CommerceBlock Network

Release 19.04

Sep 12, 2019

CommerceBlock Platform:

Commerceblock provides public blockchain based technology and infrastructure that enables the tokenisation of assets and securities on fully independent blockchains (federated sidechains) that derive trustless immutability from the Bitcoin network Proof-of-Work consensus process. We have created an open-source ecosystem that provides all the tools required to launch and operate permissioned sidechains with customisable transaction and user policy and full KYC/ID provider integration.

Independent permissioned sidechains built using CommerceBlock technology and utilising CommerceBlock services have the following advantages:

- Issuer controlled transaction and user policy
- Issuer controlled transaction fees
- Enterprise-level transaction rates and scalability
- Full KYC/ID provider integration
- Customisable block explorer and lightweight and mobile wallet implementations
- Multisig token issuance with asset management and reporting tools
- Backed by Bitcoins Proof-of-Work via the Mainstay protocol

This documentation covers the central components of the CommerceBlock technology stack, including the sidechain client Ocean, the process of sidechain creation and operation via a permissioned federation of block signing nodes, the tools for controlling user whitelists, and the tools that enable the management and mapping of issued tokens. In addition, the full protocol descriptions and documentation of the two services offered by CommerceBlock to secure individual sidechains: Mainstay and the Guardnode system, are included.

Note: All CommerceBlock software is fully open-source and free to use, available via our Github repository: github.com/commerceblock. Technical questions and suggestions for improvements can be raised as issues on the relevant repos. In addition, technical questions related to our software and processes are welcomed on our Telegram group.

CHAPTER 1

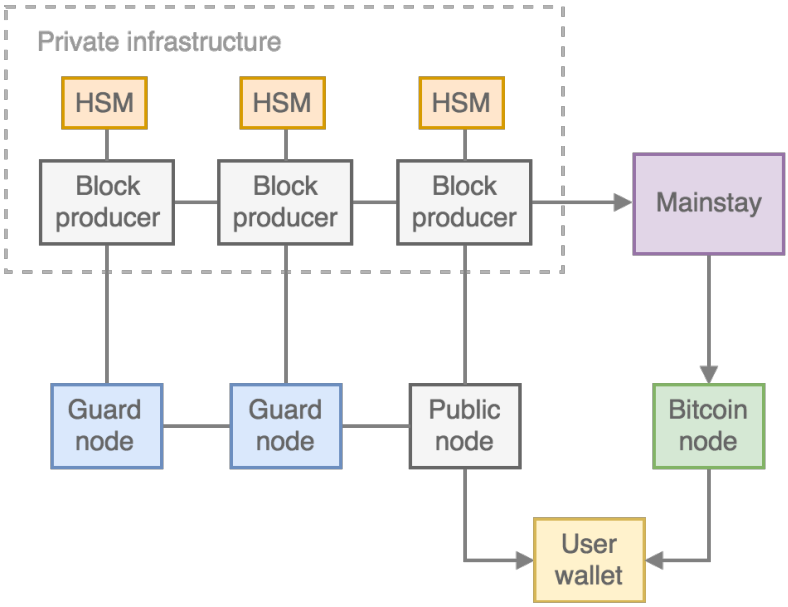
CommerceBlock Overview

CommerceBlock open-source technology, infrastructure and services have been developed to enable the tokenisation of assets and securities by companies and institutions. To remove the unnecessary second layer of trust required on a shared blockchain platform, the CommerceBlock model enables asset custodians to issue tokens on a permissioned sidechain that they control, but which has trustless immutability derived from the Bitcoin blockchain proof-of-work.

Immutability and censorship resistance are the two revolutionary properties that the decentralisation the Bitcoin blockchain has provided. However, for many applications token issuers do not require or desire full censorship resistance—certainly for asset or equity backed tokens, as there is intrinsic permission required from the issuer to redeem an asset. In addition, issuers of security tokens may be legally required to control who can transact with a token and when. The value of having tokens issued on a blockchain in this design is that the ledger is publicly verifiable: it provides independent and legal proof of ownership and transfer of ownership of a token representing an asset or security—a cryptographic proof based on the possession of private keys. If the blockchain on which the token is transacted is also trustlessly immutable, then you can prove your ownership of the token, and hence the asset, independently.

The CommerceBlock ecosystem provides a platform which simultaneously achieves 1) the *trustless* immutability that can only be provided by public, global proof-of-work via Bitcoin and 2) the legal control, scalability and reliability offered by publicly verifiable permissioned blockchains. This is accomplished via an architecture where asset backed blockchains - controlled by the asset issuer - can individually link to the CommerceBlock *Mainstay* service that immutably binds the sidechain to the Bitcoin blockchain. Companies, institutions and consortiums can then launch customised and configurable federated blockchains with tokenized asset support and full ID/KYC integration, according to their own requirements and policies. Assets can then be issued on these blockchains and transacted peer-to-peer using CommerceBlock multi-asset wallet tooling.

Asset-backed sidechains on the CommerceBlock network cab be configured as fully public blockchains, and in order to increase the decentralisation, resilience and security of the system, organisations and individuals over the globe can be incentivised to run fully validating *Guardnodes*, which enforce the sidechain consensus rules and provide services to the network. Operators of Guardnodes are free to chose the sidechains they wish to provide services to, and cab be rewarded with transaction fees generated on the asset backed sidechains.



1.1 CommerceBlock Sidechain features

1.1.1 Immutability

Sidehchains linked to the Mainstay service are as trustlessly immutable as Bitcoin, backed by the global decentralised Proof-of-Work consensus system. Sidechains are fully verifiable, and their immutability can be independently verified by third-parties via Mainstay tooling with a connection to a Bitcoin full node.

1.1.2 Permissions

Sidechains operated by token issuers have the capability to incorporate user address whitelists and blacklists into the block-signing node policy rules, which enables token issuers to control transaction permissions. The control of these policy lists is performed via the blockchain itself (with special permission control private keys), so no additional infrastructure or databases are required. The permission control can be integrated directly with KYC/ID check providers (such as Onfido) for a seamless user experience.

1.1.3 Security

Blocks are created via a Byzantine fault-tolerance consensus of a federation of block-signing nodes, which can be under the control of a single legal entity or a number of separate legal entities. The block signing protocol is fully integrated with the major hardware security module (HSM) interfaces (PKCS11/JCE/JCA). Forking or double-spending on the sidechain is prevented with Mainstay and Bitcoin's Proof-of-Work consensus - removing the requirement for full trust in the federation nodes.

1.1.4 Control

The issuance and creation of tokens on a sidechain can be configured to have custom multisignature permissions, where a number of separate parties are required to sign an issuance transaction. Asset management and mapping tools enable tokens to be linked with real-world assets and securities, tokens can be reissued and redeemed.

1.1.5 Sovereignty

Sidechains remain under the full control of the federation (or token issuer) and can operate completely independently of CommerceBlock and CommerceBlock services if desired at any time. Token owners/holders control their own private keys, and CommerceBlock's lightweight wallet client (based on the Electrum protocol) can integrate easily with hardware wallets.

1.1.6 Transaction Fees

Full control of the sidechain enables asset issuers to set the transaction fee policy according to their own requirements, which can be fixed or proportional to transaction size or value. This enables users and token holders to have long term guarantees on the cost of using the platform. This is in contrast to fully public blockchain networks, where the transaction fees and confirmation times are unpredictable and can potentially prevent token holders from transacting.

1.1.7 Scalability

Sidechains are independently controlled, so transaction throughput is not constrained by a separate network. Scalability can be controlled by the asset issuer and the federation, and is only really limited by hardware. Ocean nodes can be launched easily on cloud infrastructure, being fully containerised (with Docker images for AWS etc.). Attestation to Bitcoin via Mainstay requires only one Bitcoin transaction every 10 minutes, the cost of which is shared among all sidechains using the CommerceBlock Mainstay service.

CHAPTER 2

Introduction

Ocean is an open source federated blockchain and sidechain platform, which is developed and maintained by CommerceBlock. It incorporates functionality for token issuance, on-chain transaction policy rules including address whitelisting and for linking to Bitcoin via the Mainstay protocol.

Ocean protocol based blockchains employ a *federated* consensus mechanism to extend the chain and produce new blocks. Blocks are validated when they acquire a threshold of signatures from pre-defined *signing nodes* - the public keys of these signing nodes define the blockchain and the permissions to create blocks. Federated blockchains provide Byzantine fault tolerance as well as enabling fast, regular block times (< 1 minute) and a high degree of scalability. In addition, the fixed validator set of a federated blockchain enables transaction and address policies to be enforced - this means that user addresses can be whitelisted, blacklisted and frozen by the federation controllers facilitating the KYC and user ID verification requirements of tokenised asset and securities issuers. A unique feature of the Ocean platform is that transaction control whitelists and freezelists (*policy lists*) can be administered entirely on-chain without any requirement for separate systems and databases.

The Ocean client (incorporating the full node and wallet) is derived from the open-source Elements project, which in turn was built from the main Bitcoin Core codebase. The core routines, data structures and cryptographic algorithms are the same as those used in the Bitcoin protocol, which is the most secure and battle-tested blockchain platform ever created.

Note: Ocean is released under the terms of the MIT license.

Hint: For a more extensive set of documentation for the Elements platform, including easy to understand descriptions of the underlying technologies and detailed tutorials and examples, visit elementsproject.org.

2.1 Contents

2.1.1 Token issuance permissions

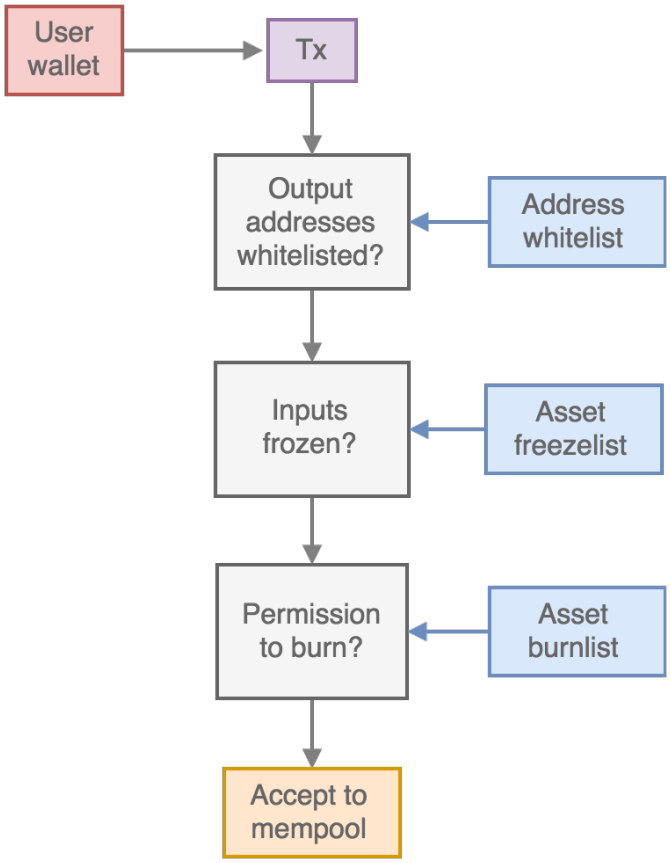
A principle function of Ocean blockchains/sidechains is the support of multiple token types, with global token identity and state being enforced by client consensus rules. Tokens issued can represent ownership of any asset or security, are divisible to arbitrary precision, and can be transferred between user wallets with atomic operations. Any number of different token types can be issued on an Ocean blockchain, and any token type can be provably inflated (re-issued). The ability to issue new tokens or re-issue existing tokens on an Ocean blockchain can be restricted with custom permissions and security policies. An Ocean blockchain can be configured so that to issue or reissue any tokens requires multiple signatures from security officers (*controllers*) via a multisig script.

2.1.2 Policy lists and transaction control

A core functionality of the Ocean platform is the ability of the blockchain/sidechain operators (the asset issuers) to control and restrict the user transaction permissions, while token outputs remain *owned* (via the control of private keys) by the legal holder. This control can determine:

- Which user addresses (public keys) tokens can be paid to (the *whitelist*)
- Which user addresses tokens can be spent from (the *freezelist*)
- Which user addresses have permission to permanently destroy (burn) tokens (the *burnlist*)

These three *policy lists* then restrict which transactions that are generated and submitted by individual users can be added to new blocks generated by the federation signing nodes, and therefore be allowed to transfer ownership of a token value peer-to-peer.



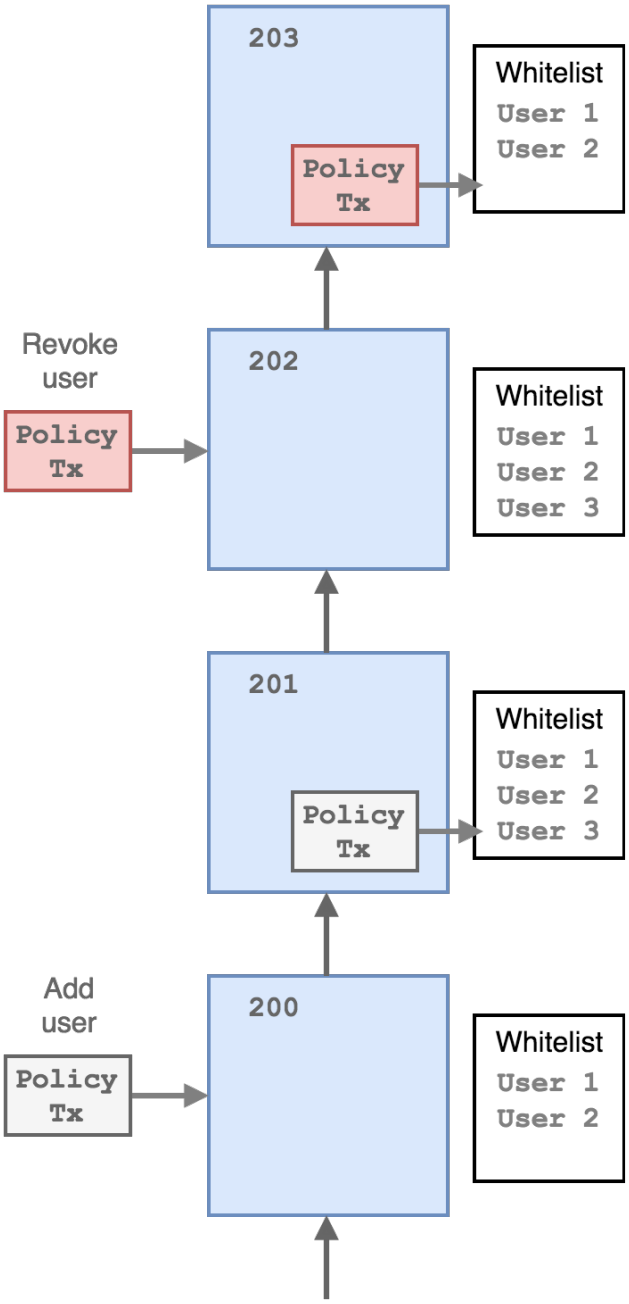
The policy lists are used by the federation nodes to determine which transactions are added to the *mempool*, and therefore in-turn added to new blocks. The restrictions are applied as follows:

1. The transaction is checked for any *spendable* outputs (i.e. P2PKH or P2SH). If *all* spendable output addresses are contained in the address whitelist, the transaction is accepted, if not it is rejected.
2. Each of the transaction inputs are checked - if *any* of the transaction input addresses are on the address freezelist, the transactions is rejected.
3. If any of the transactions outputs burn token value (i.e. send tokens to an `OP_RETURN` output) then *all* of the input addresses must be on the address burnlist.

2.1.3 Policy transactions

The three policy lists (whitelist, freezelist and burnlist) are kept in the client memory for fast lookup, and are activated and applied as mempool policy if the configuration options `-whitelist=1`, `freezelist=1` and `-burnlist=1` are set. The addresses in each policy list can be added, removed or queried via the client RPC interface. In addition, to enable scalable and modular deployment of federated nodes, the policy lists can also be controlled (i.e. addresses added and removed) via on-chain transactions: *policy transactions*. This enables addresses to be added and removed from the policy lists by remote agents outside of the federated signing nodes without any external connections except for the peer-to-peer protocol.

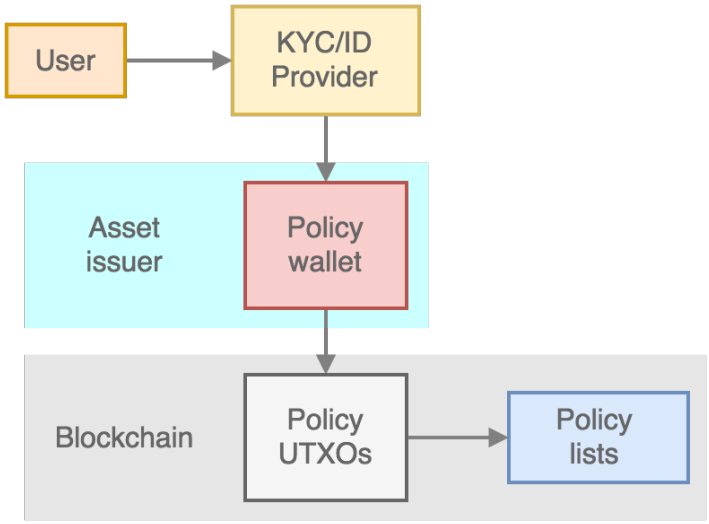
In this process, a special policy transaction containing an address to be added to a policy list is sent to the network (by an authorised wallet) and included in a block. Once confirmed, the address encoded in the policy transaction is added to the policy list by the signing node clients, and it is then enforced. A second policy transaction spending this previous output is then be used to remove the address from the list.



2.1.4 Policy tokens

Permission to modify the policy lists via policy transactions is controlled via *policy tokens*. Policy tokens for the modification of each policy list are created in the genesis block under the control of a specified `scriptPubKey`. These are specified in the configuration as `whitelistcoinsdestination`, `freezelistcoinsdestination` and `burnlistcoinsdestination` respectively.

The wallets that control the private keys to these policy tokens then have the ability to modify the corresponding policy lists via policy transactions from outside of the federation signing nodes. These wallets to be integrated directly with a KYC/ID service provider for onboarding new users of a blockchain/sidechain or controlled manually by agents or staff of the asset issuer.



2.1.5 User onboarding

A protocol is incorporated that streamlines the user onboarding process, and allows users to self-register of validated whitelist addresses in a way that preserves user privacy.

Preliminaries

A shared deterministic wallet is generated and copied to the whitelisting node and the signing nodes. This private keys from the wallet are used for encrypting and decrypting whitelisting transactions as described below.

A WHITELIST asset is defined and published in the genesis block. This asset is initially assigned to an output owned by the wallet of the "whitelisting node". The whitelist asset is required for initial address whitelisting (user onboarding) and blacklisting transactions.

The asset issuer creates deterministic "wallet" key pairs `pub_kyc` (referred to as "KYC public keys") and `priv_kyc` and publishes the `pub_kyc` keys to the blockchain via a policy transaction using the WHITELIST asset as the asset type. The `priv_kyc` are known by the signing nodes and the whitelisting node, as they all share the same deterministic wallet.

Onboarding

1. The user randomly selects a `pub_kyc` from the unassigned `pub_kyc` keys, generates a public private key pair (`pub_uob`, `priv_uob`) and creates file containing `pub_kyc` and `pub_uob`, tweaked address and corresponding untweaked public key data data for the addresses they want to register. The address data are encrypted using a shared secret generated from `priv_uob` and `pub_kyc`. Therefore, the addresses can be read by the user, the signing nodes and the whitelisting node only. This "KYC file" is forwarded to the KYC vendor together with the user's ID details. The KYC file is generated from `ocean` using the `dumpkycfile` command, or from the CommerceBlock `electrum` wallet from the `Wallet->Register` menu item.
2. The KYC vendor forwards the result of the checks together with the KYC file data to a webhook.
3. If the user passed the KYC/AML checks then `pub_kyc` (or a newly assigned one if the original `pub_kyc` has been assigned to another user) is recorded in the blockchain together with the user's wallet addresses in a `OP_REGISTERID` transaction. Again, the WHITELIST asset is required.

On reading the transactions, the signing nodes and whitelisting nodes will build whitelisted address tables in RAM for fast lookup.

User address self-registration

Submission

After the user's wallet has been onboarded, the user can register additional addresses to the whitelist.

The user submits a transaction that includes the following information:

- the tweaked address, encrypted with `pub_c`
- The operation code (`OP_REGISTERADDRESS`)

Processing

1. The signing node looks up the `pub_c` from the `addr:pub_c` map using the transactions input address (users will request new addresses using existing addresses).
2. If the `pub_c` is already whitelisted, the node decrypts `addr_e`, adds it to the whitelist and updates the `pub_c:addr` map.

Node restart

In case of node restart, the whitelist is rebuilt from the blockchain.

Privacy

Access to the whitelisting wallet master key or a `priv_kyc` is required in order to link users to addresses.

Auditing

Each user has their own pub/priv pair, so one user's addresses can be revealed if required by revealing their `priv_kyc`, without revealing any other user's addresses.

OP codes

The below opcode is used for address registration transactions.

- OP_REGISTERADDRESS

2.1.6 Ocean configuration

Ocean nodes are configured via `oceand` command line arguments or parameters specified in the `ocean.conf` configuration file located in `-datadir`. Ocean inherits all of the configuration options from *Elements 0.14* (which in turn inherits all of the configuration options from *Bitcoin 0.13*). Ocean specific configuration options are described below, with examples and whether they form part of the genesis block and are therefore critical to the definition of the chain.

Sample config

```
chain=asset_main

rpcuser=user
rpcpassword=pass
rpcport=18886
rpcallowip=10.0.3.0/24
txindex=1
listen=1
connect=nodeX:7042
pkhwhitelist=1
freezelist=1
burnlist=1
reindex=1
rescan=1
freezelistcoinsdestination=76a9149d2eaa0bb68b5b9ba11250994fdfce78f41fdc0188ac
burnlistcoinsdestination=76a91415de997afac9857dc97cdd43803cf1138f3aaef788ac
whitelistcoinsdestination=76a9144ff9b5c6885f87fb5519cc45c1474f301a73224a88ac


policycoins=2100000000000000
signblockscript=532103041f9d9edc4e494b07eec7d3f36cedd4b2cfbb6fe038b6efaa5f56b9636abd7b21037c06b0c66c9
con_
↪mandatorycoinbase=51210282e9e791e3ade9242eee103284315d61933afcb5ae3006bd61560a5819dc9cd451ae
issuancecoinsdestination=52210333f1635e1140dcf12dfc25ab2b1f993e7d9f9fd69481808af7d57b5892fe2a6e21028e
issuecontrolscript=52210333f1635e1140dcf12dfc25ab2b1f993e7d9f9fd69481808af7d57b5892fe2a6e21028e029a83

attestationhash=18b4355a10177cd6d1e11985086aa252e0a64ae59d95dcba0d882cdd99fa3564
```

Options

Information on all the config arguments used, their importance and how to derive them.

–chain

genesis-hash critical	Optional
yes	yes (but ocean defaults to test params)

Specifies the name of the chain. Chain name is important as based on this the client decides which chain parameters to use. Chain parameters differentiate in address/key prefixes and are thus crucial in generating multisig scripts for other config arguments and also when importing a private keys to full node wallets.

Currently the following hardcoded names exist (from `chainparamsbase.h`):

```
#define CHAINPARAMS_OCEAN_MAIN "ocean_main"
#define CHAINPARAMS_OCEAN_TEST "ocean_test"
#define CHAINPARAMS_ASSET_MAIN "asset_main"
```

Unless the `--chain` parameter is specified then the default **ocean_test** name is chosen.

Based on this parameter the chain params are chosen as (from `chainparams.cpp`):

```
std::unique_ptr<CChainParams> CreateChainParams(const std::string& chain)
{
    if (chain == CBaseChainParams::MAIN)
        return std::unique_ptr<CChainParams>(new CMainParams(chain));
    if (chain == CBaseChainParams::ASSET)
        return std::unique_ptr<CChainParams>(new CAssetParams(chain));
    return std::unique_ptr<CChainParams>(new CCustomParams(chain));
}
```

Example values:

- for ocean main: `chain=ocean_main`
- for asset mainnet: `chain=asset_main`
- for asset test: `chain=asset_test` or any other value

Note:

Creating custom parameters requires corresponding changes to the electrum server. Configuration options for the electrum server live in `cb-electrum-server/electrumx/lib/coins.py`. The best practice is to override the class Ocean or OceanTestnet, depending on whether it's a mainnet chain or a testnet chain, and override the address/key prefixes (mainnet only) **as well as** the `GENESIS_HASH`.

Code change PRs:

- https://github.com/commerceblock/ocean/pull/73
- https://github.com/commerceblock/cb-electrum-server/pull/8
- https://github.com/commerceblock/cb-electrum-server/pull/9

terms and conditions

genesis-hash critical	Optional
yes	yes

The terms and conditions are required in the derivation of new ocean addresses, where the contract hash is used to tweak the corresponding priv/pub key (per BIP175). This hash is, for reference and validation, included in the genesis block of the chain, therefore any ocean node attempting to connect to a specific chain that has this functionality enabled will need to have a copy of the terms and conditions in the datadir.

The terms and conditions are copied as part of building the Dockerfile but will need to be copied manually when running ocean independently. The latest contracts can be found in `ocean/doc/$chain`. The `chain` name specified above is also used as the directory name under `doc` to specify where the contract for each chain is stored.

Example:

For `chain=ocean_main` contract `doc/ocean_main/latest.txt` is used and so on...

Note:

The same terms and conditions will need to be used by the electrum wallet client. A public copy will be provided.

Code change PRs:

- https://github.com/commerceblock/ocean/pull/74

scripts

In order to generate multisig scripts and corresponding private keys a simple python script can be used.

This requires specifying number of keys, number of signatures and WIF (wallet private key format) prefix. The WIF can be found in the chosen chain parameters (named SECRET_KEY).

Current values:

- Main Params (ocean main): 128
- Asset Params (asset main): 180
- Custom Params (any other chain / testnet): 239

–issuecontrolscript

genesis-hash critical	Optional
yes	yes

Script determining ownership of the issuance process.

–issuancecoinsdestination

genesis-hash critical	Optional
yes	no

Script destination for coins, required for issuance. Usually same target as `issuecontrolscript`. The number of coins is specified by `policycoins`.

-freezelistcoinsdestination

genesis-hash critical	Optional
yes	yes

Script destination for freezelist coins, required if freezelist is enabled (option -freezelist=1). The number of coins is specified by `policycoins`.

-burnlistcoinsdestination

genesis-hash critical	Optional
yes	yes

Script destination for burnlist coins, required if burnlist is enabled (option -burnlist=1). The number of coins is specified by `policycoins`.

-whitelistcoinsdestination

genesis-hash critical	Optional
yes	yes

Script destination for public key hash whitelist coins, required if whitelist is enabled (option -pkhwhitelist=1). The number of coins is specified by `policycoins`.

–con_mandatorycoinbase

genesis-hash critical	Optional
yes	no

Script destination for all sidechain fees. Coinbase transaction of each new block pays all fees to this script.

–signblockscript

genesis-hash critical	Optional
yes	no

The signblock script is responsible for block generation in the chain. On non-HSM chains a similar approach to the other scripts should be used.

For HSM chains the Dockerfile can be used to generate keys and provide the multisig script. This Dockerfile requires providing appropriate config/secrets for the HSM.

–pkhwhitelist

genesis-hash critical	Optional
yes	yes

Wether to enforce whitelisting rules at this node. Set to pkhwhiteliist=1 for signing nodes is whitelisting is to be used.

–pkhwhitelist-scan

genesis-hash critical	Optional
no	yes

Wether to scan the blockchain for whitelisted addresses and KYC public keys. pkhwhitelist-scan=1 is required for all client nodes for used to transact on the blockchain or whitelist new addresses if the signing nodes enforce whitelisting rules.

–rescan

genesis-hash critical	Optional
no	yes

Rescan the blockchain for wallet addresses when restarting nodes, or adding new private keys to the wallet. Set rescan=1 for all nodes in the network using either pkhwhitelist=1 or pkhwhitelist-scan=1.

–reindex

genesis-hash critical	Optional
no	yes

Rescan the UTXO set when restarting nodes. Set reindex=1 for all nodes in the network using either pkhwhitelist=1 or pkhwhitelist-scan=1.

–attestationhash

genesis-hash critical	Optional
yes	no (if using Mainstay)

Reference to a transaction id hash in the mainstay staychain. Either first/latest staychain hash or the hash at which our chain genesis was committed.

2.1.7 Running with Docker

Instructions for launching a full Ocean node with Docker.

Requirements

Docker engine release: 18.02.0 or latest

docker-compose: 1.20.0 or latest

Download docker-compose.yml

From: commerceblock/ocean/contrib/docker/docker-compose.yml

Or

```
curl -O https://raw.githubusercontent.com/commerceblock/ocean/master/contrib/docker/
↪docker-compose.yml
```

Download and read terms and conditions

```
curl -O https://raw.githubusercontent.com/commerceblock/ocean/master/doc/terms-and-
↪conditions/ocean_test/latest.txt
```

Download image and start

```
docker-compose -p ocean up -d
```

Check status

```
docker-compose -p ocean ps
```

Output

```
    Name                   Command               State                            Ports
-------------------------------------------------------------------------------------------
↪-------------------
ocean_node_1   /docker-entrypoint.sh elem ...   Up      0.0.0.0:32768->18332/tcp, 0.0.
↪0.0:32769->7042/tcp
```

Check logs and see if node is syncing

```
docker-compose -p ocean logs --follow
```

Hit ctrl+c to stop following

Check if connected to CommerceBlock testnet

```
docker-compose -p ocean exec node ocean-cli -rpcport=18332 -rpcuser=ocean -
↪rpcpassword=oceanpass getpeerinfo

Should see: "testnet.commerceblock.com:7043"
```

Check block count

```
docker-compose -p ocean exec node ocean-cli -rpcport=18332 -rpcuser=ocean -
↪rpcpassword=oceanpass getblockcount
```

Once synced, block count should be the same as in: https://cbtexplorer.com

Data persistence

```
mkdir ~/ocean_full_node
mkdir -p ~/ocean_full_node/terms-and-conditions/ocean_test
cp latest.txt ~/ocean_full_node/terms-and-conditions/ocean_test/

edit: docker-compose.yml, adding:

 image: commerceblock/ocean:latest
 volumes:
   - /home/your_username/ocean_full_node:/home/bitcoin/.bitcoin
```

Using docker secrets

Add a secrets block to contrib/docker/docker-compose.yml

```
secrets:
  ocean_user:
    file: ocean_user
  ocean_pass:
    file: ocean_pass
```

Modify the service block to use the newly defined secrets:

```
services:
  node:
    secrets:
      - ocean_user
      - ocean_pass
```

Remove the rpc authentication arguments from the command:

```
-rpcuser=${BITCOIN_RPC_USER:-ocean}
-rpcpassword=${BITCOIN_RPC_PASSWORD:-oceanpass}
```

Create and populate ocean_user and ocean_pass files with credentials in the same directory.

Dig deeper

As root

```
docker-compose -p ocean exec node bash`
```

As bitcoin

```
docker-compose -p ocean exec -u bitcoin node bash
```

Then: ocean-cli / ocean-tx available from within inside of container.

Note: if running as root, need to specify: -datadir=/home/bitcoin/.bitcoin

Execute shell commands

```
docker-compose -p ocean exec node ip a
```

Scale containers

Up

```
docker-compose -p ocean scale node=2
```

Down

```
docker-compose -p ocean scale node=1
```

Stop

```
docker-compose -p ocean stop
```

Remove stack

```
docker-compose -p ocean rm -f
```

2.1.8 Federated block signing

The federation client/daemon is used to validate and sign blocks in an Ocean blockchain via a fault tolerant multiparty protocol. The federation is defined by the public keys in the M-of-N multisig blocksigning script which is specified in the Ocean node configuration. Each federation signing node coordinates with the other federation nodes to prodice valid blocks and combining M signatures on a valid block extending the blockchain over a specified block creation interval. In addition to block creation and signing, the federation client can perform token reissuance operations with the same fault-tolerance properties enabling secure and verifiable inflation of assets accordning to a specified schedule.

Instructions

1. `pip3 install -r requirements.txt`
2. `python3 setup.py build && python3 setup.py install`
3. For the demo run `./run_demo` or `python3 -m demo`
4. For the federation run `./run_federation` or `python3 -m federation` and provide the following arguments: `--rpcconnect $HOST --rpocport $PORT --rpcuser $USER --rpcpass $PASS --id $NODE_ID --msgtype $MSG_TYPE --nodes $NODES_LIST`

Federation arguments:

- `--rpconnect`: rpc host of Ocean node
- `--rpcport`: rpc port of Ocean node
- `--rpcuser`: rpc username

- `--rpcpassword`: rpc password
- `--id`: federation node id
- *–msg_type*: Messenger type used. Possible values: 'kafka', 'zmq' (optional, default='kafka')
- `--nodes`: List of node ip/domain names for zmq only
- `--hsm`: Flag to enable signing with HSM
- `--inflationrate`: Inflation rate
- `--inflationperiod`: Inflation period (in minutes)
- `--inflationaddress`: Address for inflation payments
- `--reissuancescript`: Reissuance token script
- `--reissuanceprivkey`: Reissuance private key

Example use:

- zmq: `python3 -m federation --rpcconnect 127.0.0.1 --rpcport 18443 --rpcuser user --rpcpass pass --id 1 --msgtype zmq --nodes "node0:1503,node1:1502"`
- kafka: `python3 -m federation --rpcconnect 127.0.0.1 --rpcport 18443 --rpcuser user --rpcpass pass --id 1` (check federation.py - defaults to 5 nodes)

Using HSMs

Initialisation

Assuming hsm and pkcs11 libraries setup and all config/secrets files are in place run:

```
docker build --build-arg user_pin=$USER_PIN --build-arg key_label=$KEY_LABEL
-f Dockerfile.hsm.init .
```

This will generate a multisig script that should be used as the `signblockarg` in the ocean sidechain.

Running

To build the federation container with hsm signing run:

```
docker build --build-arg user_pin=$USER_PIN --build-arg key_label=$KEY_LABEL
-f Dockerfile.hsm .
```

Inside this container federation can be initiated by:

```
python3 -u -m federation --rpcconnect signing1 --rpcport 18886 --rpcuser
username1 --rpcpass password1 --id 1 --msgtype zmq --nodes "federation0:6666,
federation1:7777,federation2:8888" --hsm 1
```

Inflating assets

The federation nodes can be used to reissue issued assets according to a fixed inflation schedule. This is enabled by setting the `--inflationrate` argument to a non zero value. The assets are then reissued every `--inflationperiod` blocks, and to the specified address. The reissuance tokens must be paid to the P2SH address of the supplied multisig script (`--reissuancescript`). The corresponding private key for the signing

node (for the reissuance script) is supplied as `--reissuanceprivkey`. If inflation is enabled, the `-rescan=1` and `-recordinflation=1` flags must be set in the signing node `ocean.conf` file.

Federation protocol demo

A demonstration of protocols used by the Ocean network, including federated signing and asset issuance.

Instructions

```
./scripts/restart_kafka.sh python3 -m demo
```

Running Kafka

- Install kafka

  ```
  brew install kafka
  ```

- Add bin path to PATH in bash profile

  ```
  export PATH="$PATH:/usr/local/Cellar/kafka/1.1.0/bin/"
  ```

- Different services
  - `brew services start kafka`
  - `brew services stop kafka`
  - `kafka-topics --zookeeper localhost:2181 --delete --topic new-block`

MultiSig

Generate multisig script and keys using the MultiSig class (M out of N).

Federated Signing

Implement federation signing using the BlockSigning class. Federation signing uses a Kafka broker. Nodes take turns proposing / signing blocks. One node will generate a new block hex and send it to a topic marked as 'new-block' in the Kafka broker. The rest of the nodes will fetch this and sign it, sending their signature to a topic marked as 'new-sigX', where X is the node id. The node that generated the block will collect the signatures, combine them and submit the block.

Asset Issuance

Issue assets and generate transactions with these assets using the AssetIssuance class.

2.1.9 Ocean API reference

The Ocean client includes all of the Remote Procedure Calls (RPCs) of the Elements platform (described in the reference here) as well as additional RPCs that control advanced and extended features unique to the Ocean client. This document describes these new RPCs and their function as well as additional Ocean client configuration options that enable them.

For any RPC supported in the Ocean client (including those inherited from Elements and Bitcoin), you can get information about function and correct usage from the command line using the `help` RPC. For example,

```
ocean-cli help getblockchaininfo
```

Quick reference

The following RPCs are unique to the Ocean client

Wallet

- *dumpderivedkeys*
- *validatederivedkeys*
- [dumpkycfile][]
- [readkycfile][]
- *createkycfile*
- *getderivedkeys*
- *getcontract*
- *getcontracthash*
- *getmappinghash*
- *getethaddress*
- *getethpeginaddress*
- *getethpegin*
- *createrawethpegin*
- *validateethpegin*
- *claimethpegin*
- *sendtoethmainchain*
- *sendanytoaddress*

Utility

- *getutxoassetinfo*
- *createrawissuance*
- *createrawreissuance*
- *createrawburn*
- *testmempoolaccept*
- *createrawpolicytx*
- *createrawrequesttx*
- *getrequests*

Policy

- *addtowhitelist*
- *addmultitowhitelist*
- *readwhitelist*
- *querywhitelist*
- *removefromwhitelist*
- *clearwhitelist*
- *dumpwhitelist*
- *sendaddtowhitelisttx*
- *sendaddmultitowhitelisttx*
- *addtofreezelist*
- *queryfreezelist*
- *removefromfreezelist*
- *clearfreezelist*
- *addtoburnlist*
- *queryburnlist*
- *removefromburnlist*
- *clearburnlist*

Configuration options

- *pkhwhitelist*
- *freezelist*
- *burnlist*
- *issuanceblock*
- *disablect*
- *embedcontract*
- *attestationhash*
- *embedmapping*
- *issuecontrolscript*
- *policycoins*
- *initialfreecoinsdestination*
- *freezelistcoinsdestination*
- *burnlistcoinsdestination*
- *issuancecoinsdestination*
- *permissioncoinsdestination*

- *mainchainrpchost*
- *mainchainrpcport*
- *validatepegin*
- *parentgenesisblockhash*
- *parentcontract*
- *fedpegaddress*
- *peginconfirmationdepth*

dumpderivedkeys

The `dumpderivedkeys` RPC outputs a list of all contract tweaked addresses in the key pool along with the corresponding non-tweaked basis public keys to a specified file.

Parameter #1—the filename of the output file

Example

```
ocean-cli dumpderivedkeys dumpfile.txt
```

validatederivedkeys

The `validatederivedkeys` RPC reads in a list of tweaked addresses with corresponding base public keys (as produced by *dumpderivedkeys*) from a specified file, and then checks that the address corresponds to the corresponding public key when tweaked with the current contract hash.

Parameter #1—the filename of the input file

Result—nothing if valid keys, RPC errors if invalid keys found

Example

```
ocean-cli validatederivedkeys
```

createkycfile

The `createkycfile` RPC creates an encrypted kyc file that stores p2pkh and p2sh address data to be whitelisted when onboarding

Parameter #1—the created KYC file name

Parameter #2—P2PKH data for whitelisting in an onboarding transaction

Parameter #3—P2SH data for whitelisting in an onboarding transaction

Parameter #4—the public key issued by the server for onboarding encryption.

Result—onboarding user public key if successful, null or rpc errors if passed data is invalid or wallet is not available

Example

```
ocean-cli createkycfile test [{"address":2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz,"pubkey
↪":028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58}] [{"nmultisig
↪":1,"pubkeys":[028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58,
↪0263a73eca5334af77037a1c8844b5220017bf6fb627c5a57c862dff20ea001d99]}]
```

(continues on next page)

(continued from previous page)

Result:

```
028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58
```

getderivedkeys

The `getderivedkeys` RPC returns a list of contract tweaked addresses in the key pool along with the corresponding non-tweaked basis public keys as a JSON object.

Parameters: none

Result—the txid and vout of the reissuance output

Example

```
ocean-cli getderivedkeys
```

Result:

```
{
  "address": ["2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz",
↪"2daBDLApGapXjW4xErMsYDAHWd2QzFHHxvB"],
  "bpubkey": ["028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58",
↪"0263a73eca5334af77037a1c8844b5220017bf6fb627c5a57c862dff20ea001d99"]
}
```

getcontract

The `getcontract` RPC returns the plain text of the currently enforced contract.

Parameters: none

Result—the full plain text of the current contract

Example

```
ocean-cli getcontract
```

Result:

```
{
  "contract": "These are the current terms and conditions that govern participation␣
↪in the Ocean network. 1. Be awsome to each other. 2. No smoking."
}
```

getcontracthash

The `getcontracthash` RPC returns the hash of the contract in force at a given block height. If the block height is not supplied, the current contract hash is returned.

Parameter #1—the blockheight at which a contract was in force

Result—the contract hash

Example

```
ocean-cli getcontracthash
```

Result:

```
f4f30db53238a7529bc51fcda04ea22bd8f8b188622a6488da12281874b71f72
```

getmappinghash

The `getmappinghash` RPC returns the hash of the mapping object in force at a given block height. If the block height is not supplied, the current mapping hash is returned.

Parameter #1—the blockheight at which a mapping was in force

Result—the mapping hash

Example

```
ocean-cli getmappinghash
```

Result:

```
f4f30db53238a7529bc51fcda04ea22bd8f8b188622a6488da12281874b71f72
```

getethaddress

The `getethaddress` RPC returns an ethereum address from an EC private key.

Parameter #1 — (hex, required) private key

Example

```
ocean-cli getethaddress␣
↪3ecb44df2159c26e0f995712d4f39b6f6e499b40749b1cf1246c37f9516cb6a4
```

Result:

```
8a40bfaa73256b60764c1bf40675a99083efb075
```

getethpeginaddress

The `getethpeginaddress` RPC returns information needed for claimethpegin to move coins to the sidechain. The user should send CBT coins from their eth wallet to the eth_mainchain_address returned. The user needs to provide their eth priv key, which is used to generate a claim pubkey that is added to the pegin transaction. The transaction is then signed with the key provided.

IMPORTANT: getethpeginaddress adds new secrets to wallet.dat, necessitating backup on a regular basis.

Parameter #1 — (hex, required) private key

Example

```
ocean-cli getethpeginaddress␣
↪3ecb44df2159c26e0f995712d4f39b6f6e499b40749b1cf1246c37f9516cb6a4
```

Result:


```
{
  "eth_mainchain_address": "b6872561de5ba19d38071a7616d9d434b9e37860",
  "eth_claim_pubkey":
↪"0397466f2b32bc3bb76d4741ae51cd1d8578b48d3f1e68da206d47321aec267ce7"
}
```


getethpegin

The `getethpegin` RPC returns an eth ERC-20 peg-in transaction via geth rpc connectivity.

Parameter #1 — (hex, required) eth transaction id

Example

```
ocean-cli getethpegin 8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7
```

Result:


```
{
  "blockHash": "0x39f2086a66c17a568c9b6e6de8eb5f972f850aa9f282325c4c8d6471ac43182c",
  "blockNumber": "0x48e23c",
  "contractAddress": null,
  "cumulativeGasUsed": "0x1d6c50",
  "from": "0x8027af3c8ab3a7b603873cf17f521d049d615862",
  "gasUsed": "0xca30",
  "logs": [
    {
      "address": "0x076c97e1c869072ee22f8c91978c99b4bcb02591",
      "topics": [
        "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
        "0x0000000000000000000000008027af3c8ab3a7b603873cf17f521d049d615862",
        "0x000000000000000000000002e6e830d10fb4f4f4211e84ae5d90083328684cd"
      ],
      "data": "0x0000000000000000000000000000000000000000000000df56b9541c229fce000000",
      "blockNumber": "0x48e23c",
      "transactionHash":
↪"0x8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7",
      "transactionIndex": "0x4b",
      "blockHash": "0x39f2086a66c17a568c9b6e6de8eb5f972f850aa9f282325c4c8d6471ac43182c
↪",
      "logIndex": "0xf",
      "removed": false
    }
  ],
  "logsBloom":
↪"0x00000000000000000000080000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
  "status": "0x1",
  "to": "0x076c97e1c869072ee22f8c91978c99b4bcb02591",
  "transactionHash":
↪"0x8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7",
  "transactionIndex": "0x4b"
}
```

createrawethpegin

The `createrawethpegin` RPC creates a raw CBT peg-in from an eth ERC-20 transaction.

Parameter #1 — (hex, required) eth transaction id

Parameter #2 — (hex, required) eth transaction peg-in amount

Parameter #3 — (hex, required) claim pubkey generated by getethpeginaddress

Example

```
ocean-cli createrawethpegin␣
↪8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7 432.109␣
↪03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
0200000001016ca60fb08c36a2e77e0810de32181b63e8250fbf9a398f9bf9e53444cbf6803000000004000ffffffff0201bfe
```

validateethpegin

The `validateethpegin` RPC validates an eth ERC-20 transaction to be used from peg-in to Ocean.

Parameter #1 — (hex, required) eth transaction id

Parameter #2 — (hex, required) eth transaction peg-in amount

Parameter #3 — (hex, required) claim pubkey generated by getethpeginaddress

Example

```
ocean-cli validateethpegin␣
↪8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7 432.109␣
↪03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
true
```

claimethpegin

The `claimethpegin` RPC claims ERC-20 CBT tokens from eth to Ocean.

Parameter #1 — (hex, required) eth transaction id

Parameter #2 — (hex, required) eth transaction peg-in amount

Parameter #3 — (hex, required) claim pubkey generated by getethpeginaddress

Example

```
ocean-cli claimethpegin␣
↪8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7 432.109␣
↪03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
bb2364284941f08cceaf49911858125256d61f1b728e544ead6423bf06ea1e15
```

sendtoethmainchain

The `sendtoethmainchain` RPC sends sidechain funds to the given eth mainchain address via federated peg-out.

Parameter #1 — (hex, required) destination address on eth mainchain

Parameter #2 — (numeric, required) eth amount pegged-out to eth mainchain

Parameter #3 — (boolean, optional) Fee deducted from amount being pegged-out

Example

```
ocean-cli sendtoethmainchain 8e8a0ec05cc3c2b8511aabadeeb821df19ea7564 533.22 false
```

Result:

```
aa2364284941f08cceaf49911858125256d61f1b728e544ead6423bf06ea1e15
```

sendanytoaddress

The `sendanytoaddress` RPC sends a combination of any non-policy assets to the address. The cumulative sum of the assets is equal to the desired amount. This rpc should only used in chains that are comprised of non-policy assets which are fungible.

Parameter #1 — (hex, required) destination address

Parameter #2 — (numeric, required) amount to be sent to the destination

Parameter #3 — (string, optional) A comment used to store what the transaction is for.

Parameter #4 — (string, optional) A comment to store the name of the person or organization to which you're sending the transaction.

Parameter #5 — (boolean, optional) Return a transaction even when a blinding attempt fails due to number of blinded inputs/outputs if this is set to true.

Parameter #6 — (boolean, optional) Split a transaction that goes over the size limit into smaller transactions if this is set to true.

Parameter #7 — (numeric, optional) Choose which balances should be used first. 1 - descending, 2 - ascending.

Example

```
ocean-cli sendtanytoaddress 8e8a0ec05cc3c2b8511aabadeeb821df19ea7564 533.22
```

Result:

```
aa2364284941f08cceaf49911858125256d61f1b728e544ead6423bf06ea1e15
```

createrawissuance

The `createrawissuance` RPC creates an unblinded raw unsigned issuance transaction with specified outputs and spending from a specified input containing an amount of policy asset.

Parameter #1—the Base58check address for the issued asset

Parameter #2—the amount of issued asset

Parameter #3—the Base58check address for the reissuance token

Parameter #4—the amount of reissuance token

Parameter #5—the Base58check address for the issuanceAsset change

Parameter #6—the amount of issuanceAsset change

Parameter #7—the number of issuanceAsset outputs

Parameter #8—input TXID

Parameter #9—input transaction vout

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawissuance 2deJ6F3w6HUtXM8JjY5YPc8wtaXerqFX7HA 123.0␣
↪2ddSmTujABoCzDyU9hPghcp4ojAGmJRtnWk 1.23 XKSxznoA799169xt3zCm7a4qkdT1KZANv3 332.
↪9995 1 0.0005 40ac4e02a64ea14190e96d6e1c5c877b12522db3fb5adffd58b1aed0cc11150a 0
```

Result:

```
0200000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff000000
```

createrawreissuance

The `createrawreissuance` RPC creates a raw unsigned re-issuance (asset inflation) transaction with specified outputs and spending from a specified input containing a valid re-issuance token.

Parameter #1—the Base58check address for the re-issued asset

Parameter #2—the amount of re-issued asset

Parameter #3—the Base58check address for the return reissuance token

Parameter #4—the amount of reissuance token to return

Parameter #5—input TXID

Parameter #6—input transaction vout

Parameter #7—the asset entropy

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawissuance 2deJ6F3w6HUtXM8JjY5YPc8wtaXerqFX7HA 0.5␣
↪2ddSmTujABoCzDyU9hPghcp4ojAGmJRtnWk 1.00␣
↪40ac4e02a64ea14190e96d6e1c5c877b12522db3fb5adffd58b1aed0cc11150a 0␣
↪b2e15d0d7a0c94e4e2ce0fe6e8691b9e451377f6e46e8045a86f7c4b5d4f0f23
```

Result:

```
0200000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff000000
```

createrawburn

The `createrawburn` RPC creates a raw unsigned burn (OP_RETURN) transaction with a single input and single output.

Parameter #1—input TXID

Parameter #2—input transaction vout

Parameter #3—the asset type

Parameter #4—the amount to burn (must equal the input amount)

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawissuance␣
↪40ac4e02a64ea14190e96d6e1c5c877b12522db3fb5adffd58b1aed0cc11150a 0␣
↪b2e15d0d7a0c94e4e2ce0fe6e8691b9e451377f6e46e8045a86f7c4b5d4f0f23 0.342100
```

Result:

```
0200000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff000000
```

createrawrequesttx

The `createrawrequesttx` RPC creates a raw request transaction with a single input and single output.

Parameter #1—Input object with details on transaction to be spent

Parameter #2—Output object with request details

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawrequesttx '''[
  {
    "txid": "43bd75af773cce38fd190f6c0943d311ce2dd8a26c7e7a9e600c58f8b21e53d4",
    "vout": 1,
  }
]''' '''[
  {
    "pubkey": "03d5be1ca0b06b54f6a29a8e245fdf58698164538191c5b376d3b27e6d3229b81a",
    "decayConst": 5,
    "endBlockHeight": 250,
    "fee": 5,
    "genesisBlockHash":
↪"99bd75af773cce38fd190f6c0943d311ce2dd8a26c7e7a9e600c58f8b21e53d4",
    "startBlockHeight": 100,
    "tickets": 150,
    "value": 1000.0,
    "startPrice": 5.0
  }
]'''
```

Result:

```
02000000000151227925212487ef62c10e46f14aec78dce956b02eb41f7e2cce8b6d56292db40100000000feffffff0101d08
```

testmempoolaccept

The `testmempoolaccept` RPC determines the validity of a raw transaction without broadcasting it. It performs the exact same validity checks as performed on mempool acceptance, including locally configured policy rules, but without adding the transaction to the mempool.

Parameter #1—signed raw transaction

Parameter #2—accept large fee

Result—a JSON object containing the transaction ID, whether the transaction is accepted or rejected, and if rejected the reason

Example

```
ocean-cli testmempoolaccept␣
↪0200000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff00000
```

Result:

```
{
 "txid": 40ac4e02a64ea14190e96d6e1c5c877b12522db3fb5adffd58b1aed0cc11150a
 "accept": 1
}
```

createrawpolicytx

The `createrawpolicytx` RPC creates a raw unsigned policy transaction that encodes an address to be added to a policy list. To be accepted, the asset type must match the policy asset type as defined in the genesis block (via `-freezelistcoinsdestination` and `-burnlistcoinsdestination`. The policy asset input(s) are specified in an array, and the outputs are specified in an array of objects that contain a policy public key, the address to be added to to the policy list and the value. Spending these outputs removes the addresses from the policy lists.

Parameter #1—Inputs

Parameter #1—Addresses to add to policy lists and control keys

Parameter #3—locktime

Parameter #4—policy asset

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawpolicytx '''[
  {
    "txid": "43bd75af773cce38fd190f6c0943d311ce2dd8a26c7e7a9e600c58f8b21e53d4",
    "vout": 1
  }
]''' '''[
  {
    "pubkey": "03d5be1ca0b06b54f6a29a8e245fdf58698164538191c5b376d3b27e6d3229b81a",
    "value": 10.0
    "address": "1HPkc4to3GzVcEV8Le6sS4V5AXWQceH5kZ"
```

(continues on next page)

(continued from previous page)

```
  }
]'''
```

Result:

```
0200000000018eb8f2e93d81b9f904f8613b6520460c00f9313683b5574fc8b67c3e33e6155500000000000ffffffff010131e
```

getutxoassetinfo

The `getutxoassetinfo` RPC returns a summary of the total amounts of unspent (and un-burnt) assets in the UTXO set. Ammounts in transactions marked as frozen (i.e. with one output having a zero address) are listed in a separate field.

Parameters: none

Result—an array of JSON objects containing the unspent amounts for each issued asset

Example

```
ocean-cli getutxoassetinfo
```

Result:

```
[
  {
    "asset": "7f7c00ca515e46165ea6a13dd49d22759beeb26a952128f1a5af824d208a051e",
    "spendabletxouts": 1,
    "amountspendable": 3.00000000,
    "frozentxouts": 0,
    "amountfrozen": 0.00000000
  },
  {
    "asset": "b2e15d0d7a0c94e4e2ce0fe6e8691b9e451377f6e46e8045a86f7c4b5d4f0f23",
    "spendabletxouts": 107,
    "amountspendable": 500000.00000000,
    "frozentxouts": 0,
    "amountfrozen": 0.00000000
  }
]
```

getrequests

The `getrequests` RPC returns all the active client requests in the blockchain

Parameter #1—the client genesis block hash

Result—an array of JSON objects containing details for each active request

Example

```
ocean-cli getrequests
ocean-cli getrequests 123450e138b1014173844ee0e4d557ff8a2463b14fcaeab18f6a63aa7c7e1d05
```

Result:

```
[
  {
    "genesisBlock": "123450e138b1014173844ee0e4d557ff8a2463b14fcaeab18f6a63aa7c7e1d05
↪",
    "startBlockHeight": 105,
    "numTickets": 20,
    "decayConst": 2,
    "startPrice": 5.0,
    "auctionPrice": 4.8,
    "feePercentage": 5,
    "endBlockHeight": 350,
    "txid": "666450e138b1014173844ee0e4d557ff8a2463b14fcaeab18f6a63aa7c7e1d05"
  },
]
```


addtowhitelist

The `addtowhitelist` RPC adds a valid contract tweaked address to the node mempool whitelist. It requires both an address and corresponding base public key, and the RPC checks that the address is valid and has been tweaked from the supplied base public key with the current contract hash as present in the most recent block header.

Parameter #1—the Base58check contract tweaked address

Parameter #2—the base (un-tweaked) compressed public key

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addtowhitelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz␣
↪028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58
```

addmultitowhitelist

The `addmultitowhitelist` RPC adds a valid contract tweaked p2sh (multisig) address to the node mempool whitelist. It requires an address, number of required signatures and corresponding base public keys, and the RPC checks that the address is valid and has been tweaked from the supplied base public keys with the current contract hash as present in the most recent block header.

Parameter #1—the Base58check contract tweaked p2sh address

Parameter #2—the base (un-tweaked) compressed public keys that the p2sh was created with

Parameter #3—the n of Multisig

Parameter #4—the Base58 KYC address

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addmultitowhitelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz␣
↪[028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58,
↪028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58] 1
```

querywhitelist

The `querywhitelist` RPC queries if a specified address is present in the node mempool whitelist.

Parameter #1—the Base58check encoded address

Result—TRUE of FALSE

Example

```
ocean-cli querywhitelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

Result:

```
1
```

readwhitelist

The `readwhitelist` RPC adds a list of valid contract tweaked address to the node mempool whitelist. It requires a file that contains a list of both an address and corresponding base public key, and the RPC cheacks that each address is valid and has been tweaked from the supplied base public key with the current contract hash as present in the most recent block header. The file format is as decribed in *dumpderivedkeys*.

Parameter #1—the filename to read in the list

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli readwhitelist derivedkeys.txt
```

removefromwhitelist

The `removefromwhitelist` RPC removes a specified address from the node mempool whitelist.

Parameter #1—the Base58check encoded address

Result—none if valid, errors returned if invalid inoputs

Example

```
ocean-cli removefromwhitelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

clearwhitelist

The `clearwhitelist` RPC clears the mempool whitelist of all addresses.

Parameters: none

Result: nome

Example

```
ocean-cli clearwhitelist
```

dumpwhitelist

The `dumpwhitelist` RPC outputs a list of all addresses in the node mempool whitelist to a specified file.

Parameter #1—the filename of the output file

Example

```
ocean-cli dumpwhitelist dumpfile.txt
```

sendaddmultitowhitelisttx

The `sendaddmultitowhitelisttx` RPC serializes and sends an OP_REGISTERADDRESS transaction for multisig which is used to add a valid contract tweaked p2sh (multisig) address to the whitelist. It requires an address, number of required signatures and corresponding base public keys, and the RPC checks that the address is valid and has been tweaked from the supplied base public keys with the current contract hash as present in the most recent block header. Whitelist node reads the transaction and adds the address to a whitelist if details are valid.

Parameter #1—the Base58check contract tweaked p2sh address

Parameter #2—the base (un-tweaked) compressed public keys that the p2sh was created with

Parameter #3—the n of Multisig

Parameter #4—the fee asset type

Result—transaction hex if valid, errors returned if invalid inputs or null if wallet is not working

Example

```
ocean-cli sendaddmultitowhitelisttx 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz␣
↪[028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58,
↪028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58] 1
```

sendaddtowhitelisttx

The `sendaddtowhitelisttx` RPC serializes and sends an OP_REGISTERADDRESS transaction which is used to add valid contract tweaked addresses to the whitelist (they are automatically retrieved from the wallet pool). Whitelist node reads the transaction and adds the addresses to a whitelist if details are valid.

Parameter #1—Number of addresses that should be taken from the wallet pool and whitelisted

Parameter #2—the fee asset type

Result—transaction hex if valid, null if wallet is not working

Example

```
ocean-cli sendaddtowhitelisttx 100
```

addtofreezelist

The `addtofreezelist` RPC adds an address to the node mempool freezelist. Transactions spending from UTXOs with output addresses on the freezelist are blocked from entering the mempool if the '-freezelist' configuration option is enabled.

Parameter #1—the Base58check address

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addtofreezelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

queryfreezelist

The `queryfreezelist` RPC queries if a specified address is present in the node mempool freezelist.

Parameter #1—the Base58check encoded address

Result—TRUE of FALSE

Example

```
ocean-cli queryfreezelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

Result:

```
1
```

removefromfreezelist

The `removefromfreezelist` RPC removes a specified address from the node mempool freezelist.

Parameter #1—the Base58check encoded address

Result—none if valid, errors returned if invalid inoputs

Example

```
ocean-cli removefromfreezelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

clearfreezelist

The `clearfreezelist` RPC clears the mempool whitelist of all addresses.

Parameters: none

Result: nome

Example

```
ocean-cli clearfreezelist
```

addtofreezelist

The `addtoburnlist` RPC adds an address to the node mempool burnlist. Transactions spending from UTXOs with output addresses on the freezelist are alowed into the mempool if these addresses are also on the burnlist and have only TX_FEE and TX_NULL_DATA outputs (with both the `-freezelist` and `-burnlist` configurations options enabled).

Parameter #1—the Base58check address

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addtoburnlist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

queryburnlist

The `queryburnlist` RPC queries if a specified address is present in the node mempool burnlist.

Parameter #1—the Base58check encoded address

Result—TRUE of FALSE

Example

```
ocean-cli queryburnlist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

Result:

```
1
```

removefromburnlist

The `removefromburnlist` RPC removes a specified address from the node mempool burnlist.

Parameter #1—the Base58check encoded address

Result—none if valid, errors returned if invalid inoputs

Example

```
ocean-cli removefromburnlist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
```

clearburnlist

The `clearburnlist` RPC clears the mempool whitelist of all addresses.

Parameters: none

Result: none

Example

```
ocean-cli clearburnlist
```

2.1.10 Lightweight wallet

CHAPTER 3

Introduction

Mainstay is a protocol that enables the creation of a cryptographic *proof* that a given system with changing state is *unique* and that the full *history* of that state is unique. This proof is generated from a series of cryptographic commitments made to a public blockchain which in turn derives its immutability and global state from a Proof-of-Work consensus algorithm. In this way, any independent system with history of state (for example a federated blockchain/sidechain or a Git repository) can be proven to be as immutable as the Proof-of-Work blockchain it is secured by. The *proof of immutability* (PoI) generated by the Mainstay protocol is *trustless* and independently verifiable.

The underlying construction of the Mainstay protocol requires no changes to the existing proof-of-work system (i.e. Bitcoin) and requires no cooperation of miners (unlike merge mining protocols). It is specifically designed to be space-efficient and censorship resistant due to the use of homomorphic commitments based on the *pay-to-contract* method (**BIP175**), and is compatible with Simplified Payment Verification (SPV) lightweight nodes. The Mainstay *Connector* protocol enables many separate systems to obtain independent proofs of immutability via a single sequence of Bitcoin commitments, by compressing sequences of commitments in a Merkel tree of *slot-proofs*. This enables the minimisation of burden on to the Bitcoin blockchain, and thousands of sidechains can be secured with just a single Bitcoin transaction per block. This enables the realisation of highly efficient, scalable and interoperable sidechain systems that can incorporate permissioned transacting and regulatory compliance while simultaneously exploiting the full security and immutability of the Bitcoin blockchain secured via proof-of-work.

This documentation describes the theory and implementation of the Mainstay protocol, as well as an explanation of the background, motivation and how the protocol differs from trustless timestamping (i.e. Proof of *Existence* - PoE). This documentation also includes detailed instructions on using CommerceBlock's Mainstay implementation and tooling, CommerceBlock's Mainstay service and API as well as step-by-step guides for sidechain integration.

Note: All the Mainstay tooling and software is released under the terms of the MIT license.

3.1 Contents

3.1.1 Background

The invention of Bitcoin solved the issue of double-spending in a fully decentralised digital payments system by ensuring that there is a single, replicated, global ledger that all participants can agree represents the valid ordering of transactions: the blockchain. Reaching consensus on the state of this global ledger is achieved using proof-of-work: adding to the blockchain requires expensive, but easily verifiable, computations that are rewarded with tokens derived from transaction fees (and block rewards). The blockchain with the most accumulated work is considered the only valid history, and all participants are incentivised to contribute their computational work to extending it.

The use of proof-of-work, which requires the consumption of real world resources (i.e. energy), as the consensus mechanism means that Bitcoin is completely permissionless: no permission is required in order to add to the blockchain - only computational work. The work required to extend the blockchain also leads to immutability: any attempt to modify the time-order of transactions in the blockchain requires more computational power than the rest of the network combined. This leads to the Bitcoin blockchain being a unique global system of consensus on the ordering of time-stamped events without the need for any trusted authority.

Of all the cryptocurrency projects that have been launched since, Bitcoin remains by far the most secure, with the most accumulated work. The Bitcoin network has operated persistently for over 9 years, holding hundreds of billions of dollars in value and has resisted constant attack. However, these properties come at the cost of both scalability and upgradability. In order to maintain the decentralisation, security and censorship resistance of the network, block sizes must remain relatively small which limits the transaction capacity: Bitcoin can process only 3 to 6 transactions per second leading to unpredictable transaction fees and confirmation times. In addition, for the very same reasons Bitcoin is so secure, it is also very difficult to change the protocol: adding new features requires the consent of all network participants and must be done extremely conservatively so as to not risk the integrity of the system.

Alternative consensus mechanisms on separate blockchains can be used to improve scalability and build in more advanced features at the protocol level [6]. Sidechains to Bitcoin secured by federated consensus rules enable significantly better scalability, and much faster and more regular block times. In addition, these systems can incorporate more protocol-level functionality, including token issuance and cryptographic privacy and anonymity features not possible on Bitcoin. However, such systems are not able to achieve the trustless immutability of Bitcoin with permissionless proof-of-work. Blockchains that are run by a static federated consensus mechanism require collective trust in the federation members: if the federation members collude or leak a threshold of secret keys, conflicting forks of the blockchain can be created at no cost and double-spend attacks launched against token holders.

To provide federated sidechains with the same level of trustless immutability as Bitcoin, we describe a method that involves cryptographically binding these sidechains to the Bitcoin mainchain in such a way that the sidechain cannot be forked without also simultaneously forking the Bitcoin mainchain. This means that for a fixed set of federated block signers, users of a sidechain do not need to trust the federation to protect them from a double-spend attack: consensus on a single unforked version of the federated sidechain is enforced by Bitcoin's proof-of-work.

Attestation and Timestamping

It was recognised early in Bitcoin's history that the blockchain could be utilised to timestamp arbitrary data in a completely trustless and decentralised way. By embedding a cryptographic commitment to a piece of data into a valid transaction, which was then mined into the blockchain, it was possible to prove that the data existed at a particular time. To accomodate time-stamping (and other meta-protocols) in a more efficient way, a new prunable transaction output type was introduced via a new OP code: `OP_RETURN`. This allowed up to 40/80 bytes to be included in an output which was not treated as a spendable output in the UTXO set. The use of `OP_RETURN` however has significant downsides: it bloats transactions (resulting in higher transaction fees), it offers no privacy (data is included in plain text directly into the transaction) and transactions including them may be rejected (censored) by mining pools.

There are many services that employ `OP_RETURN` outputs to time-stamp single files into the Bitcoin blockchain and there are protocols that can include a much more extensive set of data into a single commitment, such as OpenTimes-

tamps which collects submitted commitments via a calendar server and compresses them into Merkle Tree, and then time-stamps the Merkle Root in a transaction. This type of time-stamping is however fundamentally limited in the type of immutability it can provide. A timestamp can only prove that a particular piece of information *existed* at a certain point in time, not that the information has any other validity or *uniqueness*. A timestamp by itself cannot prove that a commitment to conflicting data has not also been simultaneously timestamped. This is a critical concept in relation to immutability: any proof-of-existence does not act as a proof that anything else (e.g. an alternative ordering of transactions) does not also exist.

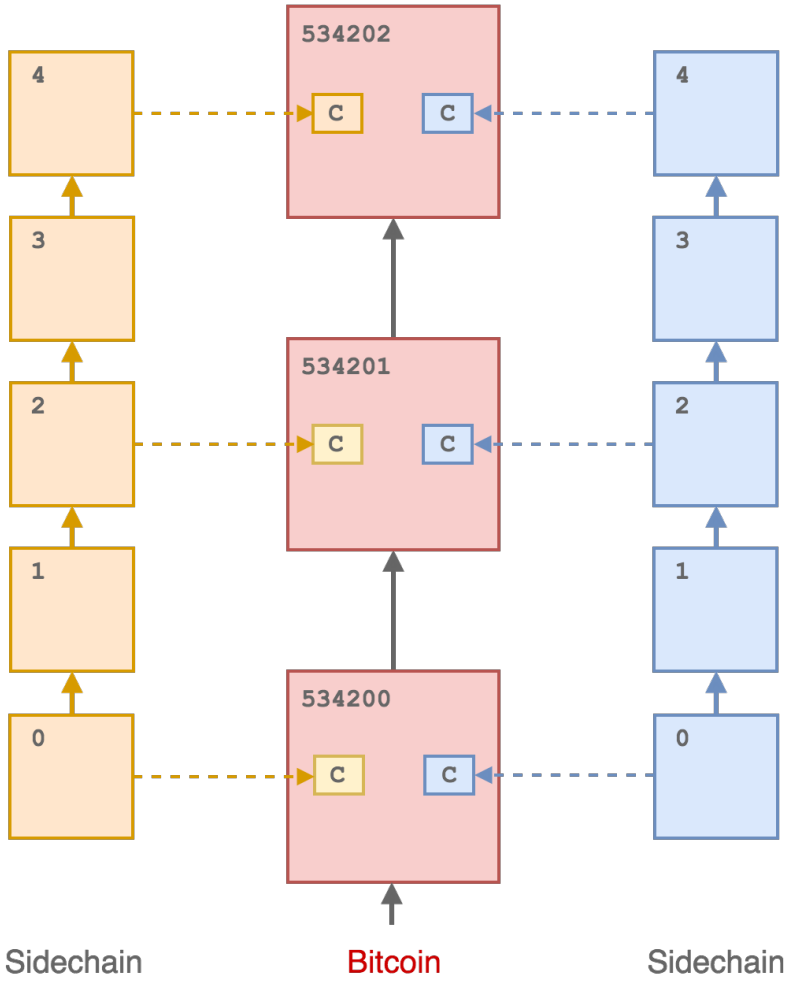


Illustration of conflicting sidechain blocks simultaneously attesting to a mainchain.

To illustrate this point, we consider a sidechain or alt-chain whose state (i.e. the chain tip block header) is periodically time-stamped into the Bitcoin blockchain. This does not lead to immutability of the sidechain, since alternative conflicting states (i.e. forks) can also be time-stamped simultaneously. Any property of immutability then must ensure that the sidechain state is linked to a specific commitment in the Bitcoin chain via some trusted mechanism - some authority (who may be effectively operating the sidechain) is responsible for defining the sequence of timestamps that correspond to the un-forked sidechain. This then relies on the integrity of the commitment mechanism: multiple versions of a sidechain can be created with multiple simultaneous timestamped commitments into Bitcoin. This could be used to execute a double spend attack by collusion of a block signing federation with the commitment authority.

The MainStay protocol is designed to eliminate the requirement for any type of trust in cryptographic proof of immutability (PoI) by initiating a *fan-in-only* transaction *staychain* within the Bitcoin blockchain that is uniquely committed to the genesis block of the sidechain, as described in the next section. The protocol does not employ OP_RETURN outputs, providing additional privacy, censorship resistance and efficiency.

3.1.2 Protocol

The MainStay protocol employs the underlying concept of a *staychain* of linked transactions within the Bitcoin mainchain, where all transactions in the staychain conform to having only a a single output, preventing branching and any possibility of alternate staychain histories. By anchoring the staychain *base* transaction ID into the genesis block of the sidechain, and then committing the state of the sidechain at regular intervals into the staychain, it becomes impossible to roll back or re-write the state of the sidechain without also rolling back the staychain, which is effectively impossible due to the might of Bitcoin's global proof- of-work. Sidechain nodes can validate these commitments and the resulting immutability of the staychain via a connection to a Bitcoin full node. When a sidechain block has been committed to a Bitcoin staychain, this block has been *reinforced* and is as immutable as a Bitcoin block of the same depth.

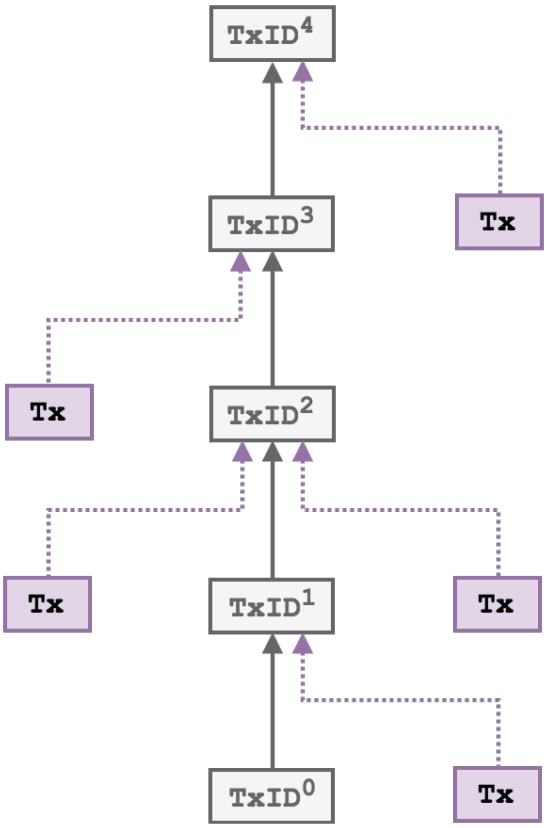
To minimise the encumbrance of the mainstay on the Bitcoin blockchain, and to prevent any potential miner censorship of transactions containing `OP_RETURN` outputs, a homomorphic commitment scheme based on the 'pay-to-contract' (BIP175) protocol is employed. In this approach, commitments from the sidechain are embedded in a single transaction output address, and the staychain is indistinguishable from normal Bitcoin payment transactions. The scheme has been designed so that it is compatible with both multisig (P2SH) and single public key (P2PKH) addresses via BIP32 derivation paths.

The aim of the MainStay protocol is to restrict a sequence of periodic commitments from an external system (referred to here as the *sidechain* without loss of generality) to an un-forkable staychain of transactions in Bitcoin, and to uniquely bind this staychain to the sidechain by commiting the transaction identifier directly the sidechain genesis block. We define a staychain as a sequence of linked transactions where each one has only a single output - transactions can have more than one input (*fan-in*), but maintaining single outputs means only one sequence of commitments is possible from a given initial transaction. Each unique transaction output then represents a *single use seal*.

If the security proposition of a sidechain depends on the Mainstay proof-of-immutability then the mechanism of propagating the staychain must be robust and reliable: if the staychain fails to propagate or is corrupted (e.g. having multiple outputs) then new sidechain state changes (i.e. blocks) will lose the guarantee of immutability - howver it will remain *fail secure* (i.e. previously reinforced transactions are provably unique).

Single-key protocol

In the following general description of the protocol, we assume a single Mainstay key and signing entity. The protocol is also presented in relation to Bitcoin as the proof-of-work mainchain, but it is in principle compatible with any PoW blockchain.



A schematic of a *fan-in-only* chain of linked transactions - a **staychain**. By enforcing single outputs only one possible sequence of transactions is possible.

Initialisation

The initial step in the protocol is the creation of the base transaction `TxID[0]`, which is performed before the initialisation of the sidechain.

1. The signing entity generates a BIP32 extended master private key `xpriv` , and corresponding extended master public key `xpub`. The extended public key is then used to create the base address: `Addr[0]` with a derivation path of `m/0`.
2. The signing entity then creates a transaction (the *base transaction* `BaseTx`) paying an amount of BTC (to cover at least initial transaction fees) to the base address `Addr[0]` as a single P2PKH output.
3. This transaction is broadcast to the Bitcoin network: once it is confirmed in the Bitcoin blockchain it acquires a globally unique transaction ID that is a pointer to the start of the staychain: `TxID[0]`.
4. The sidechain is then configured and linked to the Bitcoin staychain. The pointer `TxID[0]` is embedded directly in the genesis block of the sidechain as metadata in a defined location, along with the `xpub`.

Commitments

The frequency of state commitments is determined by the signing entity : the sidechain may generate blocks more frequently but can only attest once per Bitcoin block (average every 10 minutes). The process of attestation will occur as follows:

1. At each interval `j`, the signing entity will retrieve the sidechain best block hash `blockhash[j]`

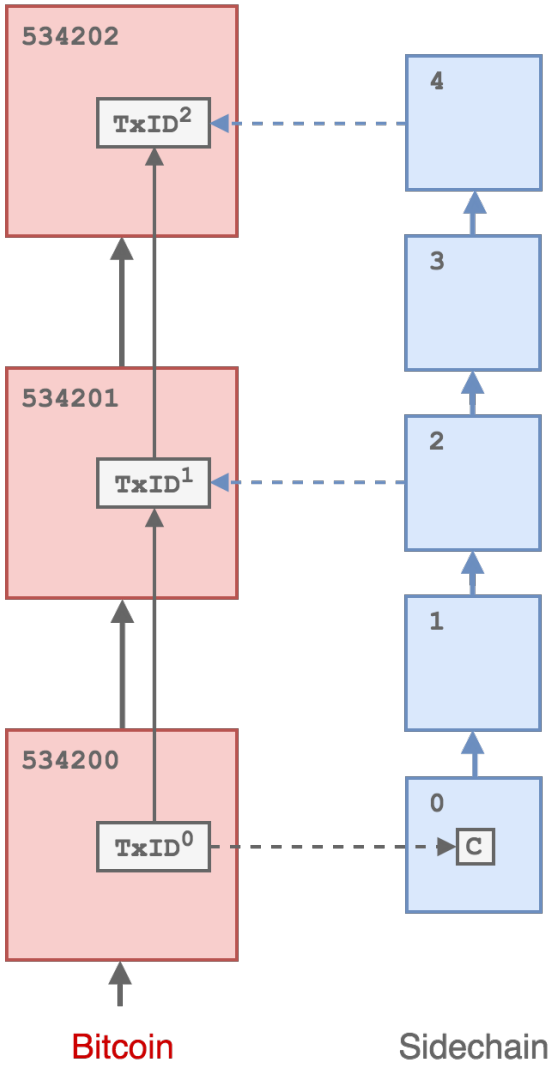
2. The 32 byte `blockhash[j]` is then split into 16 2-byte parts, which are then converted into an array of 16 integers `bhint[16]`.
3. A BIP32 derivation path (the commitment path `path[j]`) is formed from this integer array sequence, and prepended with `m/0`.
4. The commitment address `Addr[j]` is then derived from the `xpub` with `path[j]`.

For example:

```
blockhash_j = 310057788c6073640dc222466d003411cd5c1cc0bf2803fc6ebbfae03ceb4451

path_j = m/0/12544/22392/35936/29540/3522/8774/27904/13329/52572/7360/48936/1020/
↪28347/64224/15595/17489
```

5. The signing entity then creates a Bitcoin transaction with one input spending the single output of transaction `TxID[j-1]` (initially the base transaction when `j = 1`) and paying to a single P2PKH output with address `Addr[j]` 6. The transaction is then signed using the private key derived from `xpriv` with `path[j-1]` 7. The valid transaction is then broadcast to the Bitcoin network. Once it is confirmed in a block, it is referenced by transaction ID: `TxID[j]`



Schematic of the mainstay protocol. Dashed lines represent commitments.

Verification

A block generated on a sidechain that has a mainstay commitment is known as *reinforced* and has the same immutability guarantees as a confirmed Bitcoin block. For any client or user to confirm the status of a sidechain block only requires connections to both Bitcoin and sidechain full nodes. No additional information, beyond what is included in the sidechain and Bitcoin blockchains, is required to validate direct mainstay reinforcements.

This confirmation functions as follows:

1. The base transaction ID `TxID[0]` is retrieved from the sidechain genesis block along with the master `xpub`.
2. `TxID[0]` is retrieved from the Bitcoin blockchain.
3. The staychain is tracked until the unspent tip `TxID[t]`, confirming each component transaction consists of only a single output:

```
TxID[0] → TxID[1] → TxID[2] → TxID[3] → ... → TxID[t]
```

4. The single output P2PKH address of `TxID[t]` is retrieved: `Addr[t]`.
5. Starting at the tip (most recent confirmed block) of the sidechain (block `w`) with block hash `blockhash[w]`, the corresponding BIP32 path is determined: `path[w]`.
6. `Addr[w]` is derived from `path[w]` and the master `xpub`.
7. If `Addr[w]` equals `Addr[t]` block `w` on the sidechain (and all below it) are confirmed as reinforced.
8. If not true, the sidechain block height is decremented: `w ← w  1` and the check repeated.

The above protocol would only need to be followed for the initial sync of a mainstay connected node: once the staychain tip transaction `TxID[t]` has been identified, additional attestations can be confirmed by monitoring when `TxID[t]` is removed from the Bitcoin UTXO set. The new staychain tip `TxID[t+1]` will then be included in the most recent Bitcoin block.

Staychain feed in

To maintain the persistent operation of a staychain, it must be continually funded to pay for mainchain (Bitcoin) mining fees. The staychain can always be funded with a substantial amount of Bitcoin at the beginning (i.e. at the base transaction stage) however it may be required to 'top-up' the funding at a later stage. This is possible without breaking the immutability of the staychain: the only required condition for immutability is that there is always only one output of any transaction in the chain - and that the staychain cannot bifurcate. Inputs however can be added by anyone: additional funding can be added with `SIGHASH_ANYONECANPAY` inputs. The base transaction will always define the commitment sequence through to the tip.

Federated protocol

An important property of the Mainstay protocol is that it does not require trust in any party, including the entity holding the staychain base private key (`xpriv`) to confirm that a given sidechain state is immutable. However trust is required in this entity to ensure that the mainstay is persistent, and that the system continues to operate (i.e. commitments continue to be generated). If the key was stolen then an attacker could steal the Bitcoin in the staychain tip output and prevent further confirmations. To remedy this, the sidechain would need to be hard-forked to reset the mainstay (i.e. to commit a new base transaction into the sidechain).

Sidechains can be operated using a federated consensus protocol, where a fixed federation of separate entities are required to cooperate to generate a new block to add to the blockchain. This is typically implemented with `m` distinct

entities, where a threshold of n are required to add their signature to generate a new valid block. This has the advantage of being very scalable and efficient, and also retains some level of decentralisation, not requiring trust in any single entity. In the case of a federated sidechain employing Mainstay to Bitcoin, the operation of Mainstay can achieve the same security properties and guarantees as the federated block signing protocol. In this case, the staychain would be controlled with an n of m multisignature script: n signers are required to cooperate to operate the Mainstay. m n keys can be lost or compromised and the Mainstay will still function. This requires some modifications to the protocol described above, as follows.

Initialisation

1. Each signing node i where i = 1, ..., m generates a master extended private key xpriv[i] and corresponding extened public xpub[i].
2. The signing nodes then cooperate to create an n of m multisig redeem script (where m is the total number of signing nodes and n is the number of signatures required) containing m base public keys derived from each xpub[i] via a path m/0.
3. The redeem script is then hashed to create a P2SH address Addr[0].

4. A transaction is then created with Addr[0] as a single P2SH output and funded with with sufficient BTC for initial fees and then broadcast to the Bitcoin network. 5. Once confirmed, it is now publicly verifiable that the redeem script hash corresponds to the published n , m and all the xpub[i]. 6. The TxID of the transaction TxID[0] is retrieved and committed into the genesis block of the sidechain along with each xpub[i].

Commitments

1. At each attestation interval j, each of the mainstay signing nodes i will independently retrieve the sidechain tip block hash blockhash[j][i].
2. Each node splits the 32 byte blockhash[j][i] is then split into 16 2-byte parts, which are then converted into an array of 16 integers bhint[16].
3. A BIP32 derivation path (the commitment path path[j][i]) is formed from this integer array sequence, and prepended with m/0.
4. For each node i, The commitment public key pubkey[j][i] is then derived from the xpub[i] with path[j][i].
5. n of m signing nodes then combine pubkey[j][i] to derive a redeem script and corresponding P2SH address Addr[j].
6. A transaction spending the single output of TxID[j1] and paying to Addr[j] is created.
7. n of m signing nodes then verify that Addr[0] corresponds to the correctly derived base keys.
8. The transaction is then signed by each of n (any subset of m) signing nodes in turn using the derived private key xpriv[i] with path[j-1][i].
9. The transaction is then broadcast to the Bitcoin network, validated and then mined into a block, generating TxID[j].

Note: Bitcoin multisig redeem scripts are structured as follows: OP_n pubkey[1] pubkey[2] ... pubkey[m] OP_m OP_CHECKMULTISIG

Verification

1. The base transaction ID `TxID[0]` is retrieved from the sidechain genesis block along with the `n` master `xpub[i]`
2. `TxID[0]` is retrieved from the Bitcoin blockchain.
3. The staychain is tracked until the unspent tip `TxID[t]`, confirming each component transaction consists of only a single output:

```
TxID[0] → TxID[1] → TxID[2] → TxID[3] → ... → TxID[t]
```

4. The single output P2SH address of `TxID[t]` is retrieved: `Addr[t]`.
5. Starting at the tip (most recent confirmed block) of the sidechain (block `w`) with block hash `blockhash[w]`, the corresponding BIP32 path is determined: `path[w]`.
6. `Addr[w]` is derived from `path[w]` and `m` of the master `xpub[i]`
7. If `Addr[w]` equals `Addr[t]` block `w` on the sidechain (and all below it) are confirmed as reinforced.
8. If not true, the sidechain block height is decremented: `w ← w  1` and the check repeated.

3.1.3 Mainstay Service Protocol

This document describes the overall design and principles of the Mainstay connector service protocol which is used to provide trustless immutability to third party systems as a service. This immutability is derived from the Bitcoin blockchain Proof-of-Work in an extensible, scalable and efficient way.

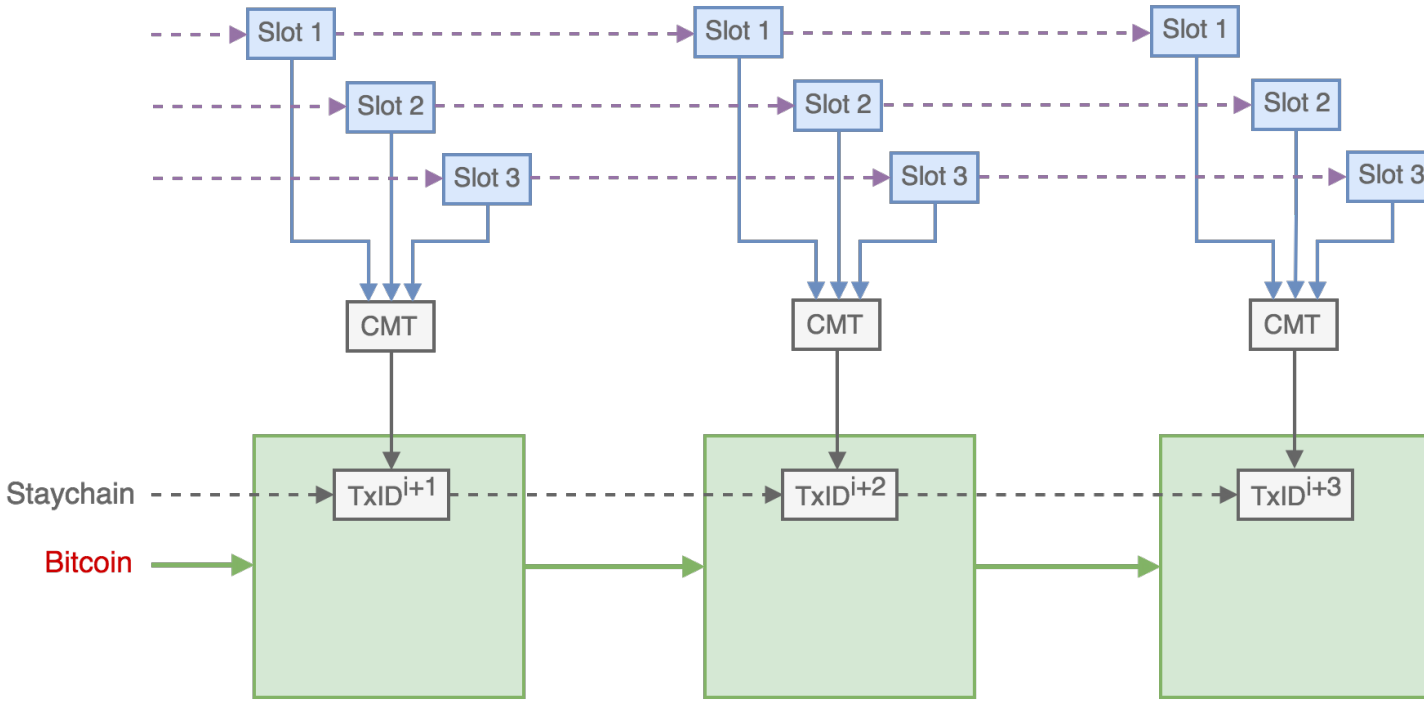
Overview

The primary purpose of the Mainstay scheme is to provide a cryptographic *Proof of Immutable State* (PoIS) for a succession of changing states of some arbitrary system or process - i.e. proof that the sequence of states has only a single, linked, verifiable history that cannot be altered (or double-spent). This PoIS is obtained via the trustless immutability inherent to the Bitcoin blockchain, where proof-of-work (via the *permissionless* mining of blocks to extend the chain) creates a practically irreversible and incorruptible ordering of transactions that does not rely on trust in any entity. The external system with a 'sequence of changing states' that can be proven as immutable via the Mainstay protocol may in some instances be a separate blockchain (i.e. a *sidechain*). However, there are many other systems and processes where a PoIS (which is a proof of a single verifiable history) is of substantial value, such as in document tracking processes, critical software development and organisational governance.

The underlying mechanism of the mainstay protocol is a sequence of successive commitments to a *fan-in-only* sequence of linked transactions on the Bitcoin blockchain, where each transaction has only a single output - referred to as the *staychain*. By enforcing the rule that all the transactions in the staychain can only have a one output, the staychain can only have a single, non-branching history from the base of the chain to the tip. Following this rule, the staychain state is as immutable as the Bitcoin blockchain and is backed by its immense proof-of-work. Verifiable state commitments to the staychain are then also immutable, and the immutability of any sequence of committed states can be proven by verifying the validity of the staychain.

State commitments are made to staychain transactions using the homomorphic *pay-to-contract* scheme, where the public key of the output is modified verifiably by the commitment value. This enables the commitments to be verified independently while the staychain remains indistinguishable from other standard bitcoin transactions. The commitment embedded in a particular staychain transaction output consists of a single 256 bit number, however this can in turn incorporate a number of separate commitments as a *Merkel Tree* where the Merkle tree *root* is committed to the staychain and a *leaf* commitment inclusion can be verified via a Merkle path proof.

In order to maintain the property of immutability for sequential commitments in sequential Merkle Trees anchored to the staychain, only one simple additional rule must be followed: each commitment from a particular seqeunce of states must always be be verifiably committed to the *same* position within the Merkle tree. If the commitment is always validated in the same position, then the sequence is as immutable (as in having only a single possible non-branching history) as the the root commitment into the staychain.



Schematic of the commitment of states from three slots to the Connector Merkle Root (CMR) which is then committed to the Bitcoin staychain, over three consecutive blocks. The sequence of commitments to a specified slot is as immutable as the the Bitcoin staychain.

The Mainstay service protocol provides a mechanism for service users to access a specific position in the commitment Merkle tree (refered to as a *slot*) which is then regularly committed to a unique Bitcoin staychain. This enables the provision of *Immutability as a Service* where a number of sidechain or other systems/processes can commit to and utilise a single Bitcoin staychain, at a substantially reduced cost (in terms of Bitcoin transaction fees) compared to operating a separate transaction staychain within Bitcoin for each individual application. The service provider operating the staychain, and the connection service, can agree service terms for each user and then assume responsibility for propagating the staychain and paying the Bitcoin fees.

Commitment Merkle Tree

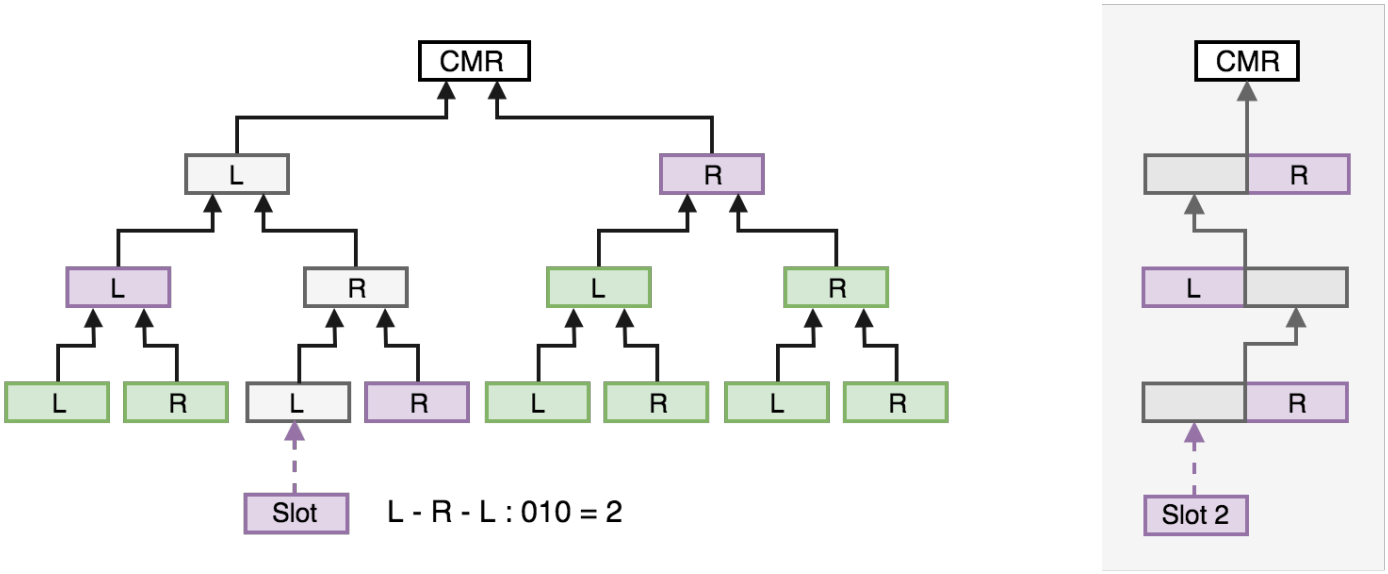
A Merkle tree is a data structure that enables a list of cryptographic commitments to be compressed into a single Merkle root with efficient and secure verification. As a result of the binary tree structure, a cryptographic proof that a specified commitment is included in the derivation of a root can be verified with O(log n) complexity, and the proof requires only O(log n) storage. A Merkle tree is defined by hash function (i.e. SHA256) and an assignment function, which maps each node to the concatenation of the hashes of its child nodes. Each parent node *N* is then defined from the left (L) and right (R) child nodes as:

```
N(Parent) = SHA256(N(L)||N(R))
```

Proof of the inclusion a commitment (as a leaf of the tree) is then generated from a traversal of the tree from the leaf through to the root, and is authenticated by verifying the path of concatenated hashes. However - for the connector

protocol - the additional requirement in order to prove immutability across successive commitments is that a particular sequence of successive commitments from an external (client) process are included in the corresponding sequence of Commitment Merkle Trees (CMTs) in the **same** leaf position each time the root is committed to the Bitcoin staychain. This specific Merkle leaf position is referred to as a *slot* and is designated by an integer `slotid`.

The `slotid` is defined according to the binary *path* from the leaf through to the Merkle root, which consists of the sequence of `L` and `R` concatenations (see Fig. 2). The `slotid` defined in this way does not change as the tree is extended with more slots and the depth of the tree is increased (increasing the depth of the tree will simply increase the size of the proofs).



Slot proof

Schematic of the structure of a CMT with 8 leaves, where the leaf position (slot) is determined by the path. The sequence of concatenated hashes from the leaf through to the root forms a slot-proof that a commitment was made is a specified position.

Slot-proofs

The Mainstay service maintains a current version of the full tree as commitments are added from users via slots (see below). If a slot is not active (i.e. is not associated with a client or user) the corresponding leaf commitment is set to zero. Once the root of the current updated tree (CMR) is committed into a new staychain transaction, then *slot-proofs* are generated for each `slotid` with a submitted commitment. The slot-proof consists of the hash sequence and concatenation order for the specific Merkle path to the commitment Merkle Root (CMR).

The slot-proof for a specific `slotid` provides cryptographic proof that a particular commtment `Com` was committed to a specified staychain (identified by the *base* transaction ID `TxID[0]`) at a staychain height `txheight` and at that specific slot position.

Example slot-proof:

```
{
    commitment: "1a39e34e881d9a1e6cdc3418b54aa57747106bc75e9e84426661f27f98ada3b7",
    ops: [
        {
            append: true,
            commitment:
↪"3a39e34e881d9a1e6cdc3418b54aa57747106bc75e9e84426661f27f98ada3b7"
```

(continues on next page)

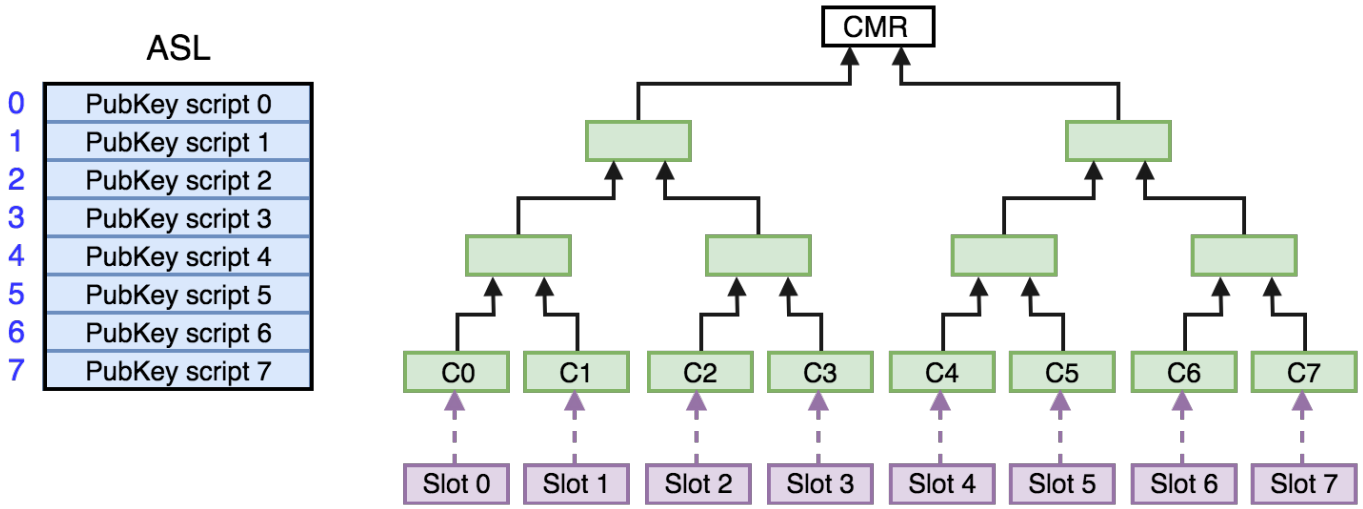
(continued from previous page)

```
        },
        {
            append: false,
            commitment:
↪"4a39e34e881d9a1e6cdc3418b54aa57747106bc75e9e84426661f27f98ada3b7"
        },
    ],
    merkle_root: "5a39e34e881d9a1e6cdc3418b54aa57747106bc75e9e84426661f27f98ada3b7"
}
```

To obtain a Proof of Immutable State (PoIS) one or more slot-proofs on same staychain and with the same `slotid` are required as described below.

Slot connection

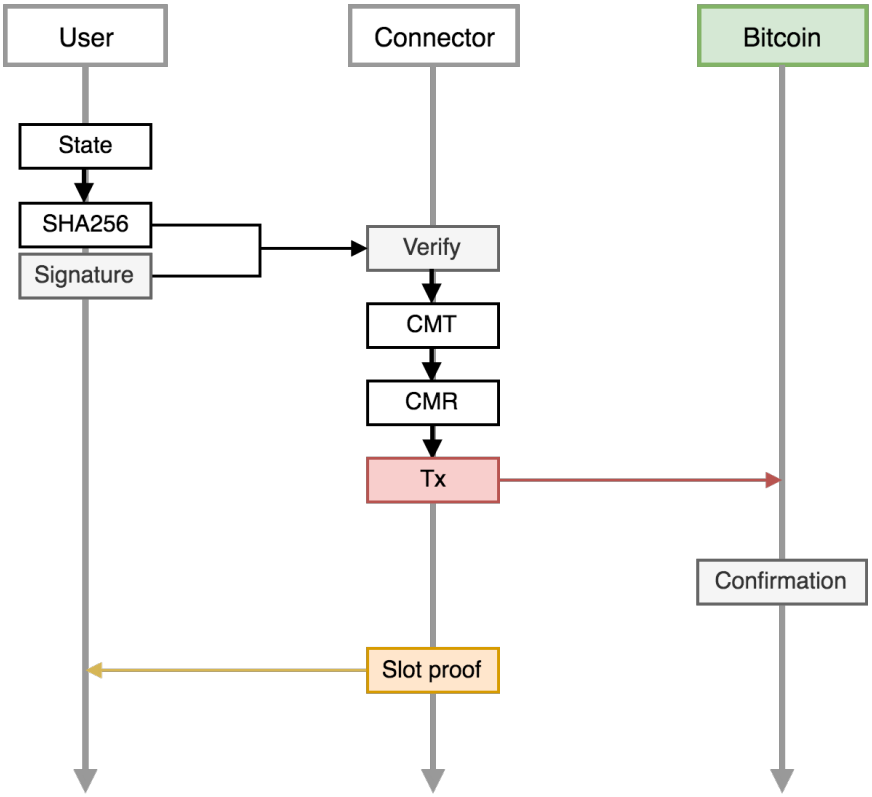
Individual users (clients) of the connector service are granted exclusive permission to add a 32 byte commitment to a specific `slotid` for as long as a service agreement remains in force. Upon the commencement of a service agreement with a client, the client will be assigned a free `slotid` (the lowest number currently unused). The client will then provide a *validation script* `PubKeyScript` which contains the policy for authenticating a submitted commitment. The policy is determined by the client, and can be a single public key requiring a single commitment signature or an *m-of-n* multisignature script (or any other policy logic). In addition, the client will be provided with API access details and tokens.



Schematic of a CMT with 8 slots. The mapping to the active slot list (ASL) is shown.

On the initiation of a connection, the `PubKeyScript` is added to the *active slot list* (ASL) in the position corresponding to `slotid`. The connector service API then recieves signed commitments (signed in accordance with the `PubKeyScript` policy) from the client and the signatures are verified using the `PubKeyScript`. If the signatures are valid then the commitment is added to the CMT at the `slotid` position. The connector server updates the cached CMT root each time a new slot commitment is recieved and verified. New verified commitments arriving for a particular slot overwrite the pervious commitment.

At intervals determined by the staychain attestation frequency, the commitment server performs commitments to the Bitcoin staychain following the BIP175 *pay-to-contract* protocol.



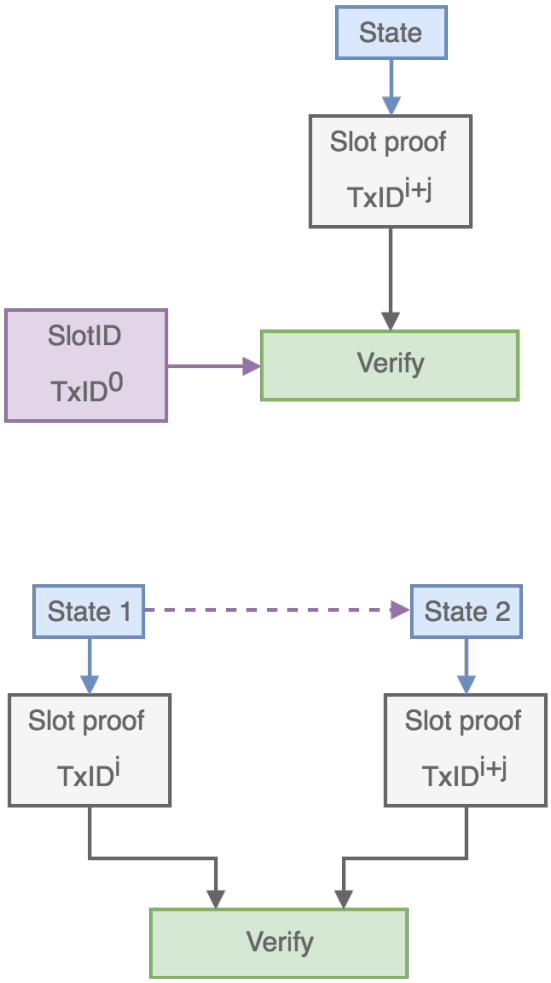
Protocol and message flow for a user interacting with the service via a single slot.

Once the commitment transaction has been confirmed, the commitment server then generates the slot-proofs for each of the active slots. These slot-proofs are then available to retrieve by the clients via the connector service API.

Proof of Immutable State

Clients retrieve slot-proofs from the connector service API in order to confirm a PoIS using a client side confirmation tool that queries a Bitcoin blockchain node via the RPC interface. The confirmation tool can be configured for a particular staychain and slot, which is defined by a *start point* Bitcoin `TxID[s]`, the staychain `xpub[i]` and the `slotid`. The start point transaction ID can be any staychain transaction before the transaction ID of the first slot-proof (the confirmation tool takes the slot-proof `TxID[j]` and traverses backward along the staychain until the `TxID[s]` is found).

Any slot-proof can then be passed to the confirmation tool, which will determine whether the slot-proof (and hence state commitment) is committed to the specified staychain at the specified slot position. This is proof that the state commitment is part of the sequence defined by the staychain and slot position (if intermediate states also form a hash-chain, then each of the intermediate states is also proven immutable). Alternatively, the confirmation tool can determine whether any two slot-proofs are on the *same* slot position and staychain (irrespective of the configuration) - this is proof that both of the slot-proof commitments are part of the same immutable sequence.



Verification pathways for state verification. Top: Verification that a particular state is committed to a particular staychain and `slotid`. Bottom: Verification that a sequence (two or more) of states are committed to the same staychain and `slotid` is sequential transactions.

Commitment frequency and fee policy

The service agreement with individual slot clients will specify the target staychain transaction frequency and fee policy. Due to the inherent nature of proof-of-work, the block generation interval on the Bitcoin blockchain is highly variable, and there is no guarantee of transaction confirmation in any particular time period which is also subject to the level of network congestion.

The staychain policy will specify a target transaction period `ctarget` (e.g. 1 hour) and the connector server will generate and broadcast a new staychain transaction containing the CMR every `ctarget` interval (irrespective of how many Bitcoin blocks have been generated). The transaction fee will initially be set at the value estimated (via a third party fee estimation app) for confirmation within 3 blocks, up to a maximum of value of `maxfee`. `maxfee` (in BTC) is the maximum fee the service will pay per hour. In the case a transaction is not confirmed within 1 hour (due to network congestion and `maxfee` being insufficient) then the staychain transaction (updated with the latest CMR) is re-broadcast with an additional `maxfee` for the next 1 hour period (i.e. the fee will now be 2x `maxfee`) using the replace-by-fee (RBF) . This will then be repeated each `ctagrget` until the transaction is confirmed.

The value of `maxfee` may be increased and `ctarget` decreased as more clients join the service, increasing the reliability and regularity of proofs.

Staychain multi-signature security

A fundamental property of the Mainstay protocol is that users do not have to trust the connector service (or anyone else) to guarantee immutability - this is provided by the global proof-of-work securing the Bitcoin blockchain combined with slot-proofs. However, in order to provide a continuous and reliable service, the staychain of commitment transactions must remain in the control of the connector service. If the private keys controlling the staychain output (i.e. the base private keys) are lost or stolen, then the new state commitments cannot be immutably linked, and users would be forced to coordinate updates to a new staychain. To provide the required security and resiliency of the service the staychain is controlled by a multi-sig script (as described in the whitepaper). In addition, each base private key (`xpriv[i]`) of the staychain is generated and secured inside of a BIP32-compatible hardware security module (HSM).

3.1.4 Application

The mainstay repository is an application that implements the Mainstay protocol - available at github.com/commerceblock/mainstay. It consists of a Go daemon that performs attestations of the Ocean network along with client commitments to Bitcoin in the form of a commitment merkle tree.

Mainstay is accompanied by a Confirmation tool that can be run in parallel with a Bitcoin network node to confirm attestations and prove the commitment inclusion in Mainstay attestations.

Prerequisites

- Go (https://github.com/golang)
- Bitcoin (https://github.com/bitcoin/bitcoin)
- Zmq (https://github.com/zeromq/libzmq)

Instructions

Attestation Service

- Install Go and the attestation service by following `scripts/build.sh`
- Setup up database collections and roles using `scripts/db-init.js`
- Setup `conf.json` file under `/config` by following config guidelines
- Run service
  - Regtest mode
    - Run service: `mainstay -regtest`
    - Run signer: `go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go -regtest`
    - Insert commitments to "ClientCommitment" database collection in order to generate new attestations
  - Testnet/Mainnet mode
    - Download and run a full Bitcoin Node on testnet mode, fully indexed and in blocksonly mode.
    - Fund this wallet node, send all the funds to a single (`m of n` sig) P2SH address and store the `TX_HASH`, `PRIVKEY_x` and `REDEEM_SCRIPT` of this transaction, where `x in [0, n-1]`.

      (In the case of an Ocean-type network the `TX_HASH` should be included in the genesis block using the config option `attestationhash`)

* Follow the same procedure to generate a single (`m of n` sig) P2SH address used to topup the service and store the `TOPUP_ADDRESS`, `TOPUP_PRIVKEY_x` and `TOPUP_SCRIPT`.
* Run the mainstay attestation service by:

  ```
  mainstay
  ```

  Command line parameters should be set in `.conf` file
* Run transaction signers of the m-of-n multisig P2SH addresses for `x in [0, n-1]` by:

  ```
  go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.
  go -pk PRIVKEY_x -pkTopup TOPUP_PRIVKEY_x -host SIGNER_HOST
  ```

  Command line parameters should be set in the corresponding signer `.conf` file

- Unit Testing
  - `/$GOPATH/src/mainstay/run-tests.sh`

Mainstay configuration

Sample Config

```
{
    "staychain": {
        "initTx": "87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1dfa62ac9e1d83cd840e1",
        "initScript":
↪"51210381324c14a482646e9ad7cf82372021e5ecb9a7e1b67ee168dddf1e97dafe40af210376c091faaeb6bb3b74e0568
↪",
        "initChaincodes":
↪"0a090f710e47968aee906804f211cf10cde9a11e14908ca0f78cc55dd190ceaa,
↪0a090f710e47968aee906804f211cf10cde9a11e14908ca0f78cc55dd190ceaa",
        "topupAddress": "2MxBi6eodnuoVCw8McGrf1nuoVhastqoBXB",
        "topupScript":
↪"51210381324c14a482646e9ad7cf92372021e5ecb9a7e1b67ee168dddf1e97dafe40af210376c091faaeb6bb3b74e0568
↪",
        "regtest": "1"
    },
    "main": {
        "rpcurl": "127.0.0.1:18000",
        "rpcuser": "USERNAME",
        "rpcpass": "PASSWORD",
        "chain": "regtest"
    },
    "clientchain": {
        "rpcurl": "127.0.0.1:19000",
        "rpcuser": "USERNAME",
        "rpcpass": "PASSWORD",
        "chain": "main"
    },
    "signer": {
        "publisher": "*:5000",
        "signers": "node0:1000,node1:1001"
    },
    "db": {
        "user":"user",
        "password":"pssword",
        "host":"localhost",
```

(continues on next page)

(continued from previous page)

```
        "port":"27017",
        "name":"mainstay"
    },
    "fees": {
        "minFee": "5",
        "maxFee": "50",
        "feeIncrement": "2"
    },
    "timing": {
        "newAttestationMinutes": "60",
        "handleUnconfirmedMinutes": "60"
    }
}
```

Config Parameters

Compulsory

Currently `main` config category is compulsory. This should be made optional in the future as tools that do not require `main` rpc connectivity options use this.

- `main` : configuration options for connection to bitcoin node
  - `rpcurl` : address for rpc connectivity
  - `rpcuser` : user name for rpc connectivity
  - `rpcpass` : password for rpc connectivity
  - `chain`: chain name for inner config, i.e. testnet/regtest/mainnet

The `staychain` category is compulsory and can be set from either .conf file or command line arguments. The configuration below is optional as preferred entry is via command line - *options*.

- `staychain` : configuration options for staychain parameters
  - `initTx` : initial transaction sets the state for the staychain
  - `initScript` : initial script used to derive subsequent staychain addresses
  - `initChaincodes`: chaincodes of init script pubkeys used to derive subsequent staychain addresses
  - `topupAddress` : address to topup the mainstay service
  - `topupScript` : script that requires signing for the topup

Several other subcategories become compulsory only if the base category exists in the `.conf` file.

For the base categories `db` and `signer` the following parameters are compulsory:

- `db` : configuration options for database
  - `user` : db user name
  - `password` : db user password
  - `host` : db host address
  - `port` : db host port
  - `name` : db name

- `signer` : zmq signer connectivity options
  - `signers` : list of comma separated addresses (host:port) for connectivity to signers

Optional

All the remaining conf options are optional. These are explained below:

- `signer`
  - `publisher` : optionally provide host address for main service zmq publisher

Default values are set in `attestation/attestsigner_zmq.go`.

- `fees` : fee configuration parameters for attestation service
  - `minFee` : minimum fee for attestation transactions
  - `maxFee` : maximum fee for attestation transactions
  - `feeIncrement` : fee increment value used when bumping fees

Default values are set in `attestation/attestfees.go`

- `timing` : various timing configuration parameters used by attestation service
  - `newAttestationMinutes` : option in minutes to set frequency of new attestations
  - `handleUnconfirmedMinutes` : option in minutes to set duration of waiting for an unconfirmed transaction before bumping fees

Default values are set in `attestation/attestservice.go`

Command Line Options

Currently only parameters in the `staychain` category can be parsed through command line arguments.

These command line arguments are:

- `tx` : argument for initTx as above
- `script` : argument for initScript as above
- `chaincodes`: argument for initChaincodes as above
- `addrTopup` : argument for topupAddress as above
- `scriptTopup` : argument for topupScript as above

Env Variables

All config parameters can be replaced with env variables. An example of this is `config/conf.json`.

The Config struct works by first looking for an env variable with the name set and if an env variable is not found then the config parameter is set to the actual value provided.

If the config argument is not to be used, **no value** should be set in the conf file. Warnings for invalid argument values are provided in runtime.

Client Chain Parameters

Parameters used for client chain confirmation tools and are not part of Config struct used by service.

- `clientchain` : configuration options for connectivity to client rpc node

Same configuration options as `main`. The `clientchain` name can be replaced with any name to match the sidechain. See `cmd/confirmationtool/conf.json`. This is not used by Config struct, only by `config::NewClientFromConfig()`.

Tools

Along with the Mainstay daemon there is various tools offered serving utilities for both Mainstay operators and clients of Mainstay. These tools and their functionality are briefly summarized below:

Transaction Signing Tool

The transaction signing tool can be used by each signer of the mainstay multisig to sign transactions.

```
go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go -pk PRIVKEY
-pkTopup TOPUP_PRIVKEY -host SIGNER_HOST
```

where:

- `PRIVKEY`: private key of address initial funds were paid to
- `TOPUP_PRIVKEY`: private key of the topup address
- `SIGNER_HOST`: host address that the signer is publishing at and for the mainstay service to subscribe to

The tool subscribes to the mainstay service in order to receive confirmed attestation hashes and new bitcoin attestation transaction pre-images. These transactions are signed and broadcast back to the mainstay service.

To do the signing ECDSA libraries are used and and no Bitcoin node connection is required.

The live release of Mainstay will be instead using an HSM interface. Thus this tool is for testing purposes only.

Client Signup Tool

The client signup tool can be used to sign up new clients to the mainstay service.

```
go run $GOPATH/src/mainstay/cmd/clientsignuptool/clientsignuptool.go
```

Connectivity to the mainstay db instance is required. Config can be set in `cmd/clientsignuptool/conf.json`.

The client will need to provide an ECDSA public key. The corresponding private key will be used by the client to sign the commitment send to the mainstay API. The signature is then verified by the API using the public key provided.

The tool assigns a new position to the client in the commitment merkle tree and also provides a unique auth_token for authorizing API POST requests submitted by the client. For random auth-token generation only, token generator tool `cmd/tokengeneratortool` can be used.

For examples check

Token Generator Tool

The token generator tool can be used to generate unique authorization tokens for client signup.

```
go run $GOPATH/src/mainstay/cmd/tokengeneratortool/tokengeneratortool.go
```

Client Confirmation Tool

The confirmation tool can be used to confirm all the attestations of a client Ocean-type sidechain to Bitcoin and wait for any new attestations that will be happening.

Running this tool will require a full Bitcoin testnet node and a full Ocean node. Connection details for these should be included in `cmd/confirmationtool/conf.json`.

The `API_HOST` field should be set to the mainstay URL. This can be updated in `cmd/confirmationtool/confirmationtool.go`.

To run this tool you need to first fetch the `TX_HASH` from the `attestationhash` field in the Ocean genesis block, as well as the publicly available `REDEEM_SCRIPT` of the attestation service multisig. The tool can also be started with any other `TX_HASH` attestation found in the mainstay website. A client should use his designated `CLIENT_POSITION` that was assigned during signup and run the tool using:

```
go run cmd/confirmationtool/confirmationtool.go -tx TX_HASH -script
REDEEM_SCRIPT -position CLIENT_POSITION -apiHost https://mainstay.xyz
```

This will initially take some time to sync up all the attestations that have been committed so far and then will wait for any new attestations. Logging is displayed for each attestation and for full details the `-detailed` flag can be used.

Commitment Tool

The commitment tool can be used to send hash commitments to the Mainstay API.

The tool functions in three different modes:

- Init mode to generate ECDSA keys
- One time commitment mode
- Recurrent commitment of Ocean blockhashes mode

Various command line arguments need to be provided:

- *-apiHost*: host address of Mainstay API (default: https://mainstay.xyz)
- *-init*: init mode to generate ECDSA pubkey/privkey (default: false)
- *-ocean*: ocean mode to use recurrent commitment mode (default: false)
- *-delay*: delay in minutes between sending commitments in ocean mode (default: 60)
- `-position`: client position on commitment merkle tree
- `-authtoken`: client authorization token generated on registration
- `-privkey`: Client private key, if signature has not been generated using a different source

Ocean connectivity details need to be provided in the `cmd/commitmenttool/conf.json` file if Ocean mode is selected.

For examples check

Multisig Tool

The multisig tool can be used to generate multisig scripts and P2SH addresses for Mainstay configuration.

Two modes:

- Regtest mode (multisig/P2SH generation for regtest and unit tests)
- Main mode

Command line arguments:

- `-chain`: set bitcoin chain configuration to regtest/testnet/mainnet (defaults to mainnet)
- `-nKeys`: num of keys (main mode)
- `-nSigs`: num of sigs (main mode)
- `-keys`: list of comma separated pub keys in hex format (main mode)
- `-keysX`: list of comma separated pub key X coordinates (main mode if -keys not set)
- `-keysY`: list of comma separated pub key Y coordinates (main mode if -keys not set)

The multisig generated can be used as the Mainstay `initScript` config option.

The P2SH address generated can be used to pay funds to initiate Mainstay.

Examples on how to run:

- `go run $GOPATH/src/mainstay/cmd/multisigtool/multisigtool.go -chain=mainnet -nKeys=2 -nSigs=1 -keysX=1707394401087380176538581041992839646429902776804130532161562185465146941303980993275118670323268013022806344211302215001475 -keysY=475813022329769762590164284448176075334749443379722569322944728779216384721,112227001874758666872359482845413579097178565373926604945912057881796816853651`
- `go run $GOPATH/src/mainstay/cmd/multisigtool/multisigtool.go -chain=testnet -nKeys=2 -nSigs=1 -keys=03e52cf15e0a5cf6612314f077bb65cf9a6596b76c0fcb3403e52cf15e0a5cf6612314f077bb65cf9a6596b76c0fcb34b682f673a8314c7b33`
- `go run $GOPATH/src/mainstay/cmd/multisigtool/multisigtool.go -chain=regtest`

For example use cases go to docs.

Initialising Mainstay

A set of instructions for setting up the Mainstay service. This requires running 2 Bitcoin full nodes and setting up a 1 of 2 P2SH multisig address for the attestations. The Mainstay service coordinates with the transaction signing tools via zmq, sending attested hashes, new commitments and transactions to sign. All transactions are committted through the main service.

Alternatively HSM interfaces can be used instead of the transaction signing tools. In this case, the P2SH address is generated by combining the pubkeys of the HSMs instead. The rest of the functionality should work in a very similar manner.

Initial attestation

Generate 2 addresses

```
$ bitcoin-cli -datadir=testnetbtc-datadir/ getnewaddress
2MwvCUjtecBAFcc7SWhEu8NyT1bLsCRtN6J

$ bitcoin-cli -datadir=testnetbtc-datadir/ getnewaddress
2N4FJ6xpbGdUvC8RjfMmQ6bzWXwEfWCFcYF
```

Generate multisig 1 of 2 address

```
$ bitcoin-cli -datadir=testnetbtc-datadir/ addmultisigaddress 1 "[\
↪"2NFBB5okotyGFLmceXK7q18ufuv11NmefUJ\",\"2NE8WKRRuj53udVsuyj5GbVfyUNN6ZSE4ia\"]" ""␣
↪legacy

{
  "address": "2N5ckx6eXY5vx3DLwBwSZsNVShiZ6k6mSGd",
  "redeemScript":
↪"512103d11753d31309988c323142a0171e5b2319a8651479835afa4ab8ecb6442141b921034f0538871c910019b8e15a3
↪"
}
```

Dump priv keys

```
$ bitcoin-cli -datadir=testnetbtc-datadir/ dumpprivkey␣
↪2NFBB5okotyGFLmceXK7q18ufuv11NmefUJ␣
↪cTgsB8DjF2vjhFrtCPopvknmNWP6CTQeb1Sd9zvXxRF5qHp9V4ct

$ bitcoin-cli -datadir=testnetbtc-datadir/ dumpprivkey␣
↪2NE8WKRRuj53udVsuyj5GbVfyUNN6ZSE4ia␣
↪cNGEfurnx9oL6z8XUuigPXxoxs5cmMxfwwnDbAa258StQ4AQTH8P
```

Import generated multisig address

```
$ bitcoin-cli -datadir=testnetbtc-datadir/ importaddress␣
↪2N5ckx6eXY5vx3DLwBwSZsNVShiZ6k6mSGd "" false
```

Send funds to generated multisig address

```
$ bitcoin-cli -datadir=testnetbtc-datadir/ getbalance
0.07926900

bitcoin-cli -datadir=testnetbtc-datadir/ sendtoaddress␣
↪2N5ckx6eXY5vx3DLwBwSZsNVShiZ6k6mSGd 0.07926900 "" "" true

87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1dfa62ac9e1d83cd840e1

bitcoin-cli -datadir=testnetbtc-datadir/ sendrawtransaction␣
↪02000000000101bbd869ef95b3280aad7e6d8c77582d1d7a3d0dc60fc3c3c0228df6931c31561b00000000232200205dec
87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1dfa62ac9e1d83cd840e1
```

Topup information

Generate 2 addresses

```
$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ getnewaddress
2MtEZ7J8ZXoieL7iHyUQw91TZpLEcVQTAyK

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ getnewaddress
2MwvCUjtecBAFcc7SWhEu8NyT1bLsCRtN6J
```

Generate multisig 1 of 2 address

```
$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ addmultisigaddress 1 "[\
↪"2MtEZ7J8ZXoieL7iHyUQw91TZpLEcVQTAyK\",\"2MwvCUjtecBAFcc7SWhEu8NyT1bLsCRtN6J\"]" ""␣
↪legacy

{
  "address": "2NBYFyyMpPeLCb67bykLBHMByu1dSRGsim1",
  "redeemScript":
↪"512102a2411030da6082ac32d0166fc19f03e264c6c2a138f83a29120d0b59969670792103d11753d31309988c323142a(
↪"
}
```

Dump priv keys

```
$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ dumpprivkey␣
↪2MtEZ7J8ZXoieL7iHyUQw91TZpLEcVQTAyK
cPLfW9BRRJjZNwNHwrz6B5XEmsTHRFsHYYFRTAQChULT5nUn8FkW

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ dumpprivkey␣
↪2MwvCUjtecBAFcc7SWhEu8NyT1bLsCRtN6J
cTgsB8DjF2vjhFrtCPopvknmNWP6CTQeb1Sd9zvXxRF5qHp9V4ct
```

Running the service

```
go build && go install && mainstay
```

Running the signing tools

- signer 1

```
go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go
-pk cTgsB8DjF2vjhFrtCPopvknmNWP6CTQeb1Sd9zvXxRF5qHp9V4ct -pkTopup
cPLfW9BRRJjZNwNHwrz6B5XEmsTHRFsHYYFRTAQChULT5nUn8FkW -host *:5001
```

- signer 2

```
go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go
-pk cNGEfurnx9oL6z8XUuigPXxoxs5cmMxfwwnDbAa258StQ4AQTH8P -pkTopup
cTgsB8DjF2vjhFrtCPopvknmNWP6CTQeb1Sd9zvXxRF5qHp9V4ct -host *:5002
```

Commitment examples

Signing and Sending Commitments

The commitment tool can be used to send signed commitments to the Mainstay API. The commitment is a 32 byte hash. While for a typical Ocean sidechain Client this will be the latest blockhash, any form of data can be hashed into this form. Various tools exist that can achieve this.

To use the commitment tool the client `private key`, `auth token` and `position` assigned during signup are required, along with the 32 byte hash `commitment` to be signed and send.

- The commitment tool can be used one off to produce a signature for a commitment hash using the client's private key:
- These data can then be submitted via the Mainstay website:

- Alternatively the commitment tool can be used directly to sign and send the commitment to the Mainstay API:
- Or both in one go (or Go):

Key Init

The commitment tool can also be used to generate private/public ECDSA key pairs if run on `init` mode. This is displayed below:

Commitment Verification

To verify if a commitment has been included in an attestation the following API call can be used:

https://mainstay.xyz/api/v1/commitment/verify?position=3&commitment=879d36232614b868a52549cf6961caaa4c8f09d3ecffb63714d

To get more detailed information on the commitment use:

https://mainstay.xyz/api/v1/commitment/commitment?commitment=879d36232614b868a52549cf6961caaa4c8f09d3ecffb63714d40d60

More information on the Mainstay API can be found on github.

3.1.5 Mainstay Service API

Short documentation for the public API offered on the Mainstay website at `https://testnet.mainstay.xyz/api/v1`.

REST framework structure

```
response = json response object

response['error'] : json response error field

timestamp : timestamp in ms

allowance : time taken to respond in ns
```

Public Endpoints

Index

API index page.

request: https://testnet.mainstay.xyz/api/v1

response:

```
{
    "response": "Mainstay-API-v1",
    "timestamp": 1548329067489,
    "allowance":
    {
        "cost": 4562
    }
}
```

Latest Attestation

Provide information on latest Merkle root commitment to the staychain.

request: https://testnet.mainstay.xyz/api/v1/latestattestation

response:

```
{
    "response":
    {
        "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
        "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13"
    },
    "timestamp": 1548329116999,
    "allowance":
    {
        "cost": 1796883
    }
}
```

Latest Commitment

Provide information on latest commitment for a specific slot position.

request: https://testnet.mainstay.xyz/api/v1/latestcommitment?position=3

response:


```
{
    "response":
    {
        "commitment":
↪"d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e",
        "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
        "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13"
    },
    "timestamp": 1548329166363,
    "allowance":
    {
        "cost": 3119659
    }
}
```


Commitment

Fetch commitment information for a specific slot position and merkle_root.

request: https://testnet.mainstay.xyz/api/v1/commitment?position=3&merkle_root=f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b

response:


```
{
    "response":
    {
        "commitment":
↪"d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e",
        "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b"
    },
    "timestamp": 1548329204516,
    "allowance":
    {
        "cost": 1484074
    }
}
```


Commitment Latest Proof

Fetch latest commitment proof for a specific slot position.

request: https://testnet.mainstay.xyz/api/v1/commitment/latestproof?position=1

response:


```
{
    "response":
    {
        "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13",
        "commitment":
↪"d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e",
```


(continues on next page)

(continued from previous page)


```
        "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
        "ops": [
        {
            "append": false,
            "commitment":
↪"5309053b9d4db8f86d2c7ec164645bdf1669111280e49e04c036c323b58f4709"
        },
        {
            "append": false,
            "commitment":
↪"213e122aaec314a94f111dd8dc797814660b680f7258f1d95adec56318eabd7c"
        },
        {
            "append": true,
            "commitment":
↪"406ab5d975ae922753fad4db83c3716ed4d2d1c6a0191f8336c76000962f63ba"
        }]
    },
    "timestamp": 1548330374527,
    "allowance":
    {
        "cost": 19732506
    }
}
```


Commitment Verify

Check if a commitment for a specific slot position is included in an Merkle root.

request: https://testnet.mainstay.xyz/api/v1/commitment/verify?position=1&commitment=5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053

response:


```
{
    "response":
    {
        "confirmed": true
    },
    "timestamp": 1548329867868,
    "allowance":
    {
        "cost": 30212539
    }
}
```


Commitment Proof

Get the merkle commitment proof (*slot proof*) for a specific slot position and merkle root.

request: https://testnet.mainstay.xyz/api/v1/commitment/proof?position=1&merkle_root=f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b

response:

```
{
    "response":
    {
        "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
        "commitment":
↪"5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
        "ops": [
        {
            "append": false,
            "commitment":
↪"21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee869830d617e4f519c"
        },
        {
            "append": true,
            "commitment":
↪"622d1b5efe11e9031f1b25aac11587e0ff81a37e9565ded16ee8e82bbc0c2fc1"
        },
        {
            "append": true,
            "commitment":
↪"406ab5d975ae922753fad4db83c3716ed4d2d1c6a0191f8336c76000962f63ba"
        }]
    },
    "timestamp": 1548330450896,
    "allowance":
    {
        "cost": 2098095
    }
}
```


Commitment Data

Get staychain information on a specific commitment.

request: https://testnet.mainstay.xyz/api/v1/commitment/commitment?commitment=5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053

response:


```
{
    "response":
    {
        "attestation":
        {
            "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
            "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13
↪",
            "confirmed": true,
            "inserted_at": "16:06:41 23/01/19"
        },
        "merkleproof":
        {
            "position": 1,
            "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
```


(continues on next page)

(continued from previous page)


```
            "commitment":
↪"5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
            "ops": [
            {
                "append": false,
                "commitment":
↪"21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee869830d617e4f519c"
            },
            {
                "append": true,
                "commitment":
↪"622d1b5efe11e9031f1b25aac11587e0ff81a37e9565ded16ee8e82bbc0c2fc1"
            },
            {
                "append": true,
                "commitment":
↪"406ab5d975ae922753fad4db83c3716ed4d2d1c6a0191f8336c76000962f63ba"
            }]
        }
    },
    "timestamp": 1548330505898,
    "allowance":
    {
        "cost": 60414043
    }
}
```


Merle Tree

Get information on the commitments to a Merkle tree.

request: https://testnet.mainstay.xyz/api/v1/merkleroot?merkle_root=f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35ef

response:


```
{
    "response":
    {
        "attestation":
        {
            "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
            "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13
↪",
            "confirmed": true,
            "inserted_at": "16:06:41 23/01/19"
        },
        "merkle_commitment": [
        {
            "position": 0,
            "commitment":
↪"21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee869830d617e4f519c"
        },
        {
            "position": 1,
```


(continues on next page)

(continued from previous page)


```
            "commitment":
↪"5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053"
        },
        {
            "position": 2,
            "commitment":
↪"5309053b9d4db8f86d2c7ec164645bdf1669111280e49e04c036c323b58f4709"
        },
        {
            "position": 3,
            "commitment":
↪"d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e"
        },
        {
            "position": 4,
            "commitment":
↪"9b07569d4fd42ae3a19c0803b7401443e0275feb728e8103330d7d8615eecb62"
        }]
    },
    "timestamp": 1548330553639,
    "allowance":
    {
        "cost": 3318936
    }
}
```


Slot Position

Get information on a client slot position.

request: https://testnet.mainstay.xyz/api/v1/position?position=1

response:


```
{
    "response":
    {
        "position": [
        {
            "position": 1,
            "merkle_root":
↪"300ab922905c67631e46e6d014be286fe1bb6dc550ae2df83484fcb1ccb21011",
            "commitment":
↪"5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
            "ops": [
            {
                "append": false,
                "commitment":
↪"2851174cf04f206e6fdfd78a9208c90a324fea5e97ee5b0629d35b5a853fbcfc"
            },
            {
                "append": true,
                "commitment":
↪"622d1b5efe11e9031f1b25aac11587e0ff81a37e9565ded16ee8e82bbc0c2fc1"
            },
```


(continues on next page)

(continued from previous page)


```
            {
                "append": true,
                "commitment":
↪"406ab5d975ae922753fad4db83c3716ed4d2d1c6a0191f8336c76000962f63ba"
            }]
        },
        {
            "position": 1,
            "merkle_root":
↪"2522e16722cfb1b29d01bbe6bfabe54ef7dd69b8bf8a00f911103284eebf4e3e",
            "commitment":
↪"5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
            "ops": [
            {
                "append": false,
                "commitment":
↪"586f199625d902706e0ebf24e2720e62f3f4343a5d7b2ddc2fac155fb359ca3a"
            },
            {
                "append": true,
                "commitment":
↪"622d1b5efe11e9031f1b25aac11587e0ff81a37e9565ded16ee8e82bbc0c2fc1"
            },
            {
                "append": true,
                "commitment":
↪"406ab5d975ae922753fad4db83c3716ed4d2d1c6a0191f8336c76000962f63ba"
            }]
        }, ]
    },
    "timestamp": 1548330579389,
    "allowance":
    {
        "cost": 31613129
    }
}
```


Attestation

Get information on an attestation.

request: https://testnet.mainstay.xyz/api/v1/attestation?txid=38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b1

response:


```
{
    "response":
    {
        "attestation":
        {
            "merkle_root":
↪"f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
            "txid": "38fa2c6e103673925aaec50e5aadcbb6fd0bf1677c5c88e27a9e4b0229197b13
↪",
            "confirmed": true,
```


(continues on next page)

(continued from previous page)


```
            "inserted_at": "16:06:41 23/01/19"
        },
        "attestationInfo":
        {
            "txid": "86b372fb70e0935bfff4d6ba112e78cb9a3201ca15251dcd7db7cbf135b342b5
↪",
            "amount": 149.9999155,
            "blockhash":
↪"3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14",
            "time": 1548253554
        }
    },
    "timestamp": 1548330644403,
    "allowance":
    {
        "cost": 7959634
    }
}
```


Block

Get information on a Bitcoin block if it contains a Mainstay Merkle root commitment.

request: https://testnet.mainstay.xyz/api/v1/blockhash?hash=3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe

response:


```
{
    "response":
    {
        "blockhash":
        {
            "txid": "86b372fb70e0935bfff4d6ba112e78cb9a3201ca15251dcd7db7cbf135b342b5
↪",
            "amount": 149.9999155,
            "blockhash":
↪"3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14",
            "time": "14:25:54 23/01/19"
        }
    },
    "timestamp": 1548330671498,
    "allowance":
    {
        "cost": 1543490
    }
}
```


Authenticated Endpoints

Commitment Send

Node.js example

```
const request = require('request');
let elliptic = require('elliptic');
let ec = new elliptic.ec('secp256k1');

const url = "https://testnet.mainstay.xyz/api/v1";
const route = '/commitment/send'
const pubKey = '1CsSceq9GWnmozaky3DGa24UER6gRDgibf';
const pvtKey =
    'bac52bbea2194e7ea1cd3da6585b66d28f1a7a3683eca91af4ba6373d323d24f';
const commitment =
    'F011111111111111111111111111111111111111111111111111111111110F';


let keyPair = ec.keyFromPrivate(
↪"97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a");
let privKey = keyPair.getPrivate("hex");
let pubKey = keyPair.getPublic();

let signature = ec.sign(commitment, privKey, "hex", {canonical: true}).toDER('base64
↪');

var payload = {
  commitment: commitment,
  position: 0,
  token: '4c8c006d-4cee-4fef-8e06-bb8112db6314',
};

payload = new Buffer(JSON.stringify(payload)).toString('base64');

const options = {
  url: url + route,
  headers: {
    'X-MAINSTAY-PAYLOAD': payload,
    'X-MAINSTAY-SIGNATURE': signature
  }
};

request.post(options, (error, response, body) => {
  if (error)
    return console.log(error);
  ...
});
```

Curl example

```
curl --header "Content-Type: application/json" --request POST --data '{"X-MAINSTAY-
↪PLAYLOAD":
↪"eyJwb3NpdGlvbiI6MCwiY29tbWl0bWVudCI6IkYwMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMT
↪","X-MAINSTAY-SIGNATURE":
↪"IJbqe50XtfZbQ1b0jr+J1tswSPfZlWwZugXCpYbwYMPuRl+htqSb7wTLYY9RtQ6Bw9Ym5dw0vMNRaDwR8pked2Y=
↪"}' http://localhost:9000/api/v1/commitment/send
```

response

{"response":"feedback","timestamp":1541761540171,"allowance":{"cost":4832691}}

3.1.6 Sidechain integration

This document describes the protocol for linking a blockchain to the Mainstay connector service to obtain trustlessly immutable transaction confirmations. This linked blockchain can in principle be public or private, permissioned or permissionless, but a defined entity (or federation of entities) will be responsible for performing state commitments to the service. In the case of a permissioned federated blockchain, this role can be performed by the block-signing nodes as they are relied upon to continue propagating the chain and confirming transactions in new blocks. Once a blockchain is *connected* to the service, it becomes a *sidechain* i.e. a blockchain that is linked to and dependent upon data encoded on a separate blockchain: the *mainchain*. In the CommerceBlock Mainstay serivce, the mainchain is the public blockchain with the highest security and greatest global cumulative proof-of-work: Bitcoin.

Slot initialisation

The connection to the Mainstay service requires the reservation of a *slot* position which is designated with a `slotid`. The slot reservation is activated via a service agreement with CommerceBlock for a specified duration, and requires the sidechain federation (or controller) to provide a `PubKeyScript` against which signatures required to authenticate the state commitments from given sidechain are validated. The service provider (CommerceBlock) then guarantees the operation of the service for the specified period, and provides the API access credentials for both committing the sidechain state to the slot, and retrieving the corresponding slot-proofs required for confirmation.

Sidechain initialisation

To initialise the Mainstay connector, the *base* of the transaction staychain (`TxID0`) *and* the provided `slotid` in the Bitcoin blockchain must be committed to the sidechain in a defined location in order to enable sidechain state commitments to be trustlessly immutable. `TxID0` and `slotid` are concatenated and committed to the sidechain (they can be retrieved from the Mainstay API). This can in principle be in any position at any block-height (for pre-existing blockchains that have already been running for some time).

If this is an Ocean client based sidechain being linked before launch, then the hex encoded `txid0||slotid` is embedded in the genesis block using the `attestationhash` configuration option in the Ocean client. For example, if the `slotid` is 14 (`0x0e`) and `txid0` is `b64f5fb1c36fe10fb74cd4797cef912cc43bdd0c2b2225fdacbbbea20b3bd365`, then in the `ocean.conf` file the following line is added:

```
attestationhash=b64f5fb1c36fe10fb74cd4797cef912cc43bdd0c2b2225fdacbbbea20b3bd365
```

This means that the unique and single Bitcoin staychain and slot position is now uniquely *paired* with the sidechain.

Sidechain state commitment

After the sidechain is initialised and begins state transitions (i.e. block production) via the federated block-signing nodes, the block producers

CHAPTER 4

Guardnode protocol design

The CommerceBlock (CB) network is designed to support a system individually permissioned blockchains (sidechains) on which tokenised assets and securities can be issued and traded with minimum friction. The individual sidechains which form the CB network are federated via permissioned block-signers, however a distributed community of incentivised *guardnodes* secure the network consensus rules and provide distributed services to lightweight clients and sidechain users. These services are provided in exchange for a proportion of fee revenues on participating sidechains, and are coordinated via the CB root chain. This general architecture is highly scalable, extensible and gives tokenized asset and security issuers control over transaction policies yet maintain transparency, reliability and trust minimisation.

The design philosophy and theory for the Guardnode system is described in the Covalence whitepaper.

4.1 Network model

At the core of the network architecture is the CommerceBlock *service* blockchain which provides the platform for the token staking, ticketing and reputation protocols that coordinate the Guardnode system. The CB root chain is a fully public blockchain, with a Byzantine fault tolerant block-signing federation, and derives its immutability from the Bitcoin blockchain via the Mainstay protocol. The native token of the CB root chain is the CommerceBlock Token (CBT) which was issued on Ethereum (as an ERC20 token). CBT can be moved to the CB root chain from the ERC20 contract on the Ethereum blockchain via a federated one-way peg.

Companies, institutions and consortia have the ability to launch customised and configurable permissioned federated sidechains with tokenized asset support, according to their own requirements and policies under their full control and on their own hardware. Tokenised assets and securities can then be issued on these sidechains and transacted peer-to-peer using CommerceBlock multi-asset wallet tooling. These so-called 'client' sidechains are capable of independent operation, however by agreeing to pay a proportion of the transaction fees generated on their chain, they can utilise the services of CB network Guardnodes to provide decentralised security and trust minimisation.

The CB service chain and connected asset-backed client chains are public (but permissioned) blockchains, and anyone is free to run a fully validating client for any or all of them, and users with significant asset-backed token holdings will do this to perform full verification of transaction confirmations on client chains. In order to increase the decentralisation, security and utility of individual client chains, organisations and individuals are incentivised to run fully validating Guardnodes, which both enforce consensus rules and provide services to client chain users and lightweight

wallets. Operators of Guardnodes are free to chose the client chains they wish to store, validate and provide services to, and are rewarded directly in tokens derived from the client chain transaction fees. In order to perform these services for the network and receive payments, individual Guardnode operators are required to bond (time-lock) tokens (CBT) on the CB root chain in return for a *ticket* to provide services for a particular client chain, which is valid for a specific amount of time. In addition, Guardnodes are required to provide regular proofs-of-blockchain-storage and service proofs to the CB block-signing federation to receive fee payments and *reputation* tokens. Tickets are obtained via a auction mechanism described below.

Client chain stakeholders (i.e. federation members and asset/security issuers) can choose the percentage of the client chain transaction fees that are paid to the pool of guardnodes and the target node count `np`. The client chain transaction fees (in asset backed tokens) are then split evenly amongst the guardnodes validating and storing that particular client chain (at specified intervals). The larger the potential transaction volume (and hence transaction fees) for a particular client chain, the more incentive there is for individuals and organisations to run guardnodes, and the greater the distributed security of that client chain.

4.2 Guardnode overview

Guardnodes fully validate and permanently store the CB service blockchain and one or more asset or security backed client sidechains, and issue network alerts and fraud proofs if they detect invalid blocks or consensus anomalies. Any individual or organisation is free to run a guardnode and to choose which client chains to validate, monitor and store, which will be influenced by the cost of the storage and network connectivity, the required ticket staking price and the expected income from leaf chain transaction fees. A single 'Guardnode' by definition then connects to and validates multiple public blockchains, with that connectivity and validation controlled by the operator (user) via a single unified interface.

The CB service chain provides the coordination platform for the guardnode system, and all request, network token, ticket and reputation operations are performed via service chain token-based transactions. This provides transparency and immutability for all network participants, and enables the CB service chain *coordinator* to control the operation of the system via token issuance. The guardnode and network interfaces interpret and abstract the on-chain transaction based logic to display the system status in a user-friendly way.

The following sections describe the full process and protocol for the operation of the guardnode system. This consists of the principle stages of:

1. Service request creation
2. Ticket auction and allocation
3. Service delivery and proof verification
4. Fee payment allocation and distribution

4.3 Request creation: client chain connection

To employ the services of distributed guardnodes on the CB network, an asset or security issuer that operates a client chain must apply to CommerceBlock to join the network and obtain permission to issue *requests* on the CB service chain. This permission is granted via the issuance of a *permission token* (`pToken`) to the address provided by the client chain *commissioner* (`cAddress`). This specific token enables the issuance of requests. The commissioner supplies the client chain information to the coordinator, where it is hosted on the CBServices portal. This information includes:

1. The client chain genesis block and block-signing script
2. The client chain federation end-points (IP addresses and port number)
3. Chain details and website URL

In return, the coordinator provides the commissioner with a client chain address (`fAddress`) to which transaction fees (or a proportion of) are paid on the client chain. This address is derived from a private key stored on a coordinator hardware security module (HSM). A request transaction consists of a single input (`pToken`) and two outputs: One time-locked (CLTV) output paying to the commissioner address and one zero value OP_RETURN output containing the encoded details of the request. The request details are as follows:

1. The client chain genesis block hash `cGen`
2. The service period start time `sStart`
3. The target number of tickets `np`
4. The auction price decay constant `dc`
5. The percentage of client chain transaction fees paid to guardnodes `Fp`
6. The guardnode services required

The time-locked `pToken` output is set as spendable after a time `sEnd` (set via `OP_CHECKLOCKTIMEVERIFY`). The target number of nodes (`np`) is the number of distributed independent Guardnodes that the client chain operator determines are needed to meet their service level, security properties and decentralisation requirements. The higher this number, the smaller the fee income per Guardnode and the smaller the eventual ticket price - reducing the incentives and hence reliability of individual nodes.

The client chain commissioner can specify the services required, which include:

- Fork detection: Guardnodes monitor the network for conflicting leaf chain blocks and broadcast alerts with header proofs if detected.
- Block validity monitoring: Guardnodes fully validate the leaf chain and construct and broadcast fraud proofs if invalid but signed blocks are detected.
- Blockchain storage: Guardnodes maintain full archival copies of leaf chains and provide proofs of retrievability.
- SPV proofs: Provision of lightweight transaction confirmation proofs (SPV proofs) to leaf chain user wallets.

The request is created and signed by the commissioner wallet interface (with the private key for `cAddress`). Once created and broadcast to the service chain, the transaction is verified by the service chain with the additional policy rules: 1. That the request is correctly formed. 2. That the token ID is of type `pToken` 3. That the client genesis hash matches a known client chain and 4. That `sEnd` > `sStart` + 1 hour > current time + 2 hour.

Once confirmed the request is active, and the ticket auction mechanism is initiated.

4.4 Ticket stake auction

Guardnode operators must hold a quantity of the service chain network token (CBT). This will correspond to a specific token type on the service chain, issued to users via the one-way peg to the ERC20 CBT token. The guardnode interface and user wallet displays the current balance of CBT, the current balance of reputation tokens (REP) and all currently active requests (where the current time < `sStart` - 1 hour).

Tickets for a specific request are allocated to guardnode operators via a uniform price Dutch auction mechanism, which determines the final staking amount of CBT for all the tickets in a request. The auction becomes active as soon as the request transaction is confirmed on the service chain (i.e. within 1 minute of transmission) and ends 60 minutes before the specified `sStart` time. The requester is free to choose both the length of time the auction should run (`sStart` - 1 hour - request confirmation time), and the value of the stake price decay function constant `dc`. These should be chosen in a trade-off between maximising both participation (reaching the target number of tickets `np`) and the final stake price (the commissioner is incentivised to maximise the stake as it optimises the reliability of the guardnode service providers).

Guardnode operators can submit a bid for a ticket for a given request at any time the auction is still active (either up to the end time, or it finalises because the target ticket number `np` has been met). The guardnode interface displays the current status of a specific request (along with the request information). This status shows the current ticket stake price `sp` (which decreases every minute according to the auction decay function), the time remaining for the auction and current number of (cumulative) bids `nb`. The operator can then make a decision on bidding.

To submit a bid for a ticket allocated for given request, the operator submits a special *bid* transaction from their guardnode wallet. This transaction contains inputs of network token (CBT) equal to the current auction stake price (in addition to the network fee). If the user has any reputation tokens (REP) then the required auction price is reduced according to the reputation discount function `repdis(rtokens)`. If that is the case, the reputation tokens must also be included as an input to the bid transaction (they will be locked for the duration of the service request).

The bid transaction then pays both the staking amount (CBT) and reputation tokens (REP) to addresses controlled by the guardnode operator wallet (all staked token outputs always remain under the ownership of the holder at all times). In addition to these two outputs, a third zero value `OP_RETURN` output contains the `TxID` of the request transaction. This then links the bid to the request at the consensus layer.

Once the bid is broadcast to the service chain signing nodes, it is accepted as valid and confirmed only if the following conditions are met: that the bid amount is consistent with the request parameters and the decay function (discounted by the reputation tokens) and that there are less than `np` submitted bids (i.e. `nb` < `np`). Once confirmed the bid is finalised and at this point the bidder is *guaranteed* a ticket - but the final required stake is not determined until the auction finalises.

The auction finalises when either `nb` = `np` or the time reaches `sStart` - 1 hour (whichever is the sooner). Once the auction finalises, the final ticket stake price is set at the value of the auction decay function at the point of finalisation (i.e. the closing time or the bid of the `np` bidder) `pfinal`.

After this point, the stakes of CBT and REP in each of the confirmed bid transactions become locked and unspendable until the time `sEnd` encoded in the request is reached. The exception to this is if the value of CBT in the output is greater than `pfinal` (which is the case for all bids made before finalisation) - in this case, then one additional transaction (including the same request `TxID` as in the bid transaction) spending the CBT output is permitted with the rule that it contains two outputs: one for exactly `pfinal` and one for the difference (change). When confirmed, the `pfinal` output becomes locked and unspendable until `sEnd`, and the other can be transacted freely.

The locked output then represents the ticket for the specified request - and the holder can prove their possession by providing a signature corresponding the address using in the locked CBT output. By performing the auction via on-chain transactions and enforcing the auction via consensus rules the process is transparent and immutable (via Mainstay) and so cheating (by anyone, including the coordinator) is impossible.

4.5 Service delivery and verification

The service interval commences at time `sStart`. The guardnode interface (which has a direct connection to a service chain node) automatically determines when a ticket is valid and is about to become active. Depending on the configuration of the guardnode interface a client chain node will be instantiated either automatically or after a prompt is confirmed by the operator, and configured according to parameters retrieved from the CB coordinator (via a public API). The guardnode interface connects locally to the client chain node (running directly on infrastructure owned or controlled by the operator) which is used to monitor the client chain network.

The guardnode operator is responsible for maintaining uninterrupted and low latency TCP connections between the client chain node and the client chain P2P network, and HTTPS connections to the service chain coordinator API and the alert system API (with a fixed IP address that is sent to the coordinator at the start of the service interval). The full list of guardnode IP addresses is made public and listed on a web-page for a specific active request on the CBServices portal. As part of the connection process, the guardnode must prove ownership of the ticket by signing a message with the private key of the bit transaction output address.

4.5.1 Service proofs

It is necessary that the guardnode operator maintains constant connections to the client chain P2P network and fully validates all blocks on the client chain, according to the consensus rules and configuration of the client chain. In order to receive payment for the service, the guardnode must demonstrate that it is doing this and storing a full archival copy of the client blockchain.

This is demonstrated using a challenge-response protocol which is initiated by the coordinator at random intervals throughout the service period. The coordinator sends a request to the IP address of the guardnode interface for a particular piece of data (which is randomly chosen, such as the raw transaction for a particular transaction ID) from the client chain and measures the time taken for the response. The response time is required to be small enough that the guardnode would not be able to produce the response if they did not have a local copy (i.e. they had to query another node on the P2P network).

In addition to the challenge-response protocol, the coordinator can query the connection status of the guardnode client chain node and obtain the current peer list. This can then be used to confirm the operation and connectivity of each guardnode via a number of independent peers (i.e. that the connection status of separate guardnodes is consistent).

4.5.2 Alert system and interface

The guardnode is configured to recognise when it receives two (or more) blocks (or block headers) on the client chain at the same block height with valid signatures. This is direct proof of a consensus fork - and should not happen under any circumstances (unlike in Bitcoin) if the block-signing keys are secure. Conflicting block signatures mean that the block-signing nodes have been compromised and that a potential double-spend attack is underway (e.g. with an attacker sending different blockchain histories to different network participants). If this happens, all users should cease transacting until the situation is investigated and resolved via a network wide upgrade, and so long as a single valid history is agreed up to the conflict point, the proof of ownership of client assets is assured.

When a conflicting block is detected, the guardnode is configured to send an authenticated *fraud proof* to both the CBServices portal and third-party forums (e.g. mailing lists, Twitter etc.). The fraud-proof consists of two valid (i.e. signed with the client chain block-signing script) block-headers at the same block height. This fraud proof is signed with the ticket address key, and can then be independently verified by anyone as incontrovertible proof of chain consensus failure.

4.6 Service fee payments

At the end of the service period (as specified in the request) `sEnd` the guardnode interface can halt the client chain node (if there is no automatic renewal protocol enabled - see below) and stop responding to service proof requests.

During the service period specified in the request, the specified proportion of transaction fees generated on the client chain is paid to `fAddress` (which is controlled by the coordinator via an HSM). This payment occurs *on the client chain* in either a native token, pegged in token or an asset-backed token. At the end of the service period, the coordinator determines which of the ticket holders have satisfactorily provided the guardnode service (by timely responses to challenges) and divides the payment `fAddress` among the qualifying ticket holders. The fee portion is paid to the address of the locked CBT output of the bid/ticket transaction. (it is assumed the client chain will be an Ocean based chain, and therefore have a compatible key/address format to the service chain)

Once the fee payment is made, the coordinator then issues reputation tokens (REP) to the address of the locked REP output in the bid/ticket. The number of reputation tickets issued is proportional to the length of the service interval (one token per day of service). The reputation tokens are transferable and fungible.

CHAPTER 5

Guardnode functionality

Proposed design for Guardnode functionality in the Ocean client.

5.1 Requests

Requests in the guardnode system must be permissioned (i.e. only authorised partners - client chains - can issue requests). Permission tokens are required to issue requests. This works as follows: Permission tokens (`permissionAsset`) are (optionally) issued to a given address in the genesis block, e.g.:

'permissionassetcoins=5000000' 'permissioncoinsdestination=g4e638a7cc43...." (the scriptPubKey of the permissionasset output).

This scriptPubKey (multisig or P2PKH) is owned by the 'controller'. The controller sends a quantity of the permissionAsset to a client chain so they can submit requests.

Requests are generated by (or on behalf of) a client (leaf) chain. The request transactions are identified by having a time (blockheight) locked (`OP_CLTV`) permissionAsset output paying to a 1-of-3 multisig output encoded with metadata. The 3 multisig pubkeys are as follows:

```
key1 -> target pubkey permission asset is locked at

key2 -> <02 pubkey prefix> <32 bytes client chain genesis block hash>

key3 -> <03 pubkey prefix>
        <4 bytes service period start time (block height)>
        <4 bytes target number of tickets>
        <4 bytes auction price decay constant>
        <4 bytes fee percentage>
        <15 bytes empty>
```

The request transaction is generated with a new RPC: `createrawrequesttx`. The RPC will take the arguments: 1. txid of input 2. vout of input. 3. output address 4. permission asset ID type, 5. client chain genesis block hash 6. start time (startBlockHeight) 7. end time (endBlockHeight) 8. target no. of tickets 9. auction decay constant 10. fee percentage.

The RPC will then create a new transaction. This tx will have one input (txid and vout) and two outputs. The first output will pay to the output address and have a time-lock set to the end time (block height). The first output asset ID tag will be set to RPC argument 4 (permission asset ID type). The first output script can be constructed like: .. code-block:: json

    script = CScript() ToByteVector(endBlockHeight) << OP_CHECKLOCKTIMEVERIFY << OP_DROP << OP_DUP << OP_1 << ToByteVector(key1) << ToByteVector(key2) << ToByteVector(key3) << OP_3 << OP_CHECKMULTISIG

The raw transaction generated by this RPC will then be signed with the private key of the (`permissionAsset`) input and braodcast.

5.2 Active requests RPC

To view all active requests (i.e. requests that have been confirmed but not expired: `blockheight > endBlockHeight`) the `getrequests` RPC is used. This RPC scans the current UTXO set for request transactions (using the permission asset type) and returns details as a JSON array.

This array contains an object for each active request. Each object contains the request metadata and the blockheight at which it was confirmed.

e.g.

```
[
    {
        "startBlockHeight": 40060,
        "endBlockHeight": 90000,
        "confirmedBlockHeight": 30000,
        "genesisBlock":
↪"fa1c7d059dab70cddb8cb3cc7b8971d385eecea4d68bd86c5cb6d75949789ba1",
        "numTickets": 100,
        "startPrice": 100000,
        "auctionPrice": 80000,
        "decayConst": 1000000,
        "feePercentage": 100,
        "txid": "cc1c7d059dab70cddb8cb3cc7b8971d385eecea4d68bd86c5cb6d75949789ba1"
    }
]
```

This RPC will be called by the Guardnode operators/interface to get current requests. If `blockheight < startBlockHeight` then the auction is potentially stil active. (this RPC can be modelled on existing functions like `gettxoutsetinfo`)

5.3 Decay function

The decay function will return the current ticket bid price (in CBT sats) for given parameters, as follows:

```
CAmount CRequest::GetAuctionPrice(uint32_t height)
{
     uint32_t t = height - nConfirmedBlockHeight;
     if(t < 0) return 0; // auction not started yet
     return nStartPrice*(1 + t)/(1 + t + pow(t,3)/nDecayConst);
}
```

Given the parameters in the object above, the ticket price is shown in the figure as function of t over 4000 blocks (~ 3 days at 1min per block).

5.4 Request/bid table

An in-memory table (`rtable`) will list all current requests (if the node is configured with a `-requestlist=1` flag). The table will be updated at each new block: new requests will be added as a block is recieved (in the `ConnectBlock` function) and removed when `blockheight` > `endBlockHeight`) e.g. with a function `UpdateRequestList`. In the event of a node re-start, the `rtable` will be regenerated by scanning the UTXO set with e.g. a function `LoadRequestList`. (This can be based on the `UpdateFreezeList` and `LoadFreezeList` functions). Each entry in the table will have all the request transaction parameters and the request transaction `txid`.

In addition, each request in the table will have a vector of valid bid transactions that have been recieved against the request. As valid bids are recieved, the transaction IDs are added to this vector (along with the bif block height) up to a max of `numTickets`. A valid bid is decribed below, and are added to the vector by the `UpdateRequestList` function.

So the table will look like this:

```
[
    {
        "requestTxID":
↪"0a22fe0103a2f583f37d3feb94df941a6c90d8d0c3113548e0776f3413f33346",
        "confirmedBlockHeight": 30000,
        "startBlockHeight": 40060,
        "endBlockHeight": 90000,
        "genesisBlock":
↪"fa1c7d059dab70cddb8cb3cc7b8971d385eecea4d68bd86c5cb6d75949789ba1",
        "numTickets": 100,
        "startPrice": 100000,
        "auctionPrice": 80000,
        "decayConst": 1000000,
        "feePercentage": 100,

        "bids": [
            { "txid":
↪"65eacf082247aaf0b1624539a0d7e3bb667b73211269907b0504a3b8f8ab0a22",
              "feePubKey":
↪"0300adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656" },
            { "txid":
↪"af3d49ff538a9a2bcd78b924aa27f102fb391811c387e7b5b06fc034d56cd4d8",
              "feePubKey":
↪"0311adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656" },
            { "txid":
↪"64c787adf54983f90be8d6a72ba9c3e2523117804b2087f8b6324ccb4b29ac0d",
              "feePubKey":
↪"0322adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656"},
            { "txid":
↪"9a5afcbd6892a2b7c8b6926f764f947df2ef22bc25be4fdb743079b7a03df56f",
              "feePubKey":
↪"0333adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656"}
        ]
    }
]
```

A new RPC `getrequestbids` will output this vector of bids (with txids and block heights) for a given request

transaction ID (by querying the in memory table).

5.5 Bid transactions

Bid transactions will be created with a new RPC `createrawbidtx`. This will take as arguments: 1. input txid 2. input vout 3. lock height (i.e. the `endBlockHeight` of the request) 4. The txid of the request. 5. The bid amount (CBT). 6. Stake address (the address to which the stake will be paid back at the end of the service period) 7. Fee address (base-58 address for fee payment on the client chain). This RPC will then output a hex encoded raw unsigned bid transaction with three outputs:

1. The first output will be a CLTV locked 1-of-3 multisig (of CBT asset type)
2. The second output will be a P2PKH output paying any change from the input
3. Transaction fee.

The first output should be locked for the same duration as the ending blockheight of the request.

The 3 multisig pubkeys are as follows:

key1 -> target pubkey CBT asset is locked at key2 -> <02 pubkey prefix> <32 bytes request transaction hash> key3 -> pubkey to receive fees on client chain

Any excess amount will have to be returned to an address owned by the user, using "change" and "change" fields in the output object. These are optional and should only be included when the input amount exceeds the bid amount.

5.6 Bid transaction validity

When a bid transaction is received into a block, the `UpdateRequestBidList` function will determine its validity, and if it is valid, the TxID and other bid information will be added to the relevant request bid set in the request list. The validity will be determined as follows:

1. Check if transaction is encoded as a bid transaction.
2. Read request TxID from the second pubkey in the CLTV locked multisig
3. Get the `decayConst`, `startPrice`, `blockheight` (when the request transaction was confirmed), `startBlockHeight`, `endBlockHeight` and `numTickets` from the request list.
4. Check that `endBlockHeight` in the bid transaction time-lock CLTV is greater than or equal to the request `endBlockHeight`.
5. Calculate the current bid price based on the request parameters and the current blockheight with `ticket_auction_price`.
6. Check that the value of CBT in output 1 is greater than or equal to the current bid price.
7. Check that the auction has not ended and that the request number of tickets has not been reached.

If valid the, bid transaction TxID and bid information is added to the request bid set in the request list.

5.7 Bid output policy

The request bid set is used for two purposes: 1. to enable the coordinator to pay client chain fees to the winning bidders, and 2. to lock the winning bid outputs for the duration of the service period. The locking is performed via the CLTV locked multisig output and the bid is added to the bid set only if it matches all the above prerequisites.

This bid set will also allow winning bids to collect the change. At the end of the auction the final request bid will be calculated and guardnodes will be able to get the overbid - see the guardnode tecdoc.

CHAPTER 6

Coordinator daemon

Coordinator daemon, responsible for verifying the operation of Guardnodes in the Commerceblock Covalence system

6.1 Requirements

- CB service chain connectivity to receive active client requests
- Client chain connectivity to generate guardnode challenges
- Db instance to store requests and responses
- Listener HTTP POST endpoint to receive guardnode responses
- Public rpc api to offer information on requests and guardnode response performance

6.2 Configuration

Env variables to set:

- CO_LISTENER_HOST: host address at which the coordinator binds to receive guardnode responses
- CO_CHALLENGE_FREQUENCY: the frequency in number of blocks that new challenges are created
- CO_CHALLENGE_DURATION: challenge duration in seconds
- CO_VERIFY_DURATION: challenge transaction verify duration in seconds
- CO_LOG_LEVEL: env logger log level
- CO_API_HOST: rpc api host address
- CO_API_USER: rpc api user name
- CO_API_PASS: rpc api password
- CO_STORAGE_HOST: db storage host address

- CO_STORAGE_USER: db storage user name
- CO_STORAGE_PASS: db storage password
- CO_STORAGE_NAME: db storage database name
- CO_CLIENTCHAIN_HOST: client chain rpc host address
- CO_CLIENTCHAIN_USER: client chain rpc user name
- CO_CLIENTCHAIN_PASS: client chain rpc password
- CO_CLIENTCHAIN_ASSET_HASH: client chain challenge asset hash
- CO_CLIENTCHAIN_GENESIS_HASH: client chain genesis hash
- CO_CLIENTCHAIN_ASSET: client chain challenge asset label
- CO_CLIENTCHAIN_ASSET_KEY: client chain challenge asset key
- CO_SERVICE_HOST: service chain host address
- CO_SERVICE_USER: service chain user name
- CO_SERVICE_PASS: service chain password

6.3 Running

To run a production instance of the coordinator along with a mongo db database, edit the envs in the docker compose file file and:

docker-compose up

To test the coordinator locally:

cargo run

To test the coordinator locally with demo request creation:

./scripts/demo.sh && cargo run –example demo

6.4 RPC API

Any requests need to be sent to CO_API_HOST using Http Basic Authentication via CO_API_USER/CO_API_PASS.

The following rpc commands are offered:

- getrequests: fetches all requests for the client
- getrequest {“txid”: “hash”}: fetches the specific request
- getrequestreponses {“txid”: “hash”}: fetches the responses for a specific request

An example of how to generate a response report is showing in report.

Example ` curl -X POST -H "Content-Type: application/json" -d '{"jsonrpc": "2.0", "method": "get_challenge_responses", "params" : {"txid": "5eba0bf305ac8963225d68195fa7eb8b79667ad9c5fa6e9dcc0ce0185ad4a046"}, "id":1 }' userApi:passwordApi@localhost:3333 `

6.5 Guarnode Responses

Guardnode responses are sent via HTTP POST to the listener host at:

http://coordinator:8080/challengeproof.

The fields "txid" (challenge transaction txid), "pubkey" (guardnode public key), "hash" (guardnode bid txid) and "sig" (signed txid with guardnode pubkey) need to be included in the body of the POST request.

An example of this is shown in hyperclient.

Example Request ` { method: POST, uri: http://localhost:9999/challengeproof, version: HTTP/1.1, headers: {"content-type": "application/json"}, body: Body } `

Body ` {\"txid\":\"1234567890000000000000000000000000000000000000000000000000000000\", \"pubkey\":\"026a04ab98d9e4774ad806e302dddeb63bea16b5cb5f223ee77478e861bb583eb3\", \"hash\":\"0404040404040404040404040404040404040404040404040404040404040404\", \"sig\":\"30450221009dd76bcdc19a283654727214757b9e33ded38f00951b4f4a074e6fbe17a6f2ef022057
`

CHAPTER 7

Guardnode daemon

Guardnode daemon responding to client chain coordinator challenges and generating alerts for misbehavior on the chain.

7.1 Running

To run the daemon locally:

1. `pip3 install -r requirements.txt`
2. `python3 setup.py build && python3 setup.py install`
3. Run `./run_guardnode` or `python3 -m guardnode` providing the arguments required

To run a demo along with the coordinator daemon execute the following replacing *$txid* with the txid produced by the coordinator demo script:

```
./run_guardnode --rpcuser user1 --rpcpassword password1 --bidpubkey
029aaa76fcf7b8012041c6b4375ad476408344d842000087aa93c5a33f65d50d92
--challengeasset fae9f771019d45e31b8f78da99a15b094b17b2ba76b0940c3ac53d5e9afd8e8e
--nodelogfile /Users/nikolaos/co-client-dir/ocean_test/debug.log --bidtxid
$txid
```

7.2 Configuration

Arguments to set:

- *–rpcconnect*: Client RPC host
- *–rpcport*: Client RPC port
- *–rpcuser*: Client RPC username
- *–rpcpassword*: Client RPC password

- *–nodeaddrprefix*: Node P2PKH address prefix
- *–nodelogfile*: Node log file destination
- *–bidtxid*: Guardnode winning bid txid
- *–bidpubkey*: Guardnode winning bid public key
- *–challengehost*: Challenge host address
- *–challengeasset*: Challenge asset hash

CHAPTER 8

Guardnode guide

A step by step guide for setting up the Guardnode stack that includes a CommerceBlock service chain full node, a client chain full node and guardnode daemon guardning the client chain.

1. Running the service chain full node

Download the docker-compose file from ocean github and follow the docs instructions on how to run the node using data persistence.

The full node wallet will need to be funded with CBT in order to provide services. This can be done by paying to an address generated by the node or import a private key from another wallet.

Use the command line interface to find about active requests:

```
ocean-cli getrequests
```

(Under development) Also possible with the dedicated guardnode electrum wallet.

2. Running the client chain node

Pick one of the active requests with an active auction and run the client chain full node for this chain request by downloading the guardnode repo and running the corresponding docker-compose file in the contrib directory.

(Under development) Also available in the CB services platform.

Start the client ocean node:

```
docker-compose -f contrib/docker-compose-filename.yml up -d ocean
```

Using the client chain node generate a pubkey to receive fee rewards on:

```
addr=`ocean-cli getnewaddress`
pub=`ocean-cli validateaddress $addr | jq -r ".pubkey"`
echo $pub
```

3. Bid for a request

The following script can be used to bid for an active request. The following parameters need to be filled:

- Fee pubkey generated previously
- Request transaction id from *getrequests* rpc
- Current auction price from *getrequests* rpc

```
#!/bin/bash
shopt -s expand_aliases

alias ocl="ocean-cli -rpcport=7043 -rpcuser=ocean -rpcpassword=password"

echo "Creating bid in service chain"

# Address tokens will be locked in
addr=`ocl getnewaddress`
pub=`ocl validateaddress $addr | jq -r ".pubkey"`

# Get asset unspent
unspent=`ocl listunspent 1 9999999 [] true "CBT" | jq .[0]`
asset_hash=`echo $unspent | jq -r ".asset"`
value=`echo $unspent | jq -r ".amount"`
txid=`echo $unspent | jq ".txid"`
vout=`echo $unspent | jq -r ".vout"`

# TO UNLOCK A PREVIOUS BID
# Provide the `txid` and `vout` for that transaction
# The output can be spent after the locktime is expired
# e.g.
# value=5000
# vout=0
# txid="\"a327b15679f7fd0a8984cdb16f07c2c92063c5565af2f7ce99cff8d4750add8d\""

# Fee
fee=0.001
# Current auction price
bid=
# Change from unspent
change=$(echo "$value - $fee - $bid" | bc)

# Request id in service chain
requestid=""
# Request end height
end=
# Fee pubkey to pay fees in clientchain
feepub=""

inputs="[{\"txid\":$txid,\"vout\":$vout,\"asset\":\"$asset_hash\"}]"
outputs="{\"endBlockHeight\":$end,\"requestTxid\":\"$requestid\",\"pubkey\":\"$pub\",\
\"feePubkey\":\"$feepub\",\"value\":$bid,\"change\":$change,\"changeAddress\":\"$addr\
↪",\"fee\":$fee}"

signedtx=`ocl signrawtransaction $(ocl createrawbidtx $inputs $outputs)`
txidbid=`ocl sendrawtransaction $(echo $signedtx | jq -r ".hex")`
echo "txid: $txidbid"
```

To use the script create a file called *create_bid.sh* with the contents and do the following commands:

```
cp create_bid.sh ../datadir/
```

(continues on next page)

(continued from previous page)

```
docker-compose -f contrib/docker-compose-filename.yml exec ocean bash

./home/bitcoin/.bitcoin/create_bid.sh
```

4. Running the guardnode service

Verify that the bid has been approved by using the service node:

```
ocean-cli getrequestbids $requesttxid
```

Once verified fill the *bidpubkey* and *bidpubkey* arguments on the docker-compose file downloaded for the client chain under *guardnode* and start the guardnode service by:

```
docker-compose -f contrib/docker-compose-filename.yml up -d guardnode
```

Monitor the logs using and look out for any alerts:

```
docker-compose -f contrib/docker-compose-filename.yml logs --follow guardnode
```

CHAPTER 9

Request guide

A guide for creating service requests in the CommerceBlock chain.

1. Running the service chain full node

Download the docker-compose file from ocean github and follow the docs instructions on how to run the node using data persistence.

The full node wallet will need to be funded with PERMISSION assets in order to create requests. This can be done by paying to an address generated by the node or import a private key from another wallet.

2. Create a request

The following script can be used to create a request. The following parameters need to be filled:

- Client chain genesis hash
- Number of tickets
- Fee percentage paid
- Start block height
- End block height
- Starting auction price
- Auction decay constant

```
#!/bin/bash
shopt -s expand_aliases

alias ocl="ocean-cli -rpcport=7043 -rpcuser=ocean -rpcpassword=oceanpass"

echo "Creating request in service chain"

# Address permission tokens will be locked in
pub=`ocl validateaddress $(ocl getnewaddress) | jq -r ".pubkey"`
# Get permission asset unspent
```

(continues on next page)

(continued from previous page)

```
unspent=`ocl listunspent 1 9999999 [] true "PERMISSION" | jq .[0]`
value=`echo $unspent | jq -r ".amount"`
txid=`echo $unspent | jq ".txid"`
vout=`echo $unspent | jq -r ".vout"`

# TO UNLOCK A PREVIOUS REQUEST
# Provide the `txid` and `vout` for that transaction
# The output can be spent after the locktime is expired
# e.g.
# txid="\"1d91bae7353c0b1fb7178b92b642746ea4ace1d79e1c5d3c680526ef9f4589a7\""
# vout=0
# value=210000

# Client chain genesis block hash
genesis=""
# Request start height
start=
# Request end height
end=
# Number of tickets
tickets=
# Starting price
price=
# Fee percentage paid
fee=
# Decay constant
decay=1000000

# Generate and sign request transaction
inputs="{\"txid\":$txid,\"vout\":$vout}"
outputs="{\"decayConst\":$decay,\"endBlockHeight\":$end,\"fee\":$fee,\
↪"genesisBlockHash\":\"$genesis\",\
\"startBlockHeight\":$start,\"tickets\":$tickets,\"startPrice\":$price,\"value\":
↪$value,\"pubkey\":\"$pub\"}"

signedtx=`ocl signrawtransaction $(ocl createrawrequesttx $inputs $outputs)`
txid=`ocl sendrawtransaction $(echo $signedtx | jq -r ".hex")`
echo "txid: $txid"
```

3. Monitor a request

Check that a request has been included in the chain using:

```
ocean-cli getrequests
```

Download the report script, replace the *txid* parameter with the request id parameter and run this script to monitor the guardnode response performance and pays due to be paid to each by the end of the service. This information will only become available once the service request has started.

CHAPTER 10

ERC20-Sidechain bridge

A short guide on how to peg-in CBT tokens from the Ethereum network to Ocean and peg-out back to Ethereum.

The following steps assume that the user is running a full Ocean node connected to the CommerceBlock mainnet. The node needs to be synced up and with RPC connectivity enabled. Connectivity to a geth node is optional but it would allow doing the same validation checks that a signing node does for the peg-in transaction.

1.

Export the private key of the ethereum address that owns the CBT tokens. This should be in hex format, e.g. "0xcb850d9db23b54ebbeae09995f7192af83646f9ea232645bb5a71699e5c15a6e".

2.

Run the *getethpeginaddress* RPC using this private key (with the "0x" prefix removed):

```
ocean-cli getethpeginaddress␣
↪cb850d9db23b54ebbeae09995f7192af83646f9ea232645bb5a71699e5c15a6e
{
  "eth_mainchain_address": "b6872561de5ba19d38071a7616d9d434b9e37860",
  "eth_claim_pubkey":
↪"03664b8a3e065329c6bb3b8f9f0bb382179775f609ffa9ff564ea6f20e913ec04b"
}
```

3.

Pay the CBT tokens to the "eth_mainchain_address" returned from the *getethpeginaddress* RPC and save the transaction id. The transaction will require a minimum amount of 10 confirmations before being allowed to peg-in.

4.

Run the *claimethpegin* RPC using the "eth_claim_pubkey" returned above, the transaction id and the CBT amount as:

```
ocean-cli claimethpegin $txid $amount␣
↪03664b8a3e065329c6bb3b8f9f0bb382179775f609ffa9ff564ea6f20e913ec04b
```

5.

Run the *getbalance* RPC and verify that the CBT has been pegged in to the Ocean network.

6.

To peg-out this CBT will require running the *sendtoethmainchain* RPC specifying an address to send the CBT to as well as the amount to peg-out:

```
ocean-cli sendtoethmainchain 8e8a0ec05cc3c2b8511aabadeeb821df19ea7564 0.1
```

CHAPTER 11

Ocean asset mapping

This document describes the technical specification of the architecture and protocol for the issuance and redemption of tokenized assets on an Ocean federated sidechain. The specification is presented as a general framework, with clearly defined interfaces with the processes that will be unique to a particular asset custodian.

The protocol is centred on a data structure (the mapping database or mapping object) that links tokens created within an Ocean blockchain to the custodians of a particular asset - as such it is a central part of the definition of the legal ownership of that particular asset, along with the ownership of the corresponding token output public key on the blockchain. The mapping database is then the link between the blockchain and the 'real world'.

11.1 Mapping database

To define legal ownership, the mapping database must be accessible by token holders, and in the interests of transparency it should be as accessible as the sidechain itself (although this is not necessarily a requirement). However, the mapping can only be defined by the issuer (and custodian) of an asset - therefore the permissions to add to or modify the database must be clearly defined and controlled.

The mapping database must be persistent and consistent - at any point in time the database must have a global state which reflects the legal ownership of the physical asset. A single state of the database can be enforced via a regular commitment to the Ocean sidechain, the immutability of which is enforced via q mainstay to the Bitcoin blockchain. This can be a part of the sidechain consensus layer, e.g. the hash of the mapping object can be included in the sidechain block header, which guarantees immutability. User wallets can confirm the authenticity of the mapping object via an SPV proof mechanism.

The mapping database will provide the canonical link between asset IDs on an Ocean sidechain and physical assets. Asset IDs on Ocean are generated with on-chain entropy, and cannot be specified by the issuer - therefore the asset issuance process will require adding the generated asset IDs back into the database. If reissuance is not required, then the mapping signatories must confirm that the issuance transaction was flagged as non-reissuable (via an SPV proof) before authorising the issuance. If reissuance is required, the reissuance token becomes under the control of the issuing wallet - they key controlling the issuance token output then becomes a critical point of failure (i.e. whoever has this key can re-issue more of a valid asset). In this case, the issuance (and the issuing wallet) should be controlled by the authorised parties of the asset custodian (i.e. have the same permissions and security model as the mapping database modification permissions).

11.2 Mapping object

The mapping database will be stored and retrieved as a JSON object, with entries that consist of attribute-value pairs that link asset-IDs (blockchain generated) to asset identifiers (physical assets) and other metadata. The basic structure is outlines in the following example:

```
{
  "date": "Thu 12 Jul 2018 12:27:51"
  "assets": [
    {
      "assetid": "78214125442A472D4A614E645267556B58703273357638792F423F4528482B4D",
      "amount": 402.342313
      "vref": "b87493"
      "location": "vault03 row18 shelf4"
      "issuer": "Acme"
    },
    {
      "assetid": "5468576D5A7134743777217A25432A462D4A614E645266556A586E3272357538",
      "amount": 399.573843
      "vref": "b45010"
      "location": "vault03 row12 shelf2"
      "issuer": "Acme"
    },
    {
      "assetid": "77397A24432646294A404E635166546A576E5A72347537782141254 42A472D4B",
      "amount": 400.765738
      "vref": "b02361"
      "location": "vault02 row07 shelf4"
      "issuer": "Acme"
    }
  ],
  "signatures": [
    {
      "authkey": 1,
      "r": "404D635166546A576D5A7134743777217A25432A462D4A614E645267556B5870",
      "s": "A13F4428472D4B6150645367566B597033733676397924422645 2948404D6251"
    },
    {
      "authkey": 3,
      "r": "3B7638792F423F4528482B4D6250655368566D597133743677397A2443264629",
      "s": "C14E645267556B58703273357538782F413F4428472B4B6250655368566D5971"
    }
  ]
}
```

The JSON object is composed of three components, the time/date, the asset list and the signatures. The signature attribute contains the required signatures to validate the asset mapping against the published public keys of the asset issuer and the authorisation policy (e.g. the signatures of 2 of 3 keys are required to authenticate the mapping). The signatures are generated over the hash of the asset list and time/date. The public keys of the asset issuance authorisers are published, and can be included in (or referenced from) the genesis block of the asset sidechain.

11.3 Protocol

11.3.1 Initialisation of controllers

The controllers are the individuals/parties authorised to create and modify the mapping database. They would typically be appointed by the custodians of the physical assets, and the modification policy would be decided by the asset issuer.

There could be a single controller, however they would have complete control and responsibility over the issuance of tokens and the mapping - if they were to lose the signing key, no more assets could be issued without a network wide hard fork to all wallet and node software. A more secure policy would be to require an n of m multi-signature modification rule: i.e. there are m controllers and at least n must add their signatures to the object to modify the mapping. A typical policy may be a *2-of-3* or *3-of-5* multisig, allowing for 1 or 2 controllers losing their keys or being unavailable.

The m controllers are chosen by the issuing authority - these may be parties involved in verifying the deposits of physical assets (e.g. in a vault), and parties with legal responsibility for the security of the physical assets. In a 'ceremony' before the launch of the asset-backed sidechain, each of the controllers will generate a random private key ski ($i = 1, \ldots, m$) on an isolated piece of secure hardware (this could be an air-gapped PC, a hardware wallet or HSM). This key will then be used to generate a corresponding controller public key $pki = ski\ xG$ (G denotes multiplication of the generator point on the secp256k1 elliptic curve).

One (or n) of the controllers will have access to the Ocean sidechain wallet to which the issued asset will be sent. This wallet will then have ownership of the issued tokens, and the access policy will need to be decided upon by the issuers (e.g. this may be a multi-sig wallet). This wallet will have a direct (non-firewalled) connection to the sidechain signing nodes to enable it to create new sidechain assets.

11.3.2 Asset issuance

The proposed asset issuance protocol is as follows, where we assume that there is no re-issuance.

1. At least n controllers agree that a physical asset is secured and that a token can be issued. They also agree on the issuer reference (serial number etc.), the mass of the asset and other data related to the issuance.
2. Once agreement is reached, the controllers access the sidechain wallet and issue the tokenised asset (with an amount corresponding to the mass of the asset) using the `issueasset` wallet RPC. This will create and send the asset issuance transaction to the sidechain signing nodes, and return the 256 bit asset ID (which is randomly generated) and the transaction ID (TxID).
3. Each of the n controllers must then confirm that the issuance transaction has been confirmed in a sidechain block and that the asset ID and token amount are correct.
4. The mapping entry is then added to the JSON mapping object, along with the current time/date and then the n controllers add their digital signatures (ECDSA) to the object to authenticate it.
5. The JSON object is then uploaded to a pre-defined repository (issuer server, IPFS) where it is accessible via a public API.
6. The block-signing nodes retrieve the current JSON object, and verify the signatures against the published public keys and that the time/date supersedes the previous version. The SHA256 hash of the object is then added to the sidechain block header at every subsequent block.

The token is then issued, and is owned by the controller wallet. It may then be transacted peer-to-peer natively on the sidechain.

11.3.3 Asset redemption

Physical assets can be redeemed by destroying the corresponding token, which is performed as follows:

1. A token holder signals to the asset issuer/custodian that they wish to redeem a specified amount of a particular token, which they send to the controller wallet (the controller will provide them with a redemption address).
2. Once the controller wallet has received the token amount to be redeemed, it is destroyed using the `destroyamount` wallet RPC, which returns the destruction transaction ID.
3. At least *n* controllers then agree that an amount X of the given asset ID has been destroyed, by independently verifying and inspecting the TxID.
4. The entry in the mapping object corresponding to the destroyed token is then modified (either by reducing the mass by the equivalent token amount destroyed, or removing the entry if the entire asset is redeemed). The date/time is updated.
5. Each *n* controllers then authenticate the change and add their signatures.
6. The new signed JSON object is then uploaded and retrieved by the sidechain signing nodes.
7. The corresponding asset (or part of it) is delivered to the redeemer by the custodian.

11.3.4 Asset re-mapping

The asset issuer may wish to reserve the right to change the mapping of physical assets to sidechain asset IDs for operation reasons. If different issued asset tokens have equivalent and indistinguishable value (e.g. they represent different units of the same precious commodity) then re-mapping may simplify the logistics of redemption and collection. An example of this is where an individual may hold in their wallet several fragments of different tokens that each represent units of a single commodity (each a separate asset) and that they wish to redeem the total amount of that commodity. In this case it would add significant additional cost to physically dissect several commodity units into fragments and then deliver a number of fragments to the redeemer. In this case, remapping can enable the individual to redeem the equivalent mass of a single unit. The re-mapping procedure (or shuffling procedure) is described as follows:

1. In the current mapping object, there are N asset mapping entries. Each entry has a mass mi of physical asset reference Ai mapped to a sidechain asset ID IDi (where i=1, ..., N). There are ai=mi tokens issued on the sidechain for each asset ID IDi.
2. A token holder has *k* tokens of asset IDs *IDj* with values *xj* where *j = 1, ..., k*
3. The token holder sends these k tokens to the controller wallet.
4. The controller destroys each token amount *xj* of *IDj* (with the `destroyamount` wallet RPC).
5. The asset IDs are then re-mapped:
   - The total amount (*xt*) of the fragments is calculated.
   - Of all the *N* listed asset mappings, the one with the smallest mass which is greater than *xt* is identified (*As*).
   - The mass of *As* is reduced by *xt: ms <- ms-xt* (*xt* of *As* is then delivered to the redeemer).
   - For each asset *Aj* (*j = 1, ..., k*) a new mapping is created, which links IDs to the assets corresponding to the destroyed tokens: *IDs <-> (Aj,m <- xj)*
   - For each mapping of asset *Aj* to *IDj* (*j = 1, ..., k*) the mass *mj* is reduced corresponding to the destroyed tokens (and the reallocated mass): *mj <- mj-xj*

11.3.5 Asset map object usage

The mapping JSON object will be available via a public API. A sidechain asset SPV wallet will retrieve the object and verify the signatures against the controller public keys and policy. The SPV wallet then confirms that the object is unique by checking the hash of the object against the sidechain block header.

CHAPTER 12

Asset management library

asset-man is a Python 3 library and set of associated scripts/utilities developed to manage the issuance, redemption, mapping and monitoring of tokenised assets on an Ocean blockchain. The core mapping library functions are independent of the blockchain client interface, but the *action* scripts are designed to interface with the Ocean client RPCs.

The library enables the creation and management of a *mapping object* which contains the canonical mapping of on-chain token IDs to real-world asset references, and which forms a central part of the definition of the ownership of an asset (along with proof of ownership of the blockchain tokens via output private keys).

Detailed guides for the initial controller set-up and sidechain configuration are found in initialisation.md, for the token issuance process in issuance.md and the asset redemption process in redemption.md.

12.1 Requirements

In addition to Python 3.* the following libraries are required:

- Pybitcointools
- Trezor's BIP39 mnemonic implementation
- DataDiff
- Boto3 for Amazon S3 (remote mapping object storage)

12.2 Installation

To install the core mapping library

```
$ git clone https://github.com/commerceblock/asset-mapping
$ cd asset-mapping
$ python setup.py install
```

12.3 Core library structure and API

The core library is located in the amap module. This module contains functions for key generation and recovery, and the `MapDB` class that operates on the mapping object.

The library contains the following functions:

`controller_keygen()`

Returns a random, securely generated 32 byte private key, a correspondoing compressed secp256k1 EC public key and a 12 word BIP39 recovery seed phrase.

`controller_recover_key(recovery_phrase)`

Returns the 32 byte private key and correspondoing compressed secp256k1 EC public key when passed a 12 word BIP39 recovery seed phrase.

`token_ratio(blockheight)`

Returns the token to asset ratio at the supplied blockheight. This function is hard-coded with the initial token to asset ratio and the inflation rate as a function of block height.

A mapping object is instantated with the `MapDB` class, and the constructor is optionally passed the n-of-m multisig policy (default 2-of-3). The mapping object is internally handled as a dictionary object and writen to file as a JSON object.

The object has the following methods:

Method	Description
`add_asset(asset_ref, year_ref, mass, tokenid, manufacturer)`	Create a new entry in the mapping object: `asset_ref`, `year_ref` and `manufacturer` define the full asset reference, `mass` is the amount of asset and `tokenid` is the Ocean generated 32 byte token ID.
`update_time(ntime)`	Set the mapping object timestamp (seconds since UNIX epoch). If no argument is given, the current system time is used.
`update_height(block_height)`	Set the block height stamp in the mapping object.
`get_time()`	Return the current object timestamp
`get_height()`	Return the current object block height.
`remove_asset(asset_ref, year_ref, mass, tokenid, manufacturer)`	Remove all assets from the object with the supplied asset reference.
`verify_multisig(controller_pubkeys)`	Verify the object ECDSA signature against the object multisig policy. `controller_pubkeys` is a list of the hex encoded (compressed) controller public keys. Returns either `True` or `False`.
`verify_blockchain(commitment)`	Verify the SHA256 hash of the entire object against the supplied value. Returns either `True` or `False`.
`load_json(filename)`	Load a JSON encoded object from file. `filename` is the file path.
`sign_db(privkey, index)`	Add an ECDSA signature to the object, using a hex encoded 32 byte private key `privkey`. `index` is used to specify the private key owner (i.e. the controller number).
`export_json(filename)`	Export the object to a JSON encoded file. `filename` is the file path.
`print_json()`	Prints the JSON encoded object to screen.
`remap_assets(burnt_tokens, asset_reference, redemption_height)`	Remap the assets in the object as an arbitrary number of tokens are destroyed to redeem a single asset. `burnt_tokens` is an array of the token IDs that have been burnt and the corresponding amounts. `asset_reference` is the full reference of the asset that is being redeemed. `redemption_height` is the blockchain height at which the redemption was initiated. Returns `True` if successful, or `False` with an error message if the re-mapping fails due to incorrect supplied values.
`get_total_mass()`	Returns the total amount of all assets in the mapping object.
`get_mass_token(tokenid)`	Returns the total amount of assets mapped to `tokenid`.
`get_mass_asset(asset_ref)`	Returns the total amount of `asset_ref`.

12.4 Scripts

The various scripts and utilities included in the repository perform various functions in the lifecycle of a mapped token, including asset issuance, the redemption process and the monitoring of tokens issued on-chain. These scripts interact with both a remotely stored mapping object and with the Ocean blockchain client via the RPC interface.

The following operation scripts are included (in the `scripts` directory):

`object_init.py`

Creates an unsigned and empty mapping object `map.json` with a 2-of-3 signature policy and uploads it (and the policy asset output `ptxo.dat` file) to the S3 bucket.

`controller_setup.py`

Securely generates controller public/private key pairs and BIP39 mnemonic seed phrases. It generates a `c_privkey.dat` file with the controller private key and prints the backup seed phrase to the terminal. It also can generate a `controllers.json` file which contains the public key(s) and a `p2sh.json` file which contains the mutlisig Ocean P2SH address and redeem script corresponding to the controller public keys (for multiple controllers with a 2-of-3 policy).

`issue_token_coordinator.py`

Initiates the issuance of new tokens corresponding to controlled assets, performed by the *coordinator*. With a 2-of-3 policy, any one controller can be the coordinator and any other the *confimer*. The script retrives the mapping object and policy asset UTXO file from an S3 bucket and verifies the signatures. Issued asset details are input by the coordinator and the script generates a new partially signed mapping object `ps1_map.json` and partially signed issuance transaction `ps1_tx.json` which are uploaded to S3. `reissuanceToken` must be set to the address of the inflation controller (the block-signing federation).

`issue_token_confirm.py`

Completes the issuance of new tokens, performed by the confirmer. The script retrieves partially signed mapping object and issuance transaction, and confirms the issuance details. The confirmer signatures are then added and the issuance transaction is broadcast to the network. Once confirmation is recieved, the fully signed mapping object is uploaded to S3.

`redemption_coordinator.py`

Initiates the process for removing an asset from the the mapping object. The script requires the types and amounts of burnt tokens to be entered which is confirmed against the blockchain, and the token-asset mappings are then updated by the re-mapping algorithm. The script then outputs the modified and partially signed mapping object and uploads to S3.

`redemption_confirmer.py`

Completes the redemption/re-mapping process. The partially signed modified mapping object is retrieved from S3 and compared to the current version. The types and amounts or burnt tokens are entered and compared with the on-chain state. The script then fully signs the mapping object and uploads it to S3.

`sig_verify.py`

Reads in `map.json` and `controllers.json` and verifies the signatures over the mapping object according the the multisig policy and the controller public keys.

`token_report.py`

Retrieves mapping object from the S3 bucket and connects to the Ocean client to perform a scan of all blockchain UTXOs and produces a comparison report. Any tokens issued on chain and not included in the mapping object, or vice-versa are identified.

CHAPTER 13

Issuance process initialisation

This document describes the steps required to set-up and initialse the token issuance process and permissions on an Ocean federated blockchain.

Prerequisites:

- The `amap` library is installed and/or available in the path of the directory where the scripts are run
- The Ocean client is available via an RPC connection
- An S3 remote file storage bucket is available and configured with write permissions for each controller (all scripts are configured for the bucket `cb-mapping`)

13.1 Controller Set-up

The first part of the initialisation involves the generation of the controller key pairs and the generation of the issuance asset P2SH address and redeem script. For demonstration purposes, the single script `controller_setup.py` automatically generates 3 controller key-pairs and generates the corresponding 2-of-3 P2SH address and redeem script (which is saved to a file named `p2sh.json`).

The script must be configured with the following parameters to enable a connection to the ocean client: `rpcport`, `rpcuser`, and `rpcpassword`.

This script is run with:

```
python3 controller_setup.py
```

This script outputs the private key, public key and recovery phrase of each three controllers to screen, and writes each controller private key to a file `cX_privkey.dat` (X = 1,2,3). The three controller public keys are added to a `ConPubKey` object, which is then written to the `controllers.json` file (this file is used for the mapping object signature verification).

In the production version, each controller key-pair will be generated separately on isolated hardware (by running the script `controller_single.py`). The private key will be stored on the isolated machine, and the recovery

phrase written down by each controller to be stored securely. The public key for each controller will then be exported via removable media (or written down) and sent to a single entity (CB) where they will be used to create the `controllers.json` and `p2sh.json` files (by running the script `controller_aggregate.py`).

13.2 Sidechain configuration

Once the controller public keys have been generated and aggregated, the Ocean blockchain can be configured with the controller public keys. This is done to immutably link the controllers with the blockchain (so that the mapping object signatures can be verified directly) and to pay the issuance asset coins to a multisig controller address, which will give them permission to create new token types.

The controller public keys are added to the `ocean.conf` file via the `issuecontrolscript` parameter. This parameter is set to the hex-encoding of the redeem script generated in the `p2sh.json` file. The `issuancecoinsdestination` parameter is set with the P2SH multisig `scriptPubKey` generated in the `p2sh.json` file. In addition a quantity of issuance asset coins to be generated in the genesis block must be specified (this value is arbitrary, but must be greater than 100) with the `policycoins` parameter. For example:

This configuration then forms part of the genesis block definition.

13.3 Issuance asset UTXO list

In order to issue new tokens, issuance transactions (signed by the controllers) must spend an issuance asset output. These outputs are initially created in the genesis block, and are tracked by the issuance process (i.e. new issuance asset outputs are created as change for each issuance). The current list of available issuance asset outputs is kept in the the `ptxo.dat` wallet file, which is updated at each issuance.

To initialise the `ptxo.dat` wallet file, first determine all the issuance asset outpoints by first importing the controller P2SH address into a configured Ocean client wallet:

```
ocean-cli importaddress 3BKwcuph76AgTgo7d5nNdW4EYRL7v1PawN
```

Then list all unspent outputs:

```
ocean-cli listunspent
```

For each output, add a line to the `ptxo.dat` file with:

```
txid vout value
```

13.4 Mapping initialisation

Once the blockchain has been configured and initialised, a 'blank' and unsigned mapping object with a 2-of-3 issuance is created and exported as a file: `map.json`. This blank object and the `ptxo.dat` wallet file are then uploaded to the S3 bucket. All these operations are performed by running the `object_init.py` script.

CHAPTER 14

Token issuance protocol

This document describes the procedure for issuing a new mapped token on a controller-permissioned Ocean blockchain. It is assumed that the full initialisation process has been completed, as described in initialisation.md and there are three controllers. Each controller has generated a secure and backed-up private key (stored on isolated hardware), and also has access to a workstation/PC that:

1. Has a full internet connection
2. Has both Python3 and the `amap` library (and dependencies) installed
3. Has a configured and synced Ocean client installed and running
4. Has configured permissions for read/write access to an S3 remote filestore
5. Has the controller publiåc key file `controllers.json` in the local directory.

The default policy of the issuance is configured as 2-of-3: this means that only 2 of the 3 controllers are required to issue new tokens and modify the mapping object (the third controller is only required if one of the first two controller keys is lost). One controller is designated as the *coordinator* that initiates the issuance process, and the other is designated as the *confirmer* that completes the issuance process. The proceedure is then detailed in terms of these two roles.

14.1 Coordinator

The coordinator runs the script `issue_token_coordinator.py` which must be configured with the following parameters:

1. The S3 bucket name in the `boto3` calls (default: `cb-mapping`)
2. The Ocean node RPC details: `rpcuser`, `rpcpassword` and `rpcport`
3. The reissuance token address `reissuanceToken`

`reissuanceToken` is the base-58 encoded address that the re-issuance token for each issuance is sent to - which enables to re-issuance (inflation) of the issued token. This address should be the P2SH multisig address of the block-signing federation script (which issues new tokens as part of the inflation/demurrage schedule).

Once configured, the script is run as follows:

```
python3 issue_token_coordinator.py
```

The script is user-friendly and prints verbose verification information to screen. The current valid mapping object is retrieved from the S3 bucket and the signatures verified against the controller public keys.

Details of the issuances (including asset details and issuance destination addresses) are entered via standard input. The controller private key (as generated at initialisation) must be in the file `c1_privkey.dat` located in the local directory. [Note that in production, the signing step will occur on a separate isolated device requiring a splitting of the script].

If the script completes without error, it produces two files which are written to the local directory: the partially signed updated mapping object `ps1_map.json` and the partially signed issuance transaction(s) `ps1_tx.json`. These files are automatically uploaded to the remote S3 bucket.

14.2 Confirmer

The confirmer runs the script `issue_token_confirmer.py` which must be configured with the following parameters:

1. The S3 bucket name in the `boto3` calls (default: `cb-mapping`)
2. The Ocean node RPC details: `rpcuser`, `rpcpassword` and `rpcport`
3. The reissuance token address `reissuanceToken`

Once configured, the script is run as follows:

```
python3 issue_token_coordinator.py
```

The script retrieves both the current *valid* (i.e. fully signed) mapping object and the *new* partially signed mapping object from the remote S3 repository, verifies the signatures and then displays the differences. The confirmer is required to verify that all of the issuances are correct and they then complete the issuance - the confirmer signatures are the partially signed issuance transactions. The issuance transactions are then broadcast to the network via the RPC interface with the Ocean node, and the script waits for confirmation before checking that all tokens have been issued on the blockchain correctly.

Once blockchain issuance has been confirmed, the confirmer signature is added to the partially signed mapping object making it valid and superseding the previous version (as it has a more recent timestamp and block-height). The new mapping object is written to file locally (`map.json`) and then automatically up-loaded to the S3 bucket.

In addition, the `ptxo.dat` policy asset output database file is updated with the change in outputs and uploaded to the S3 bucket.

CHAPTER 15

Asset redemption protocol

This document describes the procedure for redeeming an asset backed mapped token on a controlled Ocean blockchain. It is assumed that the full initialisation process has been completed, as described in initialisation.md and there are two available controllers (coordinator and confirmer). It is assumed that tokens have been issued, and that a *redeemer* has sufficient tokens to redeem and asset.

The redemption process is dependent on a list of *redeemable* assets: this is a simple JSON array of asset references that can be redeemed and is a subset of all issued assets. The redeemable asset list is stored in a file `rassets.json` which is uploaded to the S3 bucket. The redemption process involves four entities (redeemer, custodian, coordinator and confirmer), which perform operations in the following sequence:

15.1 Redeemer

The redeemer wishes to redeem one of the assets listed in the `rassets.json` object (these can be displayed on the redemption website). The script `get_redeemable.py` displays the current list, along with the amount of tokens required to redeem each of the listed assets (which is a function of the asset quantity and the current token ratio).

The redeemer then chooses an asset to redeem and runs the `freeze_script.py` (after configuring `rpcuser`, `rpcpassword` and `rpcport` to their local Ocean wallet). This script prompts them to enter the reference ID of the asset they have chosen to redeem. The script then interacts with the redeemer wallet to generate a signed *redemption transaction* that is exported as a file: `rtx-assetid.dat` (where `asset` is the asset reference). This transaction pays the exact quantity of tokens (of any number and type of token ID) to redeem the asset at the current token ratio, back to addresses controlled by the redeemer wallet. In addition, the first output of the transaction consists of zero value paying to a *zero* address (i.e. `0x0000000000000000000000000000000000000000`) which tags the redemption transaction as *uninflatable* to the signing nodes. Note that the redemption transaction has not been broadcast at this point.

In addition to the redemption transaction, the redeemer must also pay a *redemption fee* in tokens. The fee amount `rfee` is displayed on the redemption website, along with custodian fee address (`fAddress`). The redeemer generates an additional raw transaction from their wallet paying the fee to this address, using any tokens, which is exported into a file `rfee.dat`.

The redeemer then initiates the redemption of the asset by sending a request to the *custodian* (via a web-interface) which included the two raw transaction files.

15.2 Custodian

The custodian is the general term for the entity processing the redemption and the delivery of the asset to the redeemer. The redemption request to the custodian (via a web-interface) then must specify the asset reference ID, and include the `rtx-assetid.dat` and `rfee.dat` transaction files (uploaded as attachments). [the request may include additional user information and delivery information].

The custodian (back-end, with connection an Ocean node) then performs the following operations:

1. Check the redemption fee tx is valid (and pays to the correct custodian address)
2. Check that the asset reference is in the `rassets.json` array.
3. Decode the raw transaction contained in the `rtx-assetid.dat`
4. Check that the transaction is valid
5. Check that the total tokens transferred in the transaction equal the asset quantity multiplied by the current token ratio.
6. The asset reference in the `rassets.json` array is marked as *locked* and the file is uploaded to the S3 bucket.

Once validity is confirmed, the transactions are submitted after the redeem transaction addresses are added to the freezelist.

1. Submit fee transaction
2. Add redemption transaction output addresses to freezelist
3. Confirm freezelist
4. Submit the redemption transaction to the network
5. Confirm - send transactions IDs to the custodian back-office (via e-mail).

This logic is implemented in the `redeem_check.py` script.

15.3 Redeemer

The redeemer is contacted directly by the custodian with further instructions. If the redemption request is not accepted, the custodian updates the freezelist database to remove the redemption transaction output addresses. These become spendable again and the redeemer can spend those outputs.

If the redemption request is accepted, the custodian adds the output addresses to the *burnlist* database. This then enables to the redeemer to submit a *burn* transaction spending the outputs of the redemption transactions to NULL (OP_RETURN) outputs.

To do this, the redeemer runs the `burn_tokens.py` script (after configuring `rpcuser`, `rpcpassword` and `rpcport` to their local Ocean wallet). This script requires the user to enter the txid and vout indexes of the redemption transaction. The script then outputs the token IDs and values that have been burnt to the file `burntassets.dat`, which is sent by the redeemer to the custodian (via email etc).

15.4 Custodian

The custodian receives the `burntassets.dat` from the redeemer, along with the burn transaction txid. The custodian then checks that burn transaction is confirmed, and the redeemer takes possession of the asset. The custodian then contacts the controllers (and sends them the `burntassets.dat` file) to finalise the re-mapping of the mapping object.

15.5 Coordinator

The coordinator runs the script `redemption_coodinator.py` which requires the input of the data in the `burntassets.dat` file. The script automatically checks that the tokens are burnt on the blockchain and then removes the redeemed asset from the mapping object, remapping the associations if required.

The partially signed updