token through command line

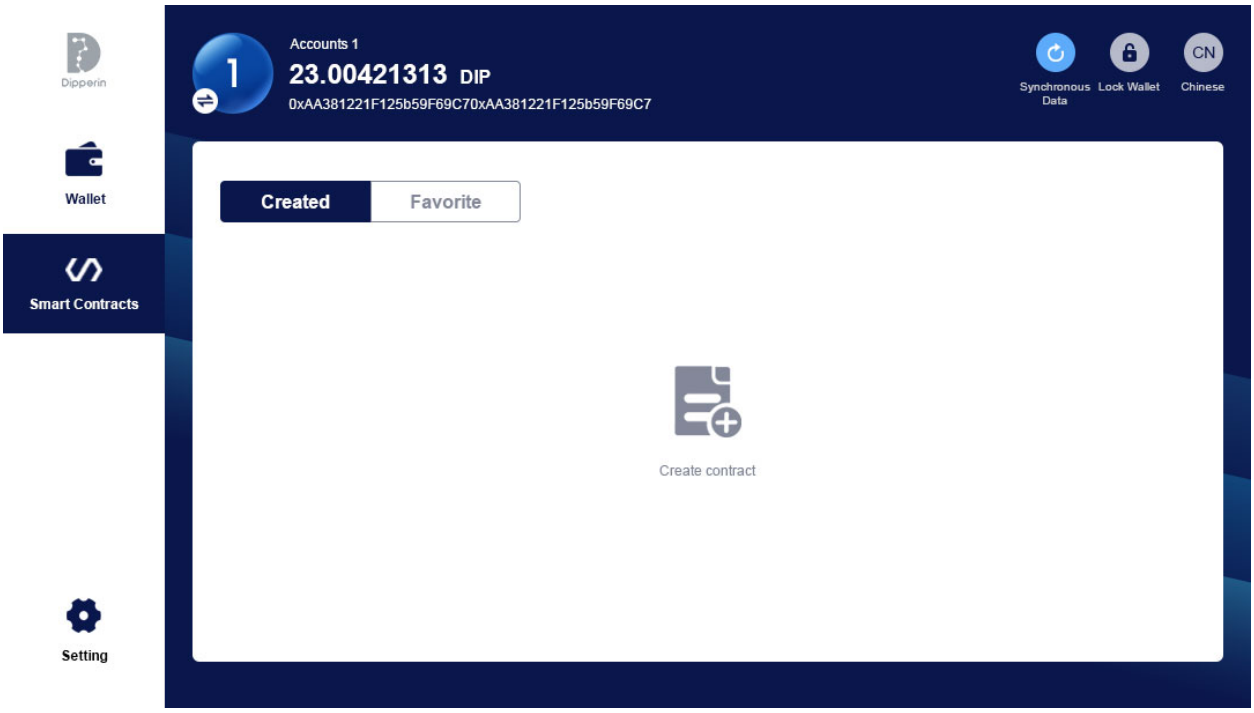
```
# Start command line
$ dipperincli
# Deploy ERC20 token
rpc -m AnnounceERC20 -p [owner_address], [token_name], [token_symbol], [token_total_
↪supply], [decimal],[transactionFee]
# Example
rpc -m AnnounceERC20 -p 0x0000D07252C7A396Cc444DC0196A8b43c1A4B6c53532,chain,stack,5,
↪3,0.00001
```

See more details for Command Line Tool

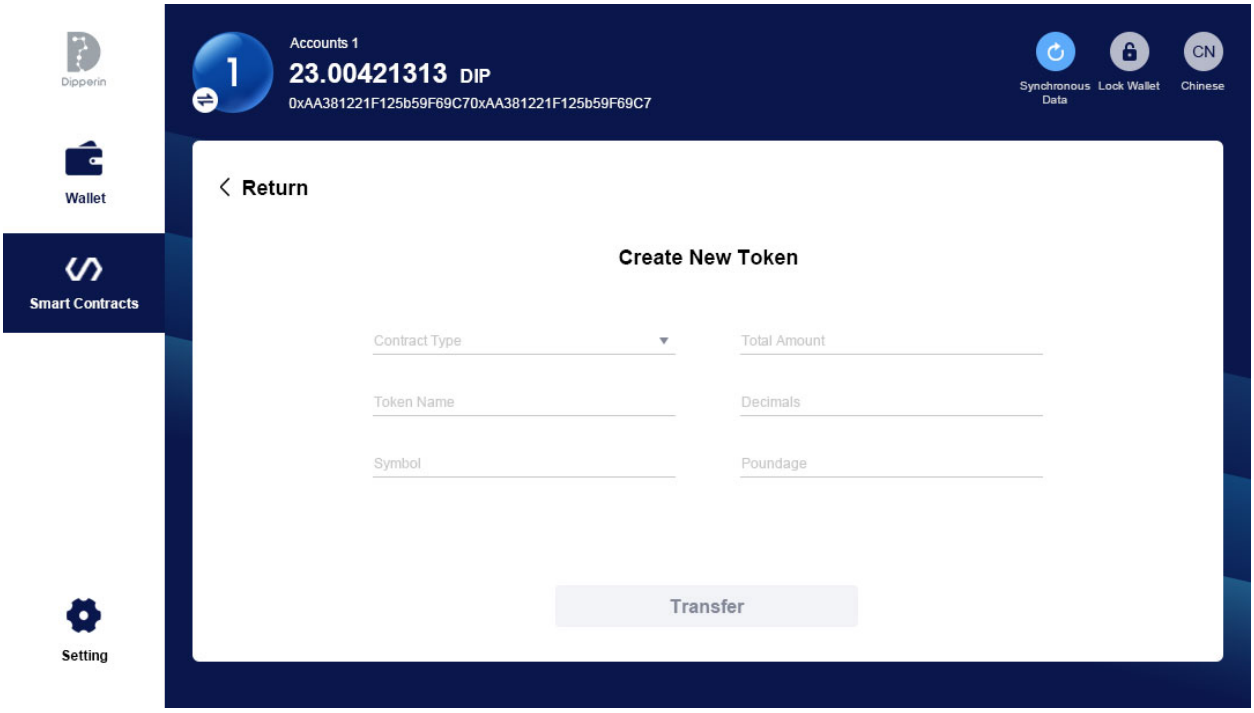
4.2.2 Deploy token through wallet application

Download and install wallet.

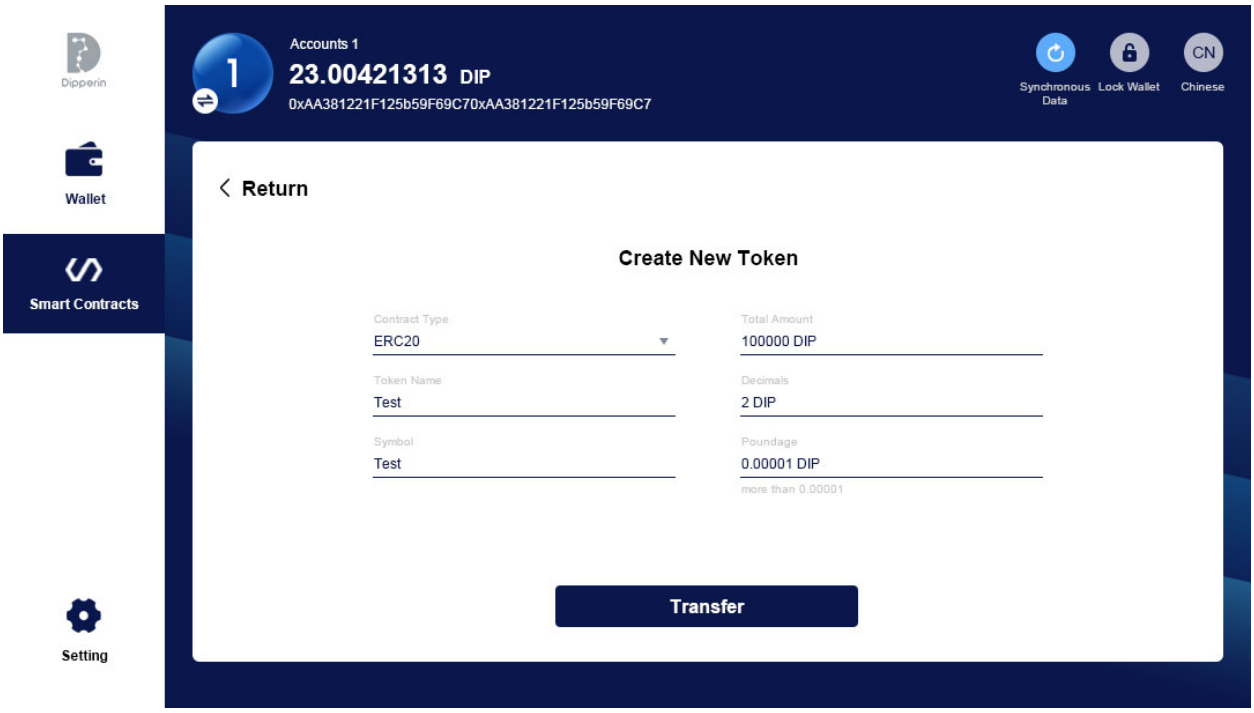
After you created your account, jump to the contract page.



Click create contract and turn to a create contract page.



Fill in the informations and click create. Done.



4.2.3 Deploy token through JavaScript API

Import dipperin.js in your JavaScript file.

```
dipperin
import dipperin { Contract Accounts } from '@dipperin/dipperin.js'
dipperin
const dipperin = new Dipperin("$YOUR_RPC_PROVIDER")
// Deploy token contract
const contract = Contract.createContract(
  {
    owner: $YOUR_CONTRACT_OWNER,
    tokenDecimals: $YOUR_TOKEN_DECIMALS,
    tokenName: $YOUR_TOKEN_NAME,
    tokenSymbol: $YOUR_TOKEN_SYMBOL,
    tokenTotalSupply: $YOUR_TOKEN_TOTAL_SUPPLY
  },
  $YOUR_TOKEN_TYPE,
  $YOUR_TOKEN_ADDRESS
)
// Create a transaction
const signedTransaction = Accounts.signTransaction(
  {
    extraData: contract.contractData,
    fee: $TRANSACTION_FEE,
    nonce: $YOUR_ACCOUNT_NONCE,
    to: $YOUR_TOKEN_ADDRESS,
    value: '0',
  },
  $YOUR_PRIVATE_KEY
)
// Send transaction
```

(continues on next page)

(continued from previous page)

```
dipperin.dr.sendSignedTransaction(signedTransaction.raw)
  .then(transactionHash => {
    // Do something
  })
// Done.
```

CHAPTER 5

Smart Contract Development

5.1 Install Compiler

Dipc is a compiler that compile Dipperin C++ smart contract code to WebAssembly program.

5.1.1 Build From Source Code

Required

- GCC 5.4+ or Clang 4.0+
- CMake 3.5+
- Git
- Python

Ubuntu

Required: 16.04+

- **Install Dependencies**

```
sudo apt install build-essential cmake libz-dev libtinfo-dev
```

- **Get Source Code**

```
git clone https://github.com/dipperin/dipc.git
cd dipc
git submodule update --init --recursive
```

- **Build Code**

```
cd dipc
mkdir build && cd build
cmake ..
make && make install
```

Windows

Required: MinGW-W64 GCC-8.1.0

NOTES: *MinGW and CMake must be installed in a directory without space.*

- **Get Source Code**

```
git clone https://github.com/dipperin/dipc.git
cd dipc
git submodule update --init --recursive
```

- **Build Code**

```
cd dipc
mkdir build && cd build
cmake -G "MinGW Makefiles" .. -DCMAKE_INSTALL_PREFIX="C:/dipc.cdt" -DCMAKE_MAKE_
↪PROGRAM=mingw32-make
mingw32-make && mingw32-make install
```

5.1.2 Use Dipc

Skeleton Smart Contract Without CMake Support

- Init a project

```
dipc-init -project example -bare
```

- Build contract

```
cd example
dipc-cpp -o example.wasm example.cpp -abigen
```

Skeleton Smart Contract With CMake Support

- Init CMake project

```
dipc-init -project cmake_example
```

- Build contract
  - Linux

```
cd cmake_example/build
cmake ..
```

  - Windows

    Required:

- MinGW-W64 GCC-8.1.0
- CMake 3.5 or higher

```
cd cmake_example/build
cmake .. -G "MinGW Makefiles" -DCMAKE_PREFIX_PATH=<cdt_install_dir>
```

5.2 Deploy Smart Contract

5.2.1 Step 1: Obtain Contract Source

Generate a seleton smart contract using dipc-init, and fill the business logic you need in the contract.

```
dipc-init -project example -bare
```

If you do not use the dipc-init tool, you need to introduce the "dipc/dipc.h" header file in the contract file and inherit the Contract class and provide an external function init.

```
#include "dipc/dipc.h"

class YourContractName : public Contract {
    EXPORT void init();
}
```

5.2.2 Step 2: Deploy a Contract

Dipperin currently offers two ways to deploy and call smart contracts through both command line and wallet. A contract is deployed through sending contract transactions. |

Deploy a Contract from Console

By calling the command

```
tx SendTransactionContract -p ${deployAddress},${value},${gasPrice},${gasLimit} --abi
↪${abiPath} --wasm ${wasmPath} --input ${init params} --is-create
```

to deploy the contract

The meaning of each parameter is:

- `deployAddress`the address of the contract issuer;
- `callAddress`: the address of the contract caller;
- `value`the number of DIPs transferred to the contract;
- `gasPrice`the gas price specified by this transaction;
- `gasLimit`: the maximum amount of gas consumed in this transaction is charged according to the actual use. If the specified value is insufficient, the transaction will fail;
- `abiPath`he path to the ABI file generated by compiling the contract file;
- `wasmPath`the path to the wasm file generated by compiling the contract file;

- `init params`if the contract's init function parameter is not empty, then the parameter needs to be passed here when creating the contract;
- `funcName`the name of the function to be called;
- `func params`:parameters that need to be passed when calling the contract function

Call a Contract from Console

Call CONSTANT Function

By calling the command

```
tx CallContract -p ${callAddress},${contractAddress} --func-name ${funcName} --input $
↪{func params}
```

to call the contract functionThe meaning of each parameter is the same as above

Call Non-CONSTANT Function

By calling the command

```
tx SendTransactionContract -p ${callAddress},${contractAddress},${value},${gasPrice},$
↪{gasLimit} --func-name ${funcName} --input ${func params}
```

to call the contract functionThe meaning of each parameter is the same as above

See *Command Line Tool* for details.

Deploy and Call a Contract from Wallet

Using Dipperin Wallet to deploy and call smart contracts is very simple, as long as you download and install the wallet, it is easy to operate according to the interface instructions.

5.3 Understanding ABI Files

When publishing a smart contract on the dipperin chain, you need to provide the ABI file generated when compiling the smart contract using the dipc tool. ABI (Application Binary Interface) is a JSON-based description that shows how to translate user operations between JSON and binary representations. ABI also describes how to convert database state to JSON or convert from JSON. Once you describe your smart contract through ABI, developers and users can seamlessly interact with your smart contract via JSON.

Special Note: ABI can be bypassed when executing a contract transaction. The messages and actions passed to the smart contract do not have to comply with the ABI. ABI is a guide, not a guard.

All methods that can be called directly by the user in the contract will be described by generating a corresponding JSON object in the ABI file.

```
[{
    "name": "init",
    "inputs": [
        {
            "name": "tokenName",
```

(continues on next page)

(continued from previous page)

```
            "type": "string"
        },
        {
            "name": "symbol",
            "type": "string"
        },
        {
            "name": "supply",
            "type": "uint64"
        }
    ],
    "outputs": [],
    "constant": "false",
    "payable": "false",
    "type": "function"
},
{
    "name": "GetBalance",
    "inputs": [
        {
            "type": "string"
        },
        {
            "type": "string"
        },
        {
            "type": "uint64"
        }
    ],
    "type": "event"
}
]
```

This is part of an ABI file for an example token contract. The meanings of their fields are:

name: indicates the name of the method in the contract or the name of the event in the contract; inputs: method parameters inputs.type: indicates the type of the input parameter; inputs.name: indicates the field name of the input parameter; outputs: the return value of the method; outputs.type: indicates the type of the return value; constant: a value of true means that the method does not change the state of the contract data, and can be called directly without sending a transaction; payable: a value of true indicates that DIP can be transferred to the contract account by this method. type: indicates the type of the abi object, which has two types: event and function.

The types supported by inputs.type and outputs.type are as follows (the types of input and return values supported in accessible functions) :

- std::string
- unsigned char
- char[]
- char *
- char
- const char*
- bool
- unsigned long long

- unsigned long
- unsigned __int128
- uint128_t
- uint64_t
- uint32_t
- unsigned short
- uint16_t
- uint8_t
- __int128
- int128_t
- long long
- int64_t
- long
- int32_t
- short
- int16_t
- int8_t
- int

5.4 Data Types

5.4.1 Storage Types

Dipc provides template types to provide data persistence to the dipperin chain.

- Uint8
- Int8
- Uint16
- Int16
- Uint
- Int
- Uint64
- Int64
- String
- Vector
- Set
- Map
- Array

- Tuple
- Deque

template types to provide data persistence to the dipperin chain.

Storage types usage example:

```
// Example one Map uses:
// Define the storage field name
char bal[] = "balance";
// Storage field name   key type    value type
Map<     bal,           std::string, uint64_t >  balance;

// Example two String uses:
// Define the storage field name
char name[] = "contract_name";
//  Storage field name
String<name> contract_name;
```

In the contract, the field defined by the storage type is used and its value is automatically stored on the dipperin chain when the contract is created.

5.4.2 Fundamental Types

Dipc supports all basic types of C++, standard library types and their arithmetic operations And the types defined in the dipclib package

- Big integer types defined using the boost library
  - bigint
  - u64
  - u128
  - u256
  - u160
  - u512
- Integer and unsigned integers encoded using VLQ
  - unsigned_int
  - signed_int
- Custom types for efficient use of memory
  - map
  - array
  - list
- Types defined by the custom FixedHash class
  - h256 //32bytes
  - h160
  - h128
  - h64

- Address

5.5 Functions

Smart contracts access and modify state variables through functions. Function can be modified with PAYABLE, CONSTANT, EXPORT. PAYABLE, CONSTANT, EXPORT need to be written before the return value of function. The modified function is externally accessible, and the unmodified function is an internal function that is not accessed externally.

5.5.1 Init Function

The init function is required and must be a function that can be accessed externally. A smart contract only allows one init function. The init function allows arguments to be passed and will only executed once during the contract deployment and run.

```
#include "dipc/dipc.h"

class YourContractName : public Contract {
     EXPORT void init();
}
```

5.5.2 Accessible Functions

The parameter types and return value types of functions that can be accessed externally are restricted. Currently only simple types are supported:

- std::string
- unsigned char
- char[]
- char *
- char
- const char*
- bool
- unsigned long long
- unsigned long
- unsigned __int128
- uint128_t
- uint64_t
- uint32_t
- unsigned short
- uint16_t
- uint8_t
- __int128

- int128_t
- long long
- int64_t
- long
- int32_t
- short
- int16_t
- int8_t
- int

Input parameters and return values do not support storage types and other custom types. Accessible Functions of the same name are not supported, ie overloads of accessible functions are not supported. There are three types of accessible functions: CONSTANT, PAYABLE, and EXPORT. The usage and functions of the CONSTANT, PAYABLE, and EXPORT macro definitions in dipc are:

| Macro | Utilized Location | Functions | | — | — | — | — | |EXPORT | Before the return value of the method declaration and definition | Indicates that the method is an external method | EXPORT void init(); | |CONSTANT | The same as above | Indicates that the method does not change the state of the contract data, and can be called directly without sending a transaction. | CONSTANT uint64 getBalance(string addr); | |PAYABLE | The same as above | Indicates that DIP can be transferred to the contract account by this method. | PAYABLE void transfer(string toAddr, uint_64 value); | These three macros are independent of each other and cannot be used at the same time.

5.5.3 Internal Functions

The internal function follows the definition of the C++ language function and does not impose any restrictions.

5.5.4 Return Value Display

If there is a query request for the return value of the externally accessible function, you need to manually call the DIPC_EMIT_EVENT in the EVENT section in the contract to save it into the Log in the receipts, and then query it by getting the receipts or Log.

5.5.5 Standard Library Functions

| Function Name | Function Introduction | Parameters | Return Types | | ———— | ———————————————— | ——————— | ———— | | gasPrice | Get the gas price of the current transaction | | int64_t | | blockHash | Get the hash of the block based on the block height | int64_t number | h256 | | number | Get the blocknumber of the current block | | uint64_t | | gasLimit | Get the gas limit of the current transaction | | uint64_t | | timestamp | Get the packing timestamp of the block | | address | | coinbase | Get the packaged miner address of the current block | | string | | balance | Get the account balance of an account on the chain | Address adr | uint64 | | origin | Get the account address of the contract creator | | Address | | caller | Get the account address of the contract caller | | Address | | sha3 | Sha3 encryption operation | | h256 | | getCallerNonce | Get the transaction nonce of the contract caller account | | string | | callTransfer | Transfer the DIP of the contract account to the specified account | Address to ,u256 value | int64_t | | prints | Print a string variable | string | void | | prints_l | Print the first few characters of a string variable | bool condition, string msg | void | | printi | Print a 64-bit signed Integer | string msg | void | | printui | Print a 64-bit unsigned Integer | bool condition, string msg | void | | printi128 | Print a 128-bit signed Integer | (address addr, uint256 amount) | void | | printui128 | Print a 128-bit unsigned Integer | const uint128_t* value | bool | | printhex | Print data in hexadecimal format | int64 value | string | | print | Template

function to print any basic type data | any basic type | void | | println | Template function, print any basic type data, and add a newline at the end | any basic type | void | | DipcAssert | Determine if the given condition is true, if it is not true, it will throw an exception | uint64 value | | | DipcAssertEQ | Determine if two conditions are equal, and throw an exception if they are not equal | string value | | | DipcAssertNE | Determine if two conditions are not equal, and throw an exception if they are equal | two arbitrary expressions | |

5.6 Events

Events provides an abstraction of the logging capabilities of WAVM. Applications can subscribe to and listen to these events through the client's RPC interface. The event is declared by the keyword DIPC_EVENT. The event only needs to declare the event name and parameters, and no return value. The event parameter type is consistent with the parameter type restrictions of the externally accessible function.

```
//Declaration
DIPC_EVENT(event_name,int32, string);

//Call
int32 val1;
string val2;
DIPC_EMIT_EVENT(event_name,val1,val2;
```

5.7 Gas Calculation

For normal transaction and contract transaction, the calculation of gas is slightly different:

5.7.1 Normal Transaction

The total `gasUsed` value of a normal transaction is divided into two parts, `fixedGasUsed` is built into the system, and `f(txExtraData)` is calculated based on the `extraData` inside the transaction:

```
totalGasUsed = fixedGasUsed + f(txExtraData)
```

Suppose that in a normal transaction, the number of bytes whose value is 0 in `exteraData` is `ZeroBytes`, and the number of bytes whose value is non-zero is `NoZeroBytes`. Then `f(txExtraData)` is expressed as follows:

```
f(txExtraData) = TxDataZeroGas*ZeroBytes + TxDataNonZeroGas*NoZeroBytes
```

| Parameters | System Default Value | Remarks | | ———————- | ———————— | ———————————————— | | fixedGasUsed | 21000 | This is the default `fixedGasUsed` value for the current system normal transaction. | | TxDataZeroGas | 4 | When the data is 0, the `gasUsed` of unit bytes | | TxDataNonZeroGas | 68 | When the data is not 0, the `gasUsed` of unit bytes |

When the data is 0 and non-zero, its gasUsed is different because non-zero data consumes less system resources during storage and calculation.

CHAPTER 6

Contributions

The Dipperin project eagerly accepts contributions from the community.We welcome contributions to Dipperin in many forms.

6.1 Working Together

When contributing or otherwise participating, please:

- Be friendly and welcoming
- Be patient
- Be thoughtful
- Be respectful
- Be charitable
- Avoid destructive behavior

Excerpted from the Go conduct document.

6.2 Ways to contribute

6.2.1 Getting help

If you are looking for something to work on, or need some expert assistance in debugging a problem or working out a fix to an issue, our community is always eager to help. We hang out on mail `report@dipperin.com`. Questions are in fact a great way to help improve the project as they highlight where our documentation could be clearer.

6.2.2 Reporting Bugs

When you encounter a bug, please open an issue on the corresponding repository. Start the issue title with the repository/sub-repository name, like `repository_name: issue name`. We have provided a issue templates for bug report:Bug report template. If you can abide by this template, this will help us fix the bug more efficiently.

6.2.3 Suggesting Enhancements

If the scope of the enhancement is small, open an issue. If it is large, such as suggesting a new repository, sub-repository, or interface refactoring, then please @Dipperin-Project on an issue,we will pay more attention on you suggestion.

6.2.4 Your First Code Contribution

If you are a new contributor, thank you! Before your first merge, you will need to be added to the CONTRIBUTORS files. Open a pull request adding yourself to these files. All Dipperin code follows the LGPL license in the license document. We prefer that code contributions do not come with additional licensing. For exceptions, added code must also follow a LGPL license.

6.2.5 Code Contribution

If it is possible to split a large pull request into two or more smaller pull requests, please try to do so. Pull requests should include tests for any new code before merging. It is ok to start a pull request on partially implemented code to get feedback, and see if your approach to a problem is sound. You don't need to have tests, or even have code that compiles to open a pull request, although both will be needed before merge. When tests use magic numbers, please include a comment explaining the source of the number.Commit messages also follow some rules. They are best explained at the official Go "Contributing guidelines" document: golang.org/doc/contribute.html

For example:

```
Dipperin-core: add support for consensus

This change list adds support for consensus.
Previously, the Dipperin-core package was consensus slowly,sometimes leading to
a panic later on in the program execution.
Improve consensus efficiency and add some tests.

Fixes  Dipperin/Dipperin-core/core#20.
```

If the change list modifies multiple packages at the same time, include them in the commit message:

```
Dipperin-core/core,Dipperin-core/core/Dipperin: implement wrapping of Go interfaces

bla-bla

Fixes Dipperin/Dipperin-core/core#40.
```

Please always format your code with goimports. Best is to have it invoked as a hook when you save your .go files.

Files in the Dipperin repository don't list author names, both to avoid clutter and to avoid having to keep the lists up to date. Instead, your name will appear in the change log and in the CONTRIBUTORS files.

New files that you contribute should use the standard copyright header:

```
// Copyright 2019, Keychain Foundation Ltd.
// This file is part of the dipperin-core library.
//
// The dipperin-core library is free software: you can redistribute
// it and/or modify it under the terms of the GNU Lesser General Public License
// as published by the Free Software Foundation, either version 3 of the
// License, or (at your option) any later version.
//
// The Dipperin-core library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

Files in the repository are copyright the year they are added. Do not update the copyright year on files that you change.

6.3 Code Review

We follow the convention of requiring at least 1 reviewer to say LGTM(looks good to me) before a merge. When code is tricky or controversial, submitters and reviewers can request additional review from others and more LGTMs before merge. You can ask for more review by saying PTAL(please take another look) in a comment in a pull request. You can follow a PTAL with one or more @someone to get the attention of particular people. If you don't know who to ask, and aren't getting enough review after saying PTAL, then PTAL @Dipperin-Project will get more attention. Also note that you do not have to be the pull request submitter to request additional review.

6.4 Style

We use Go style.

6.5 What Can I Do to Help?

If you are looking for some way to help the Dipperin project, there are good places to start, depending on what you are comfortable with.You can search for open issues in need of resolution.You can improve documentation, or improve examples.You can add and improve tests.You can improve performance, either by improving accuracy, speed, or both.You can suggest and implement new features that you think belong in Dipperin.

This "Contributing" guide has been extracted from the Gonum project. Its guide is here.

CHAPTER 7

Release Notes

7.1 Mercury (v1.0.0)

This is the first test-net version of the dipperin-core(v1.0.0) program and it marks a milestone in the history of Dipperin. Mercury introduced the following features:

- Original Deterministic Proof of Work(DPoW) consensus mechanism.
- Verifiable Random Function(VRF) based verifier sortition mechanism.
- Invertible Bloom filter Lookup Table(IBLT) based block propagation.
- Hierarchical deterministic wallet.
- Formalized ERC20 smart contract.
- Built-in economic model.
- Map-reduce proof of work mechanism.

Dipperin-core includes the following programs:

- Node main program: dipperin
- Client program: dipperin-cli
- Mining program: miner

CHAPTER 8

Architecture Reference

This document describes the architecture design of Dipperin v1.0.0. In the process of implementing Dipperin, some details will be adjusted according to the specific situation and problems.

8.1 Advantages

1. **Random miner and fast confirmation**

   Dipperin retains the randomness of the block produced by the mining mechanism, and the miners with high computational ability will only have a higher probability to produce block, but will not have absolute right to produce block. The block can be accepted by the whole network only after this block verified and signed by the 22 verifiers selected by the whole network.

2. **Fair verifier campaign**

   Unlike most verifier campaign in the market, Dipperin uses a verifiable random algorithm and reputation value election mechanism, which guarantees that there are two factors affecting the results of each round of elections, `rank = w * Random + m * P`, where W and m are the weights of random factors and reputation values.

   See Yellow Paper for details

3. **Well-designed economic model**

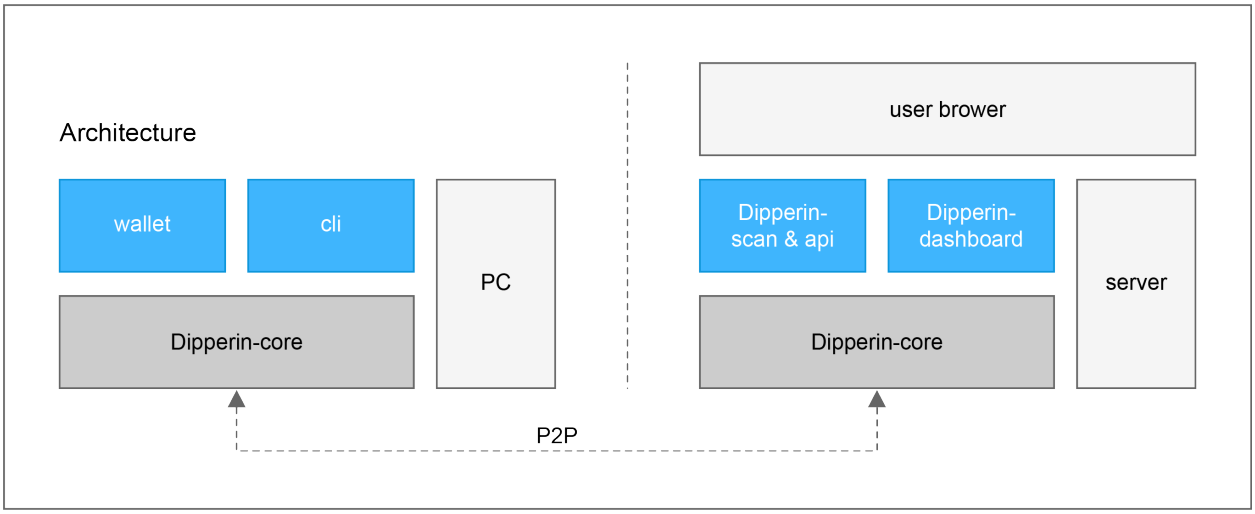
   Dipperin has a well-designed economic model to ensure that the rewards for miners and validators are reasonable, and attackers have no inspire to behave malicious.

4. **Well-designed software architecture**

   The architecture of Dipperin-core is well designed to make the program flexible, modular and plugable. Interface is largely used in Dipperin's implementation, to decoupled modules. Design patterns like middleware, decorator are carefully used in the system.

8.2 System architecture

Dipperin is a decentralized blockchain network. Below is the architecture of Dipperin nodes. The left side of the picture is the architecture of PC user and the right side is the architecture of service provider.



As it shown in the picture, Dipperin-core is the core program, responsible to maintain the ledger and communication with other nodes. To join the Dipperin blockchain network as a full node, user should run a Dipperin-core program. Dipperin-core expose RPC interface to wallets and other applications.

At the PC node, user can call functions of node through wallet and command line tool. At the service provider node, services like blockchain browser can subscribe events, and fetch chain data from Dipperin-core. Provider more functions through web service.

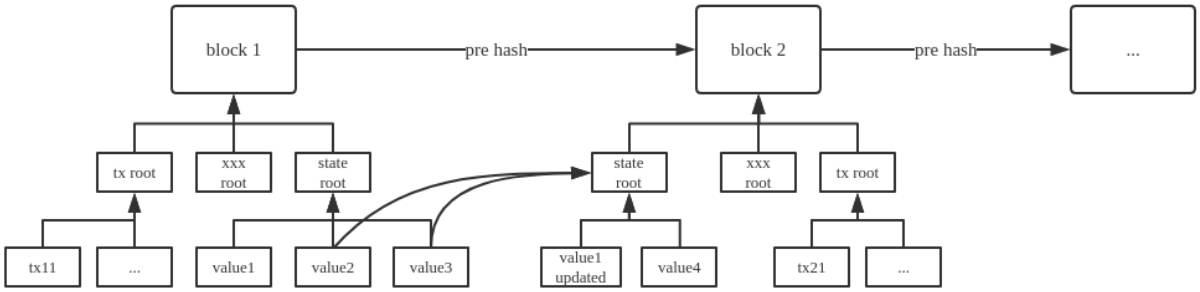
Dipperin-core User run Dipperin-core program to join the Dipperin network. Node programs reach consensus through P2P communication, and the nodes expose RPC functional interfaces to wallets or other applications.

Each node needs to maintain a full-node data locally, which includes block data and link state data. These data are independently maintained by the Dipperin-core node program, and are not affected by external users and programs. Users can only change these data information by sending the correct transaction package to the chain.

Each Dipperin-core node program can be used as a miner or verifier. After the miner node packs the blocks that meet the difficulty requirements, it broadcasts the blocks to the verifier node through P2P communication for verification. The verified blocks are broadcasted to the whole network after the voting consensus is reached through P2P communication between the verifiers.

8.3 Linked Data Structure

We use linked data structures in block and state data.



- Block data:

  Blocks are connected in series by the hash of the former block contained in the latter block. The hash calculation of each block contains information such as transaction and status, so users can not change the data in the block chain at will. Even if they modify the data on their own nodes, they can not pass the consensus check of other nodes.

- State data:

  In Dipperin, state data includes: balance of address, address used to calculate priority parameters, verifier registration list, etc. These data are saved to KV database through Merkle Patricia Tree, and MPT root calculated from them will be put into block header to verify in consensus conditions.

8.4 Transaction

There are different types of transactions in Dipperin. Different transactions are distinguished by counterparty addresses,

Example:

ordinary counterparty address:

```
0x0000 + counterparty's PubKey Hash,
```

registered counterparty address:

```
0x0002 + 00000000000000000000000000000000000000000000000000
```

1. **Normal transaction**

   User make normal transaction to transfer their coins. This kind of transaction only affect accounts' balances.

2. **Verifier register transaction**

   User register as potential verifier by sending verifier register transaction. User specified amount of money will be frozen as a deposit.

3. **Verifier logout transaction**

   If a registered verifier would like to logout, he/she should make a verifier logout transaction. After logout, the user will not be selected as the verifier, but this transaction will not redeem the user's deposit.

4. **Verifier unstake transaction**

   Users can send unstake transaction only after two slots of sending verifier logout transaction to retrieval the deposit. If there is no malpractice evidence transaction towards him/her during this period, which results in the deduction of the user's deposit, the deposit can be successfully redeemed.

5. **Verifier malpractice evidence transaction**

   Once someone discovers that a verifier signs two different block at a same round, he/she can issue a evidence transaction. After the Dipperin cluster confirms the evidence is correct, the verifier deposit is deducted.

6. **Smart contract transaction**

   User deploy smart contract or call contract functions by sending smart contract transactions.

8.5 Various types of nodes

Dipperin-core is the main program of a node. The communication between nodes is realized through the RPC interface provided by Dipperin-core. Although Dipperin-core provides a way to register third-party services, users can embed their own RPC interface and monitor block events at the code level, we do not recommend such strongly coupled integration.

- **wallet**

  Dipperin-wallet is the minimal Dipperin-decentralized wallet. The wallet will start a Dipperin-core, supports sending transactions, balance queries and other operations through RPC.

- **Dipperincli**

  Dipperincli is the command line tool, can perform all defined operations to the node through RPC interface. It is more powerful than Dipperin-wallet. User can start mining or work as verifier by run commands in Dipperincli.

- **Dipperin**

  There are some clients such as browser plug-ins, which can not synchronize a large amount of data to local. Users can open the RPC interface of a node to the outside and serve browser plug-ins and other applications. User can keep their private keys always stored in client sides, only send signed transactions to the server node when need.

8.6 Future work

We would like to keep our main net simple and secure. The next phase of our plan will focus on complementing several key block chain technologies to the main network:

1. **Virtual machine**

   Under the premise of guaranteeing the efficient operation of the main network, Dipperin introduces the mechanism of virtual machine in the side chain to ensure that it can satisfy the scenarios of unlicensed innovation and secure multi-party computing in the future.

2. **Ultra-light node**

   There are many security risks in the light client implemented by API provided by the central server, such as the central server doing evil, maliciously deceiving the user's private key and so on. Therefore, the implementation of ultra-light node is helpful to solve the situation of these centralized servers doing evil.

3. **Cross-chain transaction**

   At present, whether in public or alliance chains, there will be a scenario of multi-chain communication. Dipperin will also promote optimization in multi-chain communication, cross-chain transaction and so on.

CHAPTER 9

Command Line Tool

- *How to use command line tool*
- *How to operate Test Node*
- *Related Functional Operations*

9.1 How to use command line tool

9.1.1 Connect to test environment

dipperin command line tools are located in the $GOBIN directory: `~/go/bin/dipperincli`.

Monitor for test environment: `http://10.200.0.139:8887`.

PBFT whole process demonstration: `http://10.200.0.139:8888`.

If the startup command line needs to manipulate other startup nodes, it can be done by specifying parameters in the startup command.

Example:

Assuming that the local cluster is currently started, the command line tool needs to manipulate the V0 node.

IP: `127.0.0.1`.

HttpPort: `50007`.

```
dipperincli -- http_host 127.0.0.1 --http_port 50007
```

Connect to the test environment:

```
boots_env=test ~/go/bin/dipperincli
```

Or set temporary environment variables first:

```
export boots_env=test
```

9.1.2 Start dipperin node

The following command is to start a node, which requires a wallet password.

If no wallet path is specified, the default system path is used: `~/.dipperin/`.

```
dipperincli --node_type [type] --soft_wallet_pwd [password]
```

Example:

Local startup miner:

```
dipperincli --node_type 1 --soft_wallet_pwd 123
```

Local startup miner(start mining):

```
dipperincli --node_type 1 --soft_wallet_pwd 123 --is_start_mine 1
```

Local startup verifier:

```
dipperincli --node_type 2 --soft_wallet_pwd 123
```

Connect to the test environment:

```
boots_env=test ~/go/bin/dipperincli -- soft_wallet_pwd 123
```

9.1.3 Error

If dipperincli started in a wrong way, it may be that the local link data is not synchronized with the link state, and the local link data needs to be deleted:

```
cd ~
rm .dipperin -fr
```

restart command line tool

9.2 Related Functional Operations

Separate multiple parameters by','

```
[ModuleName] [MethodName] -p [parameters]
```

9.2.1 Transaction methods

AnnounceERC20:

```
tx AnnounceERC20 -p [owner_address],[token_name],[token_symbol],[token_total_supply],
↪[decimal],[gasPrice],[gasLimit]
tx AnnounceERC20 -p 0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,wjw,dip,10000,3,
↪10wu,100000
```

(continues on next page)

(continued from previous page)

ERC20Transfer:

```
tx ERC20Transfer -p [contract_address],[owner],[to_address],[amount],[gasPrice],
↪[gasLimit]
tx ERC20Transfer -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130,
↪0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0000970e8128aB834E8EAC17aB8E3812f010678CF791,1000,10wu,100000
```

ERC20TransferFrom:

```
tx ERC20TransferFrom -p [contract_address],[owner],[from_address],[to_address],
↪[amount],[gasPrice],[gasLimit]
tx ERC20TransferFrom -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130,
↪0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0000970e8128aB834E8EAC17aB8E3812f010678CF791,1,10wu,100000
```

ERC20Allowance:

```
tx ERC20Allowance -p [contract_address],[owner],[spender]
tx ERC20Allowance -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130,
↪0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0000970e8128aB834E8EAC17aB8E3812f010678CF791
```

ERC20Approve:

```
tx ERC20Approve -p [contract_address],[owner],[to_address],[amount],[gasPrice],
↪[gasLimit]
tx ERC20Approve -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130,
↪0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0000970e8128aB834E8EAC17aB8E3812f010678CF791,1000,10wu,100000
```

ERC20Balance:

```
tx ERC20Balance -p [contract_address],[owner_address]
tx ERC20Balance -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130,
↪0x0000970e8128aB834E8EAC17aB8E3812f010678CF791
```

ERC20GetInfo:

```
tx ERC20GetInfo -p [contract_address]
tx ERC20GetInfo -p 0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130
```

Register verifier:

```
tx SendRegisterTx -p [stake],[gasPrice],[gasLimit]
tx SendRegisterTx -p 1000dip,1wu,21000
```

Unregister verifier:

```
tx SendCancelTx -p [gasPrice],[gasLimit]
tx SendCancelTx -p 1wu,21000
```

Redemption of the deposit:

```
tx SendUnStakeTx -p [gasPrice],[gasLimit]
tx SendUnStakeTx -p 1wu,21000
```

Send transaction:

```
tx SendTx -p [to],[value],[gasPrice],[gasLimit]
tx SendTx -p 0x0000970e8128aB834E8EAC17aB8E3812f010678CF791,100dip,1wu,21000
```

Create contract:

```
tx SendTransactionContract -p [from],[value],[gasPrice],[gasLimit] --abi [abiPath] --
↪wasm [wasmPath] --is-create --input [params]
tx SendTransactionContract -p 0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,0dip,1wu,
↪5000000 --abi /home/qydev/testData/token-payable/token-payable.cpp.abi.json --wasm /
↪home/qydev/testData/token-payable/token-payable.wasm --is-create --input liu,wjw,
↪123456
```

Get contract address:

```
tx GetContractAddressByTxHash -p [txHash]
tx GetContractAddressByTxHash -p␣
↪0xb57c391ee4993a1b05712806eff7646c014e29882a2062fc29249d5339a72863
```

Estimate gas:

```
chain EstimateGas -p [from],[value],[gasPrice],[gasLimit] --abi [abiPath] --wasm␣
↪[wasmPath] --is-create --input [params]
chain EstimateGas -p 0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,0dip,1wu,5000000 -
↪-abi /home/qydev/testData/token-payable/token.cpp.abi.json --wasm /home/qydev/
↪testData/token-payable/token.wasm --is-create --input liu,wjw,123456
```

Call contract:

