(EOSIO v2.0.0-DAPPPRO2.0.0)
全球首个双合约生态应用启动

DAPPPRO2.0.0 是一个完全去中心化的开源项目，无需开发人员分配。用户需要利用 PRO 令牌通过将 PRO 发送到智能令牌 dapp.pro 来完成整个 PRO 的分配应用，任意一个获取 PRO 令牌者都可自行开发设置及组建生态。

1. 双合约生态应用概念
1.1 全球首个无需登陆及打开 dapp 链接可直接运行的双合约。
1.1.1 全球首个区块自执行，账户累计 100PRO 令牌双合约将自动赎回无需人工操作，完全实现去中心化。
1.1.2 想获取更多 PRO 令牌只需转入 PRO 令牌至 dapp.pro 双合约自动执行放大收益。

1.2 激活 PRO 生态只需转入 EOS 令牌双合约自动生产执行

2.赚 PRO 令牌参与以下生态应用即可获取更多令牌

2.2 四种应用生态获取更多 PRO 令牌

2.3 （推广分享）首次激活账户必须转入 10EOS 至（dapp.pro）需要备注邀请您的 EOS 账户合约自动锁定关系，如不备注则无法锁定邀请关系，双合约自 动锁定关联无法篡改。

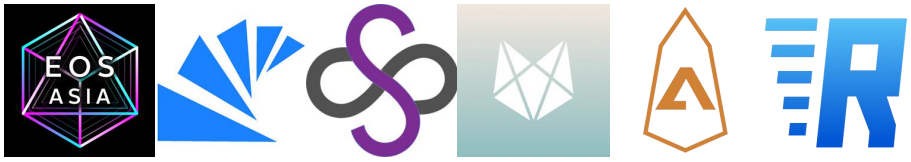
2.4 获取 PRO 令牌的锁定数量可用计算公式 100 的自动赎回公式核算

2.5 开发者可根据以下入口提交生态应用，PRO 已经成为全球唯一 一个双合约生态拥有者，同时也是全球完全去中心化的开源社区之一。

2.6 双合约生态不影响 PRO 之前所有生态详情见 github

2.7 github 开源链接 （eosio.cdt） https://github.com/EOSIO/welcome

EOSIO v2.0.0-DAPPPRO2.0.0）



EOSIO v2.0.0-DAPPPRO2.0.0）
Developer entry directory

1、Overview

1.1、Platform and Toolchain

The EOSIO platform is made up of the following components and toolchain:

nodeos (node + EOSIO = nodeos): the core EOSIO node daemon that can be configured with plugins to run a node. Example uses are block production, dedicated API endpoints, and local development. cleos (CLI + EOSIO = cleos): the command line interface to interact with the blockchain and to manage wallets. keosd (key + EOSIO = keosd): the component that securely stores EOSIO keys in wallets. EOSIO.CDT: toolchain for WebAssembly (Wasm) and a set of tools to facilitate smart contract writing for the EOSIO platform. The basic relationship between these components is illustrated in the following diagram:

EOSIO Development Lifecycle

Note EOSIO also provides a frontend library for javascript development called EOSJS along with Swift and Java SDKs for native mobile applications development.

Nodeos Nodeos is the core EOSIO node daemon. Nodeos handles the blockchain data persistence layer, peer-to-peer networking, and contract code scheduling. For development environments, nodeos enables you to set up a single node blockchain network. Nodeos offers a wide range of features through plugins which can be enabled or disabled at start time via the command line parameters or configuration files.

You can read detailed documentation about nodeos here.

Cleos cleos is a command line tool that interfaces with the REST APIs exposed by nodeos. You can also use cleos to deploy and test EOSIO smart contracts.

You can read detailed documentation about cleos here.

Keosd keosd is a key manager daemon for storing private keys and signing digital messages. keosd provides a secure key storage medium for keys to be encrypted in the associated wallet file. The keosd daemon also defines a secure enclave for signing transaction created by cleos or a third party library.

Note keosd can be accessed using the wallet API, but it is important to note that the intended usage is for local light client applications. keosd is not for cross network access by web applications trying to access users' wallets.

You can read detailed documentation about keosd here.

EOSIO.CDT EOSIO.CDT is a toolchain for WebAssembly (Wasm) and a set of tools to facilitate contract writing for the EOSIO platform. In addition to being a general-purpose WebAssembly toolchain, EOSIO-specific optimizations are available to support building EOSIO smart contracts. This new toolchain is built around Clang 7, which means that EOSIO.CDT has most of the current optimizations and analyses from LLVM.

EOSJS A Javascript API SDK for integration with EOSIO-based blockchains using the EOSIO RPC API.

1.2、Core Concepts

Accounts, Wallets and Permissions Accounts

An account is a human-readable name that is stored on the blockchain. It can be owned through authorization by an individual or group of individuals depending on permissions configuration. An account is required to transfer or push any valid transaction to the blockchain.

Wallets

Wallets are clients that store keys that may or may not be associated with the permissions of one or more accounts. Ideally, a wallet has a locked (encrypted) and unlocked (decrypted) state that is protected by a high entropy password. The EOSIO/eos repository comes bundled with a CLI client called cleos that interfaces with a lite-client called keosd and together, they demonstrate this pattern.

Authorization and Permissions

Permissions are arbitrary names used to define the requirements for a transaction sent on behalf of that permission. Permissions can be assigned for authority over specific contract actions by linking authorization or linkauth.

For more information about these concepts, see the Accounts and Permissions documentation.

Smart Contracts A smart contract is a piece of code that can execute on a blockchain and keep the state of contract execution as a part of the immutable history of that blockchain instance. Therefore, developers can rely on that blockchain as a trusted computation environment in which inputs, execution, and the results of a smart contract are independent and free of external influence.

Delegated Proof of Stake (DPOS) The EOSIO platform implements a proven decentralized consensus algorithm capable of meeting the performance requirements of applications on the blockchain called the Delegated Proof of Stake (DPOS). Under this algorithm, if you hold tokens on a EOSIO-based blockchain, you can select block

producers through a continuous approval voting system. Anyone can choose to participate in the block production and will be given an opportunity to produce blocks, provided they can persuade token holders to vote for them.

For more information about DPOS BFT, see EOSIO Consensus.

System Resources RAM RAM, in a EOSIO-based blockchain, is one of the important system resources consumed by blockchain accounts and smart contracts. RAM acts as a permanent storage and is used to store account names, permissions, token balance and other data for speedy on-chain data access. RAM needs to be purchased and is not based on staking as it is a limited persistent resource.

More details about RAM as a system resource can be found here.

CPU CPU, in a EOSIO-based blockchain, represents the processing time of an action and is measured in microseconds (μs). CPU is referred to as cpu bandwidth in the cleos get account command output and indicates the amount of processing time an account has at its disposal when pushing actions to a contract. CPU is a transient system resource and falls under the staking mechanism of EOSIO.

More details about CPU as a system resource can be found here.

Network (NET) Besides CPU and RAM, NET is also a very important resource in EOSIO-based blockchains. NET is the network bandwidth, measured in bytes, of transactions and is referred to as net bandwidth on the cleos get account command. NET is a also a transient system resource and falls under the staking mechanism of EOSIO.

More details about NET as a system resource can be found here.

1.3、Technical Features

WebAssembly C++ Compilation EOSIO uses C++ as the smart contract programming language. If you are a C++ developer, you do not need to learn a new programming language to understand Smart Contract APIs and how EOSIO supports smart contract development through C++ classes and structures. With your existing C++ programming capabilities, you can onboard EOSIO development faster and be able to program EOSIO smart contracts using specific EOSIO canonical C++ code constructs and Smart Contract APIs in no time.

On top of the EOSIO core layer, a WebAssembly (Wasm) virtual machine, EOS VM, executes smart contract code, and it is designed from the ground up for the high demands of blockchain applications which require far more from a WebAssembly engine than those designed for web browsers or standards development. The design choice of using Wasm enables EOSIO to reuse optimized and battle-tested compilers and toolchains which are being maintained and improved by a broader community. In

addition, adopting Wasm standard also makes it easier for compiler developers to port other programming languages onto the EOSIO platform.

High Throughput, Faster Confirmations and Lower Latency EOSIO is designed with high transactions throughput in mind and each new version is achieving significant improvements.

The consensus mechanism of Delegated Proof of Stake (DPOS) achieves high transaction throughputs because DPOS does not need to wait for all the nodes to complete a transaction to achieve finality. This behavior results in faster confirmations and lower latency.

The EOS-VM mentioned earlier, and all its blockchain-dedicated features and improvements add significant contribution to the overall performance of the EOSIO-based blockchains.

Programmable Economics and Governance The resource allocation and governance mechanism of any EOSIO-based blockchains are programmable through smart contracts. You can modify the system smart contracts to customize the resource allocation model and governance rules of a EOSIO blockchain. The on-chain governance mechanism can be modified using system smart contracts, as the core layer code does not always have to be updated for the changes to take place.

Staking Mechanism In EOSIO-based blockchains, access to the system resources is regulated by a process called the staking mechanism. The system resources that fall under the scope of the staking mechanism are CPU and NET. You can directly interact with the blockchain through the cleos CLI, the RPC APIs, or an application to access CPU and NET by staking system tokens.

Note RAM is a persistent system resource on the EOSIO blockchain and does not fall in the scope of the staking mechanism.

When you stake tokens for CPU and NET, you gain access to system resources proportional to the total amount of tokens staked by all other users for the same resource at the same time. This means you can perform transactions at no cost but in the limitations of the staked tokens. The staked tokens guarantee the proportional amount of resources regardless of any variations in the free market.

You can also allocate the system resources to EOSIO accounts in a programmatic manner by customizing the resource allocation model in the system smart contract. This flexibility is provided by the programmable economics of the EOSIO platform.

Business Model Flexibility Applications built on EOSIO can adopt a freemium model in which application users do not need to pay for the cost of resources needed to execute transactions.

The execution of the costless transactions under the freemium model is facilitated by applications co-signing the transactions with the user. Alternatively, applications can also stake enough system tokens to guarantee the resources needed.

Comprehensive Permission Schema The EOSIO platform has a comprehensive permission system for creating custom permission schemata for various use cases. For example, you can create a custom permission and use it to protect a specific feature of a smart contract. You can also split the authority required to modify a smart contract across multiple accounts with different levels of authority. This comprehensive permission system allows you to build a permissioned application on top of a flexible infrastructure.

Upgradability Applications deployed on EOSIO-based blockchains are upgradeable. This means you can deploy code fix, add features, and change the application’ s logic, as long as sufficient authority is provided. As a developer, you can iterate your application without the risk of being locked-in to a software bug permanently. It is also possible, however, to deploy smart contracts that cannot be modified on a EOSIO-based blockchain. These decisions are at the discretion of developers rather than restricted by the protocol.

Efficient energy consumption With DPOS as the consensus mechanism, EOSIO consumes much less energy to validate transactions and secure a blockchain compared to other consensus algorithms.

2、Getting Started

2.1、Try EOSIO

Try EOSIO in your web browser If you do not want to install EOSIO binaries on your local machine, or if you have other types of restrictions installing EOSIO binaries, you can try EOSIO in a web browser. This approach is powered by Gitpod.io and Docker, and it provides developers with a personal single-node EOSIO blockchain for development and testing purposes running in Gitpod.io's cloud accessible from your web browser.

You can give it a try in a matter of seconds right now and you can read more details about it on the eosio-web-ide project page.

What's Next? Getting Started: Install EOSIO in your local development environment.

2.1、Development Environm

2.1.1、Prerequisites

EOSIO versions The subsequent tutorials are up to date with the following EOSIO components.

Component Version eosio 2.0.0 eosio.cdt 1.7.0 eosio.contracts 1.9.0 Development Experience EOSIO based blockchains execute user-generated applications and code using WebAssembly (WASM). WASM is an emerging web standard with widespread support from Google, Microsoft, Apple, and industry leading companies.

At the moment the most mature toolchain for building applications that compile to WASM is clang/llvm with their C/C++ compiler. For best compatibility, it is recommended that you use the EOSIO C++ toolchain.

Other toolchains in development by 3rd parties include: Rust, Python, and Solidity. While these other languages may appear simpler, their performance will likely impact the scale of application you can build. We expect that C++ will be the best language for developing high-performance and secure smart contracts and plan to use C++ for the foreseeable future.

Operating System The EOSIO software supports the following environments for development and/or deployment:

Amazon Linux 2 Centos 7 Ubuntu 16.04 Ubuntu 18.04 MacOS 10.14 (Mojave) and higher Note if you are developing on Windows, unfortunately we do not provide PowerShell ports and instructions at this time. In the future we may append PowerShell commands. In the mean-time your best bet is to use a VM with Ubuntu, and set up your development environment inside this VM. If you're an advanced Window's developer familiar with porting Linux instructions, you should encounter minimal issues.

Command Line Knowledge There are a variety of tools provided along with EOSIO which requires you to have basic command line knowledge in order to interact with.

Development Tools We can use any text editor that, preferably, supports C++ syntax highlighting. Some of the popular editors are Sublime Text and Atom. Another option is an IDE, which provides a more sophisticated code completion and more complete development experience. You are welcome to use the software of your personal preference, but if you're unsure what to use we've provided some options for you to explore.

Potential Editors and IDEs Sublime Text Atom Editor CLion Eclipse Visual Studio Code The resources listed above are developed, offered, and maintained by third-parties and not by block.one. Providing information, material, or commentaries about such third-party resources does not mean we endorse or

recommend any of these resources. We are not responsible, and disclaim any responsibility or liability, for your use of or reliance on any of these resources. Third-party resources may be updated, changed or terminated at any time, so the information below may be out of date or inaccurate. USAGE AND RELIANCE IS ENTIRELY AT YOUR OWN RISK

Alternatively, you can try out some community driven IDEs specifically developed for EOSIO:

EOS Studio What you'll learn Only a sample of what you'll learn

How to quickly spin up a node Manage wallets and keys Create Accounts Write some contracts Compilation and ABI Deploy contracts What's Next? Before You Begin: Steps to download and install binaries on your system.

2.1.2、Before You Begin

Step 1: Install binaries To get started as quickly as possible we recommend using pre-built binaries. Building from source is a more advanced option but will set you back an hour or more and you may encounter build errors.

You can find how to build EOSIO from source here.

The below commands will download binaries for respective operating systems.

Mac OS X Brew Install: brew tap eosio/eosio brew install eosio If you don't have Brew installed, follow the installation instructions on the official Brew website.

Ubuntu 18.04 Debian Package Install: wget https://github.com/EOSIO/eos/releases/download/v2.0.0/eosio_2.0.0-1-ubuntu-18.04_amd64.deb sudo apt install ./eosio_2.0.0-1-ubuntu-18.04_amd64.deb Ubuntu 16.04 Debian Package Install: wget https://github.com/EOSIO/eos/releases/download/v2.0.0/eosio_2.0.0-1-ubuntu-16.04_amd64.deb sudo apt install ./eosio_2.0.0-1-ubuntu-16.04_amd64.deb CentOS RPM Package Install: wget https://github.com/EOSIO/eos/releases/download/v2.0.0/eosio-2.0.0-1.el7.x86_64.rpm sudo yum install ./eosio-2.0.0-1.el7.x86_64.rpm If you have previous versions of eosio installed on your system, please uninstall before proceeding. For detailed instructions, see here.

Step 2: Setup a development directory, stick to it. You're going to need to pick a directory to work from, it's suggested to create a contracts directory somewhere on your local drive.

mkdir contracts cd contracts Step 3: Enter your local directory below. Get the path of that directory and save it for later, as you're going to need it, you can use the following command to get your absolute path.

pwd Enter the absolute path to your contract directory below, and it will be inserted throughout the documentation to make your life a bit easier. This functionality requires cookies

cli

Absolute Path to Contract Directory

What's Next? Install the CDT: Steps to install the EOSIO Contract Development Toolkit (CDT) on your system.

2.1.3、Install The CDT

Install the Contract Dev. Toolkit The EOSIO Contract Development Toolkit, CDT for short, is a collection of tools related to contract compilation. Subsequent tutorials use the CDT primarily for compiling contracts and generating ABIs.

Starting from 1.3.x, CDT supports Mac OS X brew, Linux Debian and RPM packages. The easiest option to install would be using one of these package systems. Pick one installation method only.

If you have versions of eosio.cdt prior to 1.3.0 installed on your system, please uninstall before proceeding

Homebrew (Mac OS X) Install brew tap eosio/eosio.cdt brew install eosio.cdt Uninstall brew remove eosio.cdt Ubuntu (Debian) Install wget https://github.com/EOSIO/eosio.cdt/releases/download/v1.6.3/eosio.cdt_1.6.3-1-ubuntu-18.04_amd64.deb

sudo apt install ./eosio.cdt_1.6.3-1_amd64.deb Uninstall sudo apt remove eosio.cdt CentOS/Redhat (RPM) Install wget https://github.com/EOSIO/eosio.cdt/releases/download/v1.6.3/eosio.cdt-1.6.3-1.el7.x86_64.rpm

sudo yum install ./eosio.cdt-1.6.3-1.el7.x86_64.rpm Uninstall $ sudo yum remove eosio.cdt Install from Source The location where eosio.cdt is cloned is not that important because you will be installing eosio.cdt as a local binary in later steps. For now, you can clone eosio.cdt to your "contracts" directory previously created, or really anywhere else on your local system you see fit.

cd CONTRACTS_DIR Download Clone version 1.6.3 of the eosio.cdt repository.

git clone --recursive https://github.com/eosio/eosio.cdt --branch v1.6.3 --single-branch cd eosio.cdt It may take up to 30 minutes to clone the repository

Virtual machine environment In case you are using a virtual machine. It should be configured with at least 2 CPUs (does not have to be two physical ones) and 8G of memory to avoid compilation errors

Build ./build.sh Install sudo ./install.sh The above command needs to be ran with sudo because eosio.cdt's various binaries will be installed locally. You will be asked for your computer's account password.

Installing eosio.cdt will make the compiled binary global so it can be accessable anywhere. For this tutorial, it is strongly suggested that you do not skip the install step for eosio.cdt, failing to install will make it more difficult to follow this and other tutorials, and make usage in general more difficult.

Troubleshooting Getting Errors during build. Search your errors for the string "/usr/local/include/eosiolib/" If found, rm -fr /usr/local/include/eosiolib/ or navigate to /usr/local/include/ and delete eosiolib using your operating system's file browser. What's Next? Create Development Wallet: Steps to create a new development wallet used to store public-private key pair.

2.1.4、Create Development Wallet

Create Development Wallet Wallets are repositories of public-private key pairs. Private keys are needed to sign operations performed on the blockchain. Wallets are accessed using cleos.

Step 1: Create a Wallet The first step is to create a wallet. Use cleos wallet create to create a new "default" wallet using the option --to-console for simplicity. If using cleos in production, it's wise to instead use --to-file so your wallet password is not in your bash history. For development purposes and because these are development and not production keys --to-console poses no security threat.

cleos wallet create --to-console cleos will return a password, save this password somewhere as you will likely need it later in the tutorial.

Creating wallet: default Save password to use in the future to unlock this wallet. Without password imported keys will not be retrievable. "PW5Kewn9L76X8Fpd.....t42S9XCw2" About Wallets A common misconception in cryptocurrency regarding wallets is that they store tokens. However, in reality, a wallet is used to store private keys in an encrypted file to sign transactions. Wallets do not serve as a storage medium for tokens.

A user builds a transaction object, usually through an interface, sends that object to the wallet to be signed, the wallet then returns that transaction object with a signature which is then broadcast to the network. When/if the network confirms that the transaction is valid, it is included into a block on the blockchain.

Step 2: Open the Wallet Wallets are closed by default when starting a keosd instance, to begin, run the following

cleos wallet open Run the following to return a list of wallets.

cleos wallet list and it will return

Wallets: ["default"] Step 3: Unlock it The keosd wallet(s) have been opened, but is still locked. Moments ago you were provided a password, you're going to need that now.

cleos wallet unlock You will be prompted for your password, paste it and press enter.

Now run the following command

cleos wallet list It should now return

Wallets: ["default *"] Pay special attention to the asterisk (*). This means that the wallet is currently unlocked

Step 4: Import keys into your wallet Generate a private key, cleos has a helper function for this, just run the following.

cleos wallet create_key It will return something like..

Created new private key with a public key of: "EOS8PEJ5FM42xLpHK...X6PymQu97KrGDJQY5Y" Step 5: Follow this tutorial series more easily Enter the public key provided in the last step in the box below. It will persist the development public key you just generated throughout the documentation.

Development Public Key Step 6: Import the Development Key Every new EOSIO chain has a default "system" user called "eosio". This account is used to setup the chain by loading system contracts that dictate the governance and consensus of the EOSIO chain. Every new EOSIO chain comes with a development key, and this key is the same. Load this key to sign transactions on behalf of the system user (eosio)

cleos wallet import You'll be prompted for a private key, enter the eosio development key provided below

5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3 Important Never use the development key for a production account! Doing so will most certainly result in the loss of access to your account, this private key is publicly known.

Wonderful, you now have a default wallet unlocked and loaded with a key, and are ready to proceed.

What's Next? Start Your Node: Steps to start keosd and nodeos.

2.1.5、Start Your Node Setup

Start keosd and nodeos Step 1: Boot Node and Wallet Step 1.1: Start keosd First let us start keosd:

keosd & You should see some output that looks like this:

info 2018-11-26T06:54:24.789 thread-0 wallet_plugin.cpp:42 plugin_initialize] initializing wallet plugin info 2018-11-26T06:54:24.795 thread-0 http_plugin.cpp:554 add_handler] add api url: /v1/keosd/stop info 2018-11-26T06:54:24.796 thread-0 wallet_api_plugin.cpp:73 plugin_startup] starting wallet_api_plugin info 2018-11-26T06:54:24.796 thread-0 http_plugin.cpp:554 add_handler] add api url: /v1/wallet/create info 2018-11-26T06:54:24.796 thread-0 http_plugin.cpp:554 add_handler] add api url: /v1/wallet/create_key info 2018-11-26T06:54:24.796 thread-0 http_plugin.cpp:554 add_handler] add api url: /v1/wallet/get_public_keys Press enter to continue

Step 1.2: Start nodeos Start nodeos now:

nodeos -e -p eosio \ --plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ --plugin eosio::history_plugin \ --plugin eosio::history_api_plugin \ --filter-on="" \ *--access-control-allow-origin*=" \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors >> nodeos.log 2>&1 & These settings accomplish the following:

Run Nodeos. This command loads all the basic plugins, set the server address, enable CORS and add some contract debugging and logging. Enable CORS with no restrictions (*) and development logging In the above configuration, CORS is enabled for* for development purposes only, you should never enable CORS for * on a node that is publicly accessible!

Troubleshooting If in the previous step, after starting nodeos, you see an error message similar to "Database dirty flag set (likely due to unclean shutdown):

replay required" try to start nodeos with --replay-blockchain. More details on troubleshooting nodeos can be found here.

Step 2: Check the installation Step 2.1: Check that Nodeos is Producing Blocks Run the following command

tail -f nodeos.log You should see some output in the console that looks like this:

1929001ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366974ce4e2a... #13929 @ 2018-05-23T16:32:09.000 signed by eosio [trxs: 0, lib: 13928, confirmed: 0] 1929502ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366aea085023... #13930 @ 2018-05-23T16:32:09.500 signed by eosio [trxs: 0, lib: 13929, confirmed: 0] 1930002ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366b7f074fdd... #13931 @ 2018-05-23T16:32:10.000 signed by eosio [trxs: 0, lib: 13930, confirmed: 0] 1930501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366cd8222adb... #13932 @ 2018-05-23T16:32:10.500 signed by eosio [trxs: 0, lib: 13931, confirmed: 0] 1931002ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366d5c1ec38d... #13933 @ 2018-05-23T16:32:11.000 signed by eosio [trxs: 0, lib: 13932, confirmed: 0] 1931501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366e45c1f235... #13934 @ 2018-05-23T16:32:11.500 signed by eosio [trxs: 0, lib: 13933, confirmed: 0] 1932001ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366f98adb324... #13935 @ 2018-05-23T16:32:12.000 signed by eosio [trxs: 0, lib: 13934, confirmed: 0] 1932501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 00003670a0f01daa... #13936 @ 2018-05-23T16:32:12.500 signed by eosio [trxs: 0, lib: 13935, confirmed: 0] 1933001ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 00003671e8b36e1e... #13937 @ 2018-05-23T16:32:13.000 signed by eosio [trxs: 0, lib: 13936, confirmed: 0] 1933501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000367257fe1623... #13938 @ 2018-05-23T16:32:13.500 signed by eosio [trxs: 0, lib: 13937, confirmed: 0] Press ctrl + c to close the log

Step 2.2: Check the Wallet Open the shell and run the cleos command to list available wallets. We will talk more about wallets in the future. For now, we need to validate the installation and see that the command line client cleos is working as intended.

cleos wallet list You should see a response with an empty list of wallets:

Wallets: [] From this point forward, you'll be executing commands from your local system (Linux or Mac)

Step 2.3: Check Nodeos endpoints This will check that the RPC API is working correctly, pick one.

Check the get_info endpoint provided by the chain_api_plugin in your browser: http://localhost:8888/v1/chain/get_info Check the same thing, but in the console on your host machine curl http://localhost:8888/v1/chain/get_info What's Next? Create Test Accounts: Learn how to create test accounts in the EOSIO blockchain along with troubleshooting steps.

2.1.6、Create Test Accounts

Create Test Accounts What is an account? An account is a collection of authorizations, stored on the blockchain, and used to identify a sender/recipient. It has a flexible authorization structure that enables it to be owned either by an individual or group of individuals depending on how permissions have been configured. An account is required to send or receive a valid transaction to the blockchain

This tutorial series uses two "user" accounts, bob and alice, as well as the default eosio account for configuration. Additionally accounts are made for various contracts throughout this tutorial series.

Step 1: Create Test Accounts Public Key Persistence In section 1.4 Create Development Wallet, you created a development key pair and pasted the public key in the Development Public Key field for the value to persist throughout the tutorial.

In the following steps, if you see YOUR_PUBLIC_KEY instead of the public key value, you can either go back to section 1.4 Create Development Wallet and persist the value or replace YOUR_PUBLIC_KEY with the public key value manually.

Throughout these tutorials the accounts bob and alice are used. Create two accounts using cleos create account

cleos create account eosio bob YOUR_PUBLIC_KEY cleos create account eosio alice YOUR_PUBLIC_KEY You should then see a confirmation message similar to the following for each command that confirms that the transaction has been broadcast.

executed transaction: 40c605006de... 200 bytes 153 us

eosio <= eosio::newaccount {"creator":"eosio","name":"alice","owner":{"threshold":1,"keys":[{"key":"EOS5rti4LTL53xptjgQBXv9HxyU...

warning: transaction executed locally, but may not be confirmed by the network yet] Step 2: Public Key Note in cleos command a public key is associated with account alice. Each EOSIO account is associated with a public key.

Be aware that the account name is the only identifier for ownership. You can change the public key but it would not change the ownership of your EOSIO account.

Check which public key is associated with alice using cleos get account

cleos get account alice You should see a message similar to the following:

permissions: owner 1: 1 EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV active 1: 1 EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV memory: quota: unlimited used: 3.758 KiB

net bandwidth: used: unlimited available: unlimited limit: unlimited

cpu bandwidth: used: unlimited available: unlimited limit: unlimited Notice that actually alice has both owner and active public keys. EOSIO has a unique authorization structure that has added security for your account. You can minimize the exposure of your account by keeping the owner key cold, while using the key associated with your active permission. This way, if your active key were ever compromised, you could regain control over your account with your owner key.

In term of authorization, if you have a owner permission you can change the private key of active permission. But you cannot do so other way around.

Using Different Keys for Active/Owner on a PRODUCTION Network In this tutorial we are using the same public key for both owner and active for simplicity. In production network, two different keys are strongly recommended

Troubleshooting If you get an error while creating the account, make sure your wallet is unlocked

cleos wallet list You should see an asterisk (*) next to the wallet name, as seen below.

Wallets: ["default *"] What's Next? Hello World: The Hello World of EOSIO! Learn how to set up your environment and deploy your first smart contract.

2.2、Smart Contract Devel

2.2.1、Hello World Contract

Create a new directory called "hello" in the contracts directory you previously created, or through your system GUI or with cli and enter the directory.

cd CONTRACTS_DIR mkdir hello cd hello Create a new file, "hello.cpp" and open it in your favourite editor.

touch hello.cpp Below the eosio.hpp header file is included. The eosio.hpp file includes a few classes required to write a smart contract.

include

Using the eosio namespace will reduce clutter in your code. For example, by setting using namespace eosio;, eosio::print("foo") can be written print("foo")

using namespace eosio; Create a standard C++11 class. The contract class needs to extend eosio::contract class which is included earlier from the eosio.hpp header

include

using namespace eosio;

class [[eosio::contract]] hello : public contract {}; An empty contract doesn't do much good. Add a public access specifier and a using-declaration. The using declaration will allow us to write more concise code.

include

using namespace eosio;

class [[eosio::contract]] hello : public contract { public: using contract::contract; }; This contract needs to do something. In the spirit of hello world write an action that accepts a "name" parameter, and then prints that parameter out.

Actions implement the behaviour of a contract

include

using namespace eosio;

class [[eosio::contract]] hello : public contract { public: using contract::contract;

```
[[eosio::action]]
void hi( name user ) {
   print( "Hello, ", user);
}
```

}; The above action accepts a parameter called user that's a name type. EOSIO comes with a number of typedefs, one of the most common typedefs you'll encounter is name. Using the eosio::print library previously included, concatenate a string and print the user parameter. Use the braced initialization of name{user} to make the user parameter printable.

As is, the ABI <> generator in eosio.cdt won't know about the hi() action without an attribute. Add a C++11 style attribute above the action, this way the abi generator can produce more reliable output.

include

using namespace eosio;

class [[eosio::contract]] hello : public contract { public: using contract::contract;

```
[[eosio::action]]
void hi( name user ) {
   print( "Hello, ", user);
}
```

}; Everything together, here's the completed hello world contract

include

using namespace eosio;

class [[eosio::contract]] hello : public contract { public: using contract::contract;

```
[[eosio::action]]
void hi( name user ) {
   print( "Hello, ", user);
}
```

}; The ABI Generator in eosio.cdt supports several different style of attributes, see the ABI usage guide here

You can compile your code to web assembly (.wasm) as follows:

eosio-cpp hello.cpp -o hello.wasm When a contract is deployed, it is deployed to an account, and the account becomes the interface for the contract. As mentioned earlier these tutorials use the same public key for all of the accounts to keep things simple.

cleos wallet keys Create an account for the contract using cleos create account, with the command provided below.

cleos create account eosio hello YOUR_PUBLIC_KEY -p eosio@active Deploy the compiled wasm to the blockchain with cleos set contract.

Get an error? Check if your wallet needs to be unlocked.

In previous steps you should have created a contracts directory and obtained the absolute path and then saved it into a cookie. Replace "CONTRACTS_DIR" in the command below with the absolute path to your contracts directory. cleos set contract hello CONTRACTS_DIR/hello -p hello@active Great! Now the contract is set, push an action to it.

cleos push action hello hi '["bob"]' -p bob@active executed transaction: 4c10c1426c16b1656e802f3302677594731b380b18a44851d38e8b5275072857 244 bytes 1000 cycles

hello.code <= hello.code::hi {"user":"bob"}

> Hello, bob As written, the contract will allow any account to say hi to any user

cleos push action hello hi '["bob"]' -p alice@active executed transaction: 28d92256c8ffd8b0255be324e4596b7c745f50f85722d0c4400471bc184b9a16 244 bytes 1000 cycles

hello.code <= hello.code::hi {"user":"bob"}

> Hello, bob As expected, the console output is "Hello, bob"

In this case "alice" is the one who authorized it and user is just an argument. Modify the contract so that the authorizing user, "alice" in this case, must be the same as the user the contract is responding "hi" to. Use the require_auth method. This method takes a name as a parameter, and will check if the user executing the action matches the provided parameter.

void hi(name user) { require_auth(user); print("Hello, ", name{user}); } Recompile the contract

eosio-cpp -abigen -o hello.wasm hello.cpp And then update it

cleos set contract hello CONTRACTS_DIR/hello -p hello@active Try to execute the action again, but this time with mismatched authorization.

cleos push action hello hi '["bob"]' -p alice@active As expected, require_auth halted the transaction and threw an error.

Error 3090004: Missing required authority Ensure that you have the related authority inside your transaction!; If you are currently using 'cleos push action' command, try to add the relevant authority using -p option. Now, with our change, the contract verifies the provided name user is the same as the authorising user. Try it again, but this time, with the authority of the "alice" account.

cleos push action hello hi '["alice"]' -p alice@active executed transaction: 235bd766c2097f4a698cfb948eb2e709532df8d18458b92c9c6aae74ed8e4518 244 bytes 1000 cycles

hello <= hello::hi {"user":"alice"}

> Hello, alice What's Next? Deploy, Issue and Transfer Tokens: Learn how to deploy, issue and transfer tokens.

2.2.2、Deploy, Issue and Transfer Tokens

Step 1: Obtain Contract Source Navigate to your contracts directory.

cd CONTRACTS_DIR Pull the source

git clone https://github.com/EOSIO/eosio.contracts --branch v1.7.0 --single-branch This repository contains several contracts, but it's the eosio.token contract that is important for this section. Navigate to the eosio.contracts/contracts/eosio.token directory.

cd eosio.contracts/contracts/eosio.token Step 2: Create Account for Contract Before we can deploy the token contract we must create an account to deploy it to, we'll use the eosio development key for this account.

You need to unlock your wallet prior to the next step

cleos create account eosio eosio.token EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV Step 3: Compile the Contract eosio-cpp -I include -o eosio.token.wasm src/eosio.token.cpp --abigen Step 4: Deploy the Token Contract cleos set contract eosio.token CONTRACTS_DIR/eosio.contracts/contracts/eosio.token --abi eosio.token.abi -p eosio.token@active Reading WASM from ...eosio.contracts/contracts/eosio.token/eosio.token.wasm... Publishing contract... executed transaction: a68299112725b9f2233d56e58b5392f3b37d2a4564bdf99172152c21c7dc323f 6984 bytes 6978 us

eosio <= eosio::setcode {"account":"eosio.token","vmtype":0,"vmversion":0,"code":"0061736d01000000001a0011b60000060017e006002...

eosio <= eosio::setabi {"account":"eosio.token","abi":"0e656f73696f3a3a6162692f312e310008076163636f756e740001076163636f756e6e63...

warning: transaction executed locally, but may not be confirmed by the network yet] Step 5: Create the Token To create a new token, call create action with the correct parameters. This action accepts 1 argument, it consists of:

An issuer that is an eosio account. In this case, it's alice. This issuer will be the one with the authority to call issue and/or perform other actions such as closing accounts or retiring tokens. An asset type composed of two pieces of data, a floating-point number sets the maximum supply and a symbol in capitalized alpha characters which represents the asset. For example, "1.0000 SYS". Below is a concise way to call this method, using positional arguments:

cleos push action eosio.token create '["alice", "1000000000.0000 SYS"]' -p eosio.token@active The command above created a new token SYS with a precision of 4 decimals and a maximum supply of 1000000000.0000 SYS. It also designates alice as the issuer. To create this token, the contract requires the permission of the eosio.token account. For this reason, -p eosio.token@active was passed to authorize this action.

An alternate approach uses named arguments:

cleos push action eosio.token create '{"issuer":"alice", "maximum_supply":"1000000000.0000 SYS"}' -p eosio.token@active Execute the command above:

executed transaction: 10cfe1f7e522ed743dec39d83285963333f19d15c5d7f0c120b7db652689a997 120 bytes 1864 us

eosio.token <= eosio.token::create {"issuer":"alice","maximum_supply":"1000000000.0000 SYS"}

warning: transaction executed locally, but may not be confirmed by the network yet] Step 6: Issue Tokens The issuer alice can now issue new tokens. As mentioned earlier only the issuer can do so, therefore, -p alice@active must be provided to authorize the issue action.

cleos push action eosio.token issue '["alice", "100.0000 SYS", "memo"]' -p alice@active executed transaction: d1466bb28eb63a9328d92ddddc660461a16c405dffc500ce4a75a10aa173347a 128 bytes 205 us

eosio.token <= eosio.token::issue {"to":"alice","quantity":"100.0000 SYS","memo":"memo"}

warning: transaction executed locally, but may not be confirmed by the network yet] This time the output contains several actions: one issue action and three transfer actions. While the only action signed was issue, the issue action performed an inline transfer and the inline transfer notified the sender and receiver accounts. The output indicates all the action handlers that were called, the order they were called in, and whether any output was generated by the action.

Technically, the eosio.token contract could have skipped the inline transfer and opted to just modify the balances directly. However, in this case the eosio.token contract is following a token convention that requires that all account balances be derivable by the sum of the transfer actions that reference them. It also requires that the sender and receiver of funds be notified so they can automate handling deposits and withdrawals.

To inspect the transaction, try using the -d -j options, which indicate "don't broadcast" and "return the transaction as json", which you may find useful during development.

cleos push action eosio.token issue '["alice", "100.0000 SYS", "memo"]' -p alice@active -d -j Step 7: Transfer Tokens Now that account alice has been issued tokens, transfer some of them to account bob.

cleos push action eosio.token transfer '["alice", "bob", "25.0000 SYS", "m"]' -p alice@active executed transaction: 800835f28659d405748f4ac0ec9e327335eae579a0d8e8ef6330e78c9ee1b67c 128 bytes 1073 us

eosio.token <= eosio.token::transfer {"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"m"}

alice <= eosio.token::transfer {"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"m"}

bob <= eosio.token::transfer {"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"m"}

warning: transaction executed locally, but may not be confirmed by the network yet] Now check if bob received the tokens using cleos get currency balance

cleos get currency balance eosio.token bob SYS 25.0000 SYS Check alice's balance. Notice that tokens were deducted from the account.

cleos get currency balance eosio.token alice SYS 75.0000 SYS Excellent! Everything adds up.

What's Next? Understanding ABI Files: Introduction to Application Binary Files (ABI) and how the ABI file correlates to the eosio.token contract.

2.2.3、Understanding ABI Files

Introduction Previously you deployed the eosio.token contract using the provided ABI file. This tutorial will overview how the ABI file correlates to the eosio.token contract.

ABI files can be generated using the eosio-cpp utility provided by eosio.cdt. However, there are several situations that may cause ABI's generation to malfunction or fail altogether. Advanced C++ patterns can trip it up and custom types can sometimes cause issues for ABI generation. For this reason, it's imperative you understand how ABI files work, so you can debug and fix if and when necessary.

What is an ABI? The Application Binary Interface (ABI) is a JSON-based description on how to convert user actions between their JSON and Binary representations. The ABI also describes how to convert the database state to/from JSON. Once you have described your contract via an ABI then developers and users will be able to interact with your contract seamlessly via JSON.

Security Note ABI can be bypassed when executing transactions. Messages and actions passed to a contract do not have to conform to the ABI. The ABI is a guide, not a gatekeeper.

Create an ABI File Start with an empty ABI, name it eosio.token.abi

{ "version": "eosio::abi/1.0", "types": [], "structs": [], "actions": [], "tables": [], "ricardian_clauses": [], "abi_extensions": [], "___comment" : "" } Types An ABI enables any client or interface to interpret and even generate a GUI for your contract. For this to work consistently, describe the custom types that are used as a parameter in any public action or struct that needs to be described in the ABI.

Built-in Types EOSIO implements a number of custom built-ins. Built-in types don't need to be described in an ABI file. If you would like to familiarize yourself with EOSIO's built-ins, they are defined here

{ "new_type_name": "name", "type": "name" } The ABI now looks like this:

{ "version": "eosio::abi/1.1", "types": [{ "new_type_name": "name", "type": "name" }], "structs": [], "actions": [], "tables": [], "ricardian_clauses": [], "abi_extensions": [] } Structs Structs that are exposed to the ABI also need to be described. By looking at eosio.token.hpp, it can be quickly determined which structs are utilized by public actions. This is particularly important for the next step.

A struct's object definition in JSON looks like the following:

{ "name": "issue", //The name "base": "", //Inheritance, parent struct "fields": [] //Array of field objects describing the struct's fields. } Fields { "name":"", // The field's name "type":"" // The field's type }
In the eosio.token contract, there's a number of structs that require definition. Please note, not all of the structs are explicitly defined, some correspond to an actions' parameters. Here's a list of structs that require an ABI description for the eosio.token contract:

Implicit Structs The following structs are implicit in that a struct was never explicitly defined in the contract. Looking at the create action, you'll find two parameters, issuer of type name and maximum_supply of type asset. For brevity this tutorial won't break down every struct, but applying the same logic, you will end up with the following:

create { "name": "create", "base": "", "fields": [{ "name":"issuer", "type":"name" }, { "name":"maximum_supply", "type":"asset" }] } issue { "name": "issue", "base": "", "fields": [{ "name":"to", "type":"name" }, { "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] } retire { "name": "retire",

"base": "", "fields": [{ "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] } transfer { "name": "transfer", "base": "", "fields": [{ "name":"from", "type":"name" }, { "name":"to", "type":"name" }, { "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] } close { "name": "close", "base": "", "fields": [{ "name":"owner", "type":"name" }, { "name":"symbol", "type":"symbol" }] } Explicit Structs These structs are explicitly defined, as they are a requirement to instantiate a multi-index table. Describing them is no different than defining the implicit structs as demonstrated above.

account { "name": "account", "base": "", "fields": [{ "name":"balance", "type":"asset" }] } Actions An action's JSON object definition looks like the following:

{ "name": "transfer", //The name of the action as defined in the contract "type": "transfer", //The name of the implicit struct as described in the ABI "ricardian_contract": "" //An optional ricardian clause to associate to this action describing its intended functionality. } Describe the actions of the eosio.token contract by aggregating all the public functions described in the eosio.token contract's header file.

Then describe each action's type according to its previously described struct. In most situations, the function name and the struct name will be equal, but are not required to be equal.

Below is a list of actions that link to their source code with example JSON provided for how each action would be described.

create { "name": "create", "type": "create", "ricardian_contract": "" } issue { "name": "issue", "type": "issue", "ricardian_contract": "" } retire { "name": "retire", "type": "retire", "ricardian_contract": "" } transfer { "name": "transfer", "type": "transfer", "ricardian_contract": "" } close { "name": "close", "type": "close", "ricardian_contract": "" } Tables Describe the tables. Here's a table's JSON object definition:

{ "name": "", //The name of the table, determined during instantiation. "type": "", //The table's corresponding struct "index_type": "", //The type of primary index of this table "key_names" : [], //An array of key names, length must equal length of key_types member "key_types" : [] //An array of key types that correspond to key names array member, length of array must equal length of key names array. } The eosio.token contract instantiates two tables, accounts and stat.

The accounts table is an i64 index, based on the account struct, has a uint64 as it's primary key

Here's how the accounts table would be described in the ABI

{ "name": "accounts", "type": "account", // Corresponds to previously defined struct "index_type": "i64", "key_names" : ["primary_key"], "key_types" : ["uint64"] } The stat table is an i64 index, based on the currency_stats struct, has a uint64 as it's primary key

Here's how the stat table would be described in the ABI

{ "name": "stat", "type": "currency_stats", "index_type": "i64", "key_names" : ["primary_key"], "key_types" : ["uint64"] } You'll notice the above tables have the same "key name." Naming your keys similar names is symbolic in that it can potentially suggest a subjective relationship. As with this implementation, implying that any given value can be used to query different tables.

Putting it all Together Finally, an ABI file that accurately describes the eosio.token contract.

{ "version": "eosio::abi/1.1", "types": [{ "new_type_name": "name", "type": "name" }], "structs": [{ "name": "create", "base": "", "fields": [{ "name":"issuer", "type":"name" }, { "name":"maximum_supply", "type":"asset" }] }, { "name": "issue", "base": "", "fields": [{ "name":"to", "type":"name" }, { "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] }, { "name": "retire", "base": "", "fields": [{ "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] }, { "name": "close", "base": "", "fields": [{ "name":"owner", "type":"name" }, { "name":"symbol", "type":"symbol" }] }, { "name": "transfer", "base": "", "fields": [{ "name":"from", "type":"name" }, { "name":"to", "type":"name" }, { "name":"quantity", "type":"asset" }, { "name":"memo", "type":"string" }] }, { "name": "account", "base": "", "fields": [{ "name":"balance", "type":"asset" }] }, { "name": "currency_stats", "base": "", "fields": [{ "name":"supply", "type":"asset" }, { "name":"max_supply", "type":"asset" }, { "name":"issuer", "type":"name" }] }], "actions": [{ "name": "transfer", "type": "transfer", "ricardian_contract": "" }, { "name": "issue", "type": "issue", "ricardian_contract": "" }, { "name": "retire", "type": "retire", "ricardian_contract": "" }, { "name": "create", "type": "create", "ricardian_contract": "" }, { "name": "close", "type": "close", "ricardian_contract": "" }], "tables": [{ "name": "accounts", "type": "account", "index_type": "i64", "key_names" : ["currency"], "key_types" : ["uint64"] }, { "name": "stat", "type": "currency_stats", "index_type": "i64", "key_names" : ["currency"], "key_types" : ["uint64"] }], "ricardian_clauses": [], "abi_extensions": [] } Cases not Covered by Token Contract Vectors When describing a vector in your ABI file, simply append the type with [], so if you need to describe a vector of permission levels, you would describe it like so: permission_level[]

Struct Base It's a rarely used property worth mentioning. You can use base ABI struct property to reference another struct for inheritance, as long as that struct is also described in the same ABI file. Base will do nothing or potentially throw an error if your smart contract logic does not support inheritance.

You can see an example of base in use in the system contract source code and ABI

Extra ABI Properties Not Covered Here A few properties of the ABI specification were skipped here for brevity, however, there is a pending ABI specification that will outline every property of the ABI in its entirety.

Ricardian Clauses Ricardian clauses describe the intended outcome of a particular actions. It may also be utilized to establish terms between the sender and the contract.

ABI Extensions A generic "future proofing" layer that allows old clients to skip the parsing of "chunks" of extension data. For now, this property is unused. In the future each extension would have its own "chunk" in that vector so that older clients skip it and newer clients that understand how to interpret it.

Maintenance Every time you change a struct, add a table, add an action or add parameters to an action, use a new type, you will need to remember to update your ABI file. In many cases failure to update your ABI file will not produce any error.

Troubleshooting Table returns no rows Check that your table is accurately described in the ABI file. For example, If you use cleos to add a table on a contract with a malformed ABI definition and then get rows from that table, you will receive an empty result. cleos will not produce an error when adding a row nor reading a row when a contract has failed to properly describe its tables in its ABI.

What's Next? Data Persistence: Learn how data persistence works on EOSIO by writing a simple smart contract that functions as an address book.

2.2.4、Data Persistence

To learn about data persistence, write a simple smart contract that functions as an address book. While this use case isn't very practical as a production smart contract for various reasons, it's a good contract to start with to learn how data persistence works on EOSIO without being distracted by business logic that does not pertain to eosio's multi_index functionality.

Step 1: Create a new directory Earlier, you created a contract directory, navigate there now.

cd CONTRACTS_DIR Create a new directory for our contract and enter the directory

mkdir addressbook cd addressbook Step 2: Create and open a new file touch addressbook.cpp Open the file in your favorite editor.

Step 3: Write an Extended Standard Class and Include EOSIO In a previous tutorial, you created a hello world contract and you learned the basics. You will be familiar with the structure below, the class has been named addressbook respectively.

include

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract
{ public:

private:

}; Step 4: Create The Data Structure for the Table Before a table can be configured and instantiated, a struct that represents the data structure of the address book needs to be written. Since it's an address book, the table will contain people, so create a struct called "person"

struct person {}; When defining the structure of a multi_index table, you will require a unique value to use as the primary key.

For this contract, use a field called "key" with type name. This contract will have one unique entry per user, so this key will be a consistent and guaranteed unique value based on the user's name

struct person { name key; }; Since this contract is an address book it probably should store some relevant details for each entry or person

struct person { name key; std::string first_name; std::string last_name; std::string street; std::string city; std::string state; }; Great. The basic data structure is now complete.

Next, define a primary_key method. Every multi_index struct requires a primary key to be set. Behind the scenes, this method is used according to the index specification of your multi_index instantiation. EOSIO wraps boost::multi_index

Create an method primary_key() and return a struct member, in this case, the key member as previously discussed.

struct person { name key; std::string first_name; std::string last_name; std::string street; std::string city; std::string state;

uint64_t primary_key() const { return key.value;} }; A table's data structure cannot be modified while it has data in it. If you need to make changes to a table's data structure in any way, you first need to remove all its rows

Step 5: Configure the Multi-Index Table Now that the data structure of the table has been defined with a struct we need to configure the table. The eosio::multi_index constructor needs to be named and configured to use the struct we previously defined.

typedef eosio::multi_index<"people"_n, person> address_index; With the above multi_index configuration there is a table named people, that

Uses the _n operator to define an eosio::name type and uses that to name the table. This table contains a number of different singular "persons", so name the table "people". Pass in the singular person struct defined in the previous step. Declare this table's type. This type will be used to instantiate this table later. There are some additional configurations, such as configuring indices, that will be covered further on. So far, our file should look like this.

include

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

private: struct [[eosio::table]] person { name key; std::string first_name; std::string last_name; std::string street; std::string city; std::string state;

```
  uint64_t primary_key() const { return key.value;}
};


typedef eosio::multi_index<"people"_n, person> address_index;
```

}; Step 6: The Constructor When working with C++ classes, the first public method you should create is a constructor.

Our constructor will be responsible for initially setting up the contract.

EOSIO contracts extend the contract class. Initialize our parent contract class with the code name of the contract and the receiver. The important parameter here is the code parameter which is the account on the blockchain that the contract is being deployed to.

addressbook(name receiver, name code, datastream ds):contract(receiver, code, ds) {} Step 7: Adding a record to the table Previously, the primary key of the multi-index table was defined to enforce that this contract will only store one record per user. To make it all work, some assumptions about the design need to be established.

The only account authorized to modify the address book is the user. the primary_key of our table is unique, based on username For usability, the contract should have the ability to both create and modify a table row with a single action. In EOSIO a chain has unique accounts, so name is an ideal candidate as a primary_key in this specific use case. The name type is a uint64_t.

Next, define an action for the user to add or update a record. This action will need to accept any values that this action needs to be able to emplace (create) or modify.

For user-experience and interface simplicity, have a single method be responsible for both creation and modification of rows. Because of this behavior, name it "upsert," a combination of "update" and "insert."

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) {} Earlier, it was mentioned that only the user has control over their own record, as this contract is opt-in. To do this, utilize the require_auth method provided by the eosio.cdt. This method accepts an name type argument and asserts that the account executing the transaction equals the provided value and has the proper permissions to do so.

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); } Previously, a multi_index table was configured, and declared as address_index. To instantiate a table, two parameters are required:

The first parameter "code", which specifies the owner of this table. As the owner, the account will be charged for storage costs. Also, only that account can modify or delete the data in this table unless another payer is specified. Here we use the get_self() function which will pass the name of this contract. The second parameter "scope" which ensures the uniqueness of the table within this contract. In this case, since we only have one table we can use the value from get_first_receiver(). get_first_receiver is the account name this contract is deployed to. Note that scopes are used to logically separate tables within a multi-index (see the eosio.token contract multi-index for an example, which scopes the

table on the token owner). Scopes were originally intended to separate table state in order to allow for parallel computation on the individual sub-tables. However, currently inter-blockchain communication has been prioritized over parallelism. Because of this, scopes are currently only used to logically separate the tables as in the case of eosio.token.

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); } Next, query the iterator, setting it to a variable since this iterator will be used several times

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); } Security has been established and the table instantiated, great! Next up, write the code for creating or modifying the table.

First, detect whether a particular user already exists in the table. To do this, use table's find method by passing the user parameter. The find method will return an iterator. Use that iterator to test it against the end method. The "end" method is an alias for "null".

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { //The user isn't in the table } else { //The user is in the table } } Create a record in the table using the multi_index method emplace. This method accepts two arguments, the "payer" of this record who pays the storage usage and a callback function.

The callback function for the emplace method must use a lamba function to create a reference. Inside the body assign the row's values with the ones provided to upsert.

void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); } else { //The user is in the table } } Next, handle the modification, or update, case of the "upsert" function. Use the modify method, passing a few arguments:

The iterator defined earlier, presently set to the user as declared when calling this action. The "payer", who will pay for the storage cost of this row, in this case, the user. The callback function that actually modifies the row. void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); } else { addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); } } The addressbook contract now has a functional action that will enable a user to create a row in the table if that record does not yet exist, and modify it if it already exists.

But what if the user wants to remove the record entirely?

Step 8: Remove record from the table Similar to the previous steps, create a public method in the addressbook, making sure to include the ABI declarations and a require_auth that tests against the action's argument user to verify only the owner of a record can modify their account.

```
void erase(name user){
  require_auth(user);
}
```

Instantiate the table. In addressbook each account has only one record. Set iterator with find

... void erase(name user){ require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); } ... A contract cannot erase a record that doesn't exist, so check that the record indeed exists before proceeding.

... void erase(name user){ require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); check(iterator != addresses.end(), "Record does not exist"); } ... Finally, call the erase method, to erase the iterator. Once the row is erased, the storage space will be free up for the original payer.

... void erase(name user) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); check(iterator != addresses.end(), "Record does not exist"); addresses.erase(iterator); } ... The contract is now mostly complete. Users can

create, modify and erase records. However, the contract is not quite ready to be compiled.

Step 9: Preparing for the ABI 9.1 ABI Action Declarations eosio.cdt includes an ABI Generator, but for it to work will require some declarations.

Above both the upsert and erase functions add the following C++11 declaration:

[[eosio::action]] The above declaration will extract the arguments of the action and create necessary ABI struct descriptions in the generated ABI file.

9.2 ABI Table Declarations Add an ABI declaration to the table. Modify the following line defined in the private region of your contract:

struct person { To this:

struct [[eosio::table]] person { The [[eosio.table]] declaration will add the necessary descriptions to the ABI file.

Now our contract is ready to be compiled.

Below is the final state of our addressbook contract:

include

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

addressbook(name receiver, name code, datastream ds): contract(receiver, code, ds) {}

[[eosio::action]] void upsert(name user, std::string first_name, std::string last_name, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_self(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); } else { addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); } }

[[eosio::action]] void erase(name user) { require_auth(user);

```
address_index addresses( get_self(), get_first_receiver().value);

auto iterator = addresses.find(user.value);
check(iterator != addresses.end(), "Record does not exist");
addresses.erase(iterator);
```

}

private: struct [[eosio::table]] person { name key; std::string first_name; std::string last_name; std::string street; std::string city; std::string state; uint64_t primary_key() const { return key.value; } }; typedef eosio::multi_index<"people"_n, person> address_index;

};
Step 10 Prepare the Ricardian Clauses [Optional] Contracts compiled without a Ricardian contract will generate a compiler warning for each action missing an entry in the Ricardian clause.

Warning, action does not have a ricardian contract Warning, action does not have a ricardian contract To define Ricardian contracts for this smart contract, create a new file called addressbook.contracts.md. Notice that the name of the Ricardian contracts must match the name of the smart contract.

touch addressbook.contracts.md Add Ricardian Contract definitions to this file:

upsert

spec-version: 0.0.2 title: Upsert summary: This action will either insert or update an entry in the address book. If an entry exists with the same name as the specified user parameter, the record is updated with the first_name, last_name, street, city, and state parameters. If a record does not exist, a new record is created. The data is stored in the multi index table. The ram costs are paid by the smart contract. icon:

erase

spec-version: 0.0.2 title: Erase summary: This action will remove an entry from the address book if an entry in the multi index table exists with the specified

name. icon: Step 11 Prepare the Ricardian Clauses [Optional] To define Ricardian clauses for this smart contract create and open a new file called addressbook.clauses.md. Notice again that the name of the Ricardian clauses must match the name of the smart contract.

touch addressbook.clauses.md Add Ricardian clause definitions to this file:

Data Storage

spec-version: 0.0.1 title: General Data Storage summary: This smart contract will store data added by the user. The user consents to the storage of this data by signing the transaction. icon:

Data Usage

spec-version: 0.0.1 title: General Data Use summary: This smart contract will store user data. The smart contract will not use the stored data for any purpose outside store and delete. icon:

Data Ownership

spec-version: 0.0.1 title: Data Ownership summary: The user of this smart contract verifies that the data is owned by the smart contract, and that the smart contract can use the data in accordance to the terms defined in the Ricardian Contract. icon:

Data Distirbution

spec-version: 0.0.1 title: Data Distirbution summary: The smart contract promises to not actively share or distribute the address data. The user of the smart contract understands that data stored in a multi index table is not private data and can be accessed by any user of the blockchain.
icon:

Data Future

spec-version: 0.0.1 title: Data Future summary: The smart contract promises to only use the data in accordance of the terms defined in the Ricardian Contract, now and at all future dates. icon: Step 12: Compile the Contract Execute the following command from your terminal.

eosio-cpp addressbook.cpp -o addressbook.wasm If you created a Ricardian contract and Ricardian clauses, the definitions will appear in the .abi file. An example for the addressbook.cpp, built including the contract and clause definitions described above is shown below.

{ "____comment": "This file was generated with eosio-abigen. DO NOT EDIT ", "version": "eosio::abi/1.1", "types": [], "structs": [{ "name": "erase", "base": "", "fields": [{ "name": "user", "type": "name" }] }, { "name": "person", "base": "", "fields": [{ "name": "key", "type": "name" }, { "name": "first_name", "type": "string" }, { "name": "last_name", "type": "string" }, { "name": "street", "type": "string" }, { "name": "city", "type": "string" }, { "name": "state", "type": "string" }] }, { "name": "upsert", "base": "", "fields": [{ "name": "user", "type": "name" }, { "name": "first_name", "type": "string" }, { "name": "last_name", "type": "string" }, { "name": "street", "type": "string" }, { "name": "city", "type": "string" }, { "name": "state", "type": "string" }] }], "actions": [{ "name": "erase", "type": "erase", "ricardian_contract": "---\nspec-version: 0.0.2\ntitle: Erase\nsummary: his action will remove an entry from the address book if an entry exists with the same name \nicon:" }, { "name": "upsert", "type": "upsert", "ricardian_contract": "---\nspec-version: 0.0.2\ntitle: Upsert\nsummary: This action will either insert or update an entry in the address book. If an entry exists with the same name as the user parameter the record is updated with the first_name, last_name, street, city and state parameters. If a record does not exist a new record is created. The data is stored in the multi index table. The ram costs are paid by the smart contract.\nicon:" }], "tables": [{ "name": "people", "type": "person", "index_type": "i64", "key_names": [], "key_types": [] }], "ricardian_clauses": [{ "id": "Data Storage", "body": "---\nspec-version: 0.0.1\ntitle: General data Storage\nsummary: This smart contract will store data added by the user. The user verifies they are happy for this data to be stored.\nicon:" }, { "id": "Data Usage", "body": "---\nspec-version: 0.0.1\ntitle: General data Use\nsummary: This smart contract will store user data. The smart contract will not use the stored data for any purpose outside store and delete \nicon:" }, { "id": "Data Ownership", "body": "---\nspec-version: 0.0.1\ntitle: Data Ownership\nsummary: The user of this smart contract verifies that the data is owned by the smart contract, and that the smart contract can use the data in accordance to the terms defined in the Ricardian Contract \nicon:" }, { "id": "Data

Distirbution", "body": "---\nspec-version: 0.0.1\ntitle: Data Ownership\nsummary: The smart contract promises to not actively share or distribute the address data. The user of the smart contract understands that data stored in a multi index table is not private data and can be accessed by any user of the blockchain. \nicon:" }, { "id": "Data Future", "body": "---\nspec-version: 0.0.1\ntitle: Data Ownership\nsummary: The smart contract promises to only use the data in accordance to the terms defined in the Ricardian Contract, now and at all future dates. \nicon:" }], "variants": [] } Step 13: Deploy the Contract Create an account for the contract, execute the following shell command

cleos create account eosio addressbook YOUR_PUBLIC_KEY -p eosio@active Deploy the addressbook contract

cleos set contract addressbook CONTRACTS_DIR/addressbook -p addressbook@active 5f78f9aea400783342b41a989b1b4821ffca006cd76ead38ebdf97428559daa0 5152 bytes 727 us

eosio <= eosio::setcode {"account":"addressbook","vmtype":0,"vmversion":0,"code":"0061736d01000000019101176007f7e7f7f7f7f7f...

eosio <= eosio::setabi {"account":"addressbook","abi":"0e656f73696f3a3a6162692f312e30010c6163636f756e745f6e616d65046e616d65...

warning: transaction executed locally, but may not be confirmed by the network yet] Step 14: Test the Contract Add a row to the table

cleos push action addressbook upsert '["alice", "alice", "liddell", "123 drink me way", "wonderland", "amsterdam"]' -p alice@active executed transaction: 003f787824c7823b2cc8210f34daed592c2cfa66cbbfd4b904308b0dfeb0c811 152 bytes 692 us

addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","street":"123 drink me way","city":"wonde...

Check that alice cannot add records for another user.

cleos push action addressbook upsert '["bob", "bob", "is a loser", "doesnt exist", "somewhere", "someplace"]' -p alice@active As expected, the require_auth in our contract prevented alice from creating/modifying another user's row.

Error 3090004: Missing required authority Ensure that you have the related authority inside your transaction!; If you are currently using 'cleos push action' command, try to add the relevant authority using -p option. Error Details: missing authority of bob Retrieve alice's record.

cleos get table addressbook addressbook people --lower alice --limit 1 { "rows": [{ "key": "alice", "first_name": "alice", "last_name": "liddell", "street": "123 drink me way", "city": "wonderland", "state": "amsterdam" }], "more": false, "next_key": "" } Test to see that alice can remove the record.

cleos push action addressbook erase '["alice"]' -p alice@active executed transaction: 0a690e21f259bb4e37242cdb57d768a49a95e39a83749a02bced652ac4b3f4ed 104 bytes 1623 us

addressbook <= addressbook::erase {"user":"alice"}

warning: transaction executed locally, but may not be confirmed by the network yet] Check that the record was removed:

cleos get table addressbook addressbook people --lower alice --limit 1 { "rows": [], "more": false, "next_key": "" } Looking good!

Wrapping Up You've learned how to configure tables, instantiate tables, create new rows, modify existing rows and work with iterators. You've learned how to test against an empty iterator result. Congrats!

What's Next? Secondary Indices: Learn how to add another index to the addressbook contract that you created in the preceding Data Persistence section.

2.2.5、Secondary Indices

EOSIO has the ability to sort tables by up to 16 indices. In the following section, we're going to add another index to the addressbook contract, so we can iterate through the records in a different way.

Step 1: Remove existing data from table As mentioned earlier, a table's struct cannot be modified when it contains data. This first step allows the removal of the data already added.

Remove all records of alice and bob that were added in previous tutorial.

cleos push action addressbook erase '["alice"]' -p alice@active cleos push action addressbook erase '["bob"]' -p bob@active Step 2: Add new index member and getter Add a new member variable and its getter to the addressbook.cpp contract. Since the secondary index needs to be numeric field, a uint64_t age variable is added.

uint64_t age; uint64_t get_secondary_1() const { return age;} Step 3: Add secondary index to addresses table configuration A field has been defined as the secondary index, next the address_index table needs to be reconfigured.

typedef eosio::multi_index<"people"_n, person, indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>

> address_index; In the third parameter, we pass a index_by struct which is used to instantiate a index.

In that index_by struct, we specify the name of index as "byage" and the second type parameter as a function call operator which extracts a const value as an index key. In this case, we point it to the getter we created earlier so this multiple index table will index records by the age variable.

indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>> Step 4: Modify code With all the changes in previous steps, we can now update the upsert function. Change the function parameter list to the following:

void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) Add additional lines to update age field in upsert function as the following:

void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_first_receiver(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; // -- Add code below -- row.age = age; row.street = street; row.city = city; row.state = state; }); } else { addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; // -- Add code below -- row.age = age; row.street = street; row.city = city; row.state = state; }); } } Step 5: Compile and Deploy Compile

eosio-cpp --abigen addressbook.cpp -o addressbook.wasm Deploy

cleos set contract addressbook CONTRACTS_DIR/addressbook Step 6: Test it Insert records

cleos push action addressbook upsert '["alice", "alice", "liddell", 9, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active cleos push action addressbook upsert '["bob", "bob", "is a guy", 49, "doesnt exist", "somewhere", "someplace"]' -p bob@active Look up alice's address by the age index. Here the --index 2 parameter is used to indicate that the query applies to the secondary index

cleos get table addressbook addressbook people --upper 10 \ --key-type i64 \ --index 2 You should see something like the following

{ "rows": [{ "key": "alice", "first_name": "alice", "last_name": "liddell", "age": 9, "street": "123 drink me way", "city": "wonderland", "state": "amsterdam" }], "more": false, "next_key": "" } Look it up by Bob's age

cleos get table addressbook addressbook people --upper 50 --key-type i64 --index 2 It should return

{ "rows": [{ "key": "alice", "first_name": "alice", "last_name": "liddell", "age": 9, "street": "123 drink me way", "city": "wonderland", "state": "amsterdam" },{ "key": "bob", "first_name": "bob", "last_name": "is a loser", "age": 49, "street": "doesnt exist", "city": "somewhere", "state": "someplace" }], "more": false } Wrapping Up The complete addressbook contract up to this point:

include

include

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

addressbook(name receiver, name code, datastream ds): contract(receiver, code, ds) {}

[[eosio::action]] void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_first_receiver(),get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.age = age; row.street = street; row.city = city; row.state = state; }); } else { addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.age = age; row.street = street; row.city = city; row.state = state; }); } }

[[eosio::action]] void erase(name user) { require_auth(user);

```
address_index addresses(get_self(), get_first_receiver().value);

auto iterator = addresses.find(user.value);
check(iterator != addresses.end(), "Record does not exist");
addresses.erase(iterator);
```

}

private: struct [[eosio::table]] person { name key; std::string first_name; std::string last_name; uint64_t age; std::string street; std::string city; std::string state;

```
uint64_t primary_key() const { return key.value; }
uint64_t get_secondary_1() const { return age; }
```

};

typedef eosio::multi_index<"people"_n, person, indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>> address_index;

}; What's Next? Adding Inline Actions: Learn how to construct actions and send those actions from within a contract.

2.2.6、Adding Inline Actions

Introduction It was previously demonstrated by authoring the addressbook contract the basics of multi-index tables. In this part of the series you'll learn how to construct actions, and send those actions from within a contract.

Step 1: Adding eosio.code to permissions In order for the inline actions to be sent from addressbook, add the eosio.code permission to the contract's account's active permission. Open your terminal and run the following code

cleos set account permission addressbook active --add-code The eosio.code authority is a pseudo authority implemented to enhance security, and enable contracts to execute inline actions.

Step 2: Notify Action If not still opened, open the addressbook.cpp contract authored in the last tutorial. Write an action that dispatches a "transaction receipt" whenever a transaction occurs. To do this, create a helper function in the addressbook class.

[[eosio::action]] void notify(name user, std::string msg) {} This function is very simple, it just accepts a user account as a name type and a message as a string type. The user parameter dictates which user gets the message that is sent.

Step 3: Copy action to sender using require_recipient This transaction needs to be copied to the user so it can be considered as a receipt. To do this, use the require_recipient method. Calling require_recipient adds an account to the require_recipient set and ensures that these accounts receive a notification of the action being executed. The notification is like sending a "carbon copy" of the action to the accounts in the require_recipient set.

[[eosio::action]] void notify(name user, std::string msg) { require_recipient(user); } This action is very simple, however, as written, any user could call this function, and "fake" a receipt from this contract. This could be used in malicious ways, and should be seen as a vulnerability. To correct this, require that the authorization provided in the call to this action is from the contract itself, for this, use get_self

[[eosio::action]] void notify(name user, std::string msg) { require_auth(get_self()); require_recipient(user); } Now if user bob calls this function directly, but passes the parameter alice the action will throw an exception.

Step 4: Notify helper for sending inline transactions Since this inline action will be called several times, write a quick helper for maximum code reuse. In the private region of your contract, define a new method.

... private: void send_summary(name user, std::string message){} Inside of this helper construct an action and send it.

Step 5: The Action Constructor Modify the addressbook contract to send a receipt to the user every time they take an action on the contract.

To begin, address the "create record" case. This is the case that fires when a record is not found in the table, i.e., when iterator == addresses.end() is true.

Save this object to an action variable called notification

... private: void send_summary(name user, std::string
message){ action(//permission_level, //code, //action, //data);
} The action constructor requires a number of parameters.

A permission_level struct The contract to call (initialised using eosio::name type) The action (initialised using eosio::name type) The data to pass to the action, a tuple of positionals that correlate to the actions being called. The Permission struct In this contract the permission should be authorized by the active authority of the contract using get_self(). As a reminder, to use the 'activeauthority inline you will need your contract's to give active authority toeosio.code` pseudo-authority (instructions above)

... private: void send_summary(name user, std::string
message){ action(permission_level{get_self(),"active"_n},); } The "code" AKA "account where contract is deployed" Since the action called is in this contract, use get_self. "addressbook"_n would also work here, but if this contract were deployed under a different account name, it wouldn't work. Because of this, get_self() is the superior option.

... private: void send_summary(name user, std::string
message){ action(permission_level{get_self(),"active"_n}, get_self(), //action
//data); } The action The notify action was previously defined to be called from this inline action. Use the _n operator here.

... private: void send_summary(name user, std::string
message){ action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n,
//data); } The Data Finally, define the data to pass to this action. The notify function accepts two parameters, an name and a string. The action constructor expects data as type bytes, so use make_tuple, a function available through std

C++ library. Data passed in the tuple is positional, and determined by the order of the parameters accepted by the action that being called.

Pass the user variable that is provided as a parameter of the upsert() action. Concatenate a string that includes the name of the user, and include the message to pass to the notify action. ... private: void send_summary(name user, std::string message){ action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n, std::make_tuple(user, name{user}.to_string() + message)); } Send the action. Finally, send the action using the send method of the action struct.

... private: void send_summary(name user, std::string message) { action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n, std::make_tuple(user, name{user}.to_string() + message)).send(); } Step 6: Call the helper and inject relevant messages. Now that the helper is defined, it should probably be called from the relevant locations. There's three specific places for the new notify helper to be called from:

After the contract emplaces a new record: send_summary(user, "successfully emplaced record to addressbook"); After the contract modifies an existing record: send_summary(user, "successfully modified record in addressbook."); After the contract erases an existing record: send_summary(user, "successfully erased record from addressbook"); Step 7: Recompile and Regenerate the ABI File Now that everything is in place, here's the current state of the addressbook contract:

include

include

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

addressbook(name receiver, name code, datastream ds): contract(receiver, code, ds) {}

[[eosio::action]] void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_first_receiver(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.age = age;

row.street = street; row.city = city; row.state = state; }); send_summary(user, " successfully emplaced record to addressbook"); } else { addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.street = street; row.city = city; row.state = state; }); send_summary(user, " successfully modified record to addressbook"); } }

[[eosio::action]] void erase(name user) { require_auth(user);

```
address_index addresses(get_first_receiver(), get_first_receiver().value);

auto iterator = addresses.find(user.value);
check(iterator != addresses.end(), "Record does not exist");
addresses.erase(iterator);
send_summary(user, " successfully erased record from addressbook");
```

}

[[eosio::action]] void notify(name user, std::string msg) { require_auth(get_self()); require_recipient(user); }

private: struct [[eosio::table]] person { name key; std::string first_name; std::string last_name; uint64_t age; std::string street; std::string city; std::string state;

```
uint64_t primary_key() const { return key.value; }
uint64_t get_secondary_1() const { return age;}
```

};

void send_summary(name user, std::string message)
{ action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n, std::make_tuple(user, name{user}.to_string() + message)).send(); };

typedef eosio::multi_index<"people"_n, person, indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>

address_index; }; Open your terminal, and navigate to CONTRACTS_DIR/addressbook

cd CONTRACTS_DIR/addressbook Now, recompile the contract, including the --abigen flag since changes have been made to the contract that affects the ABI. If you've followed the instructions carefully, you shouldn't see any errors.

eosio-cpp -o addressbook.wasm addressbook.cpp --abigen Smart contracts on EOSIO are upgradeable so the contract can be redeployed with changes.

cleos set contract addressbook CONTRACTS_DIR/addressbook Publishing contract... executed transaction: 1898d22d994c97824228b24a1741ca3bd5c7bc2eba9fea8e83446d78bfb264fd 7320 bytes 747 us

eosio <= eosio::setcode {"account":"addressbook","vmtype":0,"vmversion":0,"code":"0061736d01000000001a6011a60027f7e0060077f7e...

eosio <= eosio::setabi {"account":"addressbook","abi":"0e656f73696f3a3a6162692f312e30010c6163636f756e745f6e616d65046e616d65...

Success!

Step 8: Testing it Now that the contract has been modified and deployed, test it. In the previous tutorial, alice's addressbook record was deleted during the testing steps, so calling upsert will fire the inline action just written inside of the "create" case.

Run the following command in your terminal

cleos push action addressbook upsert '["alice", "alice", "liddell", 21, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active cleos will return some data, that includes all the actions executed in the transaction

executed transaction: e9e30524186bb6501cf490ceb744fe50654eb393ce0dd733f3bb6c68ff4b5622 160 bytes 9810 us

addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","age":21,"street":"123 drink me way","cit...

addressbook <= addressbook::notify {"user":"alice","msg":"alicesuccessfully emplaced record to addressbook"}

alice <= addressbook::notify {"user":"alice","msg":"alicesuccessfully emplaced record to addressbook"}

The last entry in the previous log is an addressbook::notify action sent to alice. Use cleos get actions to display actions executed and relevant to alice.

cleos get actions alice

seq when contract::action => receiver trx id... args

==

62 2018-09-15T12:57:09.000 addressbook::notify => alice 685ecc09... {"user":"alice","msg":"alice successfully added record to ad...

What's Next? Inline Actions to External Contracts: Learn how to construct actions and send those actions to an external contract.

2.2.7、Inline Actions to External Contracts

Previously, we sent an inline action to an action that was defined in the contract. In this part of the tutorial, we'll explore sending actions to an external contract. Since we've already gone over quite a bit of contract authoring, we'll keep this contract extremely simple. We'll author a contract that counts actions written by the contract. This contract has very little real-world use, but will demonstrate inline action calls to an external contract

Step 1: The Addressbook Counter Contract Navigate to CONTRACTS_DIR if not already there, create a directory called abcounter and then create a abcounter.cpp file

cd CONTRACTS_DIR mkdir abcounter touch abcounter.cpp Open the abcounter.cpp file in your favorite editor and paste the following code into the file. This contract is very basic, and for the most part does not cover much that we haven't already covered up until this point. There are a few exceptions though, and they are covered in full below.

include

using namespace eosio;

class [[eosio::contract("abcounter")]] abcounter : public eosio::contract { public:

```
abcounter(name receiver, name code,  datastream ds): contract(receiver, code, ds) {}

[[eosio::action]]
void count(name user, std::string type) {
  require_auth( name("addressbook"));
  count_index counts(get_first_receiver(), get_first_receiver().value);
  auto iterator = counts.find(user.value);

  if (iterator == counts.end()) {
    counts.emplace("addressbook"_n, [&]( auto& row ) {
      row.key = user;
      row.emplaced = (type == "emplace") ? 1 : 0;
      row.modified = (type == "modify") ? 1 : 0;
```

```
      row.erased = (type == "erase") ? 1 : 0;
    });
  }
  else {
    counts.modify(iterator, "addressbook"_n, [&]( auto& row ) {
      if(type == "emplace") { row.emplaced += 1; }
      if(type == "modify") { row.modified += 1; }
      if(type == "erase") { row.erased += 1; }
    });
  }
}

using count_action = action_wrapper<"count"_n, &abcounter::count>;
```

private: struct [[eosio::table]] counter { name key; uint64_t emplaced; uint64_t modified; uint64_t erased; uint64_t primary_key() const { return key.value; } };

```
using count_index = eosio::multi_index<"counts"_n, counter>;
```

}; The first new concept in the code above is that we are explicitly restricting calls to the one action to a specific account in this contract using require_auth to the addressbook contract, as seen below.

//Only the addressbook account/contract can authorize this command. require_auth(name("addressbook")); Previously, a dynamic value was used with require_auth.

Another new concept in the code above, is action wrapper. As shown below the first template parameter is the 'action' we are going to call and the second one should point to the action function

using count_action = action_wrapper<"count"_n, &abcounter::count>; Step 2: Create Account for abcounter Contract Open your terminal and execute the following command to create the abcounter user.

cleos create account eosio abcounter YOUR_PUBLIC_KEY Step 3: Compile and Deploy eosio-cpp abcounter.cpp -o abcounter.wasm Finally, deploy the abcounter contract.

cleos set contract abcounter CONTRACTS_DIR/abcounter Step 4: Modify addressbook contract to send inline-action to abcounter Navigate to your addressbook directory now.

cd CONTRACTS_DIR/addressbook Open the addressbook.cpp file in your favorite editor if not already open.

In the last part of this series, we went over inline actions to our own contract. This time, we are going to send an inline action to another contract, our new abcounter contract.

Create another helper called increment_counter under the private declaration of the contract as below:

void increment_counter(name user, std::string type) { abcounter::count_action count("abcounter"_n, {get_self(), "active"_n}); count.send(user, type); } Let's go through the code listing above.

This time we use the action wrapper instead of calling a function. To do that, we firstly initialised the count_action object defined earlier. The first parameter we pass is the callee contract name, in this case abcounter. The second parameter is the permission struct.

For the permission, get_self() returns the current addressbook contract. The active permission of addressbook is used. Unlike the Adding Inline Actions tutorial, we won't need to specify the action because the action wrapper type incorporates the action when it is defined.

In line 3 we call the action with the data, namely user and type which are required by the abcounter contract.

Now, add the following calls to the helpers in their respective action scopes.

//Emplace increment_counter(user, "emplace"); //Modify increment_counter(user, "modify"); //Erase increment_counter(user, "erase"); Now your addressbook.cpp contract should look like this.

include

include "abcounter.cpp"

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

addressbook(name receiver, name code, datastream ds): contract(receiver, code, ds) {}

[[eosio::action]] void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) { require_auth(user); address_index addresses(get_first_receiver(), get_first_receiver().value); auto iterator = addresses.find(user.value); if(iterator == addresses.end()) { addresses.emplace(user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.age = age; row.street = street; row.city = city; row.state = state; }); send_summary(user, " successfully emplaced record to addressbook"); increment_counter(user, "emplace"); } else { std::string changes; addresses.modify(iterator, user, [&](auto& row) { row.key = user; row.first_name = first_name; row.last_name = last_name; row.age = age; row.street = street; row.city = city; row.state = state; }); send_summary(user, " successfully modified record to addressbook"); increment_counter(user, "modify"); } }

[[eosio::action]] void erase(name user) { require_auth(user);

```
address_index addresses(get_first_receiver(), get_first_receiver().value);


auto iterator = addresses.find(user.value);
check(iterator != addresses.end(), "Record does not exist");
addresses.erase(iterator);
send_summary(user, " successfully erased record from addressbook");
increment_counter(user, "erase");
```

}

[[eosio::action]] void notify(name user, std::string msg) { require_auth(get_self()); require_recipient(user); }

private: struct [[eosio::table]] person { name key; uint64_t age; std::string first_name; std::string last_name; std::string street; std::string city; std::string state;

```
uint64_t primary_key() const { return key.value; }
uint64_t get_secondary_1() const { return age;}
```

};

void send_summary(name user, std::string message) { action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n, std::make_tuple(user, name{user}.to_string() + message)).send(); };

void increment_counter(name user, std::string type) { abcounter::count_action count("abcounter"_n, {get_self(), "active"_n}); count.send(user, type); }

typedef eosio::multi_index<"people"_n, person, indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>

> address_index; }; Step 5: Recompile and redeploy the addressbook contract Recompile the addressbook.cpp contract, we don't need to regenerate the ABI, because none of our changes have affected the ABI. Note here we include the abcounter contract folder with the -I option.

eosio-cpp -o addressbook.wasm addressbook.cpp -I ../abcounter/ Redeploy the contract

cleos set contract addressbook CONTRACTS_DIR/addressbook Step 6: Test It. Now that we have the abcounter deployed and addressbook redeployed, we're ready for some testing.

cleos push action addressbook upsert '["alice", "alice", "liddell", 19, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active executed transaction: cc46f20da7fc431124e418ecff90aa882d9ca017a703da78477b381a0246eaf7 152 bytes 1493 us

addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","street":"123 drink me way","city":"wonde...

addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully modified record in addressbook"}

alice <= addressbook::notify {"user":"alice","msg":"alice successfully modified record in addressbook"}

abcounter <= abcounter::count {"user":"alice","type":"modify"}

As you can see, the counter was successfully notified. Let's check the table now.

cleos get table abcounter abcounter counts --lower alice --limit 1 { "rows": [{ "key": "alice", "emplaced": 1, "modified": 0, "erased": 0 }], "more": false } Test each of the actions and check the counter. There's already a row for alice, so upsert should modify the record.

cleos push action addressbook upsert '["alice", "alice", "liddell", 21,"1 there we go", "wonderland", "amsterdam"]' -p alice@active executed transaction: c819ffeade670e3b44a40f09cf4462384d6359b5e44dd211f4367ac6d3ccbc70 152 bytes 909 us

addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","street":"1 coming down","city":"normalla...

addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully emplaced record to addressbook"}

Notified

alice <= addressbook::notify {"user":"alice","msg":"alice successfully emplaced record to addressbook"}

abcounter <= abcounter::count {"user":"alice","type":"emplace"}

warning: transaction executed locally, but may not be confirmed by the network yet] To erase:

cleos push action addressbook erase '["alice"]' -p alice@active executed transaction: aa82577cb1efecf7f2871eac062913218385f6ab2597eaf31a4c0d25ef1bd7df 104 bytes 973 us

addressbook <= addressbook::erase {"user":"alice"}

Erased

addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully erased record from addressbook"}

Notified

alice <= addressbook::notify {"user":"alice","msg":"alice successfully erased record from addressbook"}

abcounter <= abcounter::count {"user":"alice","type":"erase"}

warning: transaction executed locally, but may not be confirmed by the network yet]
Toaster:addressbook sandwich$ Next, we'll test if we can manipulate the data in abcounter contract by calling it directly.

cleos push action abcounter count '["alice", "erase"]' -p alice@active Checking the table in abcounter we'll see the following:

cleos get table abcounter abcounter counts --lower alice { "rows": [{ "key": "alice", "emplaced": 1, "modified": 1, "erased": 1 }], "more": false } Wonderful! Since we require_auth for name("addressbook"), only the addressbook contract can successfully execute this action, the call by alice to fudge the numbers had no affect on the table.

Extra Credit: More Verbose Receipts The following modification sends custom receipts based on changes made, and if no changes are made during a modification, the receipt will reflect this situation.

include

include "abcounter.cpp"

using namespace eosio;

class [[eosio::contract("addressbook")]] addressbook : public eosio::contract {

public:

addressbook(name receiver, name code, datastream ds): contract(receiver, code, ds) {}

[[eosio::action]] void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) { require_auth(user);

```
address_index addresses(get_first_receiver(), get_first_receiver().value);

auto iterator = addresses.find(user.value);
if( iterator == addresses.end() )
{
  addresses.emplace(user, [&]( auto& row ){
   row.key = user;
   row.first_name = first_name;
   row.last_name = last_name;
   row.age = age;
   row.street = street;
   row.city = city;
   row.state = state;
   send_summary(user, " successfully emplaced record to addressbook");
   increment_counter(user, "emplace");
  });
}
else {
  std::string changes;
  addresses.modify(iterator, user, [&]( auto& row ) {

    if(row.first_name != first_name) {
      row.first_name = first_name;
      changes += "first name ";
```

```
    }

    if(row.last_name != last_name) {
      row.last_name = last_name;
      changes += "last name ";
    }

    if(row.age != age) {
      row.age = age;
      changes += "age ";
    }

    if(row.street != street) {
      row.street = street;
      changes += "street ";
    }

    if(row.city != city) {
      row.city = city;
      changes += "city ";
    }

    if(row.state != state) {
      row.state = state;
      changes += "state ";
    }
  });

  if(changes.length() > 0) {
    send_summary(user, " successfully modified record in addressbook. Fields changed: "
 + changes);
    increment_counter(user, "modify");
  } else {
    send_summary(user, " called upsert, but request resulted in no changes.");
  }
}
```

}

[[eosio::action]] void erase(name user) { require_auth(user); address_index addresses(get_first_receiver(), get_first_receiver().value); auto iterator = addresses.find(user.value); check(iterator != addresses.end(), "Record does not exist"); addresses.erase(iterator); send_summary(user, " successfully erased record from addressbook"); increment_counter(user, "erase"); }

[[eosio::action]] void notify(name user, std::string msg) { require_auth(get_self()); require_recipient(user); }

private:

struct [[eosio::table]] person { name key; std::string first_name; std::string last_name; uint64_t age; std::string street; std::string city; std::string state; uint64_t primary_key() const { return key.value; } uint64_t get_secondary_1() const { return age;} };

void send_summary(name user, std::string message)
{ action(permission_level{get_self(),"active"_n}, get_self(), "notify"_n, std::make_tuple(user, name{user}.to_string() + message)).send(); };

void increment_counter(name user, std::string type) {

```
action counter = action(
  permission_level{get_self(),"active"_n},
  "abcounter"_n,
  "count"_n,
  std::make_tuple(user, type)
);

counter.send();
```

}

typedef eosio::multi_index<"people"_n, person, indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>> address_index; }; What's Next? Linking Custom Permissions: Learn how create a custom permission and how to link the permission to an action of a contract.

2.2.8、Creating and Linking Custom Permissions

Introduction On an EOSIO blockchain, you can create various custom permissions for accounts. A custom permission can later be linked to an action of a contract.

This permission system enables smart contracts to have a flexible authorization scheme.

This tutorial illustrates the creation of a custom permission, and subsequently, how to link the permission to an action. Upon completion of the steps, the contract's action will be prohibited from executing unless the authorization of the newly linked permission is provided. This allows you to have greater granularity of control over an account and its various actions.

With great power comes great responsibility. This functionality poses some challenges to the security of your contract and its users. Ensure you understand the concepts and steps prior to putting them to use.

Parent permission When you create a custom permission, the permission will always be created under a parent permission.

If you have the authority of a parent permission which a custom permission was created under, you can always execute an action which requires that custom permission.

Step 1. Create a Custom Permission Firstly, let's create a new permission level on the alice account:

cleos set account permission alice upsert YOUR_PUBLIC_KEY owner -p alice@owner A few things to note:

A new permission called upsert was created The upsert permission uses the development public key as the proof of authority This permission was created on the alice account You can also specify authorities other than a public key for this permission, for example, a set of other accounts. Check account permission for more details.

Step 2. Link Authorization to Your Custom Permission Link the authorization to invoke the upsert action with the newly created permission:

cleos set action permission alice addressbook upsert upsert In this example, we link the authorization to the upsert action created earlier in the addressbook contract.

Step 3. Test it Let's try to invoke the action with an active permission:

cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"]' -p alice@active You should see an error like the one below:

Error 3090005: Irrelevant authority included Please remove the unnecessary authority from your action! Error Details: action declares irrelevant authority '{"actor":"alice","permission":"active"}'; minimum authority is {"actor":"alice","permission":"upsert"} Now, try the upsert permission, this time, explicitly declaring the upsert permission we just created: (e.g. -p alice@upsert)

cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"]' -p alice@upsert Now it works:

cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"] -p alice@upsert executed transaction:

2fe21b1a86ca2a1a72b48cee6bebce9a2c83d30b6c48b16352c70999e4c20983 144 bytes 9489 us

addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddel","age":21,"street":"Herengracht","city":"land",...

addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully modified record to addressbook"}

eosio <= addressbook::notify {"user":"alice","msg":"alice successfully modified record to addressbook"}

abcounter <= abcounter::count {"user":"alice","type":"modify"}

What's Next? Payable Actions: Learn how write a smart contract that has payable actions.

2.2.9、Payable actions

Goal This tutorial illustrates how to write a smart contract that has payable actions. Payable actions are actions that require you to transfer some tokens to actions prior to use other functionality of the smart contract. Also, the EOSIO asset type is covered in this tutorial.

As for the logic of this smart contract, we're going to write a contract that accepts a particular token but will not allow the tokens to be withdrawn for a specific amount of time.

The token to HODL First create a standard C++ class called "hodl" that extends eosio::contract.

include

using namespace eosio;

class [[eosio::contract("hodl")]] hodl : public eosio::contract{ private: public: } This contract needs to set up a few constraints:

What symbol/token does this contract accept? When is the hodl over? Let us now define these constraints as constants:

hodl_symbol: the symbol of tokens this contract accepts. In this case, we use the "SYS" symbol. the_party constant sets the hodl to end on Tuesday, February 22, 2022 10:22:22 PM.

include

using namespace eosio;

class [[eosio::contract("hodl")]] hodl : public eosio::contract { private: static const uint32_t the_party = 1645525342; const symbol hodl_symbol; public:

} Next, let's define a table to track the number of tokens the hodl contract has received.

struct [[eosio::table]] balance { eosio::asset funds; uint64_t primary_key() const { return funds.symbol.raw(); } }; In this multiple index table declaration, a new type called asset is used. An asset is a type designed to represent a digital token asset. See more details in the asset reference documentation.

The symbol member of an asset instance will be used as the primary key. By calling the raw() function the symbol variable will be converted into an unsigned integer so it can be used as a primary key.

Constructor The constructor initializes the hodl_symbol as “SYS”, which is a token created in the Deploy, Issue and Transfer Tokens section.

public: using contract::contract; hodl(name receiver, name code, datastream ds):contract(receiver, code, ds), hodl_symbol("SYS", 4){} Get current UTC time In order to get time in UTC timezone throughout the code, create a function to easily access the current UTC time.

uint32_t now() { return current_time_point().sec_since_epoch(); } Next, we'll write the actions of the contract.

Deposit To accept a transfer we need to have a deposit action.

[[eosio::on_notify("eosio.token::transfer")]] void deposit(name hodler, name to, eosio::asset quantity, std::string memo) { if (to != get_self() || hodler == get_self()) { print("These are not the droids you are looking for."); return; }

check(now() < the_party, "You're way late"); check(quantity.amount > 0, "When pigs fly"); check(quantity.symbol == hodl_symbol, "These are not the droids you are looking for.");

balance_table balance(get_self(), hodler.value); auto hodl_it = balance.find(hodl_symbol.raw());

if (hodl_it != balance.end()) balance.modify(hodl_it, get_self(), [&](auto &row) { row.funds += quantity; }); else balance.emplace(get_self(), [&](auto &row) { row.funds = quantity; }); } This action should not introduce many new concepts if you have followed this tutorial from the beginning.

Firstly, the action checks that the contract is not transferring to itself:

if (to != get_self() || hodler == get_self()) { print("These are not the droids you are looking for."); return; } The contract needs to do so because transferring to the contract account itself would create an invalid booking situation in which an account could have more tokens than the account has in the eosio.token contract.

Then this action checks a few other conditions:

The time to withdraw has not already passed The incoming transfer has a valid amount of tokens The incoming transfer uses the token we specify in the constructor check(now() < the_party, "You're way late"); check(quantity.amount >

0, "When pigs fly"); check(quantity.symbol == hodl_symbol, "These are not the droids you are looking for."); If all constraints are passed, the action updates the balances accordingly:

balance_table balance(get_self(), hodler.value); auto hodl_it = balance.find(hodl_symbol.raw());

if (hodl_it != balance.end()) balance.modify(hodl_it, get_self(), [&](auto &row) { row.funds += quantity; }); else balance.emplace(get_self(), [&](auto &row) { row.funds = quantity; }); The important thing to note is the deposit function will actually be triggered by the eosio.token contract. To understand this behaviour we need to understand the on_notify attribute.

The on_notify attribute [[eosio::on_notify("eosio.token::transfer")]] The on_notify attribute is one of the EOSIO.CDT attributes that annotates a smart contract action.

Annotating an action with an on_notify attribute ensures any incoming notification is forwarded to the annotated action if and only if the notification is dispatched from a specified contract and from a specified action.

In this case, the on_notify attribute ensures the incoming notification is forward to the deposit action only if the notification comes from the eosio.token contract and is from the eosio.token's transfer action.

This is also why we don't need to check if the hodler actually has the appropriate amount of tokens he or she claimed, as the eosio.token contract would have done this check prior to the transfer notification reaching the hodl deposit action.

Party! The party action will only allow withdrawals after the configured the_party time has elapsed. The party action has a similar construct as the deposit action with the following conditions:

check the withdrawing account is the account which made the deposit initially find the locked balance transfer the token on behalf of the account to the account itself [[eosio::action]] void party(name hodler) { //Check the authority of hodler require_auth(hodler);

//Check the current time has passed the the_party time check(now() > the_party, "Hold your horses");

balance_table balance(get_self(), hodler.value); auto hodl_it = balance.find(hodl_symbol.raw());

//Make sure the holder is in the table check(hodl_it != balance.end(), "You're not allowed to party");

action{ permission_level{get_self(), "active"_n}, "eosio.token"_n, "transfer"_n, std::make_tuple(get_self(), hodler, hodl_it->funds, std::string("Party! Your hodl is free.")) }.send();

balance.erase(hodl_it); } The complete code listing is the following:

include

include

include

include

using namespace eosio;

class [[eosio::contract("hodl")]] hodl : public eosio::contract { private: static const uint32_t the_party = 1645525342; const symbol hodl_symbol;

```
struct [[eosio::table]] balance
{
  eosio::asset funds;
  uint64_t primary_key() const { return funds.symbol.raw(); }
};

using balance_table = eosio::multi_index<"balance"_n, balance>;

uint32_t now() {
  return current_time_point().sec_since_epoch();
}
```

public: using contract::contract;

```
hodl(name receiver, name code, datastream ds) : contract(receiver, code, ds),hodl_symbol("SYS", 4){}
```

```
[[eosio::on_notify("eosio.token::transfer")]]
void deposit(name hodler, name to, eosio::asset quantity, std::string memo) {
  if (hodler == get_self() || to != get_self())
  {
    return;
  }

  check(now() < the_party, "You're way late");
  check(quantity.amount > 0, "When pigs fly");
  check(quantity.symbol == hodl_symbol, "These are not the droids you are looking for.");

  balance_table balance(get_self(), hodler.value);
  auto hodl_it = balance.find(hodl_symbol.raw());

  if (hodl_it != balance.end())
    balance.modify(hodl_it, get_self(), [&](auto &row) {
      row.funds += quantity;
    });
  else
    balance.emplace(get_self(), [&](auto &row) {
      row.funds = quantity;
    });
}

[[eosio::action]]
void party(name hodler)
{
  //Check the authority of hodlder
  require_auth(hodler);

  // //Check the current time has pass the the party time
  check(now() > the_party, "Hold your horses");

  balance_table balance(get_self(), hodler.value);
  auto hodl_it = balance.find(hodl_symbol.raw());
```

```
  // //Make sure the holder is in the table
  check(hodl_it != balance.end(), "You're not allowed to party");

  action{
    permission_level{get_self(), "active"_n},
    "eosio.token"_n,
    "transfer"_n,
    std::make_tuple(get_self(), hodler, hodl_it->funds, std::string("Party! Your hodl is free.
"))
  }.send();

  balance.erase(hodl_it);
}
```

}; Great, let's deploy it.

Test deposit First, create an account and deploy to it:

cleos create account eosio hodl YOUR_PUBLIC_KEY eosio-cpp hodl.cpp -o hodl.wasm cleos set contract hodl ./ -p hodl@active As mentioned in a previous tutorial, this contract needs an eosio.code permission:

cleos set account permission hodl active --add-code Next, create a testing account:

cleos create account eosio han DEVELOPMENT_KEY Let's transfer some SYS tokens issued in the previous section to han:

cleos push action eosio.token transfer '["alice", "han", "100.0000 SYS", "Slecht geld verdrijft goed"]' -p alice@active Finally, transfer some SYS tokens to the hodl contract from han's account.

cleos transfer han hodl '0.0001 SYS' 'Hodl!' -p han@active Test withdraw To test the withdrawal feature, the the_party variable needs to be updated. Update the the_party variable to a point in time in the past so the withdrawal functionality can be tested.

CONTRACT hodl : public eosio::contract { private: // 9 June 2018 01:00:00 static const uint32_t the_party = 1528549200; Withdrawing the funds:

cleos push action hodl party '["han"]' -p han@active Should produce the following response:

executed transaction:
62b1e6848c8c5e6458b9a0f7600e65574eaf60445be114d224adccc5a962a09a
104 bytes 383 us

hodl <= hodl::party {"hodler":"han"}

eosio.token <= eosio.token::transfer
{"from":"hodl","to":"han","quantity":"0.
0001 SYS","memo":"Party! Your hodl is
free."}

hodl <= eosio.token::transfer
{"from":"hodl","to":"han","quantity":"0.
0001 SYS","memo":"Party! Your hodl is
free."}

han <= eosio.token::transfer
{"from":"hodl","to":"han","quantity":"0.
0001 SYS","memo":"Party! Your hodl is
free."}

Party time!

What's Next? Elemental Battles: Build a blockchain game based on EOSIO and continue building your EOSIO knowledge!

3、Tutorials

3.1、BIOS Boot Sequence

Note The steps here can be readily expanded for the networked case. Some assumptions are made here regarding how the parties involved will coordinate with each other. However, there are many ways that the community can choose to coordinate. The technical aspects of the process are objective; assumptions of how the coordination might occur are speculative. Several approaches have already been suggested by the community. You are encouraged to review the various approaches and get involved in the discussions as appropriate.

The BIOS Boot sequence undergoes two significant workflows:

Creating, configuring, and starting the genesis node Transitioning from single genesis producer to multiple producers

1. Create, Configure and Start the Genesis Node The information in this section walk you through the preparatory steps for the following:

Setting up your eos environment Starting your genesis eos node Setting up additional, interconnected eos nodes with connectivity to the genesis node After performing these steps, you will have a fully functional eos blockchain running locally.

Python Script

Alternatively, if you would like to automate these steps, you can use the bios-boot-tutorial.py python script that implements the preparatory steps. However, the script uses different and additional data values. See the file accounts.json for the producer names and the user account names that the script uses. If your goal is to build a fully functional EOS blockchain on your local machine by automation, you can run the bios-boot-tutorial.py script directly by following the README.md instructions.

If your goal is to go beyond and understand what the script is doing, you can follow this tutorial which will get you through the same steps explaining also along the way each step needed to go through.

1.1. Install the binaries Pre-compiled EOSIO Binaries

For instructions to install the nodeos binaries, see the Install EOSIO pre-compiled binaries tutorial but do not start nodeos at this stage.

EOSIO.CDT Binaries

For instructions to install the EOSIO.CDT binaries, see the Install EOSIO.CDT binaries tutorial.

1.2. Create a development wallet Create and configure your default wallet, followed by creating a public and private development keys. After the key-pair is created, import

the public and private key in your wallet. For reference purposes, we will refer the public key as EOS_PUB_DEV_KEY and the private key as EOS_PRIV_DEV_KEY.

For instructions on creating a wallet and importing the keys, see the Create development wallet tutorial.

1.3. Create ~/biosboot/genesis directory Create a new directory ~/biosboot/genesis to start the genesis node by executing nodeos with specific parameters that will create the blockchain database, the log file, and the configuration file inside the directory.

cd ~ mkdir biosboot cd biosboot mkdir genesis cd genesis 1.4. Create a JSON file in ~/biosboot/ directory Create an empty genesis.json file in the ~/biosboot/ directory and open it in your preferred text editor (demonstrated with nano editor here): cd ~/biosboot touch genesis.json nano genesis.json Copy the following JSON content to clipboard: { "initial_timestamp": "2018-12-05T08:55:11.000", "initial_key": "EOS_PUB_DEV_KEY", "initial_configuration": { "max_block_net_usage": 1048576, "target_block_net_usage_pct": 1000, "max_transaction_net_usage": 524288, "base_per_transaction_net_usage": 12, "net_usage_leeway": 500, "context_free_discount_net_usage_num": 20, "context_free_discount_net_usage_den": 100, "max_block_cpu_usage": 100000, "target_block_cpu_usage_pct": 500, "max_transaction_cpu_usage": 50000, "min_transaction_cpu_usage": 100, "max_transaction_lifetime": 3600, "deferred_trx_expiration_window": 600, "max_transaction_delay": 3888000, "max_inline_action_size": 4096, "max_inline_action_depth": 4, "max_authority_depth": 6 }, "initial_chain_id": "0000000000000000000000000000000000000000000000000000000000000000" } Paste the JSON content into the genesis.json file. Replace the EOS_PUB_DEV_KEY with the public key you created in 1.2 Create Development Wallet. Save and exit the text editor: [CTRL]+X y [ENTER] 1.5. Start the genesis node To start the genesis node:

Create a genesis_start.sh shell script file in the ~/biosnode/genesis/ directory and open the file with your preferred editor (demonstrated with nano editor here): cd ~/biosboot/genesis touch genesis_start.sh nano genesis_start.sh Copy the following shell script content and paste it to the genesis_start.sh shell script file.

!/bin/bash

DATADIR="./blockchain"

if [! -d $DATADIR]; then mkdir -p $DATADIR; fi

nodeos \ --genesis-json $DATADIR"/../../genesis.json" \ --signature-provider EOS_PUB_DEV_KEY=KEY:EOS_PRIV_DEV_KEY \ --plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ -

-plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir $DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir $DATADIR"/config" \ --producer-name eosio \ --http-server-address 127.0.0.1:8888 \ --p2p-listen-endpoint 127.0.0.1:9010 \ --access-control-allow-origin=* \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-stale-production \ --p2p-peer-address localhost:9011 \ --p2p-peer-address localhost:9012 \ --p2p-peer-address localhost:9013 \

> $DATADIR"/nodeos.log" 2>&1 & \ echo $! > $DATADIR"/eosd.pid" NOTE: Replace the EOS_PUB_DEV_KEY and EOS_PRIV_DEV_KEY with the public and private key values you generated in step 1.2 Create a development wallet.

Save and exit the text editor: [CTRL]+X y [ENTER] Assign execution privileges to the genesis_start.sh shell script file and then execute the genesis_start.sh script to start genesis nodeos: cd ~/biosboot/genesis/ chmod 755 genesis_start.sh ./genesis_start.sh The Genesis node:

Bears the name eosio Produces blocks Listens for HTTP request on 127.0.0.1:8888 Listens for peer connections requests on 127.0.0.1:9010 Initiates periodic peer connections to localhost:9011, localhost:9012, and localhost:9013; these nodes are not running yet so ignore if you see any failed connection attempts Has the parameter --contracts-console which prints contracts output to the console; in our case, this information is good for troubleshooting problems 1.5.1 Stopping the Genesis node To stop nodeos:

Create a stop.sh shell script file in the ~/biosnode/genesis/ directory and copy the following stop.sh script to it.

!/bin/bash

DATADIR="./blockchain/"

if [-f $DATADIR"/eosd.pid"]; then pid=cat $DATADIR"/eosd.pid" echo $pid kill $pid rm -r $DATADIR"/eosd.pid" echo -ne "Stoping Node" while true; do [! -d "/proc/$pid/fd"] && break echo -ne "." sleep 1 done echo -ne "\rNode Stopped. \n" fi Execute the stop.sh shell script from the same ~/biosboot/genesis/ directory: cd ~/biosboot/genesis/ chmod 755 stop.sh ./stop.sh 1.5.2 Restarting nodeos After stopping the nodeos process, you will not be able to restart it using the .genesis_start.sh script created in 1.5 Start the genesis node as once a node runs and produces blocks, the blockchain database initializes and gets populated. Thus, nodeos is not able to start with the --genesis-json parameter. Therefore, it is recommended to create a new script, start.sh by following the same steps outlined in 1.5 Start a genesis node and copy the

below content to the script. Also, assign execution privileges to the script and use this file for any future nodeos restarts after you stopped the process.

```
!/bin/bash

DATADIR="./blockchain"

if [ ! -d $DATADIR ]; then mkdir -p $DATADIR; fi

nodeos \ --signature-provider EOS_PUB_DEV_KEY=KEY:EOS_PRIV_DEV_KEY \
--plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin
eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ -
-plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir
$DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir
$DATADIR"/config" \ --producer-name eosio \ --http-server-address 127.0.0.1:8888
\ --p2p-listen-endpoint 127.0.0.1:9010 \ --access-control-allow-origin=* \ --
contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-
stale-production \ --p2p-peer-address localhost:9011 \ --p2p-peer-address
localhost:9012 \ --p2p-peer-address localhost:9013 \

        $DATADIR"/nodeos.log" 2>&1 & \ echo $! >
        $DATADIR"/eosd.pid" Troubleshooting nodeos Restart
        Errors
```

"perhaps we need to replay": This error can occur when you restart nodeos due to a missing --hard-replay parameter which replays all the transactions from the genesis node. To overcome this error, add the parameter --hard-replay in the hard_replay.sh shell script. Some other parameters that you can use to restart nodeos are:

--truncate-at-block --delete-all-blocks --replay-blockchain --hard-replay-blockchain The following is the hard_replay.sh shell script which is using the --hard-replay-blockchain parameter:

```
!/bin/bash

DATADIR="./blockchain"

if [ ! -d $DATADIR ]; then mkdir -p $DATADIR; fi

nodeos \ --signature-provider EOS_PUB_DEV_KEY=KEY:EOS_PRIV_DEV_KEY \
--plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin
eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ -
-plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir
```

$DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir $DATADIR"/config" \ --producer-name eosio \ --http-server-address 127.0.0.1:8888 \ --p2p-listen-endpoint 127.0.0.1:9010 \ --access-control-allow-origin=* \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-stale-production \ --p2p-peer-address localhost:9011 \ --p2p-peer-address localhost:9012 \ --p2p-peer-address localhost:9013 \ --hard-replay-blockchain \

$DATADIR"/nodeos.log" 2>&1 & \ echo $! > $DATADIR"/eosd.pid" Restarting nodeos from scratch

Copy the below content and create a shell script clean.sh and give execution permission to it:

!/bin/bash

rm -fr blockchain ls -al If you want to erase the current configuration, the blockchain data, configuration, and logs, first run the stop.sh script and after that run the clean.sh script which you'll have to create from below content:

cd ~/biosboot/genesis/ ./stop.sh ./clean.sh ./genesis_start.sh 1.6. Inspect the nodeos.log file Inspect the nodeos.log file with the following command, and use CTRL+C to exit the listing mode.

cd ~/biosboot/genesis/ tail -f ./blockchain/nodeos.log 1.7. Create important system accounts There are several system accounts that are needed, namely the following:

eosio.bpay eosio.msig eosio.names eosio.ram eosio.ramfee eosio.saving eosio.stake eosio.token eosio.vpay eosio.rex Repeat the following steps to create an account for each of the system accounts. In this tutorial, we will use the same key pair for both the account owner and active keys, so we only need to provide the key value once on the command line. For most general accounts, it is a good practice to use separate keys for owner and active. The script uses the same key for all of the eosio.* accounts. You can use different keys for each.

cleos create key --to-console Private key: 5KAVVPzPZnbAx8dHz6UWVPFDVFtU1P5ncUzwHGQFuTxnEbdHJL4 Public key: EOS84BLRbGbFahNJEpnnJHYCoW9QPbQEk2iHsHGGS6qcVUq9HhutG cleos wallet import --private-key 5KAVVPzPZnbAx8dHz6UWVPFDVFtU1P5ncUzwHGQFuTxnEbdHJL4 imported private key for: EOS84BLRbGbFahNJEpnnJHYCoW9QPbQEk2iHsHGGS6qcVUq9HhutG cleos create account eosio eosio.bpay EOS84BLRbGbFahNJEpnnJHYCoW9QPbQEk2iHsHGGS6qcVUq9HhutG executed transaction:

ca68bb3e931898cdd3c72d6efe373ce26e6845fc486b42bc5d185643ea7a90b1 200 bytes 280 us

eosio <= eosio::newaccount {"creator":"eosio","name":"eosio.bpay","owner":{"threshold":1,"keys":[{"key":"EOS84BLRbGbFahNJEpnnJH...

1.8. Build eosio.contracts In order to build eosio.contracts, create a dedicated directory for eosio.contracts, clone the eosio.contracts sources and build them. Print the current directory in the terminal and make a note of it. The current directory will be referred to as EOSIO_CONTRACTS_DIRECTORY.

cd ~ git clone https://github.com/EOSIO/eosio.contracts.git
cd ./eosio.contracts/ ./build.sh cd ./build/contracts/ pwd You will also need an older version of eosio.contracts, specifically v1.8.0. Follow the instructions below to build it and remember the path where it is built:

To install eosio.cdt version 1.6.3 binaries, see the Install eosio.cdt binaries tutorial. After the eosio.cdt 1.6.3 version is installed, you can compile the older version of eosio.contracts: cd ~ git clone https://github.com/EOSIO/eosio.contracts.git eosio.contracts-1.8.x cd ./eosio.contracts-1.8.x/ git checkout release/1.8.x ./build.sh cd ./build/contracts/ pwd Make note of the printed local path, we will reference to this directory as EOSIO_OLD_CONTRACTS_DIRECTORY from here onward when needed.

Restore the eosio.cdt version installed at the beginning of the tutorial. 1.9. Install the eosio.token contract Now we have to set the eosio.token contract. This contract enables you to create, issue, transfer, and get information about tokens. To set the eosio.token contract:

cleos set contract eosio.token EOSIO_CONTRACTS_DIRECTORY/eosio.token/ Output:

Reading WAST/WASM from /users/documents/eos/contracts/eosio.token/eosio.token.wasm... Using already assembled WASM... Publishing contract... executed transaction: 17fa4e06ed0b2f52cadae2cd61dee8fb3d89d3e46d5b133333816a04d23ba991 8024 bytes 974 us

eosio <= eosio::setcode {"account":"eosio.token","vmtype":0,"vmversion":0,"code":"0061736d0100000017f1560037f7e7f0060057f7e...

eosio <= eosio::setabi {"account":"eosio.token","abi":{"types":[],"structs":[{"name":"transfer","base":"","fields":[{"name"...

1.10. Set the eosio.msig contract The eosio.msig contract enables and simplifies defining and managing permission levels and performing multi-signature actions. To set the eosio.msig contract:

cleos set contract eosio.msig EOSIO_CONTRACTS_DIRECTORY/eosio.msig/ Output:

Reading WAST/WASM from /users/documents/eos/build/contracts/eosio.msig/eosio.msig.wasm... Using already assembled WASM... Publishing contract... executed transaction: 007507ad01de884377009d7dcf409bc41634e38da2feb6a117ceced8554a75bc 8840 bytes 925 us

eosio <= eosio::setcode {"account":"eosio.msig","vmtype":0,"vmversion":0,"code":"0061736d0100000001980111760017f0060047f7e7e7...

eosio <= eosio::setabi {"account":"eosio.msig","abi":{"types":[{"ne

w_type_name":"account_name","type":"name"}],"structs":[{...

1.11. Create and allocate the SYS currency Create the SYS currency with a maximum value of 10 billion tokens. Then, issue one billion tokens. Replace SYS with your specific currency designation.

In the first step, the create action from the eosio.token contract, authorized by the eosio.token account, creates 1B SYS tokens in the eosio account. This effectively creates the maximum supply of tokens, but does not put any tokens into circulation. Tokens not in circulation can be considered to be held in reserve. cleos push action eosio.token create '["eosio", "10000000000.0000 SYS"]' -p eosio.token@active Output:

executed transaction:
0440461e0d8816b4a8fd9d47c1a6a53536d3c7af54abf53eace884f008429697 120 bytes 326 us

eosio.token <= eosio.token::create {"issuer":"eosio","maximum_supply":"10000000000.0000 SYS"}

In the second step, the eosio.token contract's issue action takes 1B SYS tokens out of reserve and puts them into circulation. At the time of issue, the tokens are held within the eosio account. Since the eosio account owns the reserve of uncirculated tokens, its authority is required to do the action. cleos push action eosio.token issue '["eosio", "1000000000.0000 SYS", "memo"]' -p eosio@active Output:

executed transaction:
a53961a566c1faa95531efb422cd952611b17d728edac833c9a55582425f98ed 128 bytes 432 us

eosio.token <= eosio.token::issue {"to":"eosio","quantity":"1000000000.0000 SYS","memo":"memo"}

Note As a point of interest, from an economic point of view, moving token from reserve into circulation, such as by issuing tokens, is an inflationary action. Issuing tokens is just one way that inflation can occur.

1.12. Set the eosio.system contract Activate the PREACTIVATE_FEATURE protocol

All of the protocol upgrade features introduced in v1.8 and v2.0 first require a special protocol feature (codenamed PREACTIVATE_FEATURE) to be activated and for an updated version of the system contract that makes use of the functionality introduced by that feature to be deployed.

To activate the special protocol PREACTIVATE_FEATURE:

curl --request POST \ --url http://127.0.0.1:8888/v1/producer/schedule_protocol_feature_activations \ -d '{"protocol_features_to_activate": ["0ec7e080177b2c02b278d5088611686b49d739925a92d9bfcacd7fc6b74053bd"]}' Set the eosio.system contract

A system contract provides the actions for all token-based operational behavior. Prior to installing the system contract, actions are done independently of accounting. Once the system contract is enabled, actions now have an economic element to them. System Resources (CPU, network, memory) must be paid for and likewise, new accounts must be paid for. The system contract enables tokens to be staked and unstaked, resources to be purchased, potential producers to be registered and subsequently voted on, producer rewards to be claimed, privileges and limits to be set, and more.

In the first phase, we will install the older version of the eosio.system contract.

cleos set contract eosio EOSIO_OLD_CONTRACTS_DIRECTORY/eosio.system/ Reading WAST/WASM from /users/documents/eos/build/contracts/eosio.system/eosio.system.wasm... Using already assembled WASM... Publishing contract... executed transaction: 2150ed87e4564cd3fe98ccdea841dc9ff67351f9315b6384084e8572a35887cc 39968 bytes 4395 us

eosio <= eosio::setcode {"account":"eosio","vmtype":0,"vmversion":0,"code":"0061736d0100000001be023060027f7e0060067f7e7e7f7f...

eosio <= eosio::setabi {"account":"eosio","abi":{"types":[],"structs":[{"name":"buyrambytes","base":"","fields":[{"name":"p...

Enable Features

After you set the eosio.system contract, run the following commands to enable the rest of the features which are highly recommended to be enabled for an EOSIO-based blockchain.

NOTE: Enabling these features are optional. You can choose to enable or continue without these features.

GET_SENDER

cleos push action eosio activate '["f0af56d2c5a48d60a4a5b5c903edfb7db3a736a94ed589d0b797df33ff9d3e1d"]' -p eosio

FORWARD_SETCODE

cleos push action eosio activate '["2652f5f96006294109b3dd0bbde63693f55324af452b799ee137a81a905eed25"]' -p eosio

ONLY_BILL_FIRST_AUTHORIZER

cleos push action eosio activate '["8ba52fe7a3956c5cd3a656a3174b931d3bb2abb45578befc59f283ecd816a405"]' -p eosio

RESTRICT_ACTION_TO_SELF

cleos push action eosio activate '["ad9e3d8f650687709fd68f4b90b41f7d825a365b02c23a636cef88ac2ac00c43"]' -p eosio

DISALLOW_EMPTY_PRODUCER_SCHEDULE

cleos push action eosio activate '["68dcaa34c0517d19666e6b33add67351d8c5f69e999ca1e37931bc410a297428"]' -p eosio

FIX_LINKAUTH_RESTRICTION

cleos push action eosio activate '["e0fb64b1085cc5538970158d05a009c24e276fb94e1a0bf6a528b48fbc4ff526"]' -p eosio

REPLACE_DEFERRED

cleos push action eosio activate '["ef43112c6543b88db2283a2e077278c315ae2c84719a8b25f25cc88565fbea99"]' -p eosio

NO_DUPLICATE_DEFERRED_ID

cleos push action eosio activate '["4a90c00d55454dc5b059055ca213579c6ea856967712a56017487886a4d4cc0f"]' -p eosio

ONLY_LINK_TO_EXISTING_PERMISSION

cleos push action eosio activate '["1a99a59d87e06e09ec5b028a9cbb7749b4a5ad8819004365d02dc4379a8b7241"]' -p eosio

RAM_RESTRICTIONS

cleos push action eosio activate '["4e7bf348da00a945489b2a681749eb56f5de00b900014e137ddae39f48f69d67"]' -p eosio

WEBAUTHN_KEY

cleos push action eosio activate '["4fca8bd82bbd181e714e283f83e1b45d95ca5af40fb89ad3977b653c448f78c2"]' -p eosio

WTMSIG_BLOCK_SIGNATURES

cleos push action eosio activate '["299dcb6af692324b899b39f16d5a530a33062804e41f09dc97e9f156b4476707"]' -p eosio Deploy

Now deploy the latest version of the eosio.system contract:

cleos set contract eosio EOSIO_CONTRACTS_DIRECTORY/eosio.system/

1. Transition from single genesis producer to multiple producers In the next set of steps, we will transition from a single block producer (the genesis node) to multiple producers. Up to this point, only the built-in eosio account is privileged and can sign blocks. The target is to manage the blockchain by a collection of elected producers, operating under a rule of 2/3 + 1 producers agreeing before a block is final.

Producers are chosen by election. The list of producers can change. Rather than giving privileged authority directly to any producer, the governing rules are associated with a special built-in account named eosio.prods. This account represents the group of elected producers. The eosio.prods account (effectively the producer group) operates using permissions defined by the eosio.msig contract.

As soon as possible after installing the eosio.system contract, we want to designate eosio.msig as a privileged account so that it can authorize on behalf of the eosio account. As soon as possible, eosio will resign its authority and eosio.prods will take over.

2.1. Designate eosio.msig as privileged account To designate eosio.msig as a privileged account:

cleos push action eosio setpriv '["eosio.msig", 1]' -p eosio@active 2.2. Initialize system account To initialize the system account with code zero (needed at initialization time) and SYS token with precision 4; precision can range from [0 .. 18]:

cleos push action eosio init '["0", "4,SYS"]' -p eosio@active 2.3. Stake tokens and expand the network If you've followed the tutorial steps above to this point, you now have a single host, single-node configuration with the following contracts installed:

eosio.token eosio.msig eosio.system The accounts eosio and eosio.msig are privileged accounts. The other eosio.* accounts are created but are not privileged.

We are now ready to begin staking accounts and expanding the network of producers.

2.4. Create staked accounts Staking is the process of allocating tokens acquired by an entity in the "real world" (e.g., an individual purchasing something at a Crowdsale or some other means) to an account within the EOSIO system. Staking and unstaking are an on-going process throughout the life of a blockchain. The initial staking done during the bios boot process is special. During the bios boot sequence, accounts are staked with their tokens. However, until producers are elected, tokens are effectively in a frozen state. Thus, the goal of the initial staking done during the bios boot sequence is to get tokens allocated to their accounts and ready for use, and get the voting process going so that producers can get elected and the blockchain is running "live".

The following recommendation is given for the initial staking process:

0.1 token (literally, not 10% of the account's tokens) is staked for RAM. By default, cleos stakes 8 KB of RAM on account creation, paid by the account creator. In the initial staking, the eosio account is the account creator doing the staking. Tokens staked during the initial token staking process cannot be unstaked and made liquid until after the minimum voting requirements have been met. 0.45 token is staked for CPU, and 0.45 token is staked for network. The next available tokens up to 9 total are held as liquid tokens. Remaining tokens are staked 50/50 CPU and network. Example 1. accountnum11 has 100 SYS. It will be staked as 0.1000 SYS on RAM; 45.4500 SYS on CPU; 45.4500 SYS on network; and 9.0000 SYS held for liquid use.

Example 2. accountnum33 has 5 SYS. It will be staked as 0.1000 SYS on RAM; 0.4500 SYS on CPU; 0.4500 SYS on network; and 4.0000 SYS held for liquid use. To make the tutorial more realistic, we distribute the 1B tokens to accounts using a Pareto distribution. The Pareto distribution models an 80-20 rule, e.g., in this case, 80% of the tokens are held by 20% of the population. The examples here do not show how to generate the distribution, focusing instead on the commands to do the staking. The script bios-boot-tutorial.py that accompanies this tutorial uses the Python NumPy (numpy) library to generate a Pareto distribution.

Use the following steps to stake tokens for each account. These steps must be done individually for each account.

Note The key pair is created here for this tutorial. In a "live" scenario, the key value(s) and token share for an account should already be established through some well-defined out-of-band process.

```
$ cleos create key --to-console
```

Private key: 5K7EYY3j1YY14TSFVfqgtbWbrw3FA8BUUnSyFGgwHi8Uy61wU1o
Public key: EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt

cleos wallet import --private-key 5K7EYY3j1YY14TSFVfqgtbWbrw3FA8BUUnSyFGgwHi8Uy61wU1o imported private key for: EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt Create a staked account with initial resources and public key.

cleos system newaccount eosio --transfer accountnum11 EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt --stake-net "100000000.0000 SYS" --stake-cpu "100000000.0000 SYS" --buy-ram-kbytes 8192 775292ms thread-0 main.cpp:419 create_action] result: {"binargs":"0000000000ea30551082d4334f4d113200200000"} arg: {"code":"eosio","action":"buyrambytes","args":{"payer":"eosio","receiver":"accountnum11","bytes":8192}} 775295ms thread-0 main.cpp:419 create_action] result: {"binargs":"0000000000ea30551082d4334f4d113200ca9a3b00000000045359530000000000ca9a3b00000000045359530000000001"} arg: {"code":"eosio","action":"delegatebw","args":{"from":"eosio","receiver":"accountnum11","stake_net_quantity":"100000.0000 SYS","stake_cpu_quantity":"100000.0000 SYS","transfer":true}} executed transaction: fb47254c316e736a26873cce1290cdafff07718f04335ea4faa4cb2e58c9982a 336 bytes 1799 us

eosio <= eosio::newaccount {"creator":"eosio","name":"accountnum11","owner":{"threshold":1,"keys":[{"key":"EOS8mUftJXepGzdQ2TaC...

eosio <= eosio::buyrambytes {"payer":"eosio","receiver":"accountnum11","bytes":8192}

eosio <= eosio::delegatebw {"from":"eosio","receiver":"accountnum11",

"stake_net_quantity":"100000.0000 SYS","stake_cpu_quantity...

2.5. Register the new account as a producer To register the new account as a producer:

cleos system regproducer accountnum11 EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt https://accountnum11.com EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt 1487984ms thread-0 main.cpp:419 create_action] result: {"binargs":"1082d4334f4d11320003fedd01e019c7e91cb07c724c614bbf644a36eff83a861b36723f29ec81dc9bdb4e68747470733a2f2f6163636f756e746e756d31312e636f6d2f454f53386d5566744a586570477a64513254614364754e7553504166584a4866323275657834753431616231455676394541685774000 0"} arg: {"code":"eosio","action":"regproducer","args":{"producer":"accountnum11","producer_key":"EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt","url":"https://accountnum11.com/EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt","location":0}} executed transaction: 4ebe9258bdf1d9ac8ad3821f6fcdc730823810a345c18509ac41f7ef9b278e0c 216 bytes 896 us

eosio <= eosio::regproducer {"producer":"accountnum11","producer_key":"EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt","u...

This makes the node a candidate to be a producer, but the node will not actually be a producer unless it is elected, that is, voted for.

2.6. List the producers To facilitate the voting process, list the available producers. At this point, you will see only one account registered as a producer.

To list the producers:

cleos system listproducers Producer Producer key Url Scaled votes accountnum11 EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt https://accountnum11.com/EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22 0.0000 2.7. Set up and start a new producer We will set up now a new producer using the previously

created accountnum11 account. To set up the new producer, execute these steps to create a dedicated folder for it:

cd ~/netbios/ mkdir accountnum11 cd accountnum11 copy ~/netbios/genesis/stop.sh copy ~/netbios/genesis/clean.sh Create the following three shell script files and assign execution permission to them: genesis_start.sh, start.sh, hard_start.sh.

!/bin/bash

DATADIR="./blockchain" CURDIRNAME=${PWD##*/}

if [! -d $DATADIR]; then mkdir -p $DATADIR; fi

nodeos \ --genesis-json $DATADIR"/../../genesis.json" \ --signature-provider EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt=KEY:5K7EYY3j1YY14TSFVfqgtbWbrw3FA8BUUnSyFGgwHi8Uy61wU1o \ --plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ --plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir $DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir $DATADIR"/config" \ --producer-name $CURDIRNAME \ --http-server-address 127.0.0.1:8011 \ --p2p-listen-endpoint 127.0.0.1:9011 \ --access-control-allow-origin=* \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-stale-production \ --p2p-peer-address localhost:9010 \ --p2p-peer-address localhost:9012 \ --p2p-peer-address localhost:9013 \

$DATADIR"/nodeos.log" 2>&1 & \ echo $! > $DATADIR"/eosd.pid"

!/bin/bash

DATADIR="./blockchain" CURDIRNAME=${PWD##*/}

if [! -d $DATADIR]; then mkdir -p $DATADIR; fi

nodeos \ --signature-provider EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt=KEY:5K7EYY3j1YY14TSFVfqgtbWbrw3FA8BUUnSyFGgwHi8Uy61wU1o \ --plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ --plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir $DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir $DATADIR"/config" \ --producer-name $CURDIRNAME \ --http-server-address

127.0.0.1:8011 \ --p2p-listen-endpoint 127.0.0.1:9011 \ --access-control-allow-origin=* \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-stale-production \ --p2p-peer-address localhost:9010 \ --p2p-peer-address localhost:9012 \ --p2p-peer-address localhost:9013 \

$DATADIR"/nodeos.log" 2>&1 & \ echo $! > $DATADIR"/eosd.pid"

!/bin/bash

DATADIR="./blockchain" CURDIRNAME=${PWD##*/}

if [! -d $DATADIR]; then mkdir -p $DATADIR; fi

nodeos \ --signature-provider EOS8mUftJXepGzdQ2TaCduNuSPAfXJHf22uex4u41ab1EVv9EAhWt=KEY:5K7EYY3j1YY14TSFVfqgtbWbrw3FA8BUUnSyFGgwHi8Uy61wU1o \ --plugin eosio::producer_plugin \ --plugin eosio::producer_api_plugin \ --plugin eosio::chain_plugin \ --plugin eosio::chain_api_plugin \ --plugin eosio::http_plugin \ --plugin eosio::history_api_plugin \ --plugin eosio::history_plugin \ --data-dir $DATADIR"/data" \ --blocks-dir $DATADIR"/blocks" \ --config-dir $DATADIR"/config" \ --producer-name $CURDIRNAME \ --http-server-address 127.0.0.1:8011 \ --p2p-listen-endpoint 127.0.0.1:9011 \ --access-control-allow-origin=* \ --contracts-console \ --http-validate-host=false \ --verbose-http-errors \ --enable-stale-production \ --p2p-peer-address localhost:9010 \ --p2p-peer-address localhost:9012 \ --p2p-peer-address localhost:9013 \ --hard-replay-blockchain \

$DATADIR"/nodeos.log" 2>&1 & \ echo $! > $DATADIR"/eosd.pid" If you executed every step without an error, your folder structure should look like this:

cd ~/biosboot/accountnum11/ ls -al drwxr-xr-x 8 owner group 256 Dec 7 14:17 . drwxr-xr-x 3 owner group 960 Dec 5 10:00 .. -rwxr-xr-x 1 owner group 40 Dec 5 13:08 clean.sh -rwxr-xr-x 1 owner group 947 Dec 5 14:31 genesis_start.sh -rwxr-xr-x 1 owner group 888 Dec 5 13:08 hard_start.sh -rwxr-xr-x 1 owner group 901 Dec 6 15:44 start.sh -rwxr-xr-x 1 owner group 281 Dec 5 13:08 stop.sh You are now ready to start the second producer node by executing the following commands:

cd ~/biosboot/accountnum11/ ./genesis_start.sh tail -f blockchain/nodeos.log After executing the above commands, you should see in the command shell a live stream of nodeos.log file which is getting written to by the nodeos continuously. You can stop the live stream monitor by pressing CTRL+C keys.

To stop the new node, you have to execute the stop.sh script and to restart the node, execute the start.sh script and not the genesis_start.sh (this one is used only once in 1.5 Start the genesis node).

To erase everything and start from scratch, you can execute the following set of commands:

cd ~/biosboot/accountnum11/ ./stop.sh ./clean.sh ./genesis_start.sh tail -f blockchain/nodeos.log 2.8. Repeat the process for creating multiple producers You can now repeat the process (starting from 2.4. till 2.7) for creating as many producers as you want each with its own staked account, own dedicated directory, named accountnumXY (with X and Y int values in interval [1..5]), and their own dedicated script files: genesis_start.sh, start.sh, stop.sh, clean.sh located in their corresponding folder.

Also, be aware of how you mesh these nodes between each other, so pay particular attention to the following parameters in the genesis_start.sh, start.sh and hard_start.sh scripts:

--producer-name $CURDIRNAME \ # Producer name, set in the script to be the parent directory name ... --http-server-address 127.0.0.1:8011 \ # http listening port for API incoming requests --p2p-listen-endpoint 127.0.0.1:9011 \ # p2p listening port for incoming connection requests ... ... --p2p-peer-address localhost:9010 \ # Meshing with peer genesis node --p2p-peer-address localhost:9012 \ # Meshing with peer accountnum12 node --p2p-peer-address localhost:9013 . # Meshing with peer accountnum13 node 2.9. Vote for each of the block producers started At this point the nodes are started, meshed together in a network, and they receive blocks from genesis node but they do not produce.

15% Requirement

For the nodes to produce blocks, a total of 15% of the token supply must be staked and then voted for all available producers. We gave accountnum11 enough tokens earlier. To elect block producers, execute the following command which allows one account to vote for as up to 30 block producers identified by their account name:

cleos system voteproducer prods accountnum11 accountnum11 accountnum12 accountnum13

1. Resign eosio account and system accounts Once producers have been elected and the minimum number requirements have been met, that is, a minimum 15% of tokens have been staked to produce votes, the eosio account can resign, leaving the eosio.msig account as the only privileged account.

Resigning involves setting the keys of the eosio. *accounts to null. Use the following command to clear the eosio.* accounts' owner and active keys:

cleos push action eosio updateauth '{"account": "eosio", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio.prods", "permission": "active"}}]}}' -p eosio@owner cleos push action eosio updateauth '{"account": "eosio", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio.prods", "permission": "active"}}]}}' -p eosio@active Also, the system accounts created in step 1.7. Create important system accounts should be resigned as well by running the following commands:

cleos push action eosio updateauth '{"account": "eosio.bpay", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.bpay@owner cleos push action eosio updateauth '{"account": "eosio.bpay", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.bpay@active

cleos push action eosio updateauth '{"account": "eosio.msig", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.msig@owner cleos push action eosio updateauth '{"account": "eosio.msig", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.msig@active

cleos push action eosio updateauth '{"account": "eosio.names", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.names@owner cleos push action eosio updateauth '{"account": "eosio.names", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.names@active

cleos push action eosio updateauth '{"account": "eosio.ram", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.ram@owner cleos push action eosio updateauth '{"account": "eosio.ram", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.ram@active

cleos push action eosio updateauth '{"account": "eosio.ramfee", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.ramfee@owner cleos push action eosio updateauth '{"account": "eosio.ramfee", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.ramfee@active

cleos push action eosio updateauth '{"account": "eosio.saving", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.saving@owner cleos push action eosio updateauth '{"account": "eosio.saving", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.saving@active

cleos push action eosio updateauth '{"account": "eosio.stake", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.stake@owner cleos push action eosio updateauth '{"account": "eosio.stake", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.stake@active

cleos push action eosio updateauth '{"account": "eosio.token", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.token@owner cleos push action eosio updateauth '{"account": "eosio.token", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.token@active

cleos push action eosio updateauth '{"account": "eosio.vpay", "permission": "owner", "parent": "", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.vpay@owner cleos push action eosio updateauth '{"account": "eosio.vpay", "permission": "active", "parent": "owner", "auth": {"threshold": 1, "keys": [], "waits": [], "accounts": [{"weight": 1, "permission": {"actor": "eosio", "permission": "active"}}]}}' -p eosio.vpay@active

1. Monitor, test, monitor You can monitor each nodeos started (either the genesis node or any of the block producers nodes) by:

cd ~/biosboot/genesis/ tail -f ./blockchain/nodeos.log cd ~/biosboot/accountnum11/ tail -f ./blockchain/nodeos.log You can test various commands, create accounts, check balance on accounts, transfer tokens between accounts, etc.

For commands on creating new accounts, see the Create test accounts tutorial.

For commands on issuing, allocating and transferring token between accounts, see the [Deploy, Issue and Transfer Tokensgetting-started/smart-contract-development/deploy-issue-and-transfer-tokens) tutorial.

4、Protocol

4.1、Consensus Protocol

1. Overview An EOSIO blockchain is a highly efficient, deterministic, distributed state machine that can operate in a decentralized fashion. The blockchain keeps track of transactions within a sequence of interchanged blocks. Each block cryptographically commits to the previous blocks along the same chain. It is therefore intractable to modify a transaction recorded on a given block without breaking the cryptographic checks of successive blocks. This simple fact makes blockchain transactions immutable and secure.

1.1. Block Producers In the EOSIO ecosystem, block production and block validation are performed by special nodes called "block producers". Producers are elected by EOSIO stakeholders (see 4. Producer Voting/Scheduling). Each producer runs an instance of an EOSIO node through the nodeos service. For this reason, producers that are on the active schedule to produce blocks are also called "active" or "producing" nodes.

1.2. The Need for Consensus Block validation presents a challenge among any group of distributed nodes. A consensus model must be in place to validate such blocks in a fault tolerant way within the decentralized system. Consensus is the way for such distributed nodes and users to agree upon the current state of the blockchain (see 3. EOSIO Consensus (DPoS + aBFT)).

1. Consensus Models There are various ways to reach consensus among a group of distributed parties in a decentralized system. Most consensus models reach agreement through some proof. Two of the most popular ones are Proof of Work (PoW) and Proof of Stake (PoS), although other types of proof-based schemes exist, such as Proof of Activity (a hybrid between PoW and PoS), Proof of Burn, Proof of Capacity, Proof of Elapsed Time, etc. Other consensus schemes also exist, such as Paxos and Raft. This document focuses mainly on the EOSIO consensus model.

2.1. Proof of Work (PoW) Two of the most common consensus models used in blockchains are Proof of Work and Proof of Stake. In Proof of Work, miner nodes compete to find a nonce added to the header of a block which causes the block to have some desired property (typically a certain number of zeros in the most significant bits of the cryptographic hash of the block header). By making it computationally expensive to find such nonces that make the blocks valid, it becomes difficult for attackers to create an alternative fork of the blockchain that would be accepted by the rest of the network as the best chain. The main disadvantage of Proof of Work is that the security of the network depends on spending a lot of resources on computing power to find the nonces.

2.2. Proof of Stake (PoS) In Proof-of-Stake, nodes that own the largest stake or percentage of some asset have equivalent decision power. In other words, voting power is proportional to the stake held. One interesting variant is Delegated Proof-of-Stake (DPoS) in which a large number of participants or stakeholders elect a smaller number of delegates, which in turn make decisions for them.

1. EOSIO Consensus (DPoS + aBFT) EOSIO-based blockchains use delegated proof of stake (DPoS) to elect the active producers who will be authorized to sign valid blocks in the network. However, this is only one half of the EOSIO consensus process. The other half is involved in the actual process of confirming each block until it becomes final (irreversible), which is performed in an asynchronous byzantine fault tolerant (aBFT) way. Therefore, there are two layers involved in the EOSIO consensus model:

Layer 1 - The Native Consensus Model (aBFT). Layer 2 - Delegated Proof of Stake (DPoS). The actual native consensus model used in EOSIO has no concept of delegations/voting, stake, or even tokens. These are used by the DPoS layer to generate the first schedule of block producers and, if applicable, update the set at most every schedule round after each producer has cycled through. These two layers are functionally separate in the EOSIO software.

3.1. Layer 1: Native Consensus (aBFT) This layer ultimately decides which blocks, received and synced among the elected producers, eventually become final, and hence permanently recorded in the blockchain. It gets a schedule of producers proposed by the second layer (see 3.2. Layer 2: Delegated PoS) and uses that schedule to determine which blocks are correctly signed by the appropriate producer. For byzantine fault tolerance, the layer uses a two-stage block confirmation process by which a two-thirds supermajority of producers from the current scheduled set confirm each block twice. The first confirmation stage proposes a last irreversible block (LIB). The second stage confirms the proposed LIB as final. At this point, the block becomes irreversible. This layer is also used to signal producer schedule changes, if any, at the beginning of every schedule round.

3.1.1. EOSIO Algorithmic Finality The EOSIO consensus model achieves algorithmic finality (differing from the merely probabilistic finality that at best can be achieved in Proof of Work models) through the signatures from the chosen set of special participants (active producers) that are arranged in a schedule to determine which party is authorized to sign the block at a particular time slot. Changes to this schedule can be initiated by privileged smart contracts running on the EOSIO blockchain, but any initiated changes to the schedule do not take effect until after the block that initiated the schedule change has been finalized by two stages of confirmations. Each stage of confirmations is performed by a supermajority of producers from the current scheduled set of active producers.

3.2. Layer 2: Delegated PoS (DPoS) The Delegated PoS layer introduces the concepts of tokens, staking, voting/proxying, vote decay, vote tallying, producer ranking, and inflation pay. This layer is also in charge of generating new producer schedules from the rankings generated from producer voting. This occurs in schedule rounds of approximately two minutes (126 seconds) which is the period it takes for a block producer to be assigned a timeslot to produce and sign blocks. The timeslot lasts a total

of 6 seconds per producer, which is the producer round, where a maximum of 12 blocks can be produced and signed. The DPoS layer is enabled by WASM smart contracts.

3.2.1. Stakeholders and Delegates The actual selection of the active producers (the producer schedule) is open for voting every schedule round and it involves all EOSIO stakeholders who exercise their right to participate. In practice, the rankings of the active producers do not change often, though. The stakeholders are regular EOSIO account holders who vote for their block producers of preference to act on their behalf as DPoS delegates. A major departure from regular DPoS, however, is that once elected, all block producers have equal power regardless of the ranking of votes obtained. In other DPoS models, voting power is proportional to the number of votes obtained by each delegate.

3.3. The Consensus Process The EOSIO consensus process consists of two parts:

Producer voting/scheduling - performed by the the DPoS layer 2 Block production/validation - performed by the native consensus layer 1 These two processes are independent and can be executed in parallel, except for the very first schedule round after the boot sequence when the blockchain’ s first genesis block is created.

1. Producer Voting/Scheduling The voting of the active producers to be included in the next schedule is implemented by the DPoS layer. Strictly speaking, a token holder must first stake some tokens to become a stakeholder and thus be able to vote with a given staking power.

4.1. Voting Process Each EOSIO stakeholder can vote for up to 30 block producers in one voting action. The top 21 elected producers will then act as DPoS delegates to produce and sign blocks on behalf of the stakeholders. The remaining producers are placed in a standby list in the order of votes obtained. The voting process repeats every schedule round by adding up the number of votes obtained by each producer. Producers not voted on get to keep their old votes, albeit depreciated due to vote decay. Producers voted on also get to keep their old votes, except for the contribution of the last voting weight for each voter, which gets replaced by their new voting weight.

4.1.1. Voting Weight The voting weight of each stakeholder is computed as a function of the number of tokens staked and the time elapsed since the EOSIO block timestamp epoch, defined as January 1, 2000. In the current implementation, the voting weight is directly proportional to the number of tokens staked and base-2 exponentially proportional to the time elapsed in years since the year 2000. The actual weight increases at a rate of 2^{1/52} = 1.0134192 1/52 =1.013419 per week. This means that the voting weight changes weekly and doubles each year for the same amount of tokens staked.

4.1.2. Vote Decay Increasing the voting weight produces depreciation of the current votes held by each producer. Such vote decay is intentional and its reason is twofold:

Encourage participation by allowing newer votes to have more weight than older votes. Give more voice to those users actively involved on important governance matters. 4.2. Producers schedule After the producers are voted on and selected for the next schedule, they are simply sorted alphabetically by producer name. This determines the production order. Each producer receives the proposed set of producers for the next schedule round within the very first block to be validated from the current schedule round that is about to start. When the first block that contains the proposed schedule is deemed irreversible by a supermajority of producers plus one, the proposed schedule becomes active for the next schedule round.

4.2.1. Production Parameters The EOSIO block production schedule is divided equally among the elected producers. The producers are scheduled to produce an expected number of blocks each schedule round, based on the following parameters (per schedule round):

Parameter Description Default Layer P (producers) number of active producers 21 2 Bp (blocks/producer) number of contiguous blocks per producer 12 1 Tb (s/block) Production time per block (s: seconds) 0.5 1 It is important to mention that Bp (number of contiguous blocks per producer), and Tb (production time per block) are layer 1 consensus constants. In contrast, P (number of active producers) is a layer 2 constant configured by the DPoS layer, which is enabled by WASM contracts.

The following variables can be defined from the above parameters (per schedule round):

Variable Description Equation B (blocks) Total number of blocks Bp (blocks/producer) x P (producers) Tp (s/producer) Production time per producer Tb (s/block) x Bp (blocks/producer) T (s) Total production time Tp (s/producer) x P (producers) Therefore, the value of P, being defined at layer 2, can change dynamically in an EOSIO blockchain. In practice, however, N is strategically set to 21 producers, which means that 15 producers are required for a two-thirds supermajority of producers plus one to reach consensus.

4.2.2. Production Default Values With the current defaults: P=21 elected producers, Bp=12 blocks created per producer, and a block produced every T=0.5 seconds, current production times are as follows (per schedule round):

Variable Value Tp: Production time per producer Tp = 0.5 (s/block) x 12 (blocks/producer) ⇒ Tp = 6 (s/producer) T: Total production time T = 6 (s/producer) x 21 (producers) ⇒ T = 126 (s) When a block is not produced by a given producer during its assigned time slot, a gap results in the blockchain. If the producer continues to not produce blocks beyond a given timeout (set by a layer 2 constant), the producer is demoted to a standby list.

1. Block Lifecycle Blocks are created by the active producer on schedule during its assigned timeslot, then relayed to other producer nodes for syncing and validation. This process continues from producer to producer until a new schedule of producers is approved at a later schedule round. When a valid block meets the consensus requirements (see 3. EOSIO Consensus), the block becomes final and is considered irreversible. Therefore, blocks undergo three major phases during their lifespan: production, validation, and finality. Each phase goes through various stages as well.

5.1. Block Structure As an inter-chained sequence of blocks, the fundamental unit within the blockchain is the block. A block contains records of pre-validated transactions and additional cryptographic overhead such as hashes and signatures necessary for block confirmation, re-execution of transactions during validation, blockchain replays, protection against replay attacks, etc. (see block schema below).

block schema Name Type Description timestamp block_timestamp_type block creation time (before any transactions are included) producer name account name for producer of this block confirmed uint16_t non-zero if block confirmed by producer of this block previous block_id_type block ID of the previous block transaction_mroot checksum256_type merkle tree root hash of transaction receipts action_mroot checksum256_type merkle tree root hash of action receipts schedule_version uint32_t increments when new producer schedule is confirmed for next schedule round (must be 0; field moved to new_producers) new_producers producer_schedule_type holds schedule version, producer names and keys for new producer schedule (included in first block of schedule round) header_extensions extensions_type extends header fields to support additional features producer_signature signature_type digital signature from producer that created and signed block transactions array of transaction_receipt list of valid transaction receipts included in block block_extensions extension_type extends header fields to support additional features id uint64_t UUID of this block ID (a function of block header and block number); can be used to query a transaction within the block block_num uint32_t block number (sequential counter value since genesis block 0) ref_block_prefix uint32_t lower 32 bits of id; used to prevent replay attacks Some of the block fields are known in advance when the block is created, so they are added during block initialization. Others are computed and added during block finalization, such as the merkle root hashes for transactions and actions, the block number and block ID, the signature of the producer that created and signed the block, etc. (see Network Peer Protocol: 3.1. Block ID)

5.2. Block Production During each schedule round of block production, the producer on schedule must create Bp=12 contiguous blocks containing as many validated transactions as possible. Each block is currently produced within a span of Tb=500 ms (0.5 s). To guarantee sufficient time to produce each block and transmit to other nodes for validation, the block production time is further divided into two configurable parameters:

maximum processing interval: time window to push transactions into the block (currently set at 200 ms). minimum propagation time: time window to propagate blocks to other nodes (currently set at 300 ms). All loose transactions that have not expired yet, or dropped as a result of a previous failed validation, are kept in a local queue for both block inclusion and syncing with other nodes. During block production, the scheduled transactions are applied and validated by the producer on schedule, and if valid, pushed to the pending block within the processing interval. If the transaction falls outside this window, it is unapplied and rescheduled for inclusion in the next block. If there are no more block slots available for the current producer, the transaction is picked up eventually by another producing node (via the peer-to-peer protocol) and pushed to another block. The maximum processing interval is slightly less for the last block (from the producer round of Bp blocks) to compensate for network latencies during handoff to the next producer. By the end of the processing interval, no more transactions are allowed in the pending block, and the block goes through a finalization step before it gets broadcasted to other block producers for validation.

Blocks go through various stages during production: apply, finalize, sign, and commit.

5.2.1. Apply Block Apply block essentially pushes the transactions received and validated by the producing node into a block. Internally, this step involves the creation and initialization of the block header and the signed block instance. The signed block instance simply extends the block header with a signature field. This field eventually holds the signature of the producer that signs the block. Furthermore, recent changes in EOSIO allow multiple signatures to be included, which are stored in a header extensions field.

5.2.2. Finalize Block Produced blocks need to be finalized before they can be signed, committed, relayed, and validated. During finalization, any field in the block header that is necessary for cryptographic validation is computed and stored in the block. This includes generating both merkle tree root hashes for the list of action receipts and the list of transaction receipts pushed to the block.

5.2.3. Sign Block After the transactions have been pushed into the block and the block is finalized, the block is ready to be signed by the producer. This involves computing a signature digest from the serialized contents of the block header, which includes the transaction receipts included in the block. After the block is signed with the producer's private key, the signature digest is added to the signed block instance. This completes the block signing.

5.2.4. Commit Block After the block is signed, it is committed to the local chain. This pushes the block to the reversible block database (see Network Peer Protocol: 2.2.1. Fork Database). This makes the block available for syncing with other nodes for validation (see the Network Peer Protocol for more information about block syncing).

5.3. Block Validation Block validation is a fundamental operation necessary to reach consensus within an EOSIO blockchain. During block validation, producers receive incoming blocks from other peers and confirm the transactions included within each block. Block validation is about reaching enough quorum among active producers to agree upon:

The integrity of the block and the transactions it contains. The deterministic, chronological order of transactions within each block. The first step towards validating a block begins when a block is received by a node. At this point, some safety checks are performed on the block. If the block does not link to an already known block or it matches the block ID of any block already received and processed by the node, the block is discarded. If the block is new, it is pushed to the chain controller for processing.

5.3.1. Push Block When the block is received by the chain controller, the software must determine where to add the block within the local chain. The fork database, or Fork DB for short, is used for this purpose. The fork database holds all the branches with reversible blocks that have been received but are not yet finalized. To that end, the following steps are performed:

Add block to the fork database. If block is added to the main branch that contains the current head block, apply block (see 5.2.1. Apply Block); or If block must be added to a different branch, then:

if that branch now becomes the preferred branch compared to the current main branch: rewind all blocks up to the nearest common ancestor (and rollback the database state in the process), re-apply all blocks in the different branch, add the new block and apply it. That branch now becomes the new main branch. otherwise: add the new block to that branch in the fork database but do nothing else. In order for the block to be added to fork database, some block validation must occur. Block header validation must always be done before adding a block to the fork database. And if the block must be applied, some validation of the transactions within the block must occur. The degree to which transactions are validated depends on the validation mode that nodeos is configured with. Two block validation modes are supported: full validation (the default mode), and light validation.

5.3.2. Full Validation In full validation mode, every transaction that is applied is fully validated. This includes verifying the signatures on the transaction and checking authorizations.

5.3.3. Light Validation In light validation mode, blocks signed by trusted producers (which can be configured locally per node) can skip some of the transaction validation done during full validation. For example, signature verification is skipped and all claimed authorizations on actions are assumed to be valid.

5.4. Block Finality Block finality is the final outcome of EOSIO consensus. It is achieved after a supermajority of active producers have validated the block according to the consensus rules (see 3.1. Layer 1: Native Consensus (aBFT)). Blocks that reach finality are permanently recorded in the blockchain and cannot be undone. In this regard, the last irreversible block (LIB) in the chain refers to the most recent block that has become final. Therefore, from that point backwards the transactions that have been recorded on the blockchain cannot be reversed, tampered, or erased.

5.4.1. Goal of Finality The main point of finality is to give users confidence that transactions that were applied prior and up to the LIB block cannot be modified, rolled back, or dropped. The LIB block can also be useful for active nodes to determine quickly and efficiently which branch to build off from, regardless of which is the longest one. This is because a given branch might be longer without containing the most recent LIB, in which case a shorter branch with the most recent LIB must be selected.

4.2、Transactions Protocol

1. Overview Actions define atomic behaviors within a smart contract. At a higher level, transactions define groups of actions that execute atomically within a decentralized application. Analogously to a database transaction, the group of actions that form a blockchain transaction must all succeed, one by one, in a predefined order, or else the transaction will fail. To maintain transaction atomicity and integrity in case of a failed transaction, the blockchain state is restored to a state consistent with the state prior to processing the transaction. This guarantees that no side effects arise from any actions executed prior to the point of failure.

1.1. Actions An action can be authorized by one or more actors previously created on the blockchain. Actions can be created explicitly within a smart contract, or generated implicitly by application code. For any given actor:action pair there is at most one explicit associated minimum permission. If there are no explicit minimum permissions set, the implicit default is actor@active. Each actor can independently set their personal minimum permission for a given action. Also, a complex but flexible authorization structure is in place within the EOSIO software to allow actors to push actions on behalf of other accounts. Thus, further checks are enforced to authorize an actor to send an action (see 3.4.2. Permission Check).

There are two types of actions involved in a transaction. They mainly differ in the way they are executed by the EOSIO software:

Explicit actions, which are present in a signed transaction (see 2. Transaction Instance). Implicit (inline) actions, which are created as a side effect of processing a transaction. Implicit (inline) actions are also defined in smart contract code, just like explicit actions.

The key difference is that inline actions are not included in the actual transactions propagated through the network and eventually included in a block; they are implicit.

1.1.1. Explicit Actions Regular or explicit actions, as their name implies, are included in the actual list of actions that form a transaction. Explicit actions are encoded as action instances (see 3.4.3. Action Instance) before being pushed into the transaction. Explicit actions also contain the actual payload data, if any, associated with the action to be executed as part of the transaction.

1.1.2. Implicit Actions An implicit (inline) action is generated as a result of an explicit caller action within a transaction (or another inline action, if nested) that requires that implicit action to perform an operation for the caller action to continue. As such, inline actions work within the same scope and permissions of the caller action. Therefore, inline actions are guaranteed to execute within the same transaction.

1.2. Smart Contracts In EOSIO, smart contracts consist of a set of actions, usually grouped by functionality, and a set of type definitions which those actions depend on. Therefore, actions specify and define the actual behaviors of the contract. Several actions are implemented in the standard EOSIO contracts for account creation, producer voting, token operations, etc. Application developers can extend, replace, or disable this functionality altogether by creating custom actions within their own smart contracts and applications. Transactions, on the other hand, are typically created at the application level. Smart contracts are agnostic to them.

1.2.1. Implementation An EOSIO smart contract is implemented as a C++ class that derives from eosio::contract. Actions are implemented as C++ methods within the derived class. Transactions, on the other hand, are generated dynamically (as transaction instances) within an EOSIO application. The EOSIO software processes each transaction instance within a block and keeps track of its state as it evolves from creation, signing, validation, and execution.

1. Transaction Instance A transaction instance consists of a transaction header and the list of action instances and transaction extensions that make the actual transaction. The transaction header includes information necessary to assess the inclusion of the transaction in a block based on its expiration time, which is computed when the transaction is pushed for execution. Other fields include the block number that includes the transaction, a block ID prefix used to prevent "cross chain" or "cross fork" attacks, upper limits for CPU and network usage, and the number of seconds to delay the transaction, if applicable. The diagram below depicts a transaction instance.

Transaction Instance action list Transaction Header Transaction Extensions action n action 1 action 2 ... The action instances may consist of regular actions or context free actions. Signatures are created and validated at the transaction level. Accounts and permissions are handled on a per action basis. Each action instance contains

information to validate whether it is authorized to be executed based on the permission levels of the actors specified in the action and the actual authorizations defined in the smart contract for that action (see 3.4.2. Permission Check).

2.1. Transaction ID A transaction instance contains the minimum set of fields that distinguish one transaction from another. Consequently, a transaction ID consists of a cryptographic hash of the basic fields included in a transaction instance. Therefore, the transaction ID is solely determined by the list of actions encapsulated within the transaction, the transaction header, and any embedded transaction extensions, which are optional. The transaction instance can be further specialized into a signed transaction instance or a packed transaction instance.

2.2. Signed Transaction Instance A signed transaction extends the basic contents of the transaction schema to include the signature(s) generated by the account(s) that signed the transaction. It also includes any data associated with the context free actions, if any, that were included in the transaction instance (see signed_transaction schema below). A transaction is not ready for execution and validation unless it is signed by the applicable actors.

signed_transaction schema Name Type Description expiration time_point_sec time transaction must be confirmed by ref_block_num uint16_t block number in last 216 blocks ref_block_prefix uint32_t lower 32 bits of block ID max_net_usage_words unsigned_int upper limit on total network bandwidth billed (in 64-bit words) max_cpu_usage_ms uint8_t upper limit on total CPU time billed (in miliseconds) delay_sec unsigned_int number of seconds to delay transaction for context_free_actions array of action list of context-free actions if any actions array of action list of action instances transaction_extensions extensions_type extends fields to support additional features signatures array of signature_type digital signatures after transaction is signed context_free_data array of bytes context-free action data to send if any 2.3. Packed Transaction Instance A packed transaction is an optionally compressed signed transaction with additional housekeeping fields to allow for decompression and quick validation. Packed transactions minimize space footprint and block size in the long run (see packed_transaction schema below). A packed transaction forms the most generic type of transaction in an EOSIO blockchain. Consequently, when transactions are pushed to a block, they are actually packed transactions whether compressed or not.

packed_transaction schema Name Type Description signatures signature_type digital signatures after transaction is signed compression compression_type compression method used packed_context_free_data bytes compressed context-free data (if transaction compressed) packed_trx bytes compressed transaction (if compressed) unpacked_trx signed_transaction cached ecompressed transaction trx_id transaction_id_type transaction ID The unpacked_trx field holds the cached unpacked transaction after the transaction instance is constructed. If the signed transaction was previously compressed, it is decompressed from the packed_trx field and cached to unpacked_trx. If the signed transaction was stored uncompressed, it is simply copied

verbatim to unpacked_trx. The signatures field allows a quick signature validatio of the transaction without requiring a full decompression of the transaction.

1. Transaction Lifecycle Transactions go through various stages during their lifespan. First, a transaction is created in an application or an EOSIO client such as cleos by pushing the associated actions into the transaction. Next, the transaction is sent to the locally connected node, which in turn relays it to the active producing nodes for validation and execution via the peer-to-peer network. Next, the validated transaction is pushed to a block by the active producer on schedule along with other transactions. Finally the block that contains the transaction is pushed to all other nodes for validation. When a supermajority of producers have validated the block, and the block becomes irreversible, the transaction gets permanently recorded in the blockchain and it is considered immutable.

3.1. Create Transaction Transactions are created within an application by instantiating a transaction object and pushing the related action instances into a list within the transaction instance. An action instance contains the actual details about the receiver account to whom the action is intended, the name of the action, the list of actors and permission levels that must authorize the transaction via signatures and delays, and the actual message to be sent, if any (see action schema below).

action schema Name Type Description account name encoded 13-char account name action_name name encoded 13-char action name authorization array of permission_level list of actor:permission authorizations data bytes action data to send After the transaction instance is created at the application level, the transaction is arranged for processing. This involves two main steps: signing the transaction and pushing the signed transaction to the local node for actual propagation and execution of the transaction. These steps are typically performed within the EOSIO application.

3.2. Sign Transaction The transaction must be signed by a set of keys sufficient to satisfy the accumulated set of explicit actor:permission pairs specified in all the actions enclosed within the transaction. This linkage is done through the authority table for the given permission (see Accounts and Permissions: 3. Permissions). The actual signing key is obtained by querying the wallet associated with the signing account on the client where the application is run.

The transaction signing process takes three parameters: the transaction instance to sign, the set of public keys from which the associated private keys within the application wallet are retrieved, and the chain ID. The chain ID identifies the actual EOSIO blockchain and consists of a hash of its genesis state, which depends on the blockchain’s initial configuration parameters. Before signing the transaction, the EOSIO software first computes a digest of the transaction. The digest value is a SHA-256 hash of the chain ID, the transaction instance, and the context free data if the transaction has any context free actions. Any instance fields get serialized before

computing any cryptographic hashes to avoid including reference fields (memory addresses) in the hash computation. The transaction digest computation and the signing process are depicted below.

Wallet Manager Signed Transaction signature(s) Transaction SHA-256 Chain ID Transaction Digest Sign Signing account(s) Public Key(s) Signing account(s) Wallet Signing account(s) Private Key(s) After the transaction digest is computed, the digest is finally signed with the private key associated with the signing account' s public key. The public-private key pair is usually stored within the local machine that connects to the local node. The signing process is performed within the wallet manager associated with the signing account, which is typically the same user that deploys the application. The wallet manager provides a virtual secure enclave to perform the digital signing, so a message signature is generated without the private key ever leaving the wallet. After the signature is generated, it is finally added to the signed transaction instance.

3.3. Push Transaction After the transaction is signed, a packed transaction instance is created from the signed transaction instance and pushed from the application to the local node, which in turn relays the transaction to the active producing nodes for signature verification, execution, and validation. Every producing node that receives a transaction will attempt to execute and validate it in their local context before relaying it to the next producing node. Hence, valid transactions are relayed while invalid ones are dropped. The idea behind this is to prevent bad actors from spamming the network with bogus transactions. The expectation is for bad transactions to get filtered and dropped before reaching the active producer on schedule. When a transaction is received, no assumption is made on its validity. All transactions are validated again by the next producing node, regardless of whether it is producing blocks. The only difference is that the producer on schedule attempts to produce blocks by pushing the transactions it validates into a pending block before pushing the finalized block to its own local chain and relaying it to other nodes.

3.4. Verify Transaction The process to verify a transaction is twofold. First, the public keys associated with the accounts that signed the transaction are recovered from the set of signatures provided in the transaction. Such a recovery is cryptographically possible for ECDSA, the elliptic curve digital signature algorithm used in EOSIO. Second, the public key of each actor specified in the list of action authorizations (actor:permission) from each action included in the transaction is checked against the set of recovered keys to see if it is satisfied. Third, each satisfied actor:permission is checked against the associated minimum permission required for that actor:contract::action pair to see if it meets or exceeds that minimum. This last check is performed at the action level before any action is executed (see 3.4.2. Permission Check).

3.4.1. Transaction Context After the public keys are recovered, a transaction context is created from the transaction instance. This context keeps track of the trace of actions and the action receipt generated as each action is dispatched and executed. All state generated is kept within a transaction trace instance and a list of action receipts. The

transaction trace consists of a list of action traces. Each action trace contains information about the executed action, such as the action receipt, the action instance, whether it is a context-free action, and the transaction ID that generated the action. The action receipt is generated later during transaction execution and finalization.

3.4.2. Permission Check Since the sequence of actions contained in the transaction must be executed atomically as a whole, the EOSIO software first checks that the actors specified in each action have the minimum permission required to execute it. To that end, the software checks the following for each action:

The named permission of each actor specified in each action instance. The named permission of the corresponding actor:contract::action pair specified in the smart contract. If there is at least one actor whose set of named permissions fail to meet the minimum permission level required by the corresponding actor:contract::action pair in the smart contract, the transaction fails. The reason why action permissions are checked before any action is executed is due to performance. It is more efficient to cancel a transaction with all actions unexecuted, than doing so after a few actions executed, but later were rolled back as a result of a failed action or authorization. Any state changes incurred during a failed action must be undone to preserve data integrity. Database sessions are expensive in terms of memory usage and computing resources. Therefore, undo operations must be minimized as possible.

3.4.3. Action Instance The diagram below depicts an action instance. It consists of the receiver account, the action name, the list of actors and their permissions, and the action data containing the message to be sent, if any, to the receiver account.

Action Instance authorization list auth n auth 2 auth 1 action name account name actor 2 actor n permission n ... permission 2 action data actor 1 permission 1 3.4.4. Authority Check After the minimum permission levels are checked, the authority table for the receiver account’ s permission that matches each actor’ s permission within the action instance is checked (see Accounts and Permissions: 3. Permissions for more details).

3.5. Execute Transaction To execute the transaction, a chain database session is started and a snapshot is taken. This allows to roll back any changes made to the chain state in case any of the transaction actions fails. A corresponding transaction context keeps the transaction state during execution. To execute the transaction, each action associated with the corresponding transaction instance is dispatched for execution. Context free actions, if any, are dispatched first, followed by regular actions.

3.5.1. Apply Context To prepare for action execution, an apply context is created locally for each action. The apply context, as its name implies, contains references to the necessary resources to apply the action, such as an instance to the chain controller (see Network Peer Protocol: 2.2. Chain Controller), the chain database where state is kept, the transaction context where the transaction is running, the actual action instance, the receiver account to whom the action is intended, etc.

3.5.2. Action Trace To prepare each action for execution, both action receipt and action trace instances are initialized. First, a hash of the action instance itself is computed and stored in the action receipt. Next, the action trace is initialized with statistics about the pending block where the transaction that includes the action will be pushed to. Therefore, an action trace allows an action to be traced to the actual block and transaction that includes the action, including the actual node that produced the block. Finally, the action handler is located by matching the handler name, receiver account, and actor account with the list of action handlers maintained by the chain controller within the producing node. These action handlers are applied in the controller when the system contracts and the client application are loaded. The handlers take the receiver account name, the contract name, the action name, and the action handler.

3.5.3. Action Execution Once the proper action handler is located, the appropriate whitelists and blacklists are checked. If the node is currently producing blocks, the receiver account is checked against the account whitelist and blacklist, if any. The action blacklist is checked next, if any. If the receiver account or the action name are in a blacklist, the action is aborted. If the receiver account is already on the whitelist, the blacklist check is skipped. If all checks pass, the action is finally executed by invoking the corresponding action handler, passing the actor account in the from parameter and the receiving account in the to parameter.

3.6. Finalize Transaction After all actions included in the transaction are executed, the transaction enters the finalization stage. In this step, a corresponding action receipt is produced for each action. The action receipt contains a hash of the corresponding action instance, a few counters used for analytics, and the receiver account to which the action is intended to, if applicable.

3.6.1. Transaction Receipt After all action receipts are generated for the transaction, a transaction receipt is finally created and pushed into the signed block, along with other transaction receipts included in the block. The transaction receipt summarizes the result of the transaction (executed, unexecuted, failed, deferred, expired, etc.), including the actual amount of CPU billed in microseconds, and the total NET storage used (see transaction_receipt schema below).

transaction_receipt schema Name Type Description status uint8_t result of transaction execution attempt cpu_usage_us uint32_t total CPU used in microseconds net_usage_words unsigned int total NET used in 64-bit words trx variant holds transaction ID or packed transaction The status field is an 8-bit enumeration type that can hold one of the following results:

executed - transaction succeeded, no error handler executed. soft_fail - transaction failed, error handler succeeded. hard_fail - transaction failed, error handler failed. delayed - transaction scheduled for future execution. expired - transaction expired, CPU/NET refunded to user. The trx field holds the transaction ID or the packed transaction itself. The actual choice depends on the transaction type. Receipts generated

from Deferred Transactions and Delayed User Transactions are stored by transaction ID; all other types are stored as packed transactions.

3.6.2. Deferred Transactions Deferred transactions are generated as a side effect of processing the blockchain, so their state is stored in the chain database, not within a block. Therefore, there is no need to explicitly include their contents in the transaction receipt. All in-sync nodes should be aware of the form of a deferred transaction as a matter of consensus.

3.6.3. Delayed User Transactions Delayed user transactions contain the packed transactions when they are pushed to the network (at the start of the delay timer). However, unlike regular transactions, they bear a "delayed" status so their execution and validation can be postponed. Later on when they execute/fail/expire (at the end of the delay timer), they only contain the transaction ID. This is because any in-sync node will have the transaction content from a previously broadcast block.

3.7. Validate Transaction A transaction is verified and validated at various stages during its lifecycle: first when it propagates on the peer-to-peer network as a loose transaction (see 3.4. Verify Transaction), then during block validation as the block is confirmed among a supermajority of block producers, and optionally during a blockchain replay if nodeos is configured to fully re-validate transactions during replays. By default, recorded transactions are not completely re-validated during replays since it is assumed that the node operator has established trust in the local block log, either personally or through a side-channel so it is no longer considered a potential source of byzantine information.

3.7.1. Validation Process When validating a transaction as part of a block, multiple validations occur at various levels. In full block validation, all transactions recorded in the block are replayed and the locally calculated merkle tree root hashes (generated from the transaction receipt data and the action receipt data, respectively) are compared against the transaction_mroot and action_mroot fields in the block header. Therefore, if a recorded transaction is tampered within a block, not only the merkle tree root hashes would cause a mismatch, but also the transaction signature(s) would fail to validate. If the tampering was not performed by a bona-fide block producer, the block signature would fail to validate as well (see Consensus Protocol: 5.3. Block Validation).

4.3、Network Peer Protocol

1. Overview Nodes on an active EOSIO blockchain must be able to communicate with each other for relaying transactions, pushing blocks, and syncing state between peers. The peer to peer (p2p) protocol, part of the nodeos service that runs on every node, serves this purpose. The ability to sync state is crucial for each block to eventually reach finality within the global state of the blockchain and allow each node to advance the last irreversible block (LIB). In this regard,

the fundamental goal of the p2p protocol is to sync blocks and propagate transactions between nodes to reach consensus and advance the blockchain state.

1.1. Goals In order to add multiple transactions into a block and fit them within the specified production time of 0.5 seconds, the p2p protocol must be designed with speed and efficiency in mind. These two goals translate into maximizing transaction throughput within the effective bandwidth and reducing both network and operational latency. Some strategies to achieve this include:

Fit more transactions within a block for better economy of scale. Minimize redundant information among blocks and transactions. Allow more efficient broadcasting and syncing of node states. Minimize payload footprint with data compression and binary encoding. Most of these strategies are fully or partially implemented in the EOSIO software. Data compression, which is optional, is implemented at the transaction level. Binary encoding is implemented by the net serializer when sending object instances and protocol messages over the network.

1. Architecture The main goal of the p2p protocol is to synchronize nodes securely and efficiently. To achieve this overarching goal, the system delegates functionality into four main components:

Net Plugin: defines the protocol to sync blocks and forward transactions between peers. Chain Controller: dispatches/manages blocks and transactions received, within the node. Net Serializer: serializes messages, blocks, and transactions for network transmission. Local Chain: holds the node’s local copy of the blockchain, including reversible blocks. The interaction between the above components is depicted in the diagram below:

Node A (node) local chain Node B (peer) local chain Net Plugin Chain Controller Net Serializer Net Serializer ... Net Plugin Chain Controller ... At the highest level sits the Net Plugin, which exchanges messages between the node and its peers to sync blocks and transactions. A typical message flow goes as follows:

Node A sends a message to Node B through the Net Plugin (refer to diagram above).

Node A’s Net Serializer packs the message and sends it to Node B. Node B’s Net Serializer unpacks the message and relays it to its Net Plugin. The message is processed by Node B’s Net Plugin, dispatching the proper actions. The Net Plugin accesses the local chain via the Chain Controller if necessary to push or retrieve blocks. 2.1. Local Chain The local chain is the node’s local copy of the blockchain. It consists of both the irreversible and reversible blocks received by the node, each block being cryptographically linked to the previous one. The list of irreversible blocks contains the actual copy of the immutable blockchain. The list of reversible blocks is typically shorter in length and it is managed by the Fork Database as the Chain Controller pushes blocks to it. The local chain is depicted below.

irreversible blocks reversible blocks ... 49 50 51 (lib) 52b 52c 53b 53a 53c 54d 54c 55c (hb) Each node constructs its own local copy of the blockchain as it receives blocks and transactions and syncs their state with other peers. The reversible blocks are those new blocks received that have not yet reached finality. As such, they are likely to form branches that stem from a main common ancestor, which is the LIB (last irreversible block). Other common ancestors different from the LIB are also possible for reversible blocks. In fact, any two sibling branches always have a nearest common ancestor. For instance, in the diagram above, block 52b is the nearest common ancestor for the branches starting at block 53a and 53b that is different from the LIB. Every active branch in the local chain has the potential to become part of the blockchain.

2.1.1. LIB Block All irreversible blocks constructed in a node are expected to match those from other nodes up to the last irreversible block (LIB) of each node. This is the distributed nature of the blockchain. Eventually, as the blocks that follow the LIB block reach finality, the LIB block moves up the chain through one of the branches as it catches up with the head block (HB). When the LIB block advances, the immutable blockchain effectively grows. In this process, the head block might switch branches multiple times depending on the potential head block numbers received and their timestamps, which is ultimately used as tiebreaker.

2.2. Chain Controller The Chain Controller manages the basic operations on blocks and transactions that change the local chain state, such as validating and executing transactions, pushing blocks, etc. The Chain Controller receives commands from the Net Plugin and dispatches the proper operation on a block or a transaction based on the network message received by the Net Plugin. The network messages are exchanged continuously between the EOSIO nodes as they communicate with each other to sync the state of blocks and transactions.

2.2.1. Fork Database The Fork Database (Fork DB) provides an internal interface for the Chain Controller to perform operations on the node’ s local chain. As new blocks are received from other peers, the Chain Controller pushes these blocks to the Fork DB. Each block is then cryptographically linked to a previous block. Since there might be more than one previous block, the process is likely to produce temporary branches called mini-forks. Thus, the Fork DB serves three main purposes:

Resolve which branch the pushed block (new head block) will build off from. Advance the head block, the root block, and the LIB block. Trim off invalid branches and purge orphaned blocks. In essence, the Fork DB contains all the candidate block branches within a node that may become the actual branch that continues to grow the blockchain. The root block always marks the beginning of the reversible block tree, and will match the LIB block, except when the LIB advances, in which case the root block must catch up. The calculation of the LIB block as it advances through the new blocks within the Fork DB will ultimately decide which branch gets selected. As the LIB block advances, the root block catches up with the new LIB, and any candidate branch whose ancestor node is behind the LIB gets pruned. This is depicted below.

irreversible blocks reversible blocks ... 49 50 51 52b (inv) X 52c 53b (inv) X 53a (inv) X 54c 53c (lib) 54d 55c (hb) In the diagram above, the branch starting at block 52b gets pruned (blocks 52b, 53a, 53b are invalid) after the LIB advances from node 51 to block 52c then 53c. As the LIB moves through the reversible blocks, they are moved from the Fork DB to the local chain as they now become part of the immutable blockchain. Finally, block 54d is kept in the Fork DB since new blocks might still be built off from it.

2.3. Net Plugin The Net Plugin defines the actual peer to peer communication messages between the EOSIO nodes. The main goal of the Net Plugin is to sync valid blocks upon request and to forward valid transactions invariably. To that end, the Net Plugin delegates functionality to the following components:

Sync Manager: maintains the block syncing state of the node with respect to its peers. Dispatch Manager: maintains the list of blocks and transactions sent by the node. Connection List: list of active peers the node is currently connected to. Message Handler: dispatches protocol messages to the corresponding handler. (see 4.2. Protocol Messages). 2.3.1. Sync Manager The Sync Manager implements the functionality for syncing block state between the node and its peers. It processes the messages sent by each peer and performs the actual syncing of the blocks based on the status of the node’ s LIB or head block with respect to that peer. At any point, the node can be in any of the following sync states:

LIB Catch-Up: node is about to sync with another peer's LIB block. Head Catch-Up: node is about to sync with another peer's HEAD block. In-Sync: both LIB and HEAD blocks are in sync with the other peers. If the node’ s LIB or head block is behind, the node will generate sync request messages to retrieve the missing blocks from the connected peer. Similarly, if a connected peer’ s LIB or head block is behind, the node will send notice messages to notify the node about which blocks it needs to sync with. For more information about sync modes see 3. Operation Modes.

2.3.2. Dispatch Manager The Dispatch Manager maintains the state of blocks and loose transactions received by the node. The state contains basic information to identify a block or a transaction and it is maintained within two indexed lists of block states and transaction states:

Block State List: list of block states managed by node for all blocks received. Transaction State List: list of transaction states managed by node for all transactions received. This makes it possible to locate very quickly which peer has a given block or transaction.

2.3.2.1. Block State The block state identifies a block and the peer it came from. It is transient in nature, so it is only valid while the node is active. The block state contains the following fields:

Block State Fields Description id 256-bit block identifier. A function of the block contents and the block number. block_num 32-bit unsigned counter that locates the block sequentially in the blockchain. connection_id 32-bit unsigned integer that identifies the connected peer the block came from. have_block boolean value indicating whether the actual block has been received by the node. The list of block states is indexed by block ID, block number, and connection ID for faster lookup. This allows to query the list for any blocks given one or more of the indexed attributes.

2.3.2.2. Transaction State The transaction state identifies a loose transaction and the peer it came from. It is also transient in nature, so it is only valid while the node is active. The transaction state contains the following fields:

Transaction State Fields Description id 256-bit hash of the transaction instance, used as transaction identifier. expires expiration time since EOSIO block timestamp epoch (January 1, 2000). block_num current head block number. Transaction drops when LIB catches up to it. connection_id 32-bit integer that identifies the connected peer the transaction came from. The block_num stores the node's head block number when the transaction is received. It is used as a backup mechanism to drop the transaction when the LIB block number catches up with the head block number, regardless of expiration.

The list of transaction states is indexed by transaction ID, expiration time, block number, and connection ID for faster lookup. This allows to query the list for any transactions given one or more of the indexed attributes.

2.3.2.3. State Recycling As the LIB block advances (see 3.3.1. LIB Catch-Up Mode), all blocks prior to the new LIB block are considered finalized, so their state is removed from the local list of block states, including the list of block states owned by each peer in the list of connections maintained by the node. Likewise, transaction states are removed from the list of transactions based on expiration time. Therefore, after a transaction expires, its state is removed from all lists of transaction states.

The lists of block states and transaction states have a light footprint and feature high rotation, so they are maintained in memory for faster access. The actual contents of the blocks and transactions received by a node are stored temporarily in the fork database and the various incoming queues for applied and unapplied transactions, respectively.

2.3.3. Connection List The Connection List contains the connection state of each peer. It keeps information about the p2p protocol version, the state of the blocks and transactions from the peer that the node knows about, whether it is currently syncing with that peer, the last handshake message sent and received, whether the peer has requested information from the node, the socket state, the node ID, etc. The connection state includes the following relevant fields:

Info requested: whether the peer has requested information from the node. Socket state: a pointer to the socket structure holding the TCP connection state. Node ID: the actual

node ID that distinguishes the peer’ s node from the other peers. Last Handshake Received: last handshake message instance received from the peer. Last Handshake Sent: the last handshake message instance sent to the peer. Handshake Sent Count: the number of handshake messages sent to the peer. Syncing: whether or not the node is syncing with the peer. Protocol Version: the internal protocol version implemented by the peer’ s Net Plugin. The block state consists of the following fields:

Block ID: a hash of the serialized contents of the block. Block number: the actual block number since genesis. The transaction state consists of the following fields:

Transaction ID: a hash of the serialized contents of the transaction. Block number: the actual block number the transaction was included in. Expiration time: the time in seconds for the transaction to expire. 2.4. Net Serializer The Net Serializer has two main roles:

Serialize objects and messages that need to be transmitted over the network. Serialize objects and messages that need to be cryptographically hashed. In the first case, each serialized object or message needs to get deserialized at the other end upon receipt from the network for further processing. In the latter case, serialization of specific fields within an object instance is needed to generate cryptographic hashes of its contents. Most IDs generated for a given object type (action, transaction, block, etc.) consist of a cryptographic hash of the relevant fields from the object instance.

1. Operation Modes From an operational standpoint, a node can be in either one of three states with respect to a connected peer:

In-Sync mode: node is in sync with peer, so no blocks are required from that peer. LIB Catch-Up mode: node requires blocks since LIB block is behind that peer’ s LIB. HEAD Catch-Up mode: node requires blocks since HEAD block is behind that peer’ s Head. The operation mode for each node is stored in a sync manager context within the Net Plugin of the nodeos service. Therefore, a node is always in either in-sync mode or some variant of catchup mode with respect to its connected peers. This allows the node to switch back and forth between catchup mode and in-sync mode as the LIB and head blocks are updated and new fresh blocks are received from other peers.

3.1. Block ID The EOSIO software checks whether two blocks match or hold the same content by comparing their block IDs. A block ID is a function that depends on the contents of the block header and the block number (see Consensus Protocol: 5.1. Block Structure). Checking whether two blocks are equal is crucial for syncing a node’ s local chain with that of its peers. To generate the block ID from the block contents, the block header is serialized and a SHA-256 digest is created. The most significant 32 bits of the hash are retained while the least significant 32 bits are assigned the block number. Note that the block header includes the root hash of both the transaction merkle tree and the action merkle tree. Therefore, the block ID depends on all transactions included in the block as well as all actions included in each transaction.

3.2. In-Sync Mode During in-sync mode, the node's head block is caught up with the peer's head block, which means the node is in sync block-wise. When the node is in-sync mode, it does not request further blocks from peers, but continues to perform the other functions:

Validate transactions, drop them if invalid; forward them to other peers if valid. Validate blocks, drop them if invalid; forward them to other peers upon request if valid. Therefore, this mode trades bandwidth in favor of latency, being particularly useful for validating transactions that rely on TaPoS (transaction as proof of stake) due to lower processing overhead.

Note that loose transactions are always forwarded if valid and not expired. Blocks, on the other hand, are only forwarded if valid and if explicitly requested by a peer. This reduces network overhead.

3.3. Catch-Up Mode A node is in catchup mode when its head block is behind the peer’ s LIB or the peer’ s head block. If syncing is needed, it is performed in two sequential steps:

Sync the node’ s LIB from the nearest common ancestor + 1 up to the peer’ s LIB. Sync the node’ s head from the nearest common ancestor + 1 up to the peer’ s head. Therefore, the node’ s LIB block is updated first, followed by the node’ s head block.

3.3.1. LIB Catch-Up Mode Case 1 above, where the node’ s LIB block needs to catch up with the peer’ s LIB block, is depicted in the below diagram, before and after the sync (Note: inapplicable branches have been removed for clarity):

Node (after): Peer: Node (before): ... 88 89 90 91 92 lib/hb ... 88 89 90 91 92 (lib) 93p 94p (hb) ... 88 89 90 (lib) 91n 92n 93n (hb) In the above diagram, the node’ s local chain syncs up with the peer’ s local chain by appending finalized blocks 91 and 92 (the peer’ s LIB) to the node’ s LIB (block 90). Note that this discards the temporary fork consisting of blocks 91n, 92n, 93n. Also note that these nodes have an “n” suffix (short for node) to indicate that they are not finalized, and therefore, might be different from the peer’ s. The same applies to unfinalized blocks on the peer; they end in “p” (short for peer). After syncing, note that both the LIB (lib) and the head block (hb) have the same block number on the node.

3.3.2. Head Catch-Up Mode After the node’ s LIB block is synced with the peer’ s, there will be new blocks pushed to either chain. Case 2 above covers the case where the peer’ s chain is longer than the node’ s chain. This is depicted in the following diagram, which shows the node and the peer’ s local chains before and after the sync:

Node (after): Peer: Node (before): ... 90 91 92 (lib) 93n,p 94p 95p (hb) ... 90 91 92 (lib) 93p 94p 95p (hb) ... 90 91 92 (lib) 93n 94n (hb) In either case 1 or 2 above, the syncing process in the node involves locating the first common ancestor block starting from the

node’ s head block, traversing the chains back, and ending in the LIB blocks, which are now in sync (see 3.3.1. LIB Catch-Up Mode). In the worst case scenario, the synced LIBs are the nearest common ancestor. In the above diagram, the node’ s chain is traversed from head block 94n, 93n, etc. trying to match blocks 94p, 93p, etc. in the peer’ s chain. The first block that matches is the nearest common ancestor (block 93n and 93p in the diagram). Therefore, the following blocks 94p and 95p are retrieved and appended to the node’ s chain right after the nearest common ancestor, now re-labeled 93n,p (see 3.3.3. Block Retrieval process). Finally, block 95p becomes the node’ s head block and, since the node is fully synced with the peer, the node switches to in-sync mode.

3.3.3. Block Retrieval After the common ancestor is found, a sync request message is sent to retrieve the blocks needed by the node, starting from the next block after the nearest common ancestor and ending in the peer’ s head block.

To make effective use of bandwidth, the required blocks are obtained from various peers, rather than just one, if necessary. Depending on the number of blocks needed, the blocks are requested in chunks by specifying the start block number and the end block number to download from a given peer. The node uses the list of block states to keep track of which blocks each peer has, so this information is used to determine which connected peers to request block chunks from. This process is depicted in the diagram below:

Node-peer syncing

When both LIB and head blocks are caught up with respect to the peer, the operation mode in the Sync Manager is switched from catch-up to in-sync.

3.4. Mode Switching Eventually, both the node and its peer receive new fresh blocks from other peers, which in turn push the blocks to their respective local chains. This causes the head blocks on each chain to advance. Depending on which chain grows first, one of the following actions occur:

The node sends a catch up request message to the peer with its head block info. The node sends a catch up notice message to inform the peer it needs to sync. In the first case, the node switches the mode from in-sync to head catchup mode. In the second case, the peer switches to head catchup mode after receiving the notice message from the node. In practice, in-sync mode is short-lived. In a busy EOSIO blockchain, nodes spend most of their time in catchup mode validating transactions and syncing their chains after catchup messages are received.

1. Protocol Algorithm The p2p protocol algorithm runs on every node, forwarding validated transactions and validated blocks (starting EOSIO v2.x, a node will also forward block IDs of unvalidated blocks it has received). In general, the simplified process is as follows:

A node requests data or sends a control message to a peer. If the request can be fulfilled, the peer executes the request; repeat 1. The data messages contain the block contents or the transaction contents. The control messages make possible the syncing of blocks and transactions between the node and its peers (see Protocol Messages). In order to allow such synchronization, each node must be able to retrieve information about its own state of blocks and transactions as well as that of its peers.

4.1. Node/Peers Status Before attempting to sync state, each node needs to know the current status of its own blocks and transactions. It must also be able to query other peers to obtain the same information. In particular, nodes must be able to obtain the following on demand:

Each node can find out which blocks and transactions it currently has. All nodes can find out which blocks and transactions their peers have. Each node can find out which blocks and transactions it has requested. All nodes can find out when each node has received a given transaction. To perform these queries, and thereafter when syncing state, the Net Plugin defines specific communication messages to be exchanged between the nodes. These messages are sent by the Net Plugin when transmitted and received over a TCP connection.

4.2. Protocol Messages The p2p protocol defines the following control messages for peer to peer node communication:

Control Message Description handshake_message initiates a connection to another peer and sends LIB/head status. chain_size_message requests LIB/head status from peer. Not currently implemented. go_away_message sends disconnection notification to a connecting or connected peer. time_message transmits timestamps for peer synchronization and error detection. notice_message informs peer which blocks and transactions node currently has. request_message informs peer which blocks and transaction node currently needs. sync_request_message requests peer a range of blocks given their start/end block numbers. The protocol also defines the following data messages for exchanging the actual contents of a block or a loose transaction between peers on the p2p network:

Data Message Description signed_block serialized contents of a signed block. packed_transaction serialized contents of a packed transaction. 4.2.1. Handshake Message The handshake message is sent by a node when connecting to another peer. It is used by the connecting node to pass its chain state (LIB number/ID and head block number/ID) to the peer. It is also used by the peer to perform basic validation on the node the first time it connects, such as whether it belongs to the same blockchain, validating that fields are within range, detecting inconsistent block states on the node, such as whether its LIB is ahead of the head block, etc. The handshake message consists of the following fields:

Message Field Description network_version internal net plugin version to keep track of protocol updates. chain_id hash value of the genesis state and config options. Used to identify chain. node_id the actual node ID that distinguishes the peer’ s node from the other peers. key public key for peer to validate node; may be a producer or peer key, or empty. time timestamp the handshake message was created since epoch (Jan 1, 2000). token SHA-256 digest of timestamp to prove node owns private key of the key above. sig signature for the digest above after node signs it with private key of the key above. p2p_address IP address of node. last_irreversible_block_num the actual block count of the LIB block since genesis. last_irreversible_block_id a hash of the serialized contents of the LIB block. head_num the actual block count of the head block since genesis. head_id a hash of the serialized contents of the head block. os operating system where node runs. This is detected automatically. agent the name supplied by node to identify itself among its peers. generation counts handshake_message invocations; detects first call for validation. If all checks succeed, the peer proceeds to authenticate the connecting node based on the --allowed-connection setting specified for that peer's net plugin when nodeos started:

Any: connections are allowed without authentication. Producers: peer key is obtained via p2p protocol. Specified: peer key is provided via settings. None: the node does not allow connection requests. The peer key corresponds to the public key of the node attempting to connect to the peer. If authentication succeeds, the receiving node acknowledges the connecting node by sending a handshake message back, which the connecting node validates in the same way as above. Finally, the receiving node checks whether the peer’ s head block or its own needs syncing. This is done by checking the state of the head block and the LIB of the connecting node with respect to its own. From these checks, the receiving node determines which chain needs syncing.

4.2.2. Chain Size Message The chain size message was defined for future use, but it is currently not implemented. The idea was to send ad-hoc status notifications of the node’ s chain state after a successful connection to another peer. The chain size message consists of the following fields:

Message Field Description last_irreversible_block_num the actual block count of the LIB block since genesis. last_irreversible_block_id a hash of the serialized contents of the LIB block. head_num the actual block count of the head block since genesis. head_id a hash of the serialized contents of the head block. The chain size message is superseded by the handshake message, which also sends the status of the LIB and head blocks, but includes additional information so it is preferred.

4.2.3. Go Away Message The go away message is sent to a peer before closing the connection. It is usually the result of an error that prevents the node from continuing the p2p protocol further. The go away message consists of the following fields:

Message Field Description reason an error code signifying the reason to disconnect from peer. node_id the node ID for the disconnecting node; used for duplicate notification. The current reason codes are defined as follows:

No reason: indicate no error actually; the default value. Self: node was attempting to self connect. Duplicate: redundant connection detected from peer. Wrong chain: the peer's chain ID does not match. Wrong version: the peer's network version does not match. Forked: the peer's irreversible blocks are different Unlinkable: the peer sent a block we couldn't use Bad transaction: the peer sent a transaction that failed verification. Validation: the peer sent a block that failed validation. Benign other: reasons such as a timeout. not fatal but warrant resetting. Fatal other: a catch all for fatal errors that have not been isolated yet. Authentication: peer failed authentication. After the peer receives the go away message, the peer should also close the connection.

4.2.4. Time Message The time message is used to synchronize events among peers, measure time intervals, and detect network anomalies such as duplicate messages, invalid timestamps, broken nodes, etc. The time message consists of the following fields:

Message Field Description org origin timestamp; set when marking the beginning of a time interval. rec receive timestamp; set when a message arrives from the network. xmt transmit timestamp; set when a message is placed on the send queue. dst destination timestamp; set when marking the end of a time interval. 4.2.5. Notice Message The notice message is sent to notify a peer which blocks and loose transactions the node currently has. The notice message consists of the following fields :

Message Field Description known_trx sorted list of known transaction IDs node has available. known_blocks sorted list of known block IDs node has available. Notice messages are lightweight since they only contain block IDs and transaction IDs, not the actual block or transaction.

4.2.6. Request Message The request message is sent to notify a peer which blocks and loose transactions the node currently needs. The request message consists of the following fields:

Message Field Description req_trx sorted list of requested transaction IDs required by node. req_blocks sorted list of requested block IDs required by node. 4.2.7. Sync Request Message The sync request message requests a range of blocks from peer. The sync request message consists of the following fields:

Message Field Description start_block start block number for the range of blocks to receive from peer. end_block end block number for the range of blocks to receive from peer. Upon receipt of the sync request message, the peer sends back the actual blocks for the range of block numbers specified.

4.3. Message Handler The p2p protocol uses an event-driven model to process messages, so no polling or looping is involved when a message is received. Internally, each message is placed in a queue and the next message in line is dispatched to the corresponding message handler for processing. At a high level, the message handler can be defined as follows:

receiver/read handler: if handshake message: verify that peer's network protocol is valid if node's LIB < peer's LIB: sync LIB with peer's; continue if node's LIB > peer's LIB: send LIB catchup notice message; continue if notice message: update list of blocks/transactions known by remote peer if trx message: insert into global state as unvalidated validate transaction; drop if invalid, forward if valid else close the connection 4.4. Send Queue Protocol messages are placed in a buffer queue and sent to the appropriate connected peer. At a higher level, a node performs the following operations with each connected peer in a round-robin fashion:

send/write loop: if peer knows the LIB: if peer does not know we have a block or transaction: next iteration if peer does not know about a block: send transactions for block that peer does not know next iteration if peer does not know about transactions: sends oldest transactions unknown to remote peer next iteration wait for new validated block, transaction, or peer signal else: assume peer is in catchup mode (operating on request/response) wait for notice of sync from the read loop

1. Protocol Improvements Any software updates to the p2p protocol must also scale progressively and consistently across all nodes. This translates into installing updates that reduce operation downtime and potentially minimize it altogether while deploying new functionality in a backward compatible manner, if possible. On the other hand, data throughput can be increased by taking measures that minimize message footprint, such as using data compression and binary encoding the protocol messages.

4.4、Accounts and Permissions

1. Overview An account identifies a participant in an EOSIO blockchain. A participant can be an individual or a group depending on the assigned permissions within the account. Accounts also represent the smart contract actors that push and receive actions to and from other accounts in the blockchain. Actions are always contained within transactions. A transaction can be one or more atomic actions.

Permissions associated with an account are used to authorize actions and transactions to other accounts. Each permission is linked to an authority table which contains a threshold that must be reached in order to allow the action associated with the given permission to be authorized for execution. The following diagram illustrates the relationship between accounts, permissions, and authorities.

Accounts Permissions @alice permissions Authorities @alice owner @bob ... active @alice active authority threshold 2 accounts/keys weights bob@active 2 stacy@active 2 EOS7Hnv4... 1 EOS3Wo1p... 1 family friends lawyer ... ... The example above depicts alice's account, her named permissions along with their hierarchical dependencies, and her linked active authority table. It also shows that a weight threshold of two must be reached in alice's active authority in order to allow an action associated with the active permission to be executed by or on behalf of alice.

1. Accounts Each account is identified by a human readable name between 2 and 12 characters in length. The characters can include a-z, 1-5, and optional dots (.) except the first and last characters. This allows roughly one exa (2^{60}2 60) accounts. The exact number is:

31^{2} \cdot \sum_{n=0}^{n=10} 32^{n} = (2^{5}-1) \cdot (2^{55}-1) = 1,116,892,707,587,882,977 31 2 · n=0 ∑ n=10 32 n =(2 5 −1)· (2 55 −1)=1,116,892,707,587,882,977 which is in the order of 10^{18}10 18 .

Ownership of each account on an EOSIO blockchain is solely determined by the account name. Therefore, an account can update its keys without having to redistribute them to other parties.

2.1. Account Schema Besides the account name, the blockchain associates other fields with each account instance stored in the chain database, such as ram quota/usage, cpu/net limits/weights, voter info, etc. (see account schema below). More importantly, each account holds the list of named permissions assigned to it. This allows a flexible permission structure that makes single or multi-user authorizations possible (see 3. Permissions).

account schema Name Type Description account_name name encoded 13-char account name head_block_num uint32_t last block account was referenced head_block_time time_point last time account was referenced privileged bool privileged account? last_code_update time_point time account code was set/updated created time_point time account was created core_liquid_balance asset current balance of token asset ram_quota int64_t maximum RAM amount for account net_weight int64_t weight for net limit percentage (weight/total) cpu_weight int64_t weight for cpu limit percentage (weight/total) net_limit account_resource_limit total net used, available, and max cpu_limit account_resource_limit total cpu used, available, and max ram_usage int64_t amount of RAM in bytes used by account permissions array of permission list of named permissions total_resources variant total cpu/net weights for all accounts self_delegated_bandwidth variant cpu/net stake delegated from self refund_request variant cpu/net refund amounts for token unstaking voter_info variant name of voter, proxy or producers, vote stake rex_info variant vote stake and rex balance if applicable The name type consists of a 64-bit value that encodes alphanumeric characters into 5-bit chunks, except the last character, if any, which uses a 4-bit chunk. The name type is

used to encode account names, action names, etc. The time_point type stores timestamps in microseconds. The asset type associates a currency or token symbol with a given amount. The account_resource_limit type keeps track of the amount used, available, and maximum that can be used in a given window for the given resource (NET or CPU). The permission type holds the list of permission levels associated with the account (see 3. Permissions).

2.2. Actions and Transactions Besides identifying participants in an EOSIO blockchain, actions and transactions are the other reason for accounts to exist. An action requires one or more actors to push or send the action, and a receiver account to whom the action is directed. A receiver account is also needed when leaving proof, in an action receipt, that the action was pushed to the intended recipient.

In contrast, transactions are agnostic to accounts, although there is an indirect link to them through their associated keys. Transactions are signed using one or more signing keys belonging to the one or more actors involved in the actions that form the transaction. This can be the receiving account itself or other authorized actors specified on the authority table from the receiving account's permission.

1. Permissions Permissions control what EOSIO accounts can do and how actions are authorized. This is accomplished through a flexible permission structure that links each account to a list of hierarchical named permissions, and each named permission to an authority table (see permission schema below).

permission schema Name Type Description perm_name name named permission parent name parent's named permission required_auth authority associated authority table The parent field links the named permission level to its parent permission. This is what allows hierarchical permission levels in EOSIO.

3.1. Permission Levels A named permission may be created under another permission, thereby allowing a hierarchical parent-children permission structure. This makes implicit action authorizations possible by allowing a given actor:child-permission authorization within an action to be implicitly satisfied if the actor:parent-permission is also satisfied. An authorization quorum or "threshold" must still be met for the action to be authorized for execution (see 3.2.2. Authority Threshold).

Contract-level Permissions It is also possible to create an implicit link between two accounts with the same named permission (for authorization satisfaction purposes). This can be achieved by associating an explicit named permission to the smart contract (different from the "minimum permission" for that contract[::action]). However, defining explicit actor:permission authorizations within actions is preferred versus associating permissions to the whole contract.

Every account has two default named permissions when created, owner and active. They have a parent-child relationship by default, although this can be customized by adding other permission levels and hierarchies.

3.1.1. Owner permission The owner permission sits at the root of the permission hierarchy for every account. It is therefore the highest relative permission an account can have within its permission structure. Although the owner permission can do anything a lower level permission can, it is typically used for recovery purposes when a lower permission has been compromised. As such, keys associated with the owner permission are typically kept in cold storage, not used for signing regular operations.

3.1.2. Active permission In the current EOSIO implementation, the implicit default permission linked to all actions is active, which sits one level below the owner permission within the hierarchy structure. As a result, the active permission can do anything the owner permission can, except changing the keys associated with the owner. The active permission is typically used for voting, transferring funds, and other account operations. For more specific actions, custom permissions are typically created below the active permission and mapped to specific contracts or actions. Refer to the Creating and Linking Custom Permissions for more details.

Custom Permissions EOSIO allows to create custom hierarchical permissions that stem from the owner permission. This allows finer control over action authorizations. It also strengthens security in case the active permission gets compromised.

3.2. Authority Table Each account's permission can be linked to an authority table used to determine whether a given action authorization can be satisfied. The authority table contains the applicable permission name and threshold, the "factors" and their weights, all of which are used in the evaluation to determine whether the authorization can be satisfied. The permission threshold is the target numerical value that must be reached to satisfy the action authorization (see authority schema below).

authority schema Name Type Description threshold uint32_t threshold value to satisfy authorization keys array of key_weight list of public keys and weights accounts array of permission_level_weight list of account@permission levels and weights waits array of wait_weight list of time waits and weights The key_weight type contains the actor's public key and associated weight. The permission_level_weight type consists of the actor's account@permission level and associated weight. The wait_weight contains the time wait and associated weight (used to satisfy action authorizations in delayed user transactions (see Transactions Protocol: 3.6.3. Delayed User Transactions). All of these types allow to define lists of authority factors that are used for satisfaction of action authorizations (see 3.2.1. Authority factors below).

3.2.1. Authority Factors Every authority table linked to a given permission lists potential "factors" explicitly used in the evaluation of the action authorization. A factor type can be one of the following:

Actor's account name and permission level Actor's public key Time wait The potential actors who may execute the action are specified by either public key or account name in the authority table. Time waits are special factors which are satisfied by publishing a transaction with a delay in excess of the defined time. These carry weights as well that may contribute to satisfy the threshold.

3.2.2. Authority Threshold Authorization over a given action is determined by satisfying all explicit authorizations specified in the action instance (see Transactions Protocol: 3.4.3. Action Instance). Those are in turn individually satisfied by evaluating each "factor" (account, public key, wait) for satisfaction (potentially recursively) and summing the weights of those that are satisfied. If the sum equals or exceeds the weight threshold, the action is authorized.

3.2.3. Authority Example The authority table for alice's publish named permission is shown below. According to its contents, in order to authorize an action under that permission, a threshold of two must be reached. Since both bob@active and stacy@active factors have a weight of two, either one can satisfy the action authorization. This means that either bob or stacy with a permission level of active or higher can independently execute any action under alice's publish permission.

Permission Account / Public Key Weight Threshold publish bob@active 2 2 stacy@active 2 EOS7Hnv4iBfcw2... 1 EOS3Wo1p9er7fh... 1 Alternatively, it would require two acounts with public keys EOS7Hnv4iBfcw2... and EOS3Wo1p9er7fh... to satisfy the action authorization. This is because each public key has a weight of 1 in the authority table.

3.3. Permission Mapping Any given account can define a mapping between any of its named permissions and a smart contract or action within that contract. This sets the "minimum permission" required for that contract[::action]. It does not afford, however, any other account any access or authority to execute that contract[::action]. This is by design and the process is controlled by a permission evaluation mechanism, described next.

3.4. Permission Evaluation When determining whether an action is authorized to be executed, the EOSIO software first checks whether the signatures provided in the transaction are valid (see 3.4.2. Signature Validation). Then it proceeds to check the authorization of all the actions included in the transaction. This is where permissions are evaluated. If there is at least one action that fails to be authorized (by not meeting the authority threshold (see 3.2.2. Authority Threshold), the transaction fails.

3.4.1. Custom Permissions By default every account on the EOSIO blockchain is linked to the active permission. Again, this can be customized by creating children permissions under active or by creating alternate permissions under owner (see 3.1. Permission Levels). Creating custom permissions under owner (separate from active) is

recommended. This is because if the keys associated with the active permission are compromised, the security of the account will not be compromised.

Use Case: Social Media Say we have a publish permission created for message posting on a social media application. However, we do not want to associate that permission with sensitive actions, such as transferring or withdrawing funds. Under this scenario, it makes sense to link the social::post action to the publish permission. This allows to define an authority structure which can authorize post, but cannot satisfy the default active permission for all other actions. That authority structure could delegate itself to a different account at any named permission level. If it did so to another publish permission on another account, that would be purely coincidental.

3.4.2. Signature Validation Satisfying authorities linked to permissions involves first and foremost the validation/recovery of the public keys that signed the transaction. After a signed transaction is received by a node, the set of signatures is extracted from the transaction instance. The set of public keys are then recovered from the signatures. Then for all actions included in the transaction, the node checks that each actor:permission meets or exceeds the minimum permission as defined by the per-account permission links.

Once validated, the set of recovered keys are provided to the authorization manager instance along with the amount of time "waited". The authorization manager then proceeds to check whether the provided "factors" satisfy the authorities, potentially recursing into other linked permission levels/authorities (see 3.2. Authority Table and Transactions Protocol: 3.4. Verify Transaction for more information).

5、Software Manuals

5.1、Core

5.1.1、Nodeos

Introduction nodeos is the core service daemon that runs on every EOSIO node. It can be configured to process smart contracts, validate transactions, produce blocks containing valid transactions, and confirm blocks to record them on the blockchain.

Installation nodeos is distributed as part of the EOSIO software suite. To install nodeos, visit the EOSIO Software Installation section.

Explore Navigate the sections below to configure and use nodeos.

Usage - Configuring and using nodeos, node setups/environments. Plugins - Using plugins, plugin options, mandatory vs. optional. Replays - Replaying the

chain from a snapshot or a blocks.log file. Logging - Logging config/usage, loggers, appenders, logging levels. Upgrade Guides - EOSIO version/consensus upgrade guides. Troubleshooting - Common nodeos troubleshooting questions. Access Node A local or remote EOSIO access node running nodeos is required for a client application or smart contract to interact with the blockchain.

5.1.2、Cleos

Introduction cleos is a command line tool that interfaces with the REST API exposed by nodeos. Developers can also use cleos to deploy and test EOSIO smart contracts.

Installation cleos is distributed as part of the EOSIO software suite. To install cleos just visit the EOSIO Software Installation section.

Using Cleos To use cleos, you need the end point (IP address and port number) of a running nodeos instance. Also, the nodeos instance must be configured to load the eosio::chain_api_plugin when launched. This allows nodeos to respond to the RPC requests coming from cleos.

Cleos Commands For a list of all cleos commands, run:

cleos --help Command Line Interface to EOSIO Client Usage: cleos [OPTIONS] SUBCOMMAND

Options: -h,--help Print this help message and exit -u,--url TEXT=http://127.0.0.1:8888/ the http/https URL where nodeos is running --wallet-url TEXT=unix:///Users/username/eosio-wallet/keosd.sock the http/https URL where keosd is running -r,--header pass specific HTTP header; repeat this option to pass multiple headers -n,--no-verify don't verify peer certificate when using HTTPS --no-auto-keosd don't automatically launch a keosd if one is not currently running -v,--verbose output verbose errors and action console output --print-request print HTTP request to STDERR --print-response print HTTP response to STDERR

Subcommands: version Retrieve version information create Create various items, on and off the blockchain convert Pack and unpack transactions get Retrieve various items and information from the blockchain set Set or update blockchain state transfer Transfer tokens from account to account net Interact with local p2p network connections wallet Interact with local wallet sign Sign a transaction push Push arbitrary transactions to the blockchain multisig Multisig contract commands wrap Wrap contract commands system Send eosio.system contract action to the blockchain. Cleos Subcommands To get help with any particular subcommand, run cleos SUBCOMMAND --help. For instance:

cleos create --help Create various items, on and off the blockchain Usage: cleos create SUBCOMMAND

Subcommands: key Create a new keypair and print the public and private keys account Create a new account on the blockchain (assumes system contract does not restrict RAM usage) cleos can also provide usage help for subcommands within subcommands. For instance:

cleos create account --help Create a new account on the blockchain (assumes system contract does not restrict RAM usage) Usage: cleos create account [OPTIONS] creator name OwnerKey [ActiveKey]

Positionals: creator TEXT The name of the account creating the new account (required) name TEXT The name of the new account (required) OwnerKey TEXT The owner public key or permission level for the new account (required) ActiveKey TEXT The active public key or permission level for the new account

Options: -h,--help Print this help message and exit -x,--expiration set the time in seconds before a transaction expires, defaults to 30s -f,--force-unique force the transaction to be unique. this will consume extra bandwidth and remove any protections against accidently issuing the same transaction multiple times -s,--skip-sign Specify if unlocked wallet keys should be used to sign transaction -j,--json print result as json --json-file TEXT save result in json format into a file -d,--dont-broadcast don't broadcast transaction to the network (just print to stdout) --return-packed used in conjunction with --dont-broadcast to get the packed transaction -r,--ref-block TEXT set the reference block num or block id used for TAPOS (Transaction as Proof-of-Stake) --use-old-rpc use old RPC push_transaction, rather than new RPC send_transaction -p,--permission TEXT ... An account and permission level to authorize, as in 'account@permission' (defaults to 'creator@active') --max-cpu-usage-ms UINT set an upper limit on the milliseconds of cpu usage budget, for the execution of the transaction (defaults to 0 which means no limit) --max-net-usage UINT set an upper limit on the net usage budget, in bytes, for the transaction (defaults to 0 which means no limit) --delay-sec UINT set the delay_sec seconds, defaults to 0s Cleos Example The following cleos command creates a local wallet named mywallet and displays the password to the screen:

cleos wallet create -n mywallet --to-console Creating wallet: mywallet Save password to use in the future to unlock this wallet. Without password imported keys will not be retrievable.
"PW5JbF34UdA193Eps1bjrWVJRaNMt1VKddLn4Dx6SPVTfMDRnMBWN"

5.1.3、Keosd

Introduction keosd is a key manager service daemon for storing private keys and signing digital messages. It provides a secure key storage medium for keys to be encrypted at rest in the associated wallet file. keosd also defines a secure enclave for signing transaction created by cleos or a third part library.

Installation keosd is distributed as part of the EOSIO software suite. To install keosd just visit the EOSIO Software Installation section.

Operation When a wallet is unlocked with the corresponding password, cleos can request keosd to sign a transaction with the appropriate private keys. Also, keosd provides support for hardware-based wallets such as Secure Encalve and YubiHSM.

Audience keosd is intended to be used by EOSIO developers only.

5.1.4、eosio.cdt

Version : 1.7.0 EOSIO.CDT is a toolchain for WebAssembly (WASM) and set of tools to facilitate smart contract development for the EOSIO platform. In addition to being a general purpose WebAssembly toolchain, EOSIO specific optimizations are available to support building EOSIO smart contracts. This new toolchain is built around Clang 7, which means that EOSIO.CDT has the most currently available optimizations and analyses from LLVM, but as the WASM target is still considered experimental, some optimizations are incomplete or not available.

New Introductions As of this release two new repositories are under the suite of tools provided by EOSIO.CDT. These are the Ricardian Template Toolkit and the Ricardian Specification. The Ricardian Template Toolkit is a set of libraries to assist smart contract developers in craftinng their Ricardian contracts. The Ricardian specification is the working specification for the above mentioned toolkit. Please note that both projects are alpha releases and are subject to change.

Upgrading There's been a round of breaking changes, if you are upgrading please read the Upgrade guide from 1.2 to 1.3 and Upgrade guide from 1.5 to 1.6.

Contributing Contributing Guide

Code of Conduct

License MIT

Important See LICENSE for copyright and license terms. Block.one makes its contribution on a voluntary basis as a member of the EOSIO community and is not responsible for ensuring the overall performance of the software or any related applications. We make no representation, warranty, guarantee or undertaking in

respect of the software or any related documentation, whether expressed or implied, including but not limited to the warranties or merchantability, fitness for a particular purpose and noninfringement. In no event shall we be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or documentation or the use or other dealings in the software or documentation. Any test results or performance figures are indicative and will not reflect performance under all conditions. Any reference to any third party or third-party product, service or other resource is not an endorsement or recommendation by Block.one. We are not responsible, and disclaim any and all responsibility and liability, for your use of or reliance on any of these resources. Third-party resources may be updated, changed or terminated at any time, so the information here may be out of date or inaccurate.

5.1.5、eosio.contracts

About System Contracts The EOSIO blockchain platform is unique in that the features and characteristics of the blockchain built on it are flexible, that is, they can be changed, or modified completely to suit each business case requirement. Core blockchain features such as consensus, fee schedules, account creation and modification, token economics, block producer registration, voting, multi-sig, etc., are implemented inside smart contracts which are deployed on the blockchain built on the EOSIO platform.

Block.one implements and maintains EOSIO open source platform which contains, as an example, the system contracts encapsulating the base functionality for an EOSIO based blockchain. This document will detail each one of them, eosio.bios, eosio.system, eosio.msig, eosio.token, eosio.wrap along with a few other main concepts.

Concepts System contracts, system accounts, priviledged accounts At the genesis of an EOSIO based blockchain, there is only one account present: eosio, which is the main system account. There are other system accounts, which are created by eosio, and control specific actions of the system contracts mentioned earlier. Note that we are introducing the notion of system contract/s and system account/s. Also note that privileged accounts are accounts which can execute a transaction while skipping the standard authorization check. To ensure that this is not a security hole, the permission authority over these accounts is granted to eosio.prods.

As you just learned the relation between an account and a contract, we are adding here that not all system accounts contain a system contract, but each system account has important roles in the blockchain functionality, as follows:

Account Priviledged Has contract Description eosio Yes It contains the eosio.system contract The main system account on an EOSIO based blockchain.

eosio.msig Yes It contains the eosio.msig contract Allows the signing of a multi-sig transaction proposal for later execution if all required parties sign the proposal before the expiration time. eosio.wrap Yes It contains the eosio.wrap contract. Simplifies block producer superuser actions by making them more readable and easier to audit. eosio.token No It contains the eosio.token contract. Defines the structures and actions allowing users to create, issue, and manage tokens on EOSIO based blockchains. eosio.names No No The account which is holding funds from namespace auctions. eosio.bpay No No The account that pays the block producers for producing blocks. It assigns 0.25% of the inflation based on the amount of blocks a block producer created in the last 24 hours. eosio.prods No No The account representing the union of all current active block producers permissions. eosio.ram No No The account that keeps track of the SYS balances based on users actions of buying or selling RAM. eosio.ramfee No No The account that keeps track of the fees collected from users RAM trading actions: 0.5% from the value of each trade goes into this account. eosio.saving No No The account which holds the 4% of network inflation. eosio.stake No No The account that keeps track of all SYS tokens which have been staked for NET or CPU bandwidth. eosio.vpay No No The account that pays the block producers accordingly with the votes won. It assigns 0.75% of inflation based on the amount of votes a block producer won in the last 24 hours. eosio.rex No No The account that keeps track of fees and balances resulted from REX related actions execution. RAM RAM is the memory (space, storage) where the blockchain stores data. If your contract needs to store data on the blockchain, like in a database, then it can store it in the blockchain's RAM using either a multi-index table, which can be found explained here and here or a singleton, its definition can be found here and a sample of its usage here. The EOSIO-based blockchains are known for their high performance, which is achieved also because the data stored on the blockchain is using RAM as the storage medium, and thus access to blockchain data is very fast, helping the performance benchmarks to reach levels no other blockchain has been able to. RAM is a very important resource because of the following reasons: it is a limited resource, each EOSIO-based blockchain can have a different policy and rules around RAM, for example the public EOS blockchain started with 64GB of RAM and after that the block producers decided to increase the memory with 1KiB (1024 bytes) per day, thus increasing constantly the supply of RAM for the price of RAM to not grow too high because of the increased demand from blockchain applications; also RAM it is used in executing many actions that are available on the blockchain, creating a new account for example (it needs to store in the blockchain memory the new account's information), also when an account accepts a new type of token a new record has to be created somewhere in the blockchain memory that holds the balance of the new token accepted, and that memory, the storage space on the blockchain, has to be purchased either by the account that transfers the token or by the account that accepts the new token type. RAM is a scarce resource priced according to the unique Bancor liquidity algorithm which is implemented in the system contract here.

CPU CPU is processing power, the amount of CPU an account has is measured in microseconds, it is referred to as "cpu bandwidth" on the cleos get account command output and represents the amount of processing time an account has at its disposal when pushing actions to a contract.

NET As CPU and RAM, NET is also a very important resource in EOSIO-based blockchains. NET is the network bandwidth measured in bytes of transactions and it is referred to as "net bandwidth" on the cleos get account command. This resource like CPU must be staked so that a contract's transactions can be executed.

Stake On EOSIO based blockchains, to be able to deploy and then interact with a smart contract via its implemented actions it needs to be backed up by resources allocated on the account where the smart contract is deployed to. The three resource types an EOSIO smart contract developer needs to know about are RAM, CPU and NET. You can stake CPU and NET and you can buy RAM. You will also find that staking/unstaking is at times referred to as delegating/undelegating. The economics of staking is also to provably commit to a promise that you'll hold the staked tokens, either for NET or CPU, for a pre-established period of time, in spite of inflation caused by minting new tokens in order to reward BPs for their services every 24 hours.

Vote In a EOSIO-based network the blockchain is kept alive by nodes which are interconnected into a mesh, communicating with each other via peer to peer protocols. Some of these nodes are elected, via a voting process, by the token holders to be producer nodes. They produce blocks, validate them and reach consensus on what transactions are allowed in each block, their order, and what blocks are finalized and stored forever in the blockchain memory. This way the governance, the mechanism by which collective decisions are made, of the blockchain is achieved through the 21 active block producers which are appointed by token holders' votes. It's the 21 active block producers which continuously create the blockchain by creating blocks, and securing them by validating them, and reaching consensus. Consensus is reached when 2/3+1 active block producers agree on validity of a block, that is all transactions contained in it and their order.

System contracts defined in eosio.contracts eosio.bios eosio.system eosio.msig eosio.token eosio.wrap eosio.bios system contract The eosio.bios is the first sample of system smart contract provided by block.one through the EOSIO platform. It is a minimalist system contract because it only supplies the actions that are absolutely critical to bootstrap a chain and nothing more. This allows for a chain agnostic approach to bootstrapping a chain.

The actions implemented and publicly exposed by eosio.bios system contract are: setpriv, setalimits, setglimits, setprods, setparams, reqauth, setabi.

Action name Action description setpriv Set privilege status for an account. setalimits Set the resource limits of an account setglimits Not implemented yet. setprods Set a new list of active producers, that is, a new producers' schedule. setparams Set the blockchain parameters. reqauth Check if an account has authorization to access the current action. setabi Set the abi for a contract identified by an account name. The above actions are enough to serve the functionality of a basic blockchain, however, a keen eye would notice that the actions listed above do not allow for creation of an account, nor updating permissions, and other important features. As we mentioned earlier, this sample system contract is minimalist in its implementation, therefore it relies also on some native EOSIO actions. These native actions are not implemented in the eosio.bios system contract, they are implemented at the EOSIO chain core level. In the eosio.bios contract they are simply declared and have no implementation, so they can show in the contracts ABI definition, and therefore users can push these actions to the account that holds the eosio.bios contract. When one of these actions are pushed to the chain, to the eosio.bios contract account holder, via a cleos command for example, the corresponding native action is executed by the blockchain first, see the code here, and then the eosio.bios contract apply method is invoked, see the code here, but having no implementation and not being part of the EOSIO_DISPATCH, at the contract level, this action will be a NOP, it will do nothing when called from core EOSIO code.

Below are listed the actions which are declared in the eosio.bios contract, mapped one-to-one with the native EOSIO actions, but having no implementation at the contract level:

Action name Description newaccount Called after a new account is created. This code enforces resource-limit rules for new accounts as well as new account naming conventions. updateauth Updates the permission for an account. deleteauth Delete permission for an account. linkauth Assigns a specific action from a contract to a permission you have created. unlinkauth Assigns a specific action from a contract to a permission you have created. canceldelay Allows for cancellation of a deferred transaction. onerror Called every time an error occurs while a transaction was processed. setcode Allows for update of the contract code of an account. eosio.system system contract The eosio.system contract is another smart contract that Block.one provides an implementation for as a sample system contract. It is a version of eosio.bios only this time it is not minimalist, it contains more elaborated structures, classes, methods, and actions needed for an EOSIO based blockchain core functionality:

Users can stake tokens for CPU and Network bandwidth, and then vote for producers or delegate their vote to a proxy. Producers can register in order to be voted for, and can claim per-block and per-vote rewards. Users can buy and sell RAM at a market-determined price. Users can bid on premium names. A resource exchange system, named REX, allows token holders to lend their tokens,

and users to rent CPU and NET resources in return for a market-determined fee. The actions implemented and publicly exposed by the eosio.system system contract are presented in the table below. Just like the eosio.bios sample contract there are a few actions which are not implemented at the contract level (newaccount, updateauth, deleteauth, linkauth, unlinkauth, canceldelay, onerror, setabi, setcode), they are just declared in the contract so they will show in the contract's ABI and users will be able to push those actions to the chain via the account holding the 'eosio.system' contract, but the implementation is at the EOSIO core level. They are referred to as EOSIO native actions.

Action name Action description newaccount Called after a new account is created. This code enforces resource-limits rules for new accounts as well as new account naming conventions. updateauth Updates the permission for an account. deleteauth Delete permission for an account. linkauth Assigns a specific action from a contract to a permission you have created. unlinkauth Assigns a specific action from a contract to a permission you have created. canceldelay Allows for cancellation of a deferred transaction. onerror Called every time an error occurs while a transaction was processed. setabi Allows for updates of the contract ABI of an account. setcode Allows for updates of the contract code of an account. init Initializes the system contract for a version and a symbol. setram Set the ram supply. setramrate Set the ram increase rate. setparams Set the blockchain parameters. setpriv Set privilege status for an account (turn it on/off). setalimits Set the resource limits of an account. setacctram Set the RAM limits of an account. setacctnet Set the NET limits of an account. setacctcpu Set the CPU limits of an account. rmvproducer Deactivates a producer by name, if not found asserts. updtrevision Updates the current revision. bidname Allows an account to place a bid for a name. bidrefund Allows an account to get back the amount it bid so far on a name. deposit Deposits core tokens to user REX fund. withdraw Withdraws core tokens from user REX fund. buyrex Buys REX in exchange for tokens taken out of user's REX fund by transferring core tokens from user REX fund and converting them to REX stake. unstaketorex Use staked core tokens to buy REX. sellrex Sells REX in exchange for core tokens by converting REX stake back into core tokens at current exchange rate. cnclrexorder Cancels unfilled REX sell order by owner if one exists. rentcpu Use payment to rent as many SYS tokens as possible as determined by market price and stake them for CPU for the benefit of receiver, after 30 days the rented core delegation of CPU will expire. rentnet Use payment to rent as many SYS tokens as possible as determined by market price and stake them for NET for the benefit of receiver, after 30 days the rented core delegation of NET will expire. fundcpuloan Transfers tokens from REX fund to the fund of a specific CPU loan in order to be used for loan renewal at expiry. fundnetloan Transfers tokens from REX fund to the fund of a specific NET loan in order to be used for loan renewal at expiry. defcpuloan Withdraws tokens from the fund of a specific CPU loan and adds them to the REX fund. defnetloan Withdraws tokens from the fund of a specific NET loan and adds them to the REX

fund. updaterex Updates REX owner vote weight to current value of held REX tokens. consolidate Consolidates REX maturity buckets into one bucket that cannot be sold before 4 days. mvtosavings Moves a specified amount of REX to savings bucket. mvfrsavings Moves a specified amount of REX from savings bucket. rexexec Processes max CPU loans, max NET loans, and max queued sellrex orders. Action does not execute anything related to a specific user. closerex Deletes owner records from REX tables and frees used RAM. Owner must not have an outstanding REX balance. buyrambytes Increases receiver's ram in quantity of bytes provided. buyram Increases receiver's ram quota based upon current price and quantity of tokens provided. sellram Reduces quota my bytes and then performs an inline transfer of tokens to receiver based upon the average purchase price of the original quota. delegatebw Stakes SYS from the balance of one account for the benefit of another. undelegatebw Decreases the total tokens delegated by one account to another account and/or frees the memory associated with the delegation if there is nothing left to delegate. refund This action is called after the delegation-period to claim all pending unstaked tokens belonging to owner. regproducer Register producer action, indicates that a particular account wishes to become a producer. unregprod Deactivate the block producer with specified account. voteproducer Votes for a set of producers. This action updates the list of producers voted for, for given voter account. regproxy Set specified account as proxy. onblock This special action is triggered when a block is applied by the given producer and cannot be generated from any other source. claimrewards Claim block producing and vote rewards for block producer identified by an account. eosio.msig system contract The eosio.msig allows for the creation of proposed transactions which require authorization from a list of accounts, approval of the proposed transactions by those accounts required to approve it, and finally, it also allows the execution of the approved transactions on the blockchain.

The workflow to propose, review, approve and then executed a transaction is describe in details here, and in short it can be described by the following:

first you create a transaction json file, then you submit this proposal to the eosio.msig contract, and you also insert the account permissions required to approve this proposal into the command that submits the proposal to the blockchain, the proposal then gets stored on the blockchain by the eosio.msig contract, and is accessible for review and approval to those accounts required to approve it, after each of the appointed accounts required to approve the proposed transactions reviews and approves it, you can execute the proposed transaction. The eosio.msig contract will execute it automatically, but not before validating that the transaction has not expired, it is not cancelled, and it has been signed by all the permissions in the initial proposal's required permission list. These are the actions implemented and publicly exposed by the eosio.msig contract: |Action name|Action description| |---|---| |propose|Creates a proposal containing one

transaction.| |approve|Approves an existing proposal.| |unapprove|Revokes approval of an existing proposal.| |cancel|Cancels an existing proposal.| |exec|Allows an account to execute a proposal.| |invalidate|Invalidate proposal.|

eosio.token system contract The eosio.token contract defines the structures and actions that allow users to create, issue, and manage tokens for EOSIO based blockchains.

These are the public actions the eosio.token contract is implementing: |Action name|Action description| |---|---| |create|Allows an account to create a token in a given supply amount.| |issue|This action issues to an account a specific quantity of tokens.| |open|Allows a first account to create another account with zero balance for specified token at the expense of first account.| |close|This action is the opposite for open action, it closes the specified account for specified token.| |transfer|Allows an account to transfer to another account the specified token quantity. One account is debited and the other is credited with the specified token quantity.| |retire|This action is the opposite for create action. If all validations succeed, it debits the specified amount of tokens from the total balance.|

The eosio.token sample contract demonstrates one way to implement a smart contract which allows for creation and management of tokens. This contract gives anyone the ability to create a token. It is possible for one to create a similar contract which suits different needs. However, it is recommended that if one only needs a token with the above listed actions, that one uses the eosio.token contract instead of developing their own.

The eosio.token contract class also implements two useful public static methods: get_supply and get_balance. The first allows one to check the total supply of a specified token, created by an account and the second allows one to check the balance of a token for a specified account (the token creator account has to be specified as well).

The eosio.token contract manages the set of tokens, accounts and their corresponding balances, by using two internal multi-index structures: the accounts and stats. The accounts multi-index table holds, for each row, instances of account object and the account object holds information about the balance of one token. If we remember how multi-index tables work, see here, then we understand also that the accounts table is scoped to an eosio account, and it keeps the rows indexed based on the token's symbol. This means that when one queries the accounts multi-index table for an account name the result is all the tokens that account holds at the moment.

Similarly, the stats multi-index table, holds instances of currency_stats objects for each row, which contains information about current supply, maximum supply, and the creator account for a symbol token. The stats table is scoped to the token

symbol. Therefore, when one queries the stats table for a token symbol the result is one single entry/row corresponding to the queried symbol token if it was previously created, or nothing, otherwise.

eosio.wrap system contract The eosio.wrap system contract allows block producers to bypass authorization checks or run privileged actions with 15/21 producer approval and thus simplifies block producers superuser actions. It also makes these actions easier to audit.

It does not give block producers any additional powers or privileges that do not already exist within the EOSIO based blockchains. As it is implemented, in an EOSIO based blockchain, 15/21 block producers can change an account's permissions or modify an account's contract code if they decided it is beneficial for the blockchain and community.

However, the current method is opaque and leaves undesirable side effects on specific system accounts, and thus the eosio.wrapcontract solves this matter by providing an easier method of executing important governance actions.

The only action implemented by the eosio.wrap system contract is the exec action. This action allows for execution of a transaction, which is passed to the exec method in the form of a packed transaction in json format via the 'trx' parameter and the executer account that executes the transaction. The same executer account will also be used to pay the RAM and CPU fees needed to execute the transaction.

Why is it easier for governance actions to be executed via this contract? The answer to this question is explained in detailed here

5.2、Examples

5.2.1、Examples

eosio-java-android-example-app: Application demonstrating integration with EOSIO-based blockchains using EOSIO SDK for Java eosio-swift-ios-example-app: Application demonstrating integration with EOSIO-based blockchains using EOSIO SDK for Swift tropical-example-web-app: An example for developers showing an application built on EOSIO combining UAL, Manifest Spec, and Ricardian Contracts

5.3、Tools

5.3.1、Tools

eosio-explorer: An application providing Web GUI to communicate with EOSIO blockchain in a local development environment eosio-toppings: A monorepo composed of the various packages which work together to create a web-based development tool to help users create applications on the EOSIO blockchain eosio-web-ide: EOSIO Quickstart Web IDE lets developers start experiment building applications on EOSIO platform in a matter of minutes

5.4、Javascript SDK

5.4.1、Javascript SDK

eosjs: A Javascript library which provides an API for integrating with EOSIO-based blockchains using the EOSIO Nodeos RPC API eosjs-keygen: A Javascript library for managing keys in local storage

5.5、Swift SDK

5.5.1、Swift SDK

eosio-swift: An API for integrating with EOSIO-based blockchains using the EOSIO RPC API eosio-swift-abieos-serialization-provider: A pluggable serialization provider for EOSIO SDK for Swift. eosio-swift-ecc: A library for working with public and private keys, cryptographic signatures, encryption/decryption, etc. as part of the EOSIO SDK for Swift family of libraries eosio-swift-reference-ios-authenticator-signature-provider: A pluggable signature provider for EOSIO SDK for Swift eosio-swift-softkey-signature-provider: An example pluggable signature provider for EOSIO SDK for Swift. It allows for signing transactions using in-memory K1 keys eosio-swift-vault-signature-provider: A pluggable signature provider for EOSIO SDK for Swift eosio-swift-vault: An utility library for working with public/private keys and signing with Apple's Keychain and Secure Enclave

5.6、Java SDK

5.6.1、Java SDK

eosio-java-android-abieos-serialization-provider: A pluggable serialization provider for EOSIO SDK for Java eosio-java-android-rpc-provider: An Android RPC provider implementation for use within EOSIO SDK for Java as a plugin. eosio-java-softkey-signature-provider: An example pluggable signature provider for EOSIO SDK for Java eosio-android-keystore-signature-provider: An example pluggable signature provider for EOSIO SDK for Java written in Kotlin

5.7、EOSIO Labs

5.7.1、EOSIO Labs

eosjs-ios-browser-signature-provider-interface: A Signature Provider Interface for communicating with an authenticator from iOS Safari using the EOSIO Authentication Transport Protocol Specification eosjs-ledger-signature-provider: A SignatureProvider for communicating with eosjs from a Ledger device eosjs-signature-provider-interface: An abstract class that implements the EOSJS Signature Provider interface, and provides helper methods for interacting with an authenticator using the EOSIO Authentication Transport Protocol Specification eosjs-window-message-signature-provider-interface: A Signature Provider Interface for communicating with an authenticator over the Window Messaging API using the EOSIO Authentication Transport Protocol Specification ual-authenticator-walkthrough: A tutorial walks through the steps required to create a UAL for Ledger Authenticator ual-reactjs-renderer: A library provides a React renderer around the Universal Authenticator Library

6、Reference

6.1、Nodeos RPC API Refer

6.1.1、Nodeos RPC API Reference

Nodeos Chain API: Provides access to the blockchain information and interaction with the blockchain Nodeos Producer API: Provides access to a producer node Nodeos Net API: Provides access to the blockchain's network Nodeos DB Size API: Provides access to the blockchain's database Nodeos Test Control API: Provides control of nodeos shutdown if some test conditions are met, for testing purposes

6.2、SDK API References

6.2.1、SDK API References

eosjs Javascript API: Provides integration with EOSIO-based blockchains using the EOSIO Nodeos RPC API eosio-swift Swift API: Provides integration with EOSIO-based blockchains using the EOSIO Nodeos RPC API eosio-java Java API: Provides integration with EOSIO-based blockchains using the EOSIO Nodeos RPC API

6.3、Smart Contract Actio

6.3.1、Smart Contract Action References

eosio.system: Contract defines the structures and actions needed for blockchain's core functionality eosio.token: Contract defines the structures and actions that allow users to create, issue, and manage tokens on eosio based blockchains eosio.msig: Contract defines the structures and actions needed to manage the proposals and approvals on blockchain eosio.wrap: Contract simplifies Block Producer superuser actions by making them more readable and easier to audit eosio.bios: Contract defines the structures and actions needed for blockchain's basic core functionality

7、Resources

7.1、EOSIO Developers Com

7.1.1、EOSIO Developers Community

EOSIO StackExchange: A question and answer site for users and developers Developers Telegram: For EOSIO development discussion only, all chains welcome. No memes. No prices. No promotion

7.2、Technical Resources

7.2.1、Technical Resources

EOSIO Github: The place where all EOSIO open source software can be reviewed, used and/or forked EOSIO Deprecation Schedule: Table tracking significant deprecations in EOSIO software EOSIO White Paper: The EOSIO software introduces a new blockchain architecture designed to enable vertical and horizontal scaling of decentralized applications Elemental Battles: Learn how to build your first blockchain based game on EOSIO software

7.3、EOSIO Testnet

7.3.1、EOSIO Testnet

EOSIO Testnet: Block.one’ s official EOSIO Testnet

7.4、News

7.4.1、News

EOSIO News: Latest news about EOSIO platform brought to you by block.one EOSIO Twitter: Latest tweets about EOSIO platform brought to you by block.one EOSIO Medium: Latest articles about EOSIO platform brought to you by block.one

7.5、Events

7.5.1、Events

EOSIO Hackathons: Past and future Hackathons information and news EOSIO Bug Bounty Program: A Bug Bounty program offering to external individuals to receive compensation for reporting EOSIO bugs EOSIO Webinars: Introduction to EOSIO Blockchain Development by Block.one

7.6、General

7.6.1、General

EOSIO FAQ: Frequently Asked Questions

8、Other

8.1、Get Involved

We appreciate your interest in contributing to the EOSIO platform! We always welcome contributions from our community to make our code and docs better.

Get Involved with EOSIO Community and Code Get Involved with EOSIO Documentation Get Involved with EOSIO Community and Code The following are different ways you can get involved with the EOSIO developers community and EOSIO repositories. You can find repositories to contribut to in software manuals and on the EOSIO Organization on GitHub.

For more information about the contribution guidelines of a particular repository, look for the "Contributing" link in the right-hand sidebar of the documentation.

Contributing link location

Developers Community Involvement EOSIO Stack Exchange Engage the EOSIO developer community over EOSIO Stack Exchange. You can ask a question related to EOSIO or answer questions and share your EOSIO knowledge with the community.

EOSIO Developer Telegram Instantly engage the EOSIO developer community over EOSIO Telegram instant messenger. Get involved by sharing your expertise in EOSIO development discussions, sharing community tools and projects that can assist EOSIO developers, and more.

Get Involved with EOSIO Code EOSIO is a large open source project with over 500 repositories for its stack and core components. You can visit the repositories in the EOSIO Organization on Github and start contributing to the code base.

There are many ways you can contribute to EOSIO code. You can report an issue, submit a pull request, and review pull requests.

Reporting an Issue If you're about to raise an issue because you think you've found a problem with the stack, or you'd like to make a request for a new feature in the codebase, or any other reason, please read this first.

The GitHub issue tracker is the preferred channel for bug reports, feature requests, and submitting pull requests, but please respect the following restrictions:

Please search for existing issues. Help us keep duplicate issues to a minimum by checking to see if someone has already reported your problem or requested your idea. Please be civil. Keep the discussion on topic and respect the opinions of others. See our Code of Conduct. Bug Reports A bug is a demonstrable problem that is caused by the code in the repository and can be reported in the form of a new issue.

Guidelines for bug reports:

Use the GitHub issue search — check if the issue has already been reported. Check if the issue has been fixed — look for closed issues in the current milestone or try to reproduce it using the latest develop branch. A good bug report or a new issue shouldn't leave others needing to chase you up for more information. Be sure to include the details of your environment and relevant tests that demonstrate the failure.

Feature Requests Feature requests are welcome. Before you submit one be sure to have:

Use the GitHub search and check the feature hasn't already been requested. Take a moment to think about whether your idea fits with the scope and aims of the project. Remember, it's up to you to make a strong case to convince the project's leaders of the merits of this feature. Please provide as much detail and context as possible, this means explaining the use case and why it is likely to be common. Get Involved with EOSIO Documentation The EOSIO documentation is written in Markdown. You can either submit a Pull Request for quick edits or file a new docs issue. For Markdown syntax usage, see the Markdown Syntax Documentation.

Quick Edits Quick edits allow for faster reporting of minor content issues from typographical errors to omissions.

Most of the pages on the EOSIO developer portal provide an Edit link which redirects you to the source file on Github. If you don't see the Edit button on the documentation page, that means the specific page is not available to be changed.

To make a quick edit:

Go to the specific page where you have identified an editorial need and click the Edit icon at the top right of the page. Quick Edit Icon

Fork the repository to suggest changes. Click Fork this repository. Fork the Repository

If you are not signed in to your Github account, it will redirect you to the Github login page. Sign in with your username and password to continue. If you are new to Github, create a new account.

Make the suggested changes in the web editor using Markdown syntax. Click the Preview changes tab to see the preview of the content. After suggesting your changes, scroll down to the bottom on the page. Enter a title and a description of the changes you made and click Propose file change. Create a Pull Request by entering a title and a description. Click Create pull request to submit your suggestion to us.

If you are new to Github, see About Pull Requests for more information.

Congrats! You have submitted your suggestion. Our team will review your pull request and merge it if it's a valid change.

Filing a Docs Issue Filing an issue in a docs repository means reporting a documentation bug in the form of a Github issue.

To file a new issue:

Go to the specific page where you have identified an area of improvement and click Request Changes at the top right of the page. New Issue Icon You will be directed to the Issues tab of the specific repository with an editable Issue form.

New Issue Form

If you are not signed in to your Github account, it will redirect you to the Github login page. Sign in with your username and password to continue. If you do not have a Github account, create a new account.

Enter the issue title and describe the issue with a proposed solution if you have using Markdown syntax.

If you are filing an issue for the first time, review the contributing guidelines of the repository. Contribution Guidelines

Click Submit new issue to submit the issue to the repository. Code of Conduct While contributing, please be respectful and constructive, so that participation in our project is a positive experience for everyone.

Examples of behavior that contributes to creating a positive environment include:

Using welcoming and inclusive language Being respectful of differing viewpoints and experiences Gracefully accepting constructive criticism Focusing on what is best for the community Showing empathy towards other community members Examples of unacceptable behavior include:

The use of sexualized language or imagery and unwelcome sexual attention or advances Trolling, insulting/derogatory comments, and personal or political attacks Public or private harassment Publishing others' private information, such as a physical or electronic address, without explicit permission Other conduct which could reasonably be considered inappropriate in a professional setting Contributor License & Acknowledgments Whenever you make a contribution to this project, you license your contribution under the same terms as set out in LICENSE, and you represent and warrant that you have the right to license your contribution under those terms. Whenever you make a contribution to this project, you also certify in the terms of the Developer's Certificate of Origin set out below:

Developer Certificate of Origin Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 1 Letterman Drive Suite D4700 San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

8.2、Glossary

ABI The Application Binary Interface (ABI) is a JSON-based description on how to convert user actions between their JSON and binary representations. The ABI may also describe how to convert the database state to/from JSON. Once you have described your contract via an ABI this allows developers and users to interact with your contract seamlessly via JSON.

Synonyms

Application Binary Interface Account An account is a unique identifier and a requirement to interact with an EOSIO blockchain. Unlike most other cryptocurrencies, transfers are sent to a human readable account name instead of a public key, while keys attributed to the account are used to sign transactions.

Account Name An account name is a human-readable identifier that is stored on the blockchain. Standard account names can only contain the characters .abcdefghijklmnopqrstuvwxyz12345. a-z (lowercase), 1-5 and . (period), must start with a letter and must be 12 characters in length. Non-standard account names have all restrictions of the standard ones except they can have less than 12 characters in length.

Account Name Bidding For non-standards accounts, those under 12 characters, EOSIO based blockchains provide a system where blockchain users can bid on accounts by submitting the bids via an auction process. Only one account name with the highest bidding price can be auctioned off every 24 hours among all submitted account names, that is, the system considers a bidding to be successful only when the bidding price is the highest among all other account bidding prices submitted.

Action Functionality exposed by a smart contract that is exercised by passing the correct parameters via an approved transaction to an EOSIO network.

Related

Block Bancor Relay EOSIO adopts a free-market approach to allocating scarce resources. To facilitate this market, the eosio system contract allows users to buy RAM from the system and sell RAM back to the system in exchange for the blockchain's

native tokens. This provides liquidity in the RAM market while facilitating price discovery. The less unallocated RAM available to the market maker the higher the market maker prices the remaining RAM. The algorithm used for this market maker is known as a Bancor Relay. A Bancor Relay does not set the price of RAM. It only offers to buy and sell at previously established market rates. Anytime the current market rate is different than the current price offered by the Bancor Relay, traders will buy or sell RAM pushing to closer to the market determined price.

Synonyms

Bancor Algorithm Block A confirmable unit of a blockchain. Each block contains zero or more transactions, as well as a cryptographic connection to all prior blocks. When a block becomes "irreversibly confirmed" it's because a supermajority of block producers have agreed that the given block contains the correct transactions. Once a block is irreversibly confirmed, it becomes a permanent part of the immutable blockchain.

Related

Action Block Header A part of the block which holds metadata related to the block. In an EOSIO block header this includes things like the transaction Merkle root, the action Merkle root, the producer who produced the block, the block id of the previous block, the block id of the current block, and the block timestamp.

Related

Block Merkle Tree Block Log The block log is an append only log of blocks written to disk and contains all the irreversible blocks.

Block Producer A Block Producer is an identifiable entity composed of one or more individuals that express interest in participating in running an EOSIO network. By participating it is meant these entities will provide a full node, gather transactions, verify their validity, add them into blocks, and propose and confirm these blocks. A Block producer is generally required to have experience with system administration and security as it is expected that their full-node have constant availability.

Block Producer Schedule The list of block producers who currently have the possibility of being selected to produce the next block. This list changes with every new block.

Related

Block Producer Block-Producing Node A full node running nodeos that is actively producing blocks. A producing node, if voted out, will become a producing node on "standy". Standby producers will produce blocks and will be rewarded proportionally to their vote stake.

Blockchain Application A blockchain application is a software application that has integrated a blockchain in its architecture as the storage layer for part of, or, all of its data. This includes software applications that do not own their own contract on the blockchain and instead only interact with the system contracts of the blockchain.

Byzantine Fault Tolerance In the context of distributed systems, Byzantine Fault Tolerance (BFT) is the ability of a distributed computer network to function as desired and correctly reach a sufficient consensus despite malicious components (nodes) of the system failing or propagating incorrect information to other peers. In an EOSIO based blockchain BFT is achieved using a combination of Delegated Proof of Stake, the last irreversible block, and the fact that a producer cannot sign two blocks with the same block number.

Related

Irreversible Block DPoS CPU CPU is processing power granted to an account by an EOSIO based blockchain. The amount of CPU an account has is measured in microseconds, and represents the amount of processing time an account has at its disposal when executing its actions. CPU is recalculated after each block is produced, based on the amount of system tokens the account staked for CPU bandwidth in proportion to the amount of total system tokens staked for CPU bandwidth at that time.

Chain State The chain state or database is a memory mapped file, storing the blockchain state of each block (account details, deferred transactions, transactions, data stored using multi index tables in smart contracts, etc.). Once a block becomes irreversible the chain state is not cached anymore.

Chain State The chain state (or "database" as it is often called) is a memory mapped file, which stores the blockchain state of each block (account details, deferred transactions, transactions, data stored using multi index tables in smart contracts, etc.). Once a block becomes irreversible the chain state is no longer cached.

Cleos cleos is a command line tool that interfaces with the REST api exposed by nodeos, in other words cleos is the command line tool through which you can interface with an EOSIO based blockchain; cleos contains documentation for all of its commands. For a list of all commands known to cleos, simply run it with no arguments. cleos = command line + eos

Confirmed Transaction On completion of the transaction, a transaction receipt is generated. Receiving a transaction hash does not mean that the transaction has been confirmed, it only means that the node accepted it without error, which also means that there is a high probability other producers will accept it. A transaction is considered confirmed when a nodeos instance has received, processed, and written it to a block on the blockchain, i.e. it is in the head block or an earlier block.

Core The core is used to refer to the EOSIO blockchain native components, e.g. native actions, chain libraries, nodeos daemon, etc. The core is the EOSIO platform on which EOSIO based blockchains can be instantiated and tailored by means of deploying smart contracts (including the system smart contracts). Therefore, the system smart contracts are not considered part of the core or native blockchain implementation.

Cryptographic Hash A cryptographic hash function is a hash function which takes an input (or 'message') and returns a fixed-size alphanumeric string. The alphanumeric string is called the 'hash value', 'message digest', 'digital fingerprint', 'digest' or 'checksum'.

Custom Permission In addition to the native permissions, owner and active, an account can possess custom named permissions that are available to further extend account management. Custom permissions are incredibly flexible and address numerous possible use cases when implemented. Custom permissions are arbitrary and impotent until they have been linked to an action.

DAC A DAC is an entity that utilizes a combination of automation and input from stakeholders, normally in the form of votes. They are often governed by code which describes the purpose of the organisation.

Synonyms

Decentralized Autonomous Company DLT Distributed Ledger Technologies. A distributed ledger (also called a shared ledger, or referred to as distributed ledger technology) is a consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, or institutions.

Synonyms

Distributed Ledger Technology DPoS DPoS stands for "Delegated Proof of Stake" and is a consensus algorithm initially developed by Daniel Larimer in 2013 for Bitshares. It's sometimes referred to as "Democracy as Proof of Stake"

Synonyms

Delegated Proof-of-Stake Deferred Action Deferred actions are actions sent to a peer action that are scheduled to run, at best, at a later time, at a block producer's discretion. There is no guarantee that a deferred action will be executed. From the perspective of the originating action, i.e., the action that creates the deferred action, it can only determine whether the create request was submitted successfully or whether it failed (if it fails, it will fail immediately). Deferred actions carry the authority of the contract that sends them. A deferred action can also be cancelled by another action.

Deserialization Deserialization is the reverse process of serialization. It turns a stream of bytes into an object in memory. EOSIO structures are enhanced with two operators which implement the serialization and deserialization of data to and from the database.

Digital Signature A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender (authentication), and that the message was not altered in transit (integrity). Digital signatures are a standard element of most cryptographic protocol suites, and are commonly used for software distribution, financial transactions, contract management software, and in other cases where it is important to detect forgery or tampering.

Dispatcher Every smart contract must provide an apply action handler. The apply action handler is a function that listens to all incoming actions and performs the desired behavior. In order to respond to a particular action, code is required to identify and respond to specific action requests. apply uses the receiver, code, and action input parameters as filters to map to the desired functions that implement particular actions. To simplify the work for contract developers, EOSIO provides the EOSIO_DISPATCH macro, which encapsulates the lower level action mapping details of the apply function, enabling developers to focus on their application implementation.

Dispatcher Hooks In addition to actions and notification handlers, two "hooks" are available. The pre_dispatch hook will fire when the dispatcher is run and allow the smart contract to do some pre-validation and exit early if need be by returning false. If the function returns true then the dispatcher continues to dispatch the actions or notification handlers. The post_dispatch hook will only fire when the dispatcher has failed to match any notification handlers, this allows the user to do some meaningful last ditch validation.

EOSIO Types EOSIO source code defines a list of types which ease the developer's work when writing smart contracts, plugins, or when extending the EOSIO source code. Example types include account_name, permission_name, table_name, action_name, scope_name, weight_type, public_key, etc.

Genesis Block The genesis block is the very first block in the EOSIO blockchain. The subsequent block added after the genesis block becomes block 1 and continues the sequence. The genesis block lays the foundation for other blocks to be added to form a blockchain.

Genesis Node The genesis node is the first node in the blockchain network. The genesis node is used to perform a set of actions such as creating system accounts, initializing system, and token contracts in order to create a fully-functional blockchain with varying capabilities such as governance, resource allocation, and more.

Synonyms

Single Producer Node block 1 Governance Blockchain governance is a lot like other kinds of governance, except that it's underpinned by smart contracts and transparent voting on the blockchain. Governance, the mechanism by which collective decisions are made, of an EOSIO based blockchain is achieved through 21 active block producers which are appointed by token holder votes. The 21 active block producers continuously create the blockchain via block creation, secure the blocks by validating them, and reach consensus on the state of the blockchain as a whole. Consensus is reached when 2/3+1 active block producers agree on validity of a block , therefore on all transactions contained in it and their order.

Related

Irreversible Block Head Block The head block is the last block written to the blockchain, stored in reversible_blocks.

Indices In the context of a multiple index table, an index is a particular ordering of the elements in the table. Multiple index indices allow the same data in one table to be viewed as different data structures by specifying the specific index on the table. EOSIO multiple index tables allow for up to 16 unique indices.

Inline Action Inline actions request other actions that need to be executed as part of the original calling action. Inline actions operate with the same scopes and authorities of the original transaction, and are guaranteed to execute with the current transaction. These can effectively be thought of as nested transactions within the calling transaction. If any part of the transaction fails, the inline actions will unwind with the rest of the transaction.

Irreversible Block A block is considered irreversible (i.e. immutable) on an EOSIO based blockchain when 2/3rd of the currently elected block producers have acknowledged it.

Keosd keosd is the component that securely stores EOSIO keys in wallets. keosd = key + eos

Larimer 1/10000 of an EOS (token) 0.0001 EOS

Merkle Tree A Merkle tree is a tree in which every leaf node is labelled with the hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures.

Synonyms

Hash tree Multi-Index EOSIO wraps the boost multi-index library to provide in memory data persistence. A subset of the functionality provided by the boost multi-index is provided in the EOSIO multi-index.

Multiple Index Table Multiple Index Tables, or Multi Index Tables, are a way to cache state and/or data in RAM for fast access. Multi index tables support create, read, update, and delete (CRUD) operations, something which the blockchain doesn't (it only supports create and read). Multi index tables are stored in EOSIO RAM and each smart contract using a multi index table reserves a partition of the RAM cache. Access to each partition is controlled using the table name, code, and scope, and can have up to 16 indexes or indices defined.

Related

Indices Multisig Multisig or msig is a short term for multiple signatures. It’ s used to describe the case in which one requires more than one account's permission to execute a transaction. EOSIO provides the system account eosio.msig, which can be used to push onto the blockchain the multisig proposals and their corresponding account's permission required to approve the proposal. Multisig, when used properly, increases the security of an account, the security of a smart contract, and it's also the method by which Block Producers are able to affect changes within an EOSIO blockchain.

NET NET is required to store transactions on an EOSIO based blockchain. The amount of NET an account has is measured in bytes, representing the amount of transaction storage an account has at its disposal when creating a new transaction. NET is recalculated after each block is produced, based on the system tokens staked for NET bandwidth by the account. The amount allocated to an account is proportional with the total system tokens staked for NET by all accounts. Do not confuse NET with RAM, although it is also storage space, NET measures the size of the transactions and not contract state.

Nodeos nodeos is the core EOSIO node daemon that can be configured with plugins to run a node. Example uses are block production, dedicated API endpoints, and local development. nodeos = node + eos

Non-Producing Node A full node running nodeos that is only watching and verifying for itself each block, and maintaining its own local full copy of the blockchain. A non-producing node that is in the "standby pool" can, through the process of being voted in, become a Producing Node. A producing node, if voted out, will become a non-producing node. For large EOSIO changes, non-producing nodes are outside the realm of the "standby pool".

Related

Block-Producing Node Configuring a block-producing node Oracle An oracle, in the context of blockchains and smart contracts, is an agent that finds and verifies real-world occurrences and submits this information to a blockchain to be used by smart contracts.

Packed Transaction In order to transfer transaction content between nodes faster and to save storage space when storing transaction content in and EOSIO based blockchain database, the transactions are 'translated' from json into a packed form which is smaller in size. To get the packed version of a transaction one can use the cleos convert command.

Peer-to-peer Peer-to-peer computing or networking is a distributed application architecture that partitions tasks or workloads between peers. If peers are equally privileged, equipotent participants in an application, they are said to form a peer-to-peer network of nodes.

Pending Block The pending block is the block currently being built by each node. Transactions are added to the pending block as they are received and processed. The pending block becomes the head block once it is written to the blockchain. Note that a head block is initially reversible.

Pending Blocks The pending block is an in memory block containing transactions as they are processed into a block, this will/may eventually become the head block. If this instance of nodeos is the producing node then the pending block will eventually be distributed to other nodeos instances.

Permission A weighted security mechanism that determines whether or not a message is properly authorized by evaluating its signature(s) authority. Every account has two default permissions, owner and active, but can also have custom permissions to further secure communications from an account to contracts. Every permission name has a "parent." Parents possess the authority to change any of the permissions settings for any and all of their children.

Permission Threshold The sum of permission weights necessary for a signature to be considered valid.

Permission Weight A permission weight is a value given to an account for authorization purposes. This is typically used in the context of a mutli-sig to give one or more accounts more control over a multi-sig than others.

Synonyms

Authorization Permission level Permissions are arbitrary names used to define the requirements for a transaction sent on behalf of that permission. Permissions can be assigned for authority over specific contract actions by "linking authorization" or linkauth. Every account has two native named permissions, owner and active. Every

permission name has a "parent." Parents possess the authority to change any of the permission settings for any and all of their children. In addition to the native permissions, an account can possess custom named permissions that are available to further extend account management. Custom permissions are incredibly flexible and address numerous possible use cases when implemented correctly. Given this context permission level is used to identify a specific permission, either active, or owner, or a custom one.

Plugin nodeos plugins are software components that implement features that complement the native EOSIO blockchain basic implementation. They can be enabled or disabled through the nodeos configuration file or specifying them in the command line that launches the nodeos daemon.

Private Key A private key is a secret key used to sign transactions. In EOSIO, a private key's authority is determined by it's mapping to an EOSIO account name.

Private Network A private network is a production network, or a test network, to which access is private, that is, the API endpoints, and block producers connectivity URLs and IPs are private. Access to a private network is done via invitation, or, upon request to join, is granted by the owners of the private network.

Privileged Privileged accounts are accounts which can execute transactions while skipping the standard authorization check. To ensure that this is not a security hole, the permission authority over these accounts is granted to the eosio.prods account. An account can be set as privileged by sending the native action setpriv to an EOSIO based blockchain, specifying the account to be set as privileged, and providing the correct permission, or, by using cleos command line utility.

Privileged Account At the genesis of an EOSIO based blockchain, there is only one account present, eosio which is the main system account.

Public Key A publicly available key that can be authorized to permissions of an account and can be used to identify the origin transaction. A public key can be inferred from a signature.

Public Network A public network is a production network instantiated with the EOSIO platform. For a public network, the tokens have value on public markets, and all of the features required to run and maintain the network, such as consensus and governance, are enabled. To contrast a public network, in a private network typically tokens are not traded on public markets, and there is less emphasis placed on governance.

RAM RAM is required to store account information such as keys, balances, and contract state on an EOSIO based blockchain. Because the amount of RAM available to a single computer is limited by Moore’ s Law and other technological advances, RAM is

fundamentally scarce and must be purchased on a free-market inside an EOSIO based blockchain.

Related

RAM Market RAM Market In order to persist data on an EOSIO based blockchain, a user must first purchase RAM. The EOSIO RAM market uses the Bancor Relay algorithm in a system smart contract to offer to buy and sell RAM from users at previously established market rates.

REX The REX (Resource Exchange) is a CPU and Network resource rental market in which holders of the core token of a blockchain can buy and sell slices of the REX pool in the form of REX tokens. Blockchain users can then rent CPU and Network resources from the REX pool.

Read Mode An EOSIO based blockchain allows contract developers to persist state across actions, and consequently transaction, boundaries. For example the sample eosio.token contract keeps balances for all users in the database. Each instance of nodeos keeps the database in memory, so contracts can read and write data quickly. However, at any given time there can be multiple correct ways, called modes, to query that data.

Reversible Block Any block on an EOSIO based blockchain with a block number greater than the last irreversible block. Reversible blocks are blocks that are not currently guaranteed to be on the blockchain.

Related

Irreversible Block Ricardian Contract In the EOSIO based blockchain context Ricardian Contract is a digital document that accompanies a smart contract and defines the terms and conditions of an interaction between the smart contract and its users, written in human readable text, which is then cryptographically signed and verified. It is easily readable for both humans and programs, and aids in providing clarity to any situations that may arise in the interactions between smart contract and its users.

SYS SYS is the blockchain default token name for an EOSIO based blockchain. Any fork of the EOSIO open source has the option to rename it, through a very simple procedure, to any token name that meets the symbol validation rules.

Scope Scope is a region of data within a contract. Contracts can only write to regions in their own contracts but they can read from any other contract's regions. Proper scoping allows transactions to run in parallel for the same contract because they do not write to the same regions. Scope is not to be conflated with an account name, but contracts can use the same value for both for convenience.

Serialization Serialization is the process of turning an object in memory into a stream of bytes so you can store it on disk or send it over the network

Signature A signature is a mathematical scheme for demonstrating the authenticity of digital messages or documents

Smart Contract A smart contract is a computer protocol intended to facilitate, verify, or enforce the negotiation or performance of a contract.

Staking Staking is the act of locking tokens for resources on an EOSIO network. This includes but is not limited to, CPU time, RAM, and on-chain governance.

Standard Account Name Standard account names can only contain the characters .abcdefghijklmnopqrstuvwxyz12345. a-z (lowercase), 1-5 and . (period), must start with a letter and must be 12 characters in length.

Standby Pool A set of about 100 full nodes that have expressed the desire to be selected as block producers, and are capable of doing so on demand. Whenever the chain needs to replace an existing BP with a new one, the new one is drawn from the standby pool.

System Everything that is part of EOSIO which is not part of core, is referred to as system, e.g. system accounts, privileged accounts, system contracts. From an architectural point of view system components sit on top of the core/native components.

System Contract The design of the EOSIO blockchain calls for a number of smart contracts that are run at a privileged permission level in order to support functions such as block producer registration and voting, token staking for CPU and network bandwidth, RAM purchasing, multi-sig, etc. These smart contracts are referred to as the system contracts and are the following, eosio.bios, eosio.system, eosio.token, eosio.msig and eosio.wrap (formerly eosio.sudo) contracts.

Tables Tables on an EOSIO based blockchain are achieved via Multiple Index Table.

Related

Multiple Index Table Test Network A test network or testnet is an instantiation of the EOSIO platform that is intended for testing purposes. Generally, the native token has no value and is given away to developers so they can test. Some features of a testnet may be disabled such as consensus and governance.

Transaction A complete all-or-nothing change to the Blockchain. A combination of one or more actions. Usually the execution result of a Smart Contract.

Transaction Reciept On completion of a transaction, a transaction receipt is generated. This receipt takes the form of a hash. Receiving a transaction hash does not mean that

the transaction has been confirmed, it simply means that the node accepted it without error, which also means that there is a high probability other producers will accept it.

Transaction Trace Transaction trace is a log of all the actions that took place as a result of an initial action (inline actions, deferred actions, context free actions, etc), including the initial action, for all actions that are processed on the blockchain. A transaction trace for one transaction includes among other things, the transaction id, the block id that the transaction is part of, the time when the block was produced, the producer id, the time elapsed to process the transaction, and a list of all actions contained in the transactions.

Unconfirmed Transaction A transaction is considered unconfirmed as long as no nodeos instance has received, processed, and written it to a block on the blockchain, i.e. it is not in the head block or a block earlier than the head block.

WASM WASM stands for WebAssembly. The EOSIO based blockchains execute user-generated applications and code using WebAssembly. WASM is an emerging web standard with widespread support of Google, Microsoft, Apple, and others. At the moment the most mature toolchain for building applications that compile to WASM is clang/llvm with their C/C++ compiler. For best compatibility, it is recommended that you use the EOSIO toolchain to generate WASM.

Synonyms

Web-Assembly Machine WIF WIF stands for Wallet Import Format and is an encoding for a private EDSA key. EOSIO uses the same version, checksum, and encoding scheme as the Bitcoin WIF addresses and should be compatible with existing libraries. The following is an example of a WIF Private Key: 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAbuatmU

Wait Wait is measured in seconds and it is the time to wait until a delayed transaction is executed. Its value can not be higher than the max_transaction_delay which is set in the global configuration file.

Wallet Wallets are clients that store keys that may or may not be associated with the permissions of one or more accounts. Ideally a wallet has a locked (encrypted) and unlocked (decrypted) state that is protected by a high entropy password.