```
tx SendTransactionContract -p [from],[contract_address],[value],[gasPrice],[gasLimit]␣
↪-func-name [function_name] --input [params]
tx SendTransactionContract -p 0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0014ab28B203Fd254ac6f123cC94D7a91011eFFeaf24,10dip,1wu,5000000 -func-name␣
↪transfer --input 0x00005586B883Ec6dd4f8c26063E18eb4Bd228e59c3E9,100
```

Call contract without state change:

```
tx CallContract -p [from],[contract_address] -func-name [function_name] -input␣
↪[params]
tx CallContract -p 0x0000661A3c6c0955B5E6dbf935f0891aAA1112b9E9ca,
↪0x0014ab28B203Fd254ac6f123cC94D7a91011eFFeaf24 -func-name getBalance -input␣
↪0x00005586B883Ec6dd4f8c26063E18eb4Bd228e59c3E9
```

Get transaction:

```
tx Transaction [TxId]
tx Transaction -p 0xf8dd21db65b2adcb5e3ed3c61475eb66a1653d309b1a82354959fdf58852f023
```

9.2.2 Chain methods

Get current block:

```
chain CurrentBlock
```

Get genesis block:

```
chain GetGenesis
```

Get block by number:

```
chain GetBlockByNumber -p [blockNumber]
chain GetBlockByNumber -p 1
```

Get block by block hash:

```
chain GetBlockByHash -p [blockHash]
chain GetBlockByHash -p ␣
↪0x0f7057ff3e3048ed38c0ac2353e001dad6aded5d825d43fcc924a39221713e4c
```

Get receipt by tx hash

```
chain GetReceiptByTxHash -p [txHash]
chain GetReceiptByTxHash -p␣
↪0xb57c391ee4993a1b05712806eff7646c014e29882a2062fc29249d5339a72863
```

Get receipts by block number

```
chain GetReceiptsByBlockNum -p [blockNum]
chain GetReceiptsByBlockNum -p 100
```

Search logs

```
chain GetLogs -p [jsonFile]
chain GetLogs -p {"from_block":10,"to_block":10000,"addresses":[
↪"0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130"],"topics":[["Transfer"]]}
chain GetLogs -p {"block_hash":
↪"0x000023e18421a0abfceea172867b9b4a3bcf593edd0b504554bb7d1cf5f5e7b7","addresses":[
↪"0x0010Cb4174726E90E3ce09360B5F0488Ab29Fa5aB130"],"topics":[["Transfer"]]}
```

9.2.3 Verifier methods

GetVerifiers:

```
verifier GetCurVerifiers
verifier GetNextVerifiers
```

GetVerifiersBySlot

```
verifier GetVerifiersBySlot -p [slotNum]
verifier GetVerifiersBySlot -p 10
```

VerifierStatus

```
verifier VerifierStatus
```

9.2.4 Personal methods

Look up local wallet:

```
personal ListWallet
```

Look up local wallet account:

If the wallet type and path are not specified, the default wallet is displayed

```
personal ListWalletAccount -p [walletType],[walletPath]
personal ListWalletAccount -p SoftWallet,/home/qydev/tmp/dipperin_apps/default_v0/
↪CSWallet
```

Create new wallet:

```
personal EstablishWallet -p [walletType],[walletPath],[password]
personal EstablishWallet -p SoftWallet,/tmp/TestWallet,123
```

Recovery wallet:

```
personal RestoreWallet -p [walletType],[walletPath],[password],[passpharse],
↪[mnemonic],...,[mnemonic]
personal RestoreWallet -p SoftWallet,/tmp/TestWallet2,123,,plastic,balcony,trophy,
↪fuel,vacant,inmate,profit,rival,mimic,cute,hurdle,pig,column,pudding,visit,edge,
↪rhythm,armed,cook,federal,amount,stock,damp,bring
```

Open wallet:

If the wallet type and path are not specified, the default wallet is displayed

```
personal OpenWallet -p [walletType],[walletPath],[password]
personal OpenWallet -p SoftWallet,/tmp/TestWallet3,123
```

Close wallet:

If the wallet type and path are not specified, the default wallet is displayed

```
personal CloseWallet -p [walletType],[walletPath]
personal CloseWallet -p SoftWallet,/tmp/TestWallet3
```

Add account:

If the wallet type and path are not specified, the default wallet is displayed

```
personal AddAccount -p [walletType],[walletPath]
personal AddAccount -p SoftWallet,/tmp/TestWallet3
```

Get account current balance:

```
personal CurrentBalance -p [address]
personal CurrentBalance -p 0x0000e447B8B7851D3FBD5C6A03625D288cfE9Bb5eF0E
```

Get account deposit:

```
personal CurrentStake -p [address]
personal CurrentStake -p 0x0000e447B8B7851D3FBD5C6A03625D288cfE9Bb5eF0E
```

Get account nonce:

```
personal GetAddressNonceFromWallet -p [address]
personal GetAddressNonceFromWallet -p 0x00001c2beC8E0E4caac668cD75d520E41f827092Ce79
```

9.2.5 Miner methods

Start mining:

```
miner StartMine
```

Stop mining:

```
miner StopMine
```

Set miner address:

```
miner SetMineCoinBase -p [address]
miner SetMineCoinBase -p 0x0000e447B8B7851D3FBD5C6A03625D288cfE9Bb5eF0E
```

CHAPTER 10

yellow paper

10.1 Abstract

Dipperin is a public chain positioned for financial applications. At present, the major bottlenecks of mainstream public chain for financial applications are poor performance, insufficient security, lack of privacy protection and difficulty in supervision. Dipperin applies the following design philosophy: the decentralized level not weaker than Bitcoin, the performance that meet ten million degree DAU, permissionless innovation supported, formalized high frequency applications and the standardization of cross connector and asynchronous trading mechanisms. Dipperin sets up several kinds of roles, such as ordinary nodes, sharding servers, ordinary miners and verifiers, and applies the DPoW consensus algorithm to set working mechanism for each category based on the reasonable layered architecture. By separating the miners' and the verifiers' work, which is the feature the most interesting, Dipperin retains the level of decentralization, reduces the possibility of fork attacks and simultaneously greatly improve trading TPS. A clear economic reward and punishment mechanism plays an important role in the stable operation of the blockchain while the successful implementation of this mechanism depends on fair election methods. Dipperin has developed a cryptographic sortition algorithm based on users' reputation, which realizes the decentralized, fair, unpredictable and non-brittle election mechanism, ensuring the fairness and justice of the reward and punishment mechanism in order to maintain the stable operation of the blockchain ecology.

10.2 1 introduction

From white paper of Bitcoin published by Nakamoto in 2008 to the Bitcoin came out in 2009, and to the Ethereum shown up in 2014 , more and more distributed applications have sprung up. But all of these distributed systems have serious shortcomings in performance, security, and privacy. The current transaction performance of Bitcoin is about 7TPS, and with the increase of a new block every ten minutes, the storage requires more and more disk space , which has reached about 200G so far.

Upon analysis of the existing popular main public blockchains, it is essential to create a public chain that overrides the shortcomings in performance, security, privacy, efficiency, and decentralization level. Dipperin is created based on the expectation and is usable in the financial domain.

Bitcoin uses POW to enable the system to reach a consensus and elect a node that has block-packaging rights at a specific time. However, in addition to the huge problem of scalability in POW, there is another problem that has been

criticized, that is, all nodes in the system need to search for suitable random numbers and perform hash calculation without stop. This process requires a lot of power consumption, which causes a huge waste of energy. Futhermore, since the workload is calculated independently among the miners in the POW consensus, the search spaces of the random numbers are overlapped with each other, resulting in inefficiency in the completion of the workload when the difficulty value is fixed. This indirectly leads to system latency and low TPS. Dipperin has proposed a Deterministic PoW (DPoW) consensus algorithm, which is a miner verifier separation mechanism. Miners use the map-reduce PoW method, and the verifiers confirm the block through the Byzantine Fault Tolerance. Comparing with traditional POW, DPOW increases the TPS greatly by decreasing the difficulty value while retaining the bitcoin decentralization level, therefore reducing energy consumption and improving efficiency.

DPoS is prone to corruption problems, for example EOS representative elections of some important candidates are questioned to bribery. The selection plan of PoW assumes that most of the computing power is loyal, but it seems more reasonable that most of the money is loyal. Algorand uses cryptographic sortition, assuming that most of money is loyal. It selects verifiers randomly, and users with more money are more likely to be selected, which is more reasonable than the first two. However, Algorand's approach may need 4-13 rounds of interaction to reach consensus on a block and the block may be empty, and its throughput is not high enough. Inspired by Algorand, we have proposed a method based on user reputation, using VRF to randomly select verifiers and using PBFT to reach consensus. This approach is really fair for all potential verifiers.

The blockchain is based on the P2P network for data exchange. The P2P mode is different from the traditional client/server mode. Each node can be both a client and a server, so the HTTP protocal is not suitable for communication between nodes. Dipperin has structured the P2P network and realizes the DHT-based network model and fast routing lookup between points via the Kademlia algorithm. Block synchronization is an important part of communication between nodes. Dipperin uses reversible Bloom filter technology, so that nodes need only very small network bandwidth, and block synchronization can be realized by just one communication.

So far there are no real public chain with native multi-chain system. The market has some popular cross-chain trading systems, using notary public, two-way anchoring or hash-based locking, while they cannot completely shake off the centralization problem. Based on distributed signature technology, Dipperin combines zero-knowledge proof and homomorphic encryption technology to achieve a decentralized multi-chain system. As a native multi-chain system, Dipperin uses a mode of cooperation between the main chain and the side chain. The main chain and the side chain each have their distributed network, miners group, consensus mechanism, and digital assets. They operate in parallel and do not interfere with each other.

For technology details, we will begin with the Dipperin architecture, and move to P2P networks, block synchronization, mining and verification algorithms, verifier elections, block storage, wallets, and multi-chain systems.

10.3 2 The Structure of Dipperin

10.3.1 2.1 Block, Transaction and Account State

2.1.1 Block

In Dipperin, the block is composed of the following parts: a set of some related information pieces (named block header), and the transactions, signatures of verifiers and interlink of the super spv.

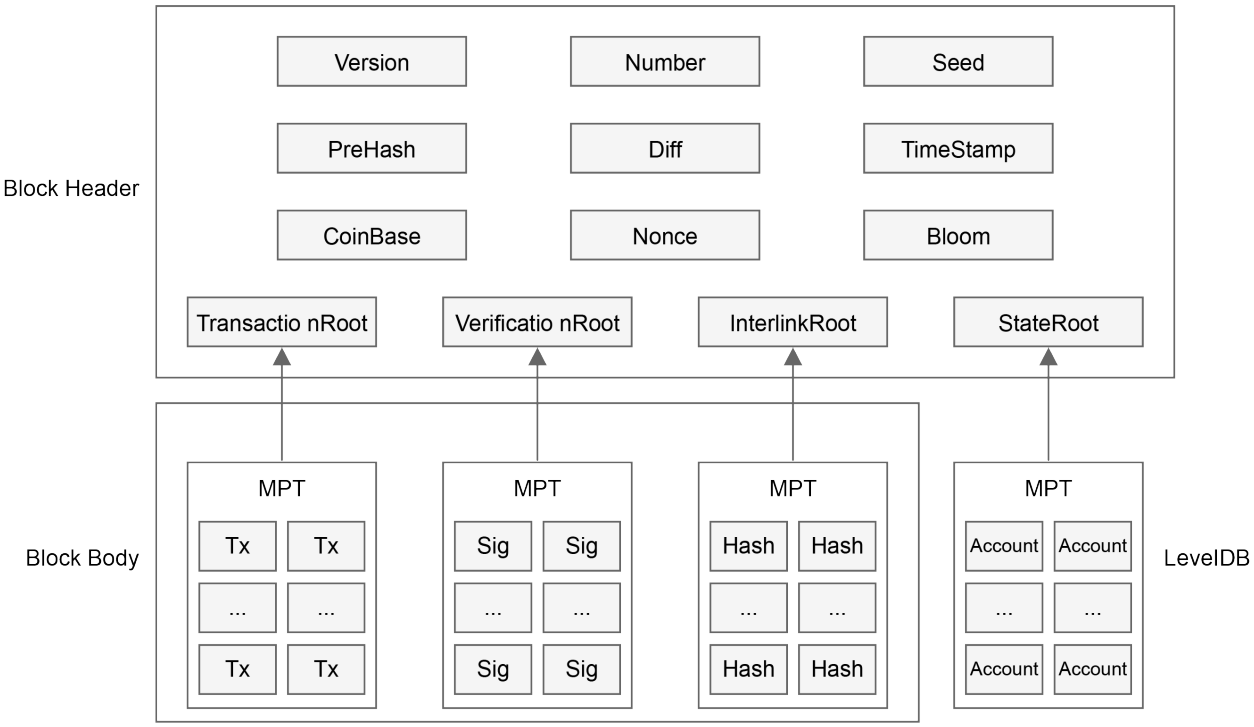
```
type Header struct {
        Version uint64                          // the corresponding version id of the␣
↪block, use different consensus conditions or different block handling methods for␣
↪compatibility with subsequent potentiel upgrades
        Number uint64                          // the height of the block
        Seed  []byte                           // the seed used for cryptographic␣
↪sortition
        Proof []byte                          // the VRF proof of the seed
```

(continues on next page)

(continued from previous page)

```
        MinerPubKey []byte                  // miner public key
        PreHash []byte                        //the hash of the last block
        Diff uint64                          // difficulty for this block
        TimeStamp []byte                    // timestamp for this block
        CoinBase []byte                    // the address of the miner who mined this␣
↪block
        Nonce uint64                          // nonce needed to be mined by the miner
        Bloom []byte                         // reserved field, used for bloom filter
        TransactionRoot []byte                // the hash of the transaction trie␣
↪composed of all transactions in the block
        StateRoot []byte                  // MPT trie root for accounts state
        VerificationRoot []byte               // MPT trie root for committed message
        InterlinkRoot []byte                 // MPT trie root for interlink message
        RegisterRoot []byte                 // MPT trie root for register
}
```

In the above four roots, the MPT trie structure corresponding to TransactionRoot, VerificationRoot and InterlinkRoot are stored separately in the block body, and the state MPT trie corresponding to StateRoot is stored in LevelDB. A discussion of the MPT tree will be discussed in detail in the storage section. The corresponding block diagram is as follows:



In this diagram we see that the block header is composed of items listed in the struct above, and the body is composed of items of transactions, of Witnesses and Hashes of interlinks. Of course, there is a huge MPT tree structure which is composed of accounts and this tree is not stored in the blockchain. But its hash root is stored in the block header.

2.1.2 Transaction

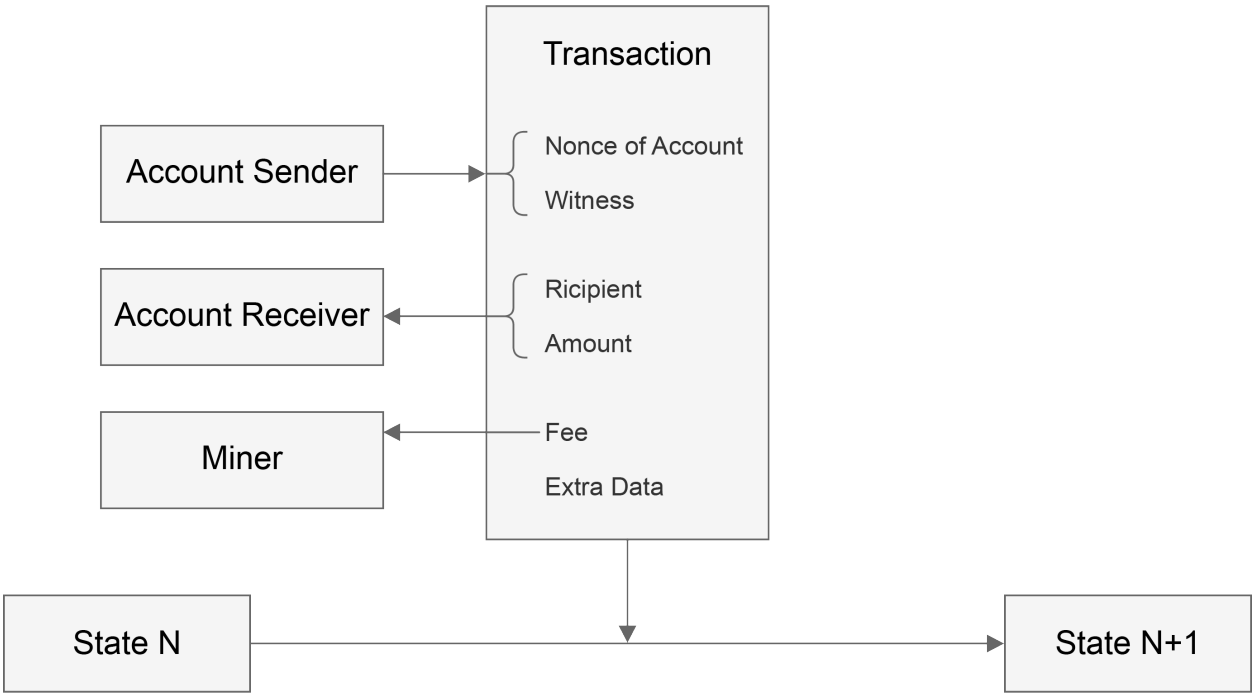
```
type Transaction struct {
        AccountNonce uint64 // The nonce value of the account who has launched this␣
↪transaction
```

(continues on next page)

(continued from previous page)

```
        Recipient []byte // the counterparty of this transaction which has a length␣
↪of 22 bytes where the first 2 bytes are used to mark the type of the transaction␣
↪and the last 20 are the address of the counterparty
        Amount []byte // transaction amount
        Fee []byte // the trasaction fee for the miner
        ExtraData []byte // additional remarks for the transaction, if it is a␣
↪contract transaction,then it's accompanied by the data of the contract operation
        R []byte // the R part of the originator’s signature on the transaction
        S []byte // the S part of the originator's signature on the transaction
        V []byte // the V part of the originator's signature on the transaction
        HashKey []byte // the hashkey of the originator's signature on the transaction
}
```

The transaction structure is as follows



In this diagram we see that the transaction launches the transition of state. The content in the transaction structure is as explained in the code block where Witness corresponds to R,S,V and HashKey. The address of the sender can be recoverd from the Witness and the nonce of account is also information of the sender.

2.1.3 Account State

```
type account struct {
        Nonce uint64 // the total number of transactions originated by this account
        Balance []byte // the balance of this account address
        Stake []byte // the amount of the deposit in this account
        CommitNum uint64 // the total number of messages committed by this account
        Performance uint64 // the performance as verifier of the account, which is an␣
↪         important factor of the reputation
        VerifyNum uint64 // the number of blocks that this account should verify in␣
↪total
        LastElect uint64 // the height of last elect transaction. The stake cannot be␣
↪retrieved within 4 periods after the height
```

(continues on next page)

(continued from previous page)

```
        ContractRoot []byte // the root hash of the contract trie created by this␣
↪account
        DataRoot []byte // the root hash of the tree structure of the data generated␣
↪by the application of the account executed off the chain
}
```

10.3.2 2.2 Fee Payments

In addition to the cost of the transaction, in order to avoid the abuse of blockchain by individuals, Dipperin charges the contract data used by the user and the data stored by the custom application, which will be included in the accounts of miners and verifiers.

10.3.3 2.3 Transaction Execution

All transactions sent to the Dipperin network can be packaged by the miners, verified by the verifier, and finally submitted to the chain after verification by consensus conditions. In the process of submitting the transaction to the chain, Dipperin will perform the operations attached to the transaction and modify the state of the chain. Different operations will be done for different types of transactions by Dipperin:

- normal transaction, modify the account balance of the sender and the receiver in the state trie
- register transaction, modify the balance and the stake of the sender in the state trie
- cancel transaction, modify the balance and the stake of the sender in the state trie
- evidence transaction, deduct the deposit from the reported party in the state trie
- contract transaction, modity the contract data in the state trie
- storage transaction, modify the storage data in the state trie

10.4 3 Consensus Mechanism of Dipperin

In the blockchain system there is no centralized node, which makes the system totally different from traditional centralized server. So in this decentralized environment, how to reach a unified opinion on a problem or proposal requires a consensus mechanism. The core issue for bitcoin or other blockchain systems is how to reach a consensus on a proposal in a decentralized environment.

However, in the blockchain consensus mechanism, there is always an impossible triangular relationship among decentralization, consistency and scalability, namely DCS triangle. That means, at most 2 of these 3 features can be fullfiled for any consensus mechanism in the blockchain system.

In the current blockchain system, the main consensus mechanisms are as follows: POW, POS\DPOS, PBFT, etc., but they all have their own defects. POW pursues a high degree of consistency and decentralisation at the cost of scalability, while POS and DPOS sacrifice decentralization and seek high consistency and scalability. In the blockchain system, if the PBFT consensus algorithm is used alone, then it cannot support the communication between a large number of nodes, but only between a small number of nodes. Furthermore, the shortcomings of these consensus mecanisms also constrain the development of blockchain system. Therefore, all current blockchain systems want to design a consensus mechanism that allows them to find the best balance in the DCS triangle of the blockchain consensus, thereby promoting the development of blockchain technology.

10.4.1 3.1 DPOW consensus

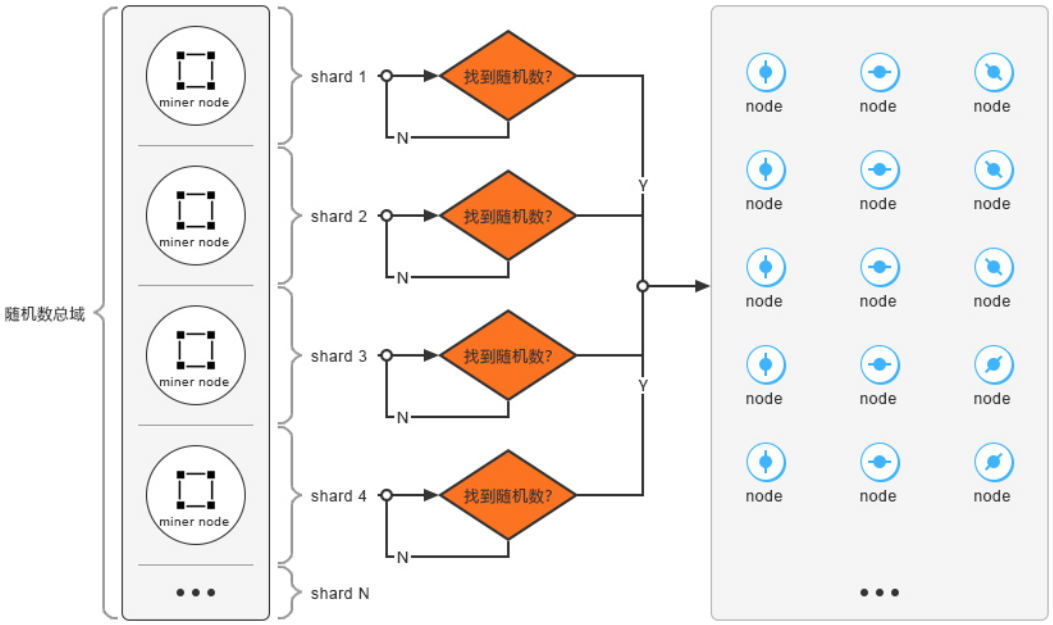
In Dipperin's deterministic proof of work consensus algorithm, nodes are divided into four roles: miners, sharding servers, verifier servers, and common nodes. Each of them assumes different responsibilities in the blockchain system to maintain the stability and security of the system together.

- Ordinary node: It holds the electronic currency circulating in the system and has the right to vote. It has no special responsibility. It can perform ordinary transaction operations and can not vote or be voted in the system. Block data can only be recorded synchronously from the nodes that own packing and accounting rights.
- Sharding Server: It's responsible for packaging transaction table and assign the hash pazzle to the miners
- Miner: It's responsible for resolving the hash puzzle assigned by sharding servers
- Verifier Server: It's mainly responsible for signing the blocks fabricated by miners

3.1.1 MapReduce Mining

The specificity of sharding proof of work lies in that the calculation is accomplished by sharding server and miners togother. The main principle is as follows:

In the pure proof of work mechanism, we find that the miners in the system work seperately. If they execute the proof of work calculations on the same transaction list, then the probability of repeating the work is high, which will result in a longer workload and waste of a lot of power. The process of proof of work using the fragmented POW proposed in this paper is as follows:



1. The sharding server divides the entire search space of proof of work into multiple parts, ensuring that there is no mutual coverage between the divided cells, and assign it to the miner.
2. The miner will calculate the hash value on the assigned search space to seek proof of completion. The miner needs to periodically monitor the message notification of the sharding server during the calculation process in order to respond in time the update of workload search space or reallocation of the task by the sharding server. Once the miner completes the proof of work, the block that meets the conditions is submitted to the sharding server.

3. After receiving the block submitted by the miner, the sharding server submits it to the verification server cluster and waits for the verification.

Here are the pseudo-code for the map-reduce PoW:

```
Precedure map-reduce POW
-------------------------------------------------------------------------
 master.start()
 worker.start()
 master.register(worker) //notify master about the worker
 master.dispatch(work)    //distribute shards to worker
 notFound = true          //miner starts mining
 while notFound do
          worker.change(coinbase) //worker changes coinbase data to make a trial
         pass = worker.seal()     //worker computes hash and returns true if␣
↪successful
         if pass then
                notFound = false
         end if
 end while
 worker.submit(work)          //worker submits proof of work to master
 pass = master.check(work)  //master verifies validation of proof of work
 if pass then
                master.broadcast(block) //master broadcasts the valid block
 end if
```

In the above process, the sharding server will listen to the P2P network and package the transaction, but the sharding server will not construct the Merkle tree of the entire transaction. It will only calculate the Merkle path for the coinbase transaction, and the sharding server will send the following data to the miner:

- the hash value of the last block header
- the difficulty of mining
- the Merkle Path of the coinbase transaction
- the transaction-out part of the coinbase transaction

Apart from the above 4 parts, sharding server will not send miners other information. It waits for the miner to complete the calculation and reply to the message, but it will set a timeout and will not wait indefinitely. Upon expiration, a new fragmentation is reconstructed and assigned to the miners to prevent the system from being blocked because of non completion of the PoW by miners. In addition, when a sharding server completes a PoW proof, the other sharding servers will also package new transactions and reassign tasks to their corresponding miners. The miner performs a hash calculation on the allocated search space after a sharding task is received from the sharding server. First of all, the miner will generate an additional data and splice it into a coinbase transaction with the transaction-out portion of the coinbase transaction sent by the sharding server. This extra data generated by the miner must satisfy a certain format, starting with the miner's ID followed by any additional information. After constructed the coinbase transaction, the miner can use the Merkle path of the coinbase transaction sent by the sharding server to calculate the Merkle root hash value of the transaction list.Then the miner can make the Merkle root hash that meets the specific difficulty value by adjusting the nonce. If the miner can't make the Merkle root hash that meets the specific difficulty value, it can regenerate the extra data and construct a new coinbase transaction. Once a new coinbase transaction is constructed, the entire nonce search space changes, so the miner can perform a workload calculation based on this new search space to obtain a nonce value that satisfies the condition. Since the ID of each miner node is different, the coinbase transactions constructed by each miner are different, so the search space of each miner is also different. Thus, for the same transaction data assigned by the sharding server, the possibility of conflicting proof of work calculations between miners is very low.

Compared with the pure POW consensus mechanism, in the deterministic POW consensus algorithm proposed in this paper, it uses the map-reduce method to distribute the workload to the miners, and avoids the direct repetitive

work of the miners. This mechanism has increased the efficiency of the miners' workload and reduced the energy consumption of the traditional POW consensus mechanism. In addition, compared with the treatment of offline mines, the map-reduce mechanism proposed in this paper has a self-determined protocol mechanism.

3.1.2 Byzantine fault tolerance consensus mechanism of verifiers

In the deterministic POW consensus algorithm proposed in this paper, the problem of uncertainty of traditional POW consensus is solved by reasonable introduction of PBFT consensus. In this consensus mechanism, by means of registration, the system selects the verification server, which is responsible for processing the blocks packaged by the miners. This mechanism is characterized by the combination of POW and PBFT. The dedicated miner is responsible for calculating the hash problem and the packaged block performs the PBFT consensus through the verification server, so that the consensus result becomes deterministic. When the miner completes the workload, it submits the block to the verifiers, who will perform a PBFT consensus to select the block so that it be recorded on the chain. The verifier consensus process is as follows:

After the miner finds a candidate block that satisfies the condition with its nonce value, the signed block needs to be sent to the verifier servers. The verifier cluster is generated through the election process and is recorded on the blockchain. When the checker cluster receives the block sent by the miner, it needs to reach a consensus on the legality of the block submitted by the miner. Once the block is verified, it will be recorded on the blockchain, and other nodes in the system can update synchronously the newly recorded block data via the P2P network. The verifier consensus process is completed by PBFT. For the Byzantine problem, the system can only tolerate faulty nodes less than 1/3 of all nodes. Therefore, in the deterministic POW consensus mechanism proposed in this paper, the number of faulty nodes should also be less than 1/3 for the consensus to function correctly.

In the consensus mechanism of the verifiers, some improvements have been made to the original PBFT protocol. The specific consensus is divided into the following steps:

1. There is a master in the verifier group, and the master will present and distribute the blocks received from the miners to other verifiers. This process is called propose
2. Each of other verifiers executes the verification on the block proposed by master upon reception, and decides whether to vote for the block. If it believes that this block can not be voted, it can vote an empty block called Nil instead. It signs this vote and distribute it to all other verifiers. This process is called prevote.
3. When the number of votes collected for the proposed block of all the verifiers (including itself) reaches 2/3 of the total number within the predetermined time, then the precommit stage of the block is entered, otherwise it enters the precommit stage of the empty block.
4. When the number of precommit information collected for the same block reaches 2/3 of all nodes, it enters the commit phase and adds the block to the chain.

It is worth noting that when a node enters the precommit stage of a certain block, it adds a state lock for this block to itself, which means, before the state lock is released, for any subsequent round it can uniquely vote for this locked block and cannot vote on any other blocks. Similarly, when entering the second step above, each node must at first verify whether it has been locked on a different block in which case it can not accept this proposed block but the empty block.

Since there are locking conditions, there are certainly unlocking conditions. Unlock happens in two situations:

- When the locked block reaches the above condition 4 and is added to the blockchain
- When it finds that in the higher round for the same height, there are other nodes locked on other blocks.

Through the above locking and unlocking mechanism, it is possible to prevent lagged synchronization caused by network delay problems for some nodes, and also prevent malicious nodes from manipulating the network so that different honest nodes be locked on different locks, namely deadlock.

When the number of Byzantine nodes in a checker cluster is less than 1/3, the block validity of any miner to the verifier cluster can be correctly verified, and the system can reach a consensus. When the number of Byzantine nodes

in the verifier cluster is between 1/3 and 2/3, the verifier cluster cannot reach a consensus on the block sent by the miners. When the Byzantine node in the verifier cluster exceeds 2/3, the Byzantine nodes in the cluster can control the consensus result by collusion. Therefore, in the deterministic POW consensus mechanism proposed in this paper, the tolerance of the verifier cluster can only be 1/3.

The deterministic POW consensus described above ensures that when a block is validated, it will be irreversible on the chain because the verifiers carry out the block confirmation process and record it to the blockchain through the PBFT consensus, ensuring that there is only one legal block at one time. As for the pure POW consensus mechanism of Bitcoin, there are probably multiple miners who complete the proof of work at the same time, which may lead to fork. In the case that the longest chain cannot be determined, the consensus result is uncertain. Due to the certainty of the consensus mechanism proposed in this paper, the blockchain system adopting this consensus mechanism will not be limited by the speed of block generation. It avoids the fork problem caused by pure POW consensus, and ensures that the system is always a longest chain without fork. By introducing PBFT to reach a consensus on the basis of traditional POW, the scalability of the system is enhanced, so that the consensus of the system is not completely dependent on the computing power competition, and the transaction can be quickly confirmed. Since the consensus between the verifiers is achieved by adopting the PBFT method, high consistency between the verifiers is also ensured.

In addition, compared with the POS and DPOS consensus mechanisms, in the deterministic POW consensus, each miner and verifier work together to maintain the normal operation of the system. It seperates the rights and uses rewards to combine the nodes. In a blockchain system that uses this consensus, each role can participate in the consensus mechanism, ensuring that the billing rights not be controlled by centralized nodes.

Since the pure PBFT consensus mechanism is not applicable to the blockchain system with too many nodes, the deterministic POW consensus requires that PBFT be used only between verifiers so that the number of nodes in this consensus break through the limitation in the PBFT consensus mechanism.

10.4.2 3.2 Sortition Mechanism Between Verifiers Based on Verifiable Random Function

Of all consensus algorithms, PBFT is particularly characterized by high consistency and efficiency. Nonetheless, at least 2/3 of all nodes in the system should be honest to ensure the safety and liveness of the system. Therefore, it is essential to select honest nodes as verifiers from so vast candidates.

Dipperin has applied a cryptographic sortition for verifiers based on VRF(Verifiable Random Function). Every user is assigned a weight according to its reputation in order to resist Sybil attack. This sortition mechanism ensures that only a small percentage of users are selected, and the chance of being selected is propotional to its weight. Futhermore, their identity can be verified by all users. Random results cannot be predicted in advance and cannot be manipulated by any malicious adversory. Dipperin's sortition mechanism provides objective security, that is, the whole process is objective, and decisions are made entirely through calculations. Human intervention cannot affect this process.

3.2.1 Weighted Users

Reputation is very important in business. Our system quantifies the reputation of users and measures the weight of users by reputation. The verifier is selected by lottery, and the candidate with higher reputation has more chance of being selected. Under hypothesis that 2/3 of the network's reputation is good, the chain's security can be guaranteed. We believe that reputation-based weights are more fair than weighting method based on computing power or based on stocks.

Reputation : Reputation = F(Stake, Performance Nonce)

There are three factors for measuring reputation, stake, performance and nonce.The stake deposit determines the cost of cheating and is introduced to defend Sybil attacks. More deposit means more reputation if other conditions are equal.Performance represents the user's past working performance as a verifier. More performance means more reputation as well if other conditions are equal. The production of performance depends on use's activity each time as a verifier and the average commit rate.

Nonce is the number of transactions related to this account. Other conditions being equal, higher number of transactions means more reputation. The introduction of the number of transactions is to defend the Sybil attack. For example, the malicious adversory deposits money into multiple new accounts and tries to participate in the election. If this is the case, the account's Nonce will be low and this will have a significant impact on reputation. If Nonce is in a normal range, this factor has little effect.

3.2.2 Cryptographic Sortition

The role of cryptographic sortition is to select candidates as block proposer or verifier whose identity can be verified by all other users.

The implementation of cryptographic sortition uses VRF: In the VRF algorithm, for a common input, the user can generate a hash value and a proof using its private key SK and a hash function. Other users can use this hash value and the proof, combined with the user's public key PK, to verify whether the hash value is generated by the user for input. In the process, the user's private key is not leaked at all from beginning to end. The user is authenticated in this way, and other users can believe his role as a verifier for a certain period of time. In this way, a set of users can be randomly selected through a common input and their identity can be verified by others.

```
Procedure Sortition(Stake, PerformanceNonce, Seed)
------------------------------------------------------------------------
reputation = Reputation(Stake, PerformanceNonce)

<hash, proof> = VRF(PrivateKey,Seed)

priority = Priority(hash,reputation)

return (priority, proof)
```

The purpose of our introduction of reputation is to make high-credit users more likely to be selected. But whether a certain user can be selected is not a deterministic event. Therefore, it is necessary to generate a random number seed that can be generally accepted by all nodes in this distributed network. In each round we need a seed that cannot be controlled by any attacker or be predicted in advance. Let us discuss now how this seed is selected.

3.2.3 the production of verifiable random value

Production of Seed The Seedr of the r round exists in the r-1 block, of which the proposer is selected by the VRF algorithm. Seedr, proof := VRF(SKr-1 proposer , Seedr-1) Verifiable random number generation:The user takes Seed as input and uses VRF to calculate the random number and Proof. <hash, proof> = VRF(PrivateKey,Seed)

```
Procedure VRF(sk,input)
------------------------------------------------------------------------
h = hash_to_curve(input)
r =  h^sk
hash = H(r)

k = radom()
pk = g^sk
c = Hash(g,pk,h,g^k,h^k)
s = k - cx mod q
proof = (r,c,s)

return (hash, proof)
```

Given a user's public key, we are able to deduce whether this pseudo-random hash is generated by the user and this input

```
Precedure VerifiyVRF(input, pk, hash, proof)
----------------------------------------------------------------------
h = hash_to_curve(input)
(r,c,s) = proof
u = (pk)^c * g^s
v = r^c * h^s

if H(g,PK,h,u,v) = c && hash = H(r)
    return ture
else
    return false
```

3.2.4 Priority

Combining the user's reputation Ri with the above random seed, we can use the following algorithm to generate priority for the user. Based on these verifiable prioritizations, the node is able to recognize if it is a verifier, and other nodes can easily identify whether this node has the right to perform the verification:Suppose this seed produces a random number Ui that is evenly distributed between 0 and 1 for each user. Then the user's priority

Priority=Ri×Ui

10.4.3 3.3 How Dipperin Resists Attacks

The consensus mechanism is the soul of the blockchain. The establishment of a reasonable consensus mechanism is the core weapon for the network to resist various internal and external attacks. Dipperin adopts the DPoW consensus and the VRF-based verifier election mechanism, which is better than the traditional PoW and DPoS consensus algorithm in resisting attacks.

3.3.1 Sybil Attack

The attacker destroys the reputation system of the peer-to-peer network by creating a large number of pseudonym identifiers and uses them to obtain a disproportionate level of control over the peer-to-peer network. For example, the malicious adversory can try controlling the block production process by occupying the verifier seats.PoW naturally are resistant to Sybil attacks. In the PBFT verification process of Dipperin, due to the comprehensive consideration of the deposit, the number of transactions and the performance as verifier into the calculation of the weight, the creation of multiple accounts by candidates will inevitably lead to lower weight of each account, thus reducing the possibility of begin elected.

3.3.2 Bribery Attack

The EOS representative election was questioned by bribery. The Dipperin verifier election process is transparent. The election result cannot be manipulated by human factor so that there is no person capable of predicting the result, and no miner capable of bribing verifiers to package specific blocks.

3.3.3 51% Attack

Bitcoin's PoW consensus algorithm cannot resist the 51% attack. By seperating PBFT verification from POW, since each block without the signatures of the verifiers cannot be accepted by nodes, a node or group of nodes who own even 51% computing power cannot restart a new branch in the Dipperin system.

3.3.4 Seed Dependant Attack

The luck of the Dipperin verifier candidate is produced by the seed, which is generated by the miners who dig into the last block of each round. It seems that the miner can try to control the seed and thus the value of the luck by iterative calculations.

While in fact, it is very difficult for miners to do so. Firstly, the miners has no idea in advance whether the blocks they have dug will be taken into the consensus round, and doing so will increase the workload and decrease the possibility of being chosen. Secondly, the sortition of each person is encrypted. Only when the seed is published, each potentiel verifier can check his priority by performing VRF calcultion using seed and ites private key before broadcasting to other nodes for verification. That is to say, even if the miners know their seed in advance, they can only obtain their own priorities without the others', and they have no idea whether they can successfully be elected as a verifier.

10.5 4. The P2P network transmission of Dipperin

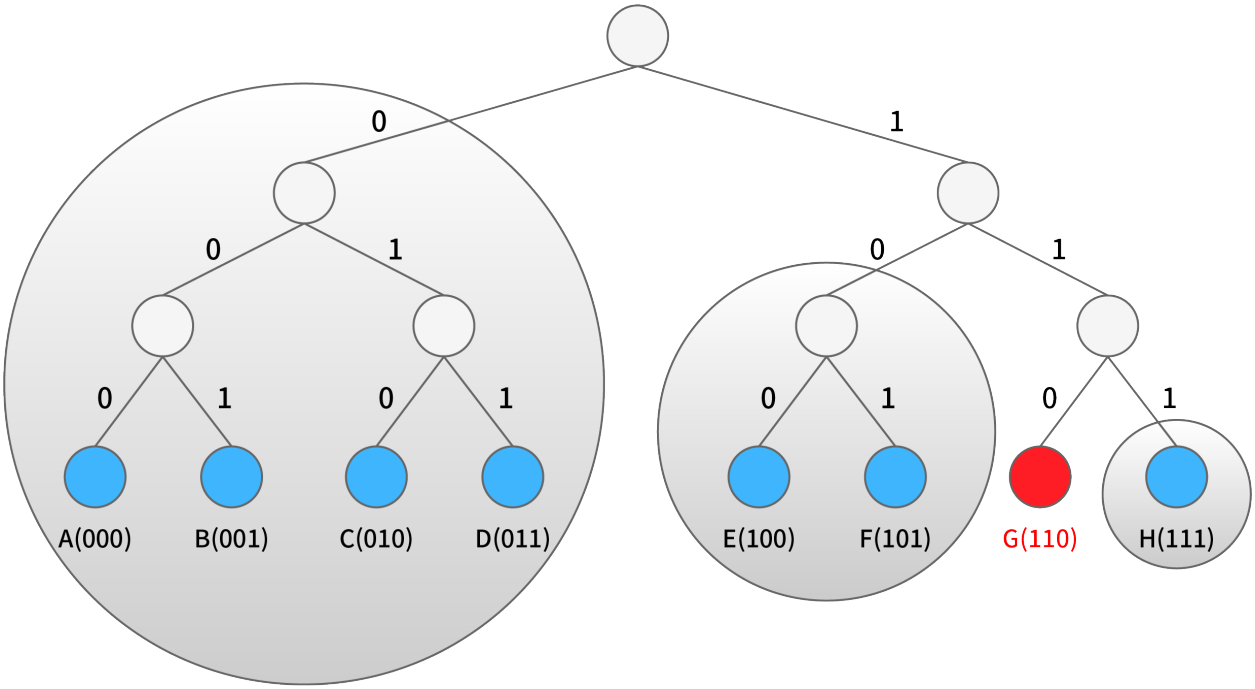
Dipperin uses DHT(Distributed Hash Table) at P2P network structure, to improve searching effectiveness between nodes and the P2P network capability that defend DOS(Denial of Service) attack. In this case even if a whole batch of nodes in the network were attacked, the availability of the network would not be significantly affected. Dipperin uses Kademlia algorithm to realize DHT.

10.5.1 4.1 The routing lookup of Kademlia

Each Node in Kademlia network will be assigned a unique Node ID. Dipperin adopts the public key generated by a temporary private key of the Node. The distance between nodes can be calculated and measured by the XOR value of the Node ID.

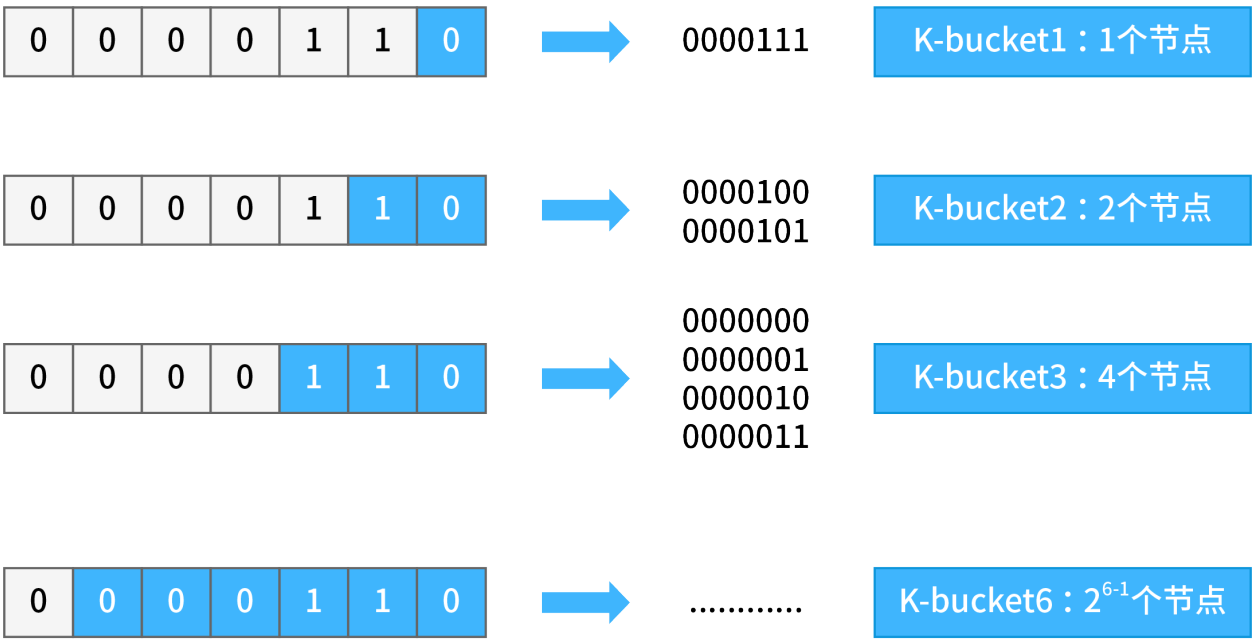
```
Distance(X, Y) = X.NodeID XOR Y.NodeID
```

Each node produces a list(K-bucket) according to the XOR value of other nodes where K is a system variation, for example 20. Each K-bucket is a list that includes no more than K items, which means one list of all nodes in the network (corresponding to a certain bit, a specific distance from the node) contains at most 20 nodes. As the corresponding bit position becomes lower (the corresponding exclusive or distance is getting shorter and shorter), the number of possible nodes contained by K-bucket decreases rapidly (this is because the closer the exclusive or distance corresponding to K-bucket is, the fewer nodes it has). Therefore, the K-bucket corresponding to lower bit contains obviously nodes of all relevant parts of the network. Since the actual number of nodes in the network is much smaller than the number of possible ID numbers, the K-bucket corresponding to those short distances may have been empty (If the exclusive-or distance is only 1, the maximum number of possible nodes can only be 1. If the node with the exclusive or distance of 1 is not found, the K-bucket that corresponding to the node with the exclusive or distance of 1 is empty).



Let's take a look at the above simple network who has at present a total of eight nodes each in a small circle (at the bottom of the tree). We consider the node G highlighted in red circle who has three K - bucket, of which the first consists of A, B, C and D (binary representation of 000,001,010 and 011), the second E and F (binary representation of 100 and 101, respectively), the third node H only (binary representation of 111). In the figure, all three K-buckets are represented by gray circles. If the size (that is, the value of K) of the K-bucket is 2, then the first K-bucket can only contain 2 of the 4 nodes. As is known to all, nodes that have been connected online for a long time are more likely to remain online in the future. Based on the law of static statistical distribution, Kademlia prefers to store those nodes into K-bucket, which increases the number of effective nodes at a certain time in the future and provides a more stable network. When a K-bucket is full and a new node corresponding to the bucket is found, the earliest visited node in the K-bucket should be checked and if the node is still alive, the new node should be placed to the list of affiliate (as an alternative cache). The alternative cache is used only when the node stops responding. In other words, newly discovered nodes are used only when the old ones disappear.

Stratification by XOR distance can be generally understood as stratification by bit number. Imagine the following scenario: on the basis of 0000110 nodes, all the digits ahead are the same as the ID of one node and only the last one digit is different. This kind of node has only one - 0000111, whose XOR value with the basic node is 0000001, which means the distance is 1; For 0000110, such nodes are grouped as "K-bucket 1". Consider another scenario: all previous digits of one node ID are the same, and are different only from the last second digit. This kind of node only has two: 0000101, 0000100, whose XOR value with basic node are 0000011 and 0000010, and so the distance range are 3 and 2; For 0000110, such nodes are grouped as "K-bucket 2"



When a node has K-bucket, it will look for other nodes. Suppose node A(node_id:00000110) wants to find node H (node_id:00010000), A need to calculate the distance between H and itself at first;

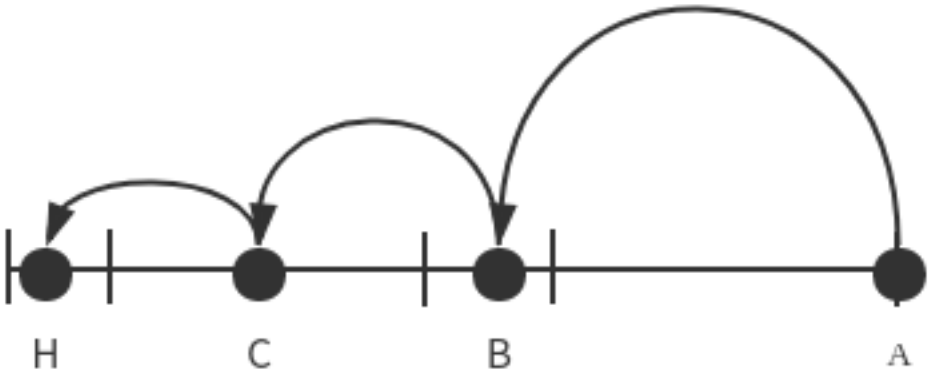
```
Distance(A, H) = A.NodeID XOR H.NodeID = 00000110 XOR 00010000

Distance(A, H) = 00010110
```

The XOR distance of node A with node H is 00010110, so the distance range is [24, 25), so node H may be in the K-bucket 5 (in other words, the Node ID of node A and H differs from the 5th bit, so node H may be in K-bucket 5). Then node A needs to check whether node H lies in its K-bucket 5 ;

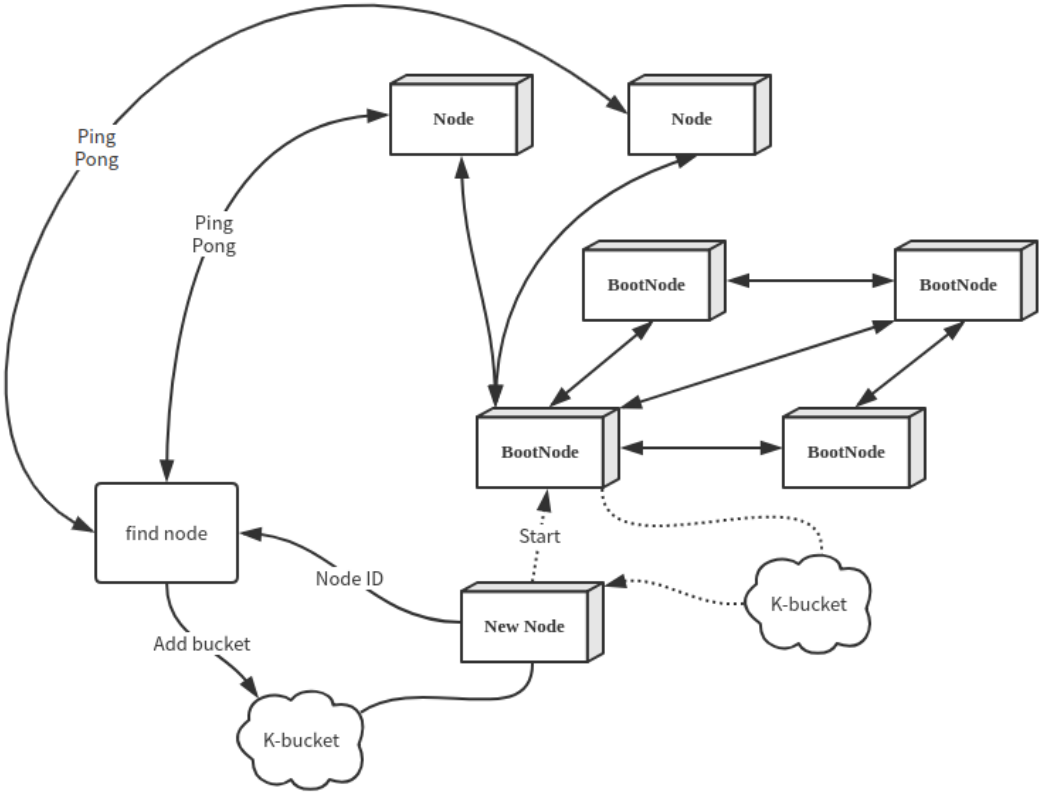
> If so, search the node H directly;
>
> If not, select arbitrarily a node B in K-bucket 5 (for any node B, its fifth digit of Node ID must be the same as Z, which means the distance with node Z will be less than 24, almost half the distance between A and Z), and request node B to search node Z in its own K-bucket by the same method.
>
> > If B knows node Z, it tells the IP Address of node Z to node A;
> >
> > If node B doesn't know node Z, node B can find node C closer to node Z in his own K - bucket by the same method (the distance between Z and C is less than 23), and recommend node C to A; node A requests node C to perform the next search.

Kademlia's query mechanism is a bit like arbitrarily folding a piece of paper to shrink the search range, ensuring that for any n nodes, only log2n times' query are needed to find the contact information of the target node (that is, for any network with [2n1 , 2n) nodes, at most n steps of search are needed to find the target node).



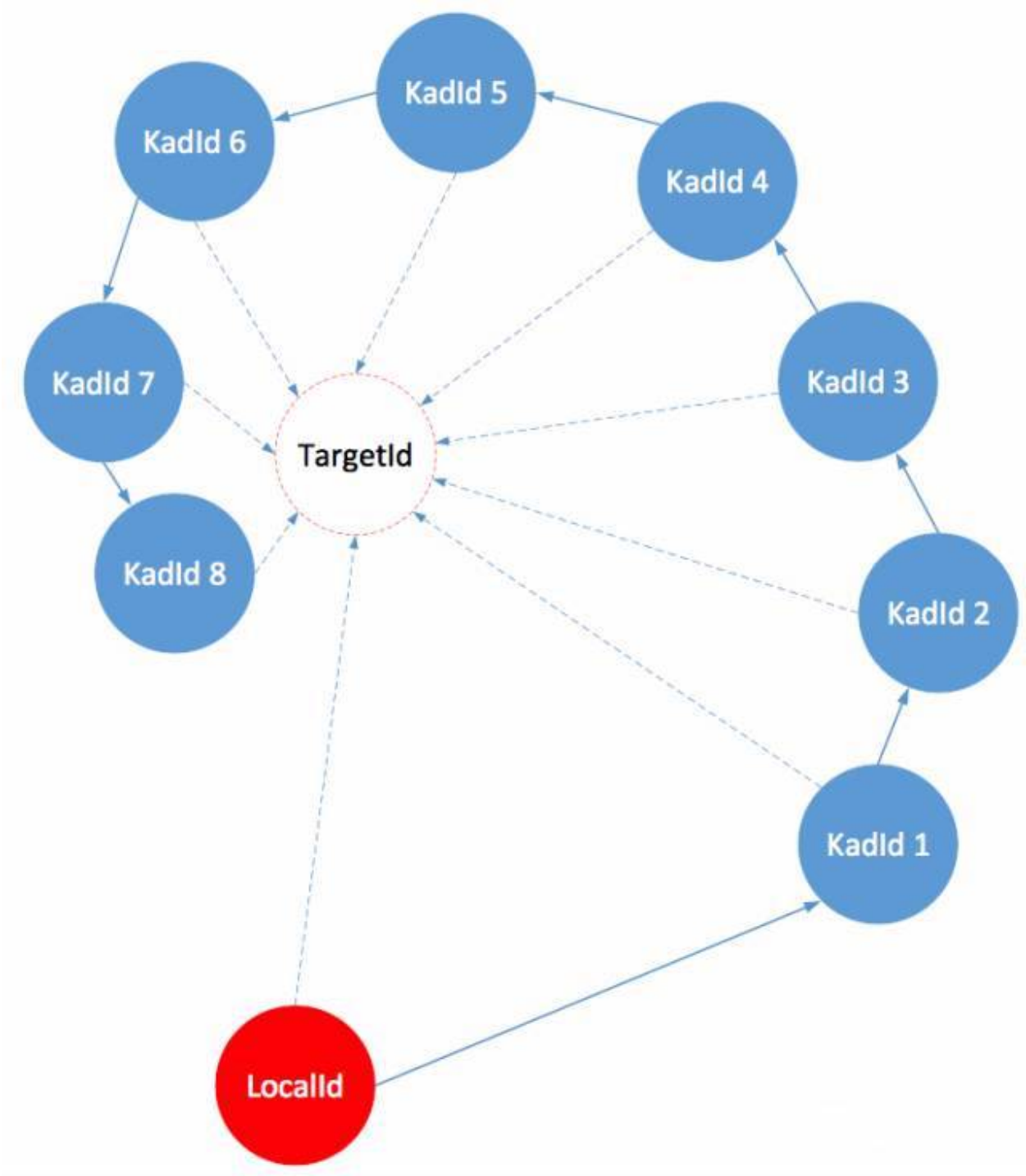
10.5.2 4.2 The concrete realization of P2P network

Before the chain release, Dipperin deploys some start nodes (BootNode), hard-coded in the program. In this way, when these nodes are started for the first time, they will connect automatically the bootnodes, exchange the K-bucket between the nodes, and obtain more nodes ID to make connections, thus joining the entire Dipperin network.



The first time the node starts to read the bootnode information, it updates the local K-bucket with the local node ID as the target, and then updates the node every interval. The process of refreshing the K-bucket is as follows:

1. Randomly generate the target node ID and mark it as TargetId. Record the number of discovery times and refresh the time from 1.
2. Locate the 16 nodes closest to the target node in the K-bucket of the current node
3. Send the findnode command to each of the nodes obtained in step 2, and receive the adjacent nodes returned by each node
4. Ping-pong test on each node returned by step 3 and update to local K-bucket
5. The above discovery processes are all based on UDP. The p2p network will randomly select the unconnected nodes in K-bucket for TCP connection at regular intervals and execute the communication in the connected TCP channels (the TCP connection coroutine will do the heartbeat to maintain this connection by itself).



10.5.3 4.3 Invertible Bloomfilter Lookup Table

In blockchain network, nodes need to execute the block synchronization with other nodes through P2P network transmission, that is, to find out transactions existing in others but not in their own nodes and add these transactions to their own blocks. We assume that most transactions in the trading pool between the nodes should be the same. The most traditional method is that node A sends the excavated block to B directly. The bandwith requirement would be the size of one block. Since most transactions in the trading pool are the same, this approach takes up a large amount of

network bandwidth and appears to be extremely inefficient. Therefore, we use a method called Invertible Bloomfilter Lookup Table (hereinafter referred to as IBLT) to achieve block synchronization. IBLT is a method based on the evolution of bloomfilter. It is an improved scheme from the disadvantage of bloomfilter that requires two communications to complete block synchronization. Refer to Appendix A for details on bloom filters.

4.3.1 Principle of IBLT

IBLT is used to restore the values of key-value pairs. It follows the prototype of bloomfilter, including k hash functions and a set of storage units, called bucket here. Instead of being 1 bit in size, a bucket can contain larger elements. The struct of a bucket is as follows:

```
type Bucket struct {
        count    // a counting item to record the number of operation
        keySum   // the sum of all keys mapped to the current bucket.
        valueSum  // the sum of all values mapped to the current bucket
        keyhashSum  // the sum of the Hash values that are mapped to all keys in the␣
↪current bucket using a particular Hash function. The usage of this field will be␣
↪discussed later when we judge whether a bucket is a pure bucket.
}
```

The usage of count is similar to that of the Bloomfilter, except that whenever there is a hash operation mapped on the current bucket, the value of count is incremented by 1. Therefore, the value of count will be the number of hash mappings, and the value range is any integer greater than or equal to 0, not just {0,1}.

We allow new elements to be added to the IBLT constantly (that is, the values of the above four fields are updated into the corresponding k buckets through k Hash functions), and only added elements are allowed to be taken out of the IBLT (unadded elements cannot be subtracted). If we want to restore all of the key-value pairs from the invertible bloomfilter above, we need to find the buckets where count=1, because only the key-value of one element is stored in these buckets. After restoring the element, we can update it by subtracting the key and value of the element from the keySum and valueSum in the other buckets to which the k hashes are mapped, and the count values in these buckets are also subtract by 1. After such transformation, we can continue to look for the bucket with count value of 1 and peel off the key and value values of the element until count=0 or count value is at least 2 in all buckets.

If there are no buckets with count 1 but still buckets with count value at least 2, then the algorithm fails. However, this rarely happens when we choose the memory size and k value size reasonably.

Through mathematical derivation and prove, we find that when the KV pair inserted is less than a certain value, IBLT has a very high probability to recover successfully. However, when the KV pairs inserted are slightly more than this threshold, the probability of IBLT's successful recovery decreases rapidly. This threshold is related to the number of bucket in IBLT. It can be deduced that at K=4, the effect of IBLT is the best, in which case inserting M KV pairs means the number of buckets greater than 1.3M, so

N>= 1.3*M, K = 4

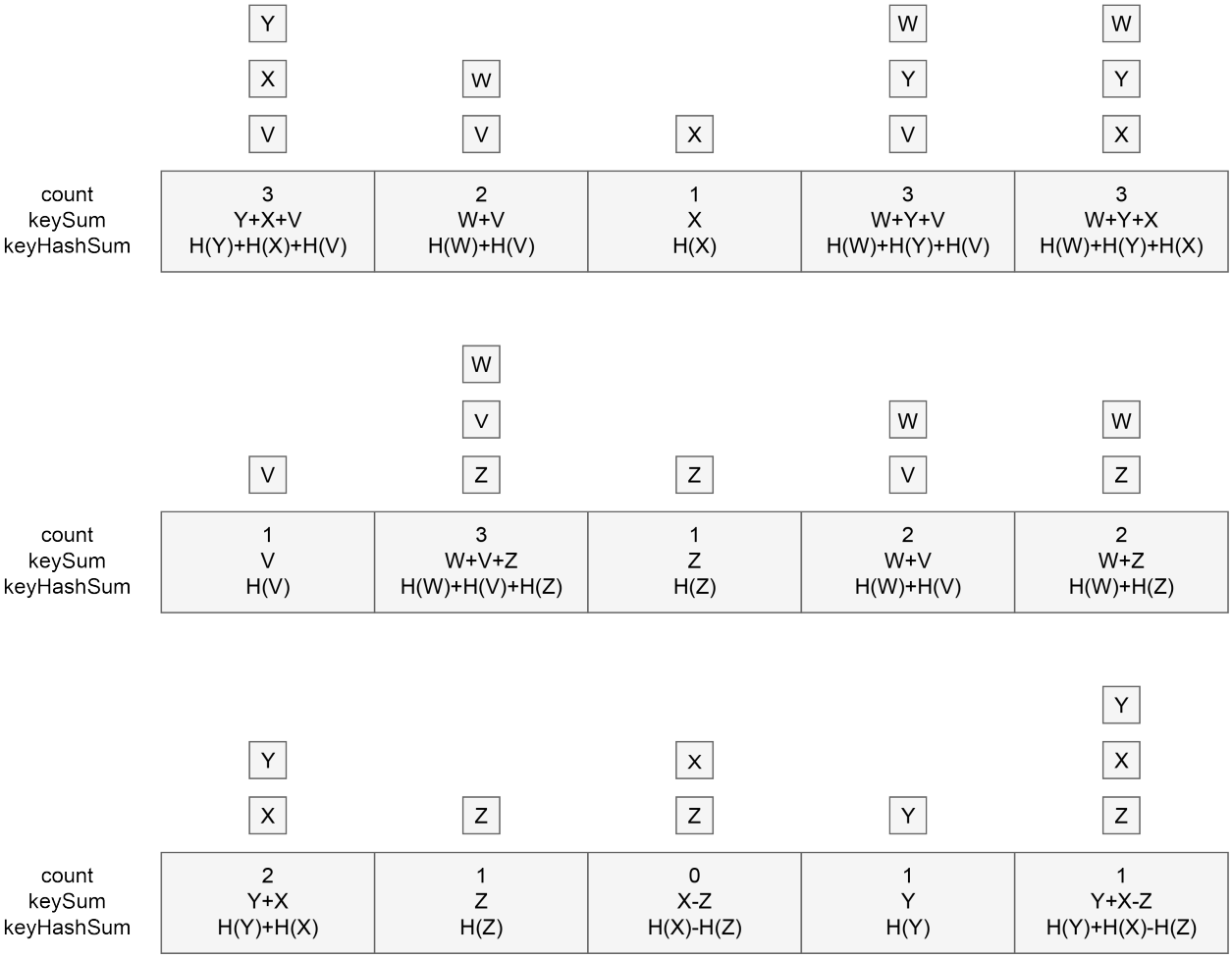
So we found that this algorithm is not meaningful for restoring the elements of an entire set. It only makes sense to find the difference of two sets (that is, the elements that another set does not have). Especially when the set itself is large, and each has only a few elements to be synchronized.

Since our purpose is to let the nodes find the transactions they don't own but other nodes have, that is to find the difference set with other nodes, without having to restore all the elements through this IBLT. So what we're more concerned about are

- How to accurately restore the filters after subtracting two IBLT.
- How to choose the size of this filter.

Subtraction of IBLT

To solve these two problems, we introduce subtraction between filters. This subtraction allows to subtract elements from IBLT that are not added to the IBLT. Upon reception of IBLT sent by other nodes, each node can substract it by the IBLT of its own to peel the element one by one fom the bucket that start from count=1.



But we find that after making the difference between two IBLTs, it might not be feasible to strip the element from a bucket with count value of 1. Because if A bucket does the calculation of A plus B minus C, count is still going to be 1, but its key is going to be key(A)+key(B)-key(C) who is not the key of a single element. So the question is how do we tell if this value is the key of a single element? In this case, the fourth element in the bucket we mentioned earlier, keyHashSum, comes in handy.

Buckets with count=1 and can be stripped off the key value of a single element (that is, keySum is the key value of a single element) are called pure buckets. We use keyHashSum to help figure out whether a bucket is pure.

KeyHashSum is the sum of all hash values mapped to the key of the bucket. Note H the hash function we choose. We know that for hash operation, H(A)+H(B)=H(A+B) does not exist, so when the count=1, the key value in the bucket might be in these two situations below:

- KeySum is the key value of some element x. In this case, keyHashSum=H(keySum). This equation can be verified immediately and inform us that the key is the key value of a single element and the bucket is a pure bucket.
- KeySum is actually the result of the sum of two key values (x,y) minus another element (z) 's key values. In this case KeyHashSum = H(x)+H(y)-H(z), which is not the same as H(keySum)=H(x+y-z). So we know that this bucket is not a pure bucket.

By starting from the pure bucket, we can peel off the transactions one by one, that is, we can restore the transactions not already owned and realise the synchronization.

The summary of IBLT

The cleverness of this approach is that we use a data structure in constant space where the growth of the data structure is linear when the element inserted is less than a threshold.But beyond this threshold, the size of the data structure does not increase, but only related to our parameter setting. We use about 1.3 times the space of the differential elements to eliminate a large number of common element transmissions. From our P2P network hypothesis, we know that most nodes have a large number of similar elements. When a node synchronizes, it wastes network bandwidth if it transfers a large number of these common elements.Therefore, we can find that the efficiency of this data structure is better when the number of common elements is larger.

4.3.2 Estimator

In the implementation process of IBLT algorithm, it is very important to estimate the size of this difference set. If the estimated size of the difference set is too large, then the purpose of saving space and improving transmission efficiency cannot be achieved; if the estimated size is too small, the IBLT may not be resolved.

In the real scenario, we have no information about the similarity between the two sets. If we fix the parameters of IBLT, then when the difference between the two sets is greater than the tolerance range of IBLT, the receiver cannot successfully solve the compressed data by subtraction between the two sets of IBLT. But if we are able to estimate difference size between two sets in advance, the size of IBLT can be adjusted in real-time according to the size of the differences, in which case successful solution is guaranteed when the collection difference is too large, while if the difference is too small, a smaller IBLT is used to reduce the amount of data that needs to be transferred for synchronization.

Let's first introduce two methods based on sampling estimation respectively, MinHash and Strata. MinHash works when there is a large difference between the two sets, and Strata works when the difference is small. We finally use the hybride method of the two to estimate the size.

MinHashMinHash is an estimation method based on random sampling. Random sampling statistics are very common on daily. Imagine we need to count the incidence of a genetic disease in a group of people, say, 1 million. We then have to pick uniformly at random 1,000 samples out of the group, and if 100 out of this 1,000 people are sick, then we can conclude with a lot of confidence that the incidence is 10 percent, and that 100,000 out of a million people are sick.

In order to ensure the accuracy of random sampling, the sampling execution process needs to meet certain preconditions:

1. Uniform and random extraction is required during extraction. For example, if we choose to carry out sampling in the department of genetic disease treatment, the prevalence rate estimated by sampling will greatly exceed the general population, leading to incorrect conclusions.
2. Reasonable selection of sample size. If the incidence of this genetic disease is very low, say 100 out of one million, then there's a high probability that we won't be able to get one sample from 1,000 people. That leads to conclude that the prevalence is zero.

MinHash first hashes all the elements in the set, and then selects the N elements with smallest hash values. As a good hash function is uniform enough, this process guarantees that our sampling is random and uniform. Compare the N hashes we extracted from one set with the N hashes that we extracted from another set, and if there are M common hashes between these 2 selections, then we estimate that the similarity between the two sets is M/N. This is the most basic and simplest idea. Obviously we find that this is affected by the second requirement of random sampling. That is, if our parameter chosen is not good, for example in the N selected elements the common elements are very few, then our estimation error will be very large.

To address this issue, we have a simple idea, which is to do multiple different samplings before calculating the average over the estimates of these samples. We pick K hash functionsand choose N elements with smallest hash values for each of these hash functions to compare with the smallest hash in another set. Assume there are M common hash values, then for each hash function the similarity is

Si=Mi/N, i=1, 2. . . , K

We can average the K outcomes to get the estimation of S

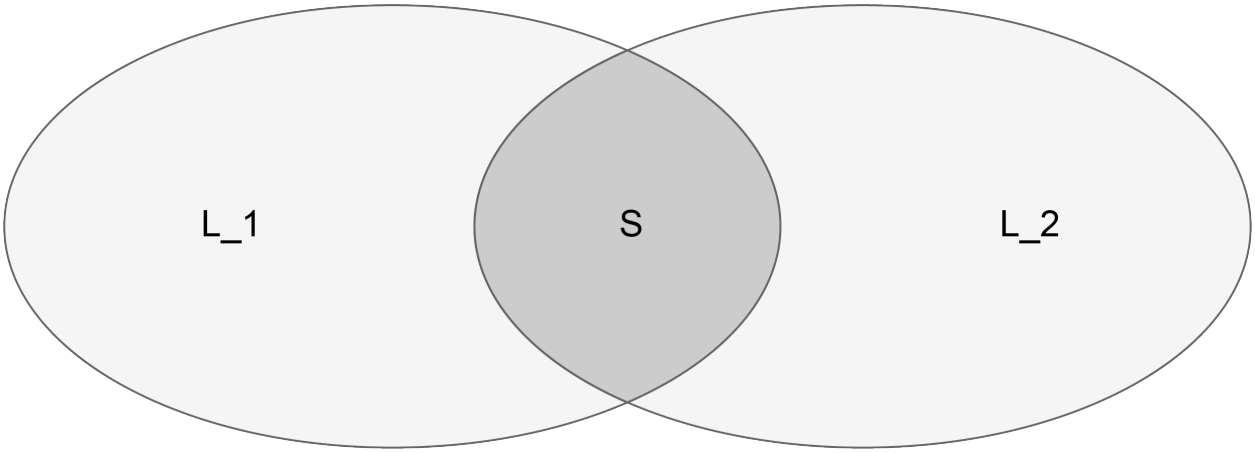
We find from this formula that this process is essentially equivalent to selecting the KN hashed elements and then finding the same number in the KN elements. The difference is that the strategy of taking K times N is allowed to have repeated selections compared to directly selecting KN elements with the smallest hash, and the K experiment should ideally be independent and random. The range of the similarity S is between 0 and 1, where 0 means that the set does not have any coincident parts while 1 means that the sets are identical.

After obtaining the value of S, the similarity is then

D = (1-S)/(1+S)*L

Where L=L1+L2 is the sum of the size of two sets

This formula is indeed quite obvious. L1+L2 includes the part of S, and 1+S id used as the denominator to standardize the number. By multiplying it with 1-S, the difference of the set is obtained.



StrataThe Strata(layer method) first layered the elements in the set, and which layer the element is assigned to is determined by its hash. The way to do that is to observe the number of zeros lie on the left side of the binary hash. Because of the randomness of the hash function, there is no difference if we count the number of zeros on the left side or the number of zeros on the right side. Here we choose the number of zeros on the left side to decide the level of each element. If the binary of an element's hash value is not 0 at the beginning of the left side, which means the binary value starts with 1, then we put it on the lowest layer (the 0th). If there is only one 0 from the left side, we then put it on the second lower level (level 1), and so on. If the number of zeros on the left of the binary value exceeds our number of levels, we then put it on the highest level.

It's not hard to see that if we're using a good hash function, then ideally we should be able to get approximately one half of total elements of the set on layer 0 and a quarter of the elements on layer 1; By analogy, we should be able to get on the ith layer 1/2i elements, that is, the distribution of the number of elements is a geometric sequence distribution with 1/2 as the proportion parameter.

Each layer of the hierarchical approach consists of an IBLT, which is fixed in length and relatively small, with a typical value of 80. when constructing, each element of the set is inserted into the corresponding layer. Then at the time of estimation, the IBLT of the two corresponding layers is subtracted from the highest level for getting resolved until the solution is not solved or all of them can be successfully solved.

Obviously, if we can solve all of them, then the sum of the number of data on each layer would be the difference between the two sets, and the difference would be accurate. While if we see that IBLT starts to fail at some level, then we can stop there, because it's bound to fail when we continue. Then we use the sum number of the elements that solved from all high levels and multiplied by 2{I+1}, i is the height of the layer at this time, namely:

D{estimated} = count * 2{I+1}

It's not hard to see, as the number of elements is evenly distributed, the sum of geometric series with 1/2 as the proportion has this following rule:

Si = Si+1 + Si+2+ . . . . . .

Namely, 1/2=1/4+1/8+. . . . . , and 1/4=1/8+1/16+. . . . . .

Take 1/4 as an example, our count includes everything from 1/8+1/16+. . . We can get a good estimate by multiplying it with a 4.

The disadvantages of the layered approach are also very obvious. If the two sets of elements are quite different, then the top layer of strata would be full of elements. When all the layers are piled up, the solution would fail from the beginning, and our estimation becomes impossible. And when we fail in comparing the high level, our count is small, which means the multiplied coefficient 2(i+1) is larger. In this case the influence of a tiny change of count is so big that the error of our estimation will be very large. The error of the Strata method is very large when the difference is large, which is also a conclusion given by the author in the original paper.

Hybrid

MinHash doesn't work when the difference is small, while strata can detect the difference without error in this case. MinHash can guarantee the extraction of sufficient elements when the difference is large, while strata will assume a large error in this case so that no layer can be solved. Then what if we combine these two methods by raising strengths and overcoming weaknesses? This is exactly what does the method of hybrid estimator.

Here is the detail of this hybrid method: We do the same stratification, but only for the relatively low layers. The high layers are set beside for MinHash. We divide 16 layers into 2 parts, where the highest 6 are reserved for Minhash while the lowerest 10 are left for Strata. Then we can take full advantage of the advantages of the two methods. However it's still difficult to fully understand the effects of inserting different numbers of elements.

When we have a fewer number of elements, there is basically no element that falls into the MinHash layer, which is equivalent to strata in this case. Its accuracy is the accuracy of strata, and the difference will not be greater than the number of elements. so we completely use strata for detection where the result is relatively confident and not affected by MinHash when the number of elements is small. Our estimation uses the sum of the results of the two detectors.

When we have more elements, most of them fall into the strata layer and a few fall into the MinHash layer. At this point, there are two situations:first, if our differences are small. Because MinHash has fewer elements, Therefore, MinHash is easily mistaken for basically no difference. At this point, because the difference is small, we can use IBLT in Strata to get our estimation. In this way, reliable results can be obtained. We also take the sum of the two results.Second: if our differences are large. At this point, although MinHash has fewer elements, it can still estimate a certain error due to its large difference while the strata also have a large error. In this case, it is an embarrassing critical point. Figure 9 of the original paper shows the figure between 100 and 1000, with strata playing a major role. We're also going to take the sum of the two results.

When we have a lot of elements with huge difference, most of them fall into the strata layer and a few fall into the MinHash layer. But at this point there are enough elements in MinHash so that we have enough confidence upon the result of its estimation. At this point, as Strata has started to fail because of the huge difference, we can just take the MinHash result.

10.6 5 Dipperin Block and Data Storage

The data structure of blocks plays an important role in public chain research. A reasonable data structure facilitates rapid transaction search and verification. The data storage of Dipperin adopts multi-layer storage mode, and the bottom layer can permanently store data by key-value pairs in levelDB.

Between the underlying database module and the business model, Dipperin sets up a local storage module, which is oriented to the business model. It can flexibly design various storage formats and units according to business needs, and connect to the underlying database at the same time. If the underlying database changes, it can greatly reduce the impact on business modules. This role is StateDB, which manages all account information through a large collection of AccountStateDB objects.

10.6.1 5.1 StateDB - Service Oriented Storage Module

StateDB has a member called storage trie or state trie, which is stored in the form of Merkle Patrica Trie (the later chapter has a detailed explanation of the tree structure). The MPT structure stores the AccountStateDB object. The object has a variable member blockStateTrie, which contains the account balance, the number of contract initiations, the hash value of the latest contract instruction set, and the vertice hash value of an MPT structure.

When the miner packages the block, all the transactions in the block will be executed in order. The account status involved in these transactions will be modified and the storage trie will be updated, and the root node hash recalculated will be recorded in the block. Here is the question: if the final block is not accepted by the network, how should the changed storage trie be rolled back?

We introduce here a snapshot mechanism, which is a fast rollback mechanism, indicating that once the miner's block is not received, the tree can immediately roll back to the state of the snapshot. When the miner transfers the block to the network, he will send a state change list as well, which records the changes caused by the block's transaction on the storage trie. In this case, when the block is received by the network, other nodes only need to use the state change list to update the state tree quickly upon reception of the block, instead of re-executing all the transactions in the block. Similarly, if the block is not received by the network, then through the list of records, the storage trie can be quickly restored back to the state it was in before the series of transactions was executed.

Although the storage trie's leaf nodes are accounts, Dipperin divides the fields of the account when it is implemented, so that when a certain state (such as balance) of the account is changed, it is not necessary to change the entire structure.

The structure used by the Storage trie is the Merkle Patricia Trie. There are three types of data in Dipperin that use the structure of the Merkle Patricia Trie tree, which is transaction tree, verification tree, and storage tree. The hash tree roots of these trees are stored in the block header and used as a check. We will introduce more details about the structure of the MPT tree in the following.

Merkle Patricia Trie is a data structure that combines the characteristics of Patricia tree and Merkle tree, hereinafter referred to as the MPT tree (refer to Appendix B for an introduction to Patricia tree and Merkle tree).

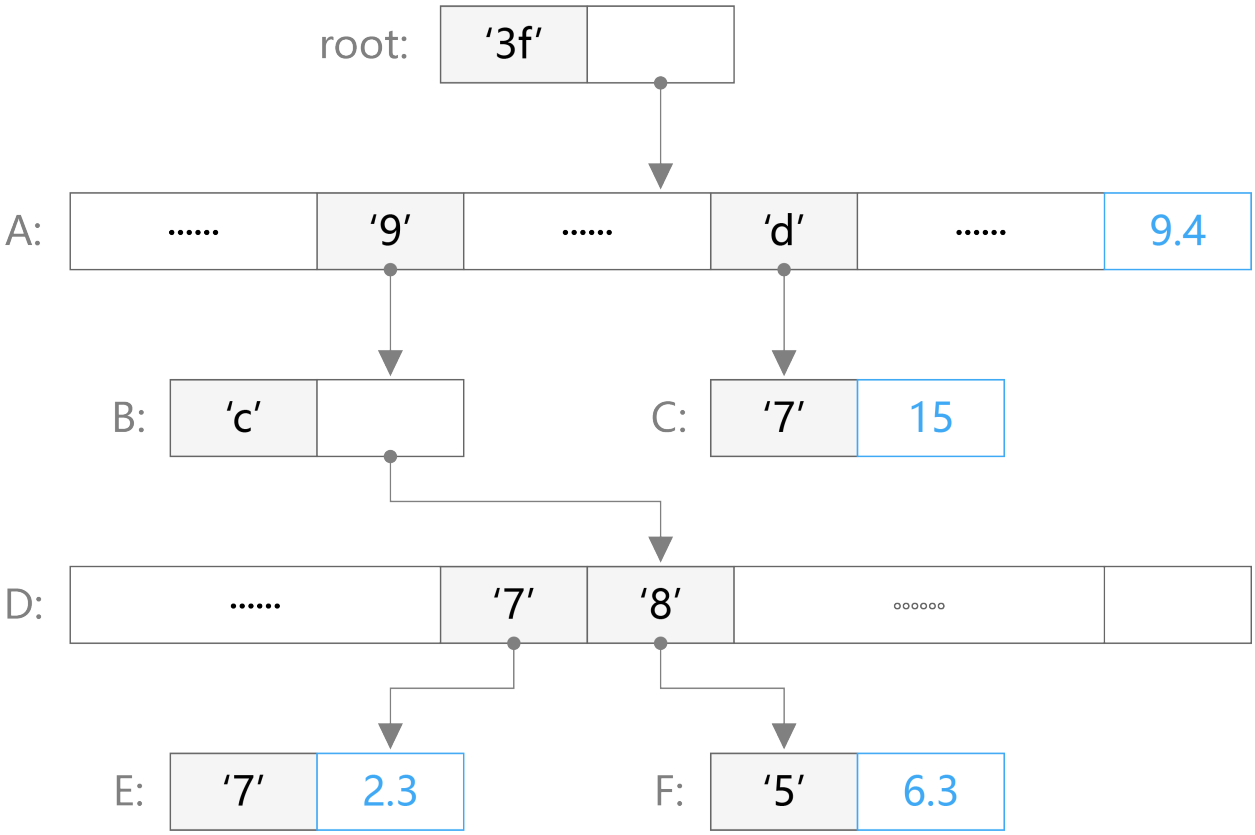
MPT has deterministic. Deterministic means the same content key value will direct to the same result and the same root hash. In efficiency, the time complexity of inserting, finding, and deleting trees is controlled at O(log(n)). MPT is easier for people to understand and code than red-black trees.

The main point MPT inherits from the Patricia trie is branch nodes that do not need too much redundancy. If a branch node has only one child node, it will merge with the child nodes into a node called an extension node. Therefore, the node types of MPT include a leaf node, an extension node, and a branch node.

Since the key in the MPT tree becomes a special hexadecimal representation by RLP coding, the branch node is a list of length 17 in which the first 16 bits store the hash values of other nodes, corresponding to different branch nodes or Leaf node. If some of the 16 bits do not correspond to any of the above nodes, the corresponding one may be an empty node. The last bit of the extension branch is the value, which can be empty (if the current node does not have a corresponding value). The extension node and the leaf node are each a key-value pair, except that the value of extension node stores the hash by RLP code of the branch note (which cannot be a leaf node because it will be merged with the leaf node in this case) , while the value of the leaf node stores the "value".

The key-value pair mentioned above is different from the key-value pair permanently stored in LevelDB by the MPT we will talk about later, where the value is the RLP encoding of the entire node, and the key is the sha3 of the encoding. hope. We will talk about it later.

The following figure is an example of an MPT tree:



The figure shows an MPT state tree that stores four key-value pairs. These four key-value pairs are [‘3f’, 9.4], [‘3fd7’, 15], [‘3f9c85’, 6.3], [‘3f9c77’, 2.3]. In the figure, nodes A and D are branch nodes, root nodes and node B are extension nodes, and nodes C, E, and F are leaf nodes. Through this figure we can see how these types of nodes are organized in the tree. When we need to find a value corresponding to a key, such as ’3f9c85’, then we only need to find ’3f’->’9’->’c’->’8’->’5’ from the root node. The last leaf node can find the corresponding value of 6.3. For the case where the key is ’3f’, since the branch appears in ’3f’, its value will appear in the value of the next branch node, which is 9.4.

10.6.2 5.2 LevelDB - Persistent Storage

LevelDB uses a key-value storage method, on which two types of data are stored permanently. The first type is ChainDB, which is the data on the chain. The second type is the StateDB mentioned above, which is status data for the account. For the data on the chain, there are multiple prefixes in the design of the key for better query. All the queries that can be supported in this module are listed in the following:

- Query Block Hash by Block Number
- Query Block Number by Block Hash
- Query the Header Hash of the last block in the database
- Query the hash of the last block in the database
- Query Block Header RLP
- Query Block Body RLP
- Query TxLookupEntry
- Query Block

- Query Tx

All important members of the Block structure are stored in the underlying database. When all the information of the Block object has been written into the database, we can use the BlockChain structure to process the entire blockchain.

When we talked about the Header structure in the Dipperin architecture, there are three elements inside: TransactionRoot, StateRoot, and VerificationRoot, which are come from the hash value of root node in three MPT type objects: txTrie, stateTrie, and VerificationTrie. These are the root hashes saved from statedb. For encryption, using a 32-byte hash to represent a tree structure with several nodes (or an array of several elements). For example, in the synchronization process of Block, it can be confirmed whether the array member transactions are synchronized by comparing the received TxHash.

We will introduce the permanent storage of the MPT tree in the LevelDB database in following paragraphs.

Each node in the MPT tree is referenced by its hash. When the node is permanently stored in the LevelDB as a key-value, the key is the hash value of the node. For coding considerations, this hash is actually a sha3 hash of the RLP encoding of the node, then the value is the RLP encoding of the node. The root node becomes the cryptographic signature of the entire tree. When the root hash of a given trie is public, everyone can provide a proof of whether a particular key contains a given value by providing a path up steps.

The code uses different implementation types of the node interface to record and store various types of nodes in the MPT tree.

The node implementation type in the Trie structure contains the following four types:

- fullNode
- shortNode
- hashNode
- valueNode

The fullNode is used to record the branch node, the shortNode is used to record the extension node or the leaf node, the hashNode and the valueNode are both byte32 types, the hashNode can only be used to record the hash of the node, and the valueNode can only be used to store the value. At the code level, both fullNode and shortNode hold a structure called nodeFlag, which has a hashNode, and the value of hashNode is the hash of the RLP code of the current node. The fullNode structure contains an array of length 17 of node type. The first 16 nodes are used to store other nodes' hashes or empty, the last node is used to store a specific value or empty. If the shortNode is an extension node , its value is hashNode, which represents the hash of another branch node. If the shortNode is a leaf node, then its value is a valueNode, carrying a specific value.

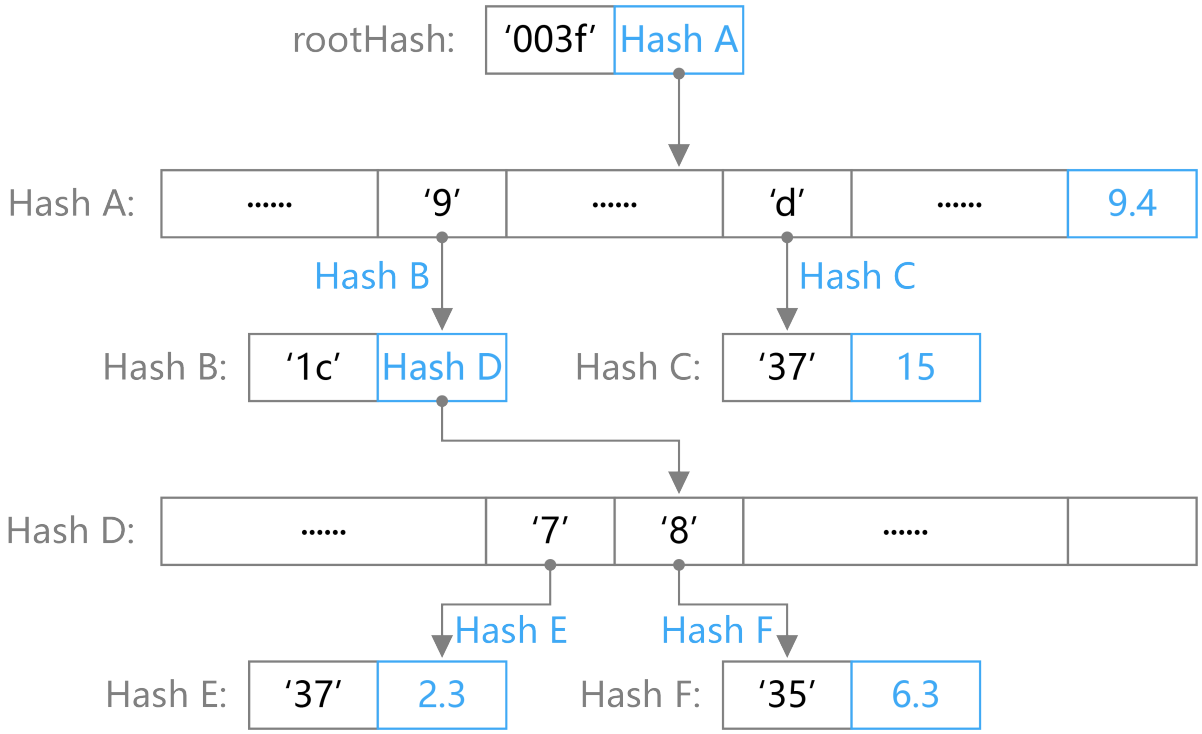
When inserting [key, value] data into this MPT tree, it will extend down from the root along the path of the key string. At the end of this insertion, it will become a valueNode first. And the key will exist in the form of key path that starts from the vertex root to the end of the node. The nodes passing by on the way may change due to this data insertion, even lead to node type changes. Similarly, each change or deletion of [key, value] data will also cause some nodes to change. For example, the above data increase may cause the parent node of the newly added leaf node to be changed from the previous leaf node to a branch node, and a deletion may cause a branch node to be degenerated into an extension node and merge with the child node.

Since shortNode can be used to represent both extended nodes and leaf nodes, we assign different hexadecimal prefixes to the two types of nodes when encoding. In addition, since we only need 4 bits to represent a hexadecimal character, 1 byte can represent two hexadecimal characters. In order to distinguish between 'f' ('00001111') and '0f' ('00001111'), we also use different encoding prefixes for hexadecimal characters of different parity lengths. The specific display is as follows:

- The prefix '00' is in front of an even-length key to indicate a hash. For example, '00f937' indicates that the key of the shortNode is 'f937', and the node is an extension node.
- The prefix '1' is in front of an odd-length key to indicate a hash. For example, '15d7' indicates that the key of the shortNode is '5d7', and the node is an extension node.

- The prefix '20' is in front of an even-length key to indicate a value. For example, '20c7ab' indicates that the key of the shortNode is 'c7ab', and the node is a leaf node.
- The prefix '3' is in front of an odd-length key to indicate a value. For example, '3a88' indicates that the key of the shortNode is 'a88', and the node is a leaf node.

Using the example in 5.1, the figure's data is permanently stored in LevelDB with followlling steps:



The key-value pairs stored in LevelDB are:

- [rootHash: ['003f',HashA]]
- [HashA: [nil,nil,nil,nil,nil,nil,nil,nil,nil,HashB,nil,nil,nil,HashC,nil,nil,9.4]]
- [hashB: ['1c',HashD]]
- [hashC: ['3d',15]]
- [HashD: [nil,nil,nil,nil,nil,nil,nil,HashE,HashF,nil,nil,nil,nil,nil,nil,nil,nil,nil]]
- [HashE: ['37',2.3]]
- [HashF: ['35',6.3]]

10.7 6 Hierachical Deterministic Wallets

The wallet and account management of Dipperin is implemented in the way of hierarchical deterministic wallet. It encrypts and stores sensitive data such as seed private key so that the security of user private key is ensured. In order to meet multiple user demands, Dipperin realizes various functions of wallet both in the front end and the back end.

10.7.1 6.1 Hierachical Deterministic Wallets

HD wallet is created from a single root seed which is a random number of 128-256 bits. All the certainties of HD wallet derive from this root seed. Any root seed compatible with HD wallet can also recreate the entire HD wallet. So transferring the root seeds of HD wallet simply allows the millions of keys contained in HD wallet to be copied, stored, exported, and imported.

Conversion function and corresponding parameters:

- pointpreturns the coordinate pairs generated by secp256k1 basis point multiplication (repeated application of EC group operations) represented by integer p.
- ser32iserializes 32-bit unsigned integer i into a 4-byte sequence for big endian (computer terminology).
- ser256pserializes the integer p into a 32-byte sequence, big endian(computer term).
- serPPserializes coordinate pairs P= (x, y) by SEC1 compression format to byte sequences (0x02 or 0x03) ‖ ser256 (x), where the header byte depends on the parity of the omitted y coordinate.
- parse256pconverts a 32-byte sequence to 256 bits for big endian (computer terminology)
- kprivete key
- K: public key
- c: chaincode

The system uses cryptographically secure pseudo-random functions to generate root seeds, which can be converted into mnemonic words for users to remember. The root seed is input into the HMAC-SHA512 algorithm to obtain a hash that can be used to create the primary private key and the main chain encoding. Then the master public key could be obtained through the master private key.

Subkey derivative equation

The hierarchical deterministic wallet uses the CKD (child key derivation) equation to derive a subkey from the parent key.

The subkey derivative equation is based on a single hash equation. This equation combines:

- Parent private key or public key (ECDSA uncompressed key)
- Chain code (256 bits)
- Index number (32 bits)

Chain code is used to introduce seemingly random data into this process, so that indexes cannot derive other subkeys adequately. Therefore, having a subkey does not mean the discovery of its own similar subkey unless you already have the chaincode. The original chaincode seeds (at the root of the cryptographic tree) are composed of random data, and the chaincode is then derived from the parent chaincode of each destination.

Extended secret key

- Extended private key; composed of a private key and a chaincode, used to derive a child private key.
- Extended public key; composed of a public key and a chain code, used to derive the child public key.

The extended secret key serves as the root of a branch of the key tree structure in HD wallet. You can spawn off the rest of this branch. Extended private key allows to create a complete branch while extended public key permits only creating a branch of the public key. Each extened key has 231 normal child subkeys and 231 hardened subkey. These subkeys have an index. Normal subkeys using the index from 0 to 231-1, while hardened index of subkey use 231 to 232-1.

Derivative normal child private key

The parent public key – chaincode – and the index number are combined and can be hashed by hmac-sha512 equation into a 512-bit hash value. The resulting hash can be split into two parts. The right half of the 256-bit hash output can be used as the chaincode for the subchain. The left half of the 256-bit hash and the index code are loaded on the parent private key to derive the child private key. The calculation process is as follows:

> Function CKDpriv ((kpar, cpar), I) to (ki, ci) calculates the child extended private key from the parent extended private key:
>
> - Check whether i 231 (child private key).
>   - If it does (hardened subkey) : let I = hmac-sha512 (Key = cpar, Data = 0x00 ||ser256 (kpar) || ser32 (I)). (note: 0x00 extends the private key to 33 bytes long.)
>   - If not (normal subkey): let I= hmac-sha512 (Key = cpar, Data =serP (point (kpar)) || ser32 (I)).
> - I is divided into two 32-byte sequences, IL and IR.
> - The returned subkey ki is parse256(IL)+kpar (mod n).
> - The returned chaincode ci is IR.
> - If parse256 (IL) n or ki = 0, then the generated secret key is invalid, and we should proceed to calculate the next i values. (note: the probability is less than 1/2127)

Changing the index allows us to extend the parent secret key and create other subkeys in the sequence, such as Child 0, child 1, child 2, and so on. Every parent key can have 231 normal subkey.

Repeat this process down the cryptographic tree, and each subkey can become the parent secret key in turn and continue to create its own child key until infinity.

Child public key derivation

A useful feature of hierarchical deterministic wallets is the ability to derive the child public key directly from the parent public key without using the private key. This gives us two ways to derive the child public key: either through the child private key, or through the parent public key directly. Therefore, the extended public key can be used to derive all (and only) the public keys in a branch of the HD wallet structure.

The parent public key – the chaincode – and the index number are combined and can be hashed by hmac-sha512 equation into a 512-bit hash value. The resulting hash can be split into two parts. The right half of the 256-bit hash output can be used as the chaincode for the subchain. The left half of the 256-bit hash and the index code are loaded on the parent private key to derive the child private key. The calculation process is as follows:

> Function CKDpub ((Kpar, cpar), I) - (Ki, ci) calculate the extended child public key from the parent extended public key. It is defined only for unhardened child secret keys.
>
> - Check whether i231 (whether the subkey is a hardened secret key)
>   - If it does(hardened subkey) : return failed
>   - If not(normal subkey): let I= hmac-sha512 (Key = cpar, Data = serP(Kpar) || ser32(i)).
> - I is divided into two 32-byte sequences, IL and IR.
> - The returned subkey Ki is pointparse256IL+ Kpar
> - The returned chaincode ci is IR.
> - If parse256(IL) n or Ki is an infinity point, then the generated key is invalid, and we should proceed to calculate the next i value.

Derived hardened sub private key

The ability to derive a branch public key from an extended public key is important, but involves some risk. Accessing the extended public key does not provide the way to access the child private key. However, because the extended public key contains the chain code, the chain code can be used to derive all other child private keys if the child private

key is known or leaked. A oblivious leaked private key and a parent chaincode can expose all the child keys. What is worse, the child private key and the parent chaincode can be used to infer the parent private key.

In response to this risk, HD wallet uses an alternative derivation equation called hardened derivation. This "breaks" the relationship between the parent public key and the child chaincode. This hardened derivative equation uses the parent private key to derive the subchain code, rather than the parent public key. This creates a "firewall" in the parent/child order – there is a chaincode but it cannot be used to calculate the child chaincode or sister private keys. The enhanced derivative equation looks almost identical to the general derived child private key except that the parent private key is used as the input of the hash equation instead of the parent public key.

When the enhanced private key derivative equation is used, the child private key and the chaincode obtained are completely different from those obtained by the general derivative equation. The resulting secret key "branch" can be used to produce an extended public key that is not vulnerable because it contains a chaincode that cannot be used to develop or expose any private key. Reinforcement derivatives are therefore used to create "gaps" in the key tree that extend the public key at the upper level.

In short, if you want to take advantage of the convenience of the extension public key to derive branches of the public key without exposing yourself to the risk of leaking the extension chaincode, you should derive the public key from strengthening the parent private key, rather than from the general parent private key. The best way to do this is to avoid pushing out the master key, and the first level of child keys derived from the master key are best used with enhanced spawn.

Index Numbers for normal and enhanced derivatives

The index number used in a derived equation is a 32-bit integer. To distinguish whether a secret key is derived from a normal derivative equation or an enhanced derivative equation, the index number is divided into two ranges. Index number from 0 to 231-1 (0x0 to 0x7fffffff) is only used in conventional derivative. Index number from 231 to 232-1 (0x80000000 to 0xFFFFFFFF) is only used in the enhanced derivative equation. As a result, the index number less than 231 means the subkey is regular, and the case greater than or equal to 231 means the subkey is enhanced.

In order to make the index number easier to read and display, the index number of enhanced child password is displayed from 0 with a small apostrophe on the top right corner. The first regular subkey is therefore represented as 0, but the first enhanced subkey (index 0x80000000) is represented as 0'. The second enhanced key has the index number 0x80000001 in order and is displayed as 1', and so on. Moreover, the HD purse index number i' means 231+i.

The path

The materialized path of tree-structured data is as follows

m / 0 / 1

where m is the primary private secret key, 0 is the secret key with the sequence number 0 in the first-layer derived subkey, and 1 is the sequence number 1. So m/0/1 represents the derivative child secret key with the number 1 under the derivative child secret key with the number 0 under the main secret key.

10.7.2 6.2 The Storage of Wallet File

AES algorithm is used to encrypt and store sensitive data such as the extended private key generated by the system. A key is generated from the scrypt derivative of user's password. In order to use the purse, the users need to enter the password to unlock the wallet file. Users can restore the wallet by mnemonic word or by wallet encrypted file combined with password.

10.8 7 Token Economy Model Design

The existing design of the general economic model is easy to fall into two misunderstandings: first, increasing the circulation means inflation and reducing the circulation means deflation; second, deflation can improve the income of

investors, so deflation is a better design. First of all, We need to be clear that deflation and inflation are not determined only by currency issuance. The relationship of supply and demand should also be taken into account. That is to say, if the increased liquidity is less than the increase in demand, there will still be deflation; if the reduced liquidity is less than the decrease in demand, inflation will still occur. Secondly, between inflation and deflation, either one is not absolutely better or worse than the other. How to choose between the two depends on the positioning of the project. Both deflation and inflation have their rationality. The deflation model is conducive to improving the storage value of the certificate and the inflation the circulation value of the certificate. From the perspective of the token economic design, Bitcoin is a deflation model while Ethereum is a model of from inflation to deflation, and Dipperin should be a model of micro-deflation to micro-inflation. Dipperin's certification standpoint is different from Bitcoin and Ethereum. In the long run, Bitcoin and Ethereum are deflation models that emphasize storage value, while Dipperin is an inflation model that emphasizes circulation value. Dipperin emphasizes the value of circulation because the design purpose of token is to use the certificate as a public chain engine to ensure that the public chain functions maximally. The token economy model of Dipperin includes two parts: the macro mechanism and the reward and punishment mechanism.

10.8.1 7.1 Macro-Mechanism

The design philosophy of Dipperin is based on the relationship of demand and supply. We assume that the public chain will develop in a positive way under which premise we estimate the future demand change and design the supply based on this estimation. Demand growth will increase rapidly in the early stage, and then the growth rate will slow down. As the product matures, the demand increase will stabilize (Figure 1). According to the change in demand, we set Dipperin's public chain issuance mode to two stages. The first stage (1-10 years) is the micro-contraction period. The supply and demand in the early stage are not balanced and the foundation reward and pre-mining stage unlock mechanism are set to meet the large demand fluctuations. Moreover the reward amount will be adjusted according to the market demand, so that the supply is slightly less than the demand. The second stage (10 years later) is the micro-inflation period. We assume that the 10-year project is mature and supply and demand tend to be balanced. Since then, the annual output of currency is 3% of the cumulative output in previous years, which means 3% inflation.

7.1.1 Public Chain Coin Offering

Mode de Coin Offering: First Stage: The productivity of the chain is constant, namely 2 CSK each block, so that approximately 7,884,000 CSK are produced as there are about 3,942,000 new blocks each year. The output of pre-mining are 52,560,000 CSK, and then the total productivity within the future 10 years will be 131,400,000CSK.

Second Stage: The annual productivity of the public chain is 3% of the pre-accumulated output, namelyP_{t+1} = 3% × \Sigma_{t=1}^n P_t, t > 10

Pt is the nth annual output. For example: the 11th and 12th annual productivity is respectivelyP_11 = 3% × 131400000 = 3942000 P_12 = 3% * (131400000+3942000) = 4060260

Settings of Pre-mining

Type	Coin#	Owned Person
Premining 1 (subject)	43,800,000	Foundation, investors, development team
Premining 2 (reward)	8,760,000	Miner, verifier, user
Premining in total	52,560,000	

Design Philosophy Pre-mining 1: For minority Pre-mining 2: For public Public Chain Mining: For public Pre-mining 1 = (accumulated output during first 10 years of public chain mining + accumulated output during first 10 years by pre-mining 2) * 50%=(7,884,000*10+8,760,000)*50%=43,800,000

At the end of the 5th year: pre-mining 1/accumulated circulated output (namely accumulated output during the first 5 years of public chain mining + accumulated output during first 5 years by pre-mining 2 + pre-mining 1) 50%.

Premining 1 as a percentage of total market circulation	
Year	Circulation ratio
1	82%
2	70%
3	61%
4	54%
5	**49%**
6	45%
7	41%
8	38%
9	36%
10	33%

During the first 5 years the proportion of pre-mining 1 is greater than 50%, mainly because it demands a relatively long period for the maturing of this project where in the first stage the support of the foundation is essential. However, considering the decentralized nature of the Dipperin public chain, and it is expected that the project will be stable within five years, there should be no more holders of more than 50% share. Relying on the Dipperin public chain alone to adjust supply and demand is also facing the challenge of lack of early bird welfare, non-compliance with product development and change of supply and demand. In order to solve these problems, early birds are awarded extra foundation token reward and market circulation are influenced by unlock periodically of the pre-mined coins on the basis of the public chain.

7.1.2 ECSK Reward of the Foundation

There are 2 kinds of rewards of the foundation: one is for miners and verifiers and the other is for normal users.

The first reward for the miners and verifiers is produced by issuing ECSK token on smart contract. Every block produced by miners and verifiers means proportional reward and is anchored to CSK. The reward curve during the first 5 years are:

R_t = 5 - 0.4 * \Sigma_{t=1}^n (t-1), t \in {1,2,3,4,5}

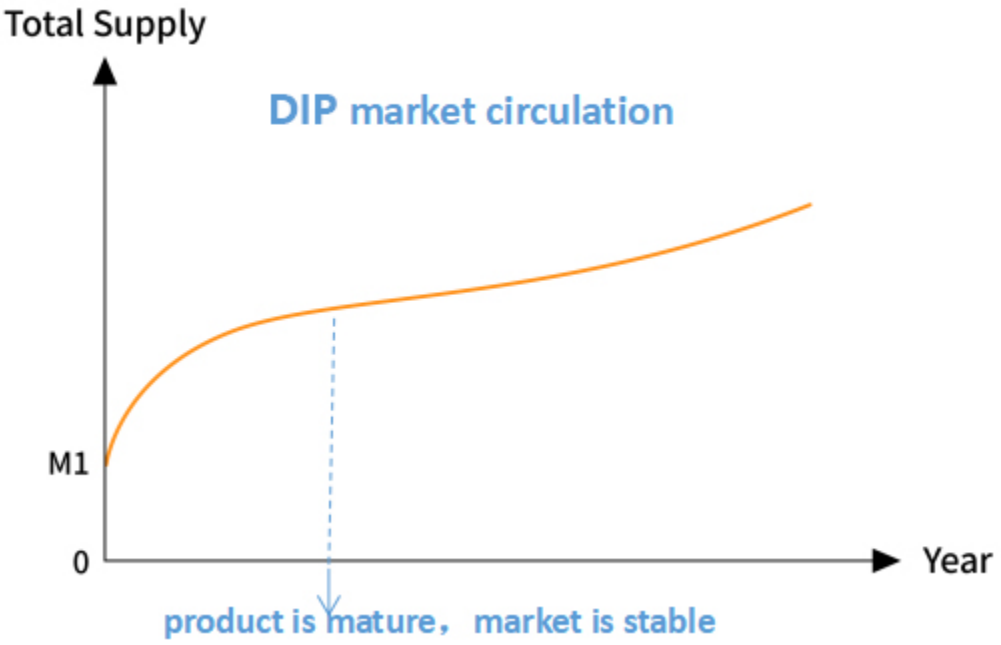
At the beginning every CSK excavated means 5 ECSK to the corresponding miners and verifiers. The initial exchange rate between ECSK and CSK is 1ECSK=0.0327CSK. Later the foundation adjusts the rate every 3 months according to the market situation.

The second reward for the normal user is awarded by the foundation according to the market and operation demand in different periods

7.1.3 Pre-mining Unlock Periodically:

The CSK obtained by the investors and the development team in the one-time creation of the Genesis block is locked at the beginning, then unlocked in the later stage, in order to avoid the influx of a large number of CSKs into the market at the beginning, causing the currency to dive, and also prevent speculation and cashing out. Different unlock period setting according to different participants also avoid centralized unlocking.

Through the adjustment of the three means, the final trend of Dipperin's currency is:



10.8.2 7.2 Reward and Punishment Mechanism

7.2.1 Reward

Both miners and verifiers are subject to rewards. All CSK generated by each block are shared by them according to the following rule: miners 87% and verifiers 13%. There are 22 verifiers during each session, which means the 13% are shared by 22 participants. These participants are divided into 3 categories: primary verifier, active verifiers(The first 15 verifiers who appears in the commit list) and passive verifiers(the last 7 not appearing in the commit list), each category having different level of rewards. There is a concrete table as follows:

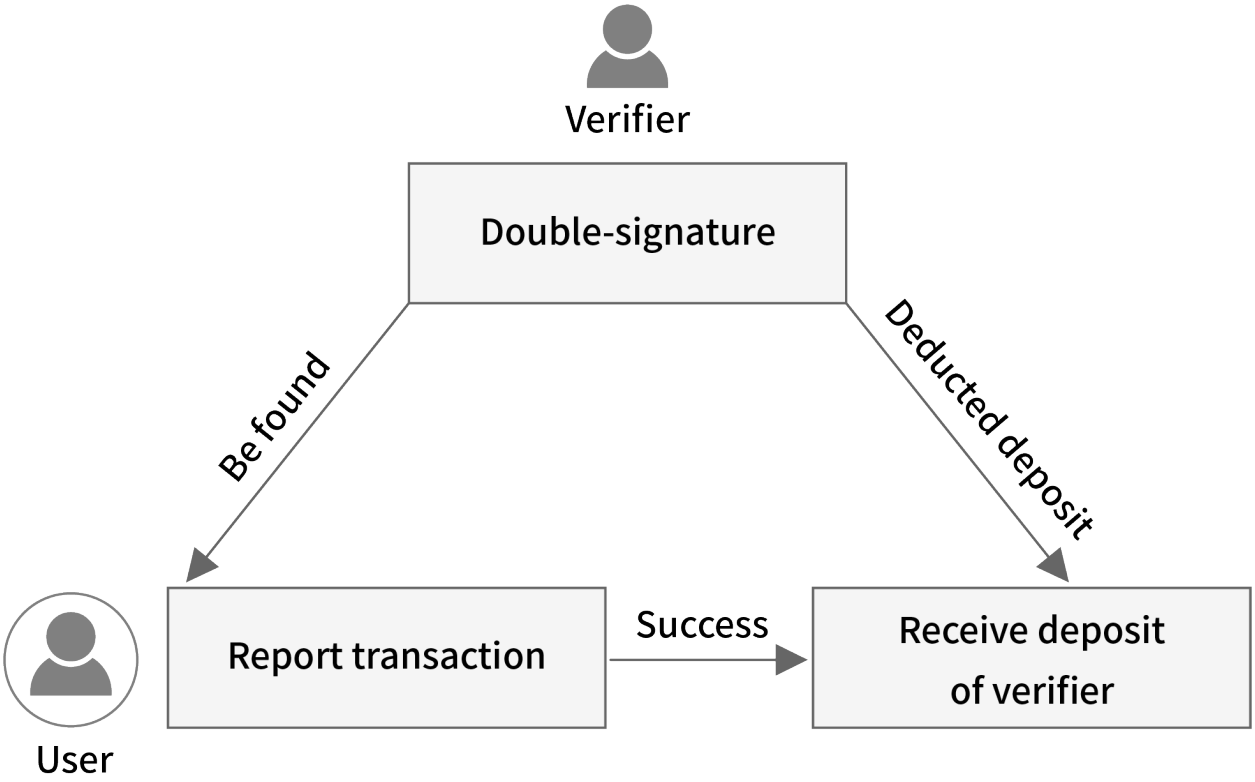
Table 3 Allocation rule of rewards among verifiers

|Three categories of verifiers | total number of participants in the category | weight | proportion | Formula | |—— | ———- | —— | —- | — | |Primary verifier's extra reward | 1 | 0.5 | 0.38% | 13%×[0.5/(10.5+151+70.25)]| |*Active Verifier(Each)* | *15* | *1* | *0.75%* | *13%×[1/(1*0.5+151+70.25)] | |Passive Verifier(Each) | 7 | 0.25 | 0.19% | 13%×[0.25/(10.5+151+7*0.25)] |

There are 110 blocks to verify for each session of verifiers, which takes approximately 14.67 minutes. During the first 10 years, each block can generate 1.74 CSK for the miner while each verifier can earn at least 0.42 CSK but at most 1.72 CSK during a session.

7.2.2 Punishment

Before talking about specific punitive measures, we need to make clear that the verifier's rewards and punishments exist not only in the measurable rewards and punishments set by us. The reputation value can be reduced by malicious behaviors and this will negatively influence the chance of being selected in the subsequent rounds. Mainly two kinds of malicious behavious are subject to punishment: slack as a verifier or double vote. Slack causes the decrease of reputation while double vote will assume severe punishment such as the confiscation of all the deposits upon confirmation of report, and the deposits will enter the account of the whistleblower. Concrete process is as follows:



10.9 8 Conclusion

In this yellowpaper, we have explained the implementation principle of Dipperin, revealing that it can achieve a number of TPS far above Bitcoin while maintaining its decentralization level, and achieve no fork. Fast routing lookup, reasonable data structure, fair economic incentives, and cutting-edge cryptographic algorithms enable security and privacy protection, make Dipperin the next generation financial blockchain model in terms of performance, security, robustness and privacy protection.

10.10 9 References

[1] Ethereum: a secure decentralised generalised transaction ledger byzantine version, Dr. Gavin Wood

[2] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto

[3] Invertible Bloom Lookup Tables, Michael T. Goodrich, Michael Mitzenmacher

[4] What's the Difference? Efficient Set Reconciliation without Prior Context, David Eppstein, Michael T. Goodrich, Frank Uyeda, George Varghese

[5] The Byzantine Generals Problem, Leslie LAMPORT, Robert SHOSTAK, and Marshall PEASE

[6] Algorand: Scaling Byzantine Agreements for Cryptocurrencies, Yossi Gilad

[7] Efficient signature generation by smart cards,Claus Peter Schnorr,Journal of Cryptology · January 1991

[8] Non-Interactive Proofs of Proof-of-Work, Aggelos Kiayias, Andrew Miller, and Dionysis Zindros

[9] The Bitcoin Backbone Protocol:Analysis and Applications, Juan A. Garay, Aggelos Kiayias, Nikos Leonardos

[10] Simple Schnorr Multi-Signatures with Applications to Bitcoin,Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille

10.11 10 Appendix

10.11.1 10.1 Bloomfilter

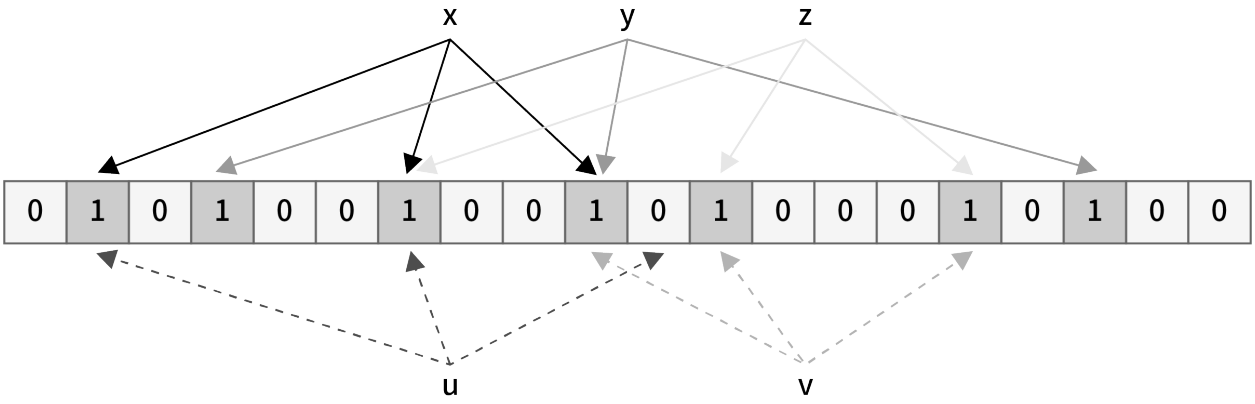
In a variety of scenarios, we need to determine whether an element is in a set, such as whether an email address is in the spam address list, and whether a citizen is on the suspect list. We can use a traditional hash table to store the set to be searched, then only one hash operation is needed to know if such an element exists in the collection. However, for this unique purpose, there is no need to save all the elements of this set. For example, for a collection with hundreds of millions of email addresses, it takes up very large storage space. Therefore, we introduce a technical means called Bloomfilter here to accomplish the above judgment in the case of ensuring the same search efficiency and greatly reducing the storage space occupation.

Problem

We assume that there are n elements to be compared in the set, and now a new element is given to determine whether it appears in the n elements.

Steps

Step 1. We use an m-bit space (that is, a binary vector of length m), so that the value of each bit in the space is 0; Step 2. We then give k independent hash generators, each of which can map the elements in the above set to a certain integer between 1 and m by a deterministic algorithm and guarantee the value of this operation is uniformly distributed in an integer from 1 to m. For any element, we use this k hash generators to map this element, and the value obtained by the mapping becomes 1 in the corresponding position of this m-bit space. If the number at this position has been changed to 1, then it remains unchanged at 1; Step 3. For all n elements, we repeat the step 2, so that we get a binary vector of length m with 0 or 1 in its each unit. This binary vector is called the "Bloomfilter"; Step 4. Perform the k hash mappings on the new element to be determined, and take out the value obtained by the mapping in the corresponding k positions in the m-bit space to see whether they are all 1.



This is a BloomFilter composed of x,y,z, we can see that u is not an element of this set, but v belongs to the set of false positive.

Conclusion

- If at least one of the k positions is 0, then this element must not be in this set.
- If the k positions are all 1, then this element is probably in this set (we will analyse the estimation of this probability later)

Model Evaluation

We find that we only need to store this Bloomfilter to determine whether the new element appears in the collection, and there is no need to save the original data. This way our storage space is changed from 16n bits to m bits. However, we have not significantly increased our calculation time: only k hashes are required.

Similarly, we mentioned in argument 2 that if the k positions are all 1, then this element is probably in this set. However there are possibilities for misjudgment, that is, an element not in the set may be incorrectly judged to be in. However, missing is not possible. If an element belongs to a certain set, it will not be judged as not belonging to this set.

In the next chapter, we execute the performance calculation of the Bloomfilter. The performance calculation includes two dimensions:

- How much storage space can be reduced by the Bloom filter?
- What is the false positive rate of the Bloom filter?

Space Efficiency of Bloomfilter

Return to our case of email. We assume that each email address occupy 2 bytes, namely 16 bits. Therefore this email set occupies a total space of 16n bits. However the size of Bloomfilter is m bits, the compression rate is 16n/m. So long as m<<16n, the compression rate would be very considerable. Now we analyse how this rate can be in combination with the bottleneck of Bloomfilter (i.e. the false positive rate)

False positive rate of BloomFilter

The probability that any one bit of the filter is marked 1 is 1-(1-1/m)nk

Then if the element is not in the collection, the probability that the k bits of the hash map are set to 1 is (1-(1-1/m)nk)k

approximately, (1-e(-nk/m))k

In order to get better performance of the compression and to make the false positive rate lower, we hope that the larger n/m the better, and the smaller nk/m the better.

Here we can take n/m=16, k=11, so the false rate is controlled below five ten thousandths.

Of course in this example, since the original data itself is not large, there is no compression effect, but if the original data and the number of bytes occupied is large, the effect will be very obvious. For example, in our scenario of block synchronization, because a transaction occupies a large space, this algorithm can save a lot of space and improve transmission efficiency.

In the scenario of our block synchronization, node A can transmit the Bloomfilter obtained by block hash calculation to other nodes, who find the transactions missing through the comparison with the filter, and tell node A to transmit the missing ones.

Although the Bloomfilter can save more transmission bandwidth and increase query efficiency, there are several obvious defects:

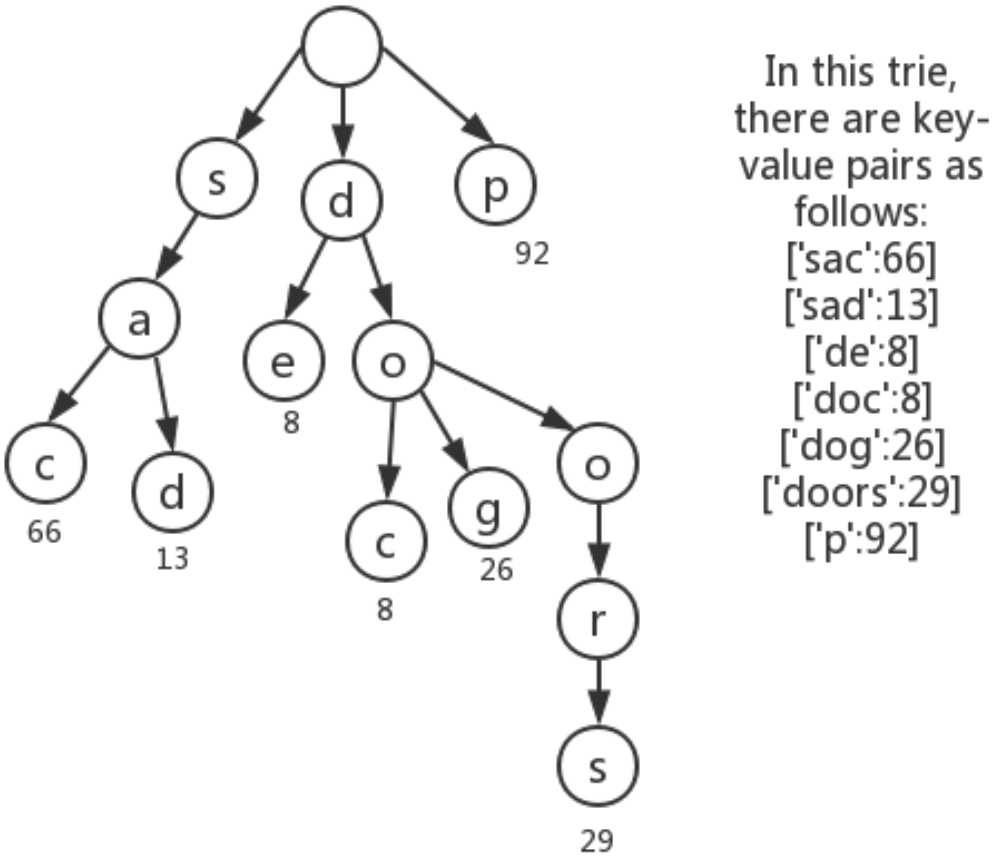
- The size of the Bloomfilter is proportional to the size of the set. If the set is large but the difference is small, then using a Bloomfilter is not cost effective.
- It can only support queries and can not restore blocks. The synchronization needs a second transfer
- There is still a certain false positive rate

10.11.2 10.2 TriePatricia Trie and Merkle Trie

Trie

We talked about the data storage in Section 5 that the storage-dependent MPT trie structure is derived from the Patricia tree and the Merkle tree. We start our discussion from a relatively simple structure called Trie.

The Trie, also known as the prefix tree, is a way to store key-value pairs. The key is searched along the tree from the trunk to the leaves, and the value is stored on the corresponding node at the location of the key discovered. The specific search method is illustrated by the following figure:



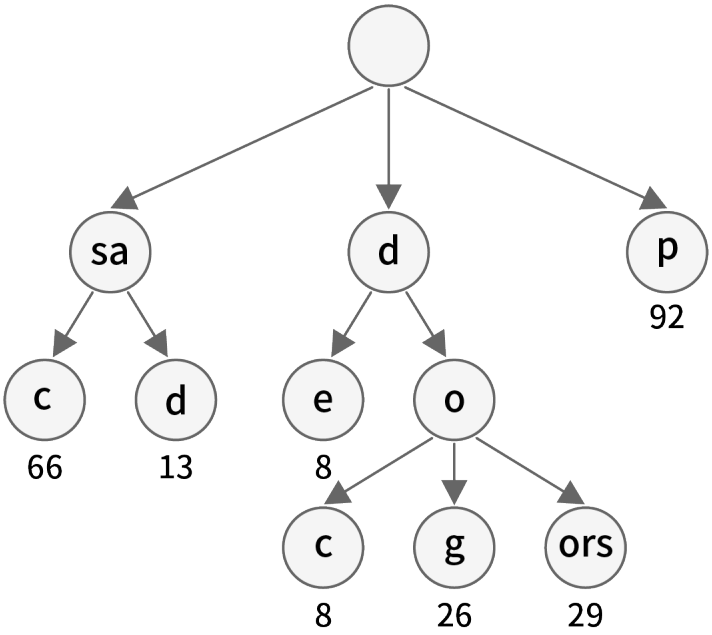
For example, if you want to find the value of the key 'dog' in this radix tree, you need to follow the path of the letter, from the tree root to the bottom leaf node of the tree and find the value corresponding to 'dog'. Specifically, first find the root node that stores the key-value pair data, the 'd' node of the next layer, then the node 'o', and finally 'g' and then the path root -> d -> o -> g is completed. This way you will eventually find the corresponding node of the value. Of course, in the coding practice, we will convert the data into hexadecimal code, so the data storage of each node will generally adopt the type of [i1, ... iN, value] where N=16. In i1 to iN are stored the hash of the child node and in 'value' is stored the value of the node (if any).

From the figure we can see that the key value with the same prefix will share the trunk part, and the leaf value can be obtained by finding the leaf node according to the key string order. Such a lookup has high search efficiency and can avoid collisions like hash tables.

Patricia Trie

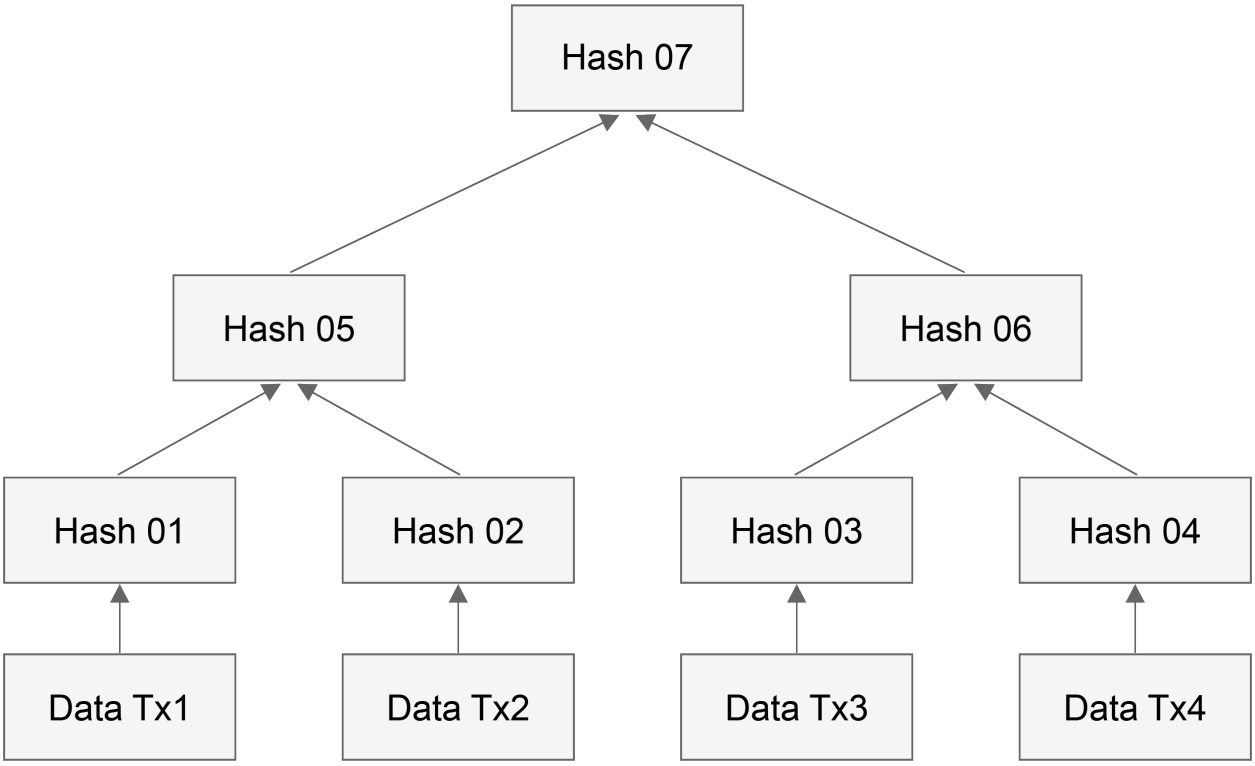
In order to avoid such lengthy lookups as the keywords 'doors' in the above figure, we construct a new tree call Patricia Trie by merging the parent node who has a unique child node with this child node.

For example, the tree in the above figure can evolve into the following Patricia trie:



Merkle Trie

The Merkle tree consists in dividing the data into many small data blocks and recording their hash values in turn on the leaf nodes. We combine the two adjacent hash values on one string and calculate the hash value of this string, which are stored on the nodes of the above layer. By continuous operation like this up to the root node, we can use the hash value check of the final root node to determine whether the data of the bottom leaf node has been falsified. Bitcoin uses the data structure of the Merkle tree. Only the corresponding Merkle path is needed to quickly perform transaction verification. As shown, Hash01 and Hash06 are the Merkle paths of Data Tx2, because for Tx2 hash calculation of the combination with Hash01 and Hash06 successively is needed to execute the comparison with the root node.


Hash 07
Hash 05
Hash 06
Hash 01
Hash 02
Hash 03
Hash 04
Data Tx1
Data Tx2
Data Tx3
Data Tx4

CHAPTER 11

Application-side

Question: How do application clients know the outcome of a transaction?

Answer: When client submit a transaction to a full node, a receipt would be returned. Clients can check the outcome of the transaction with the transaction id on the receipt from any full node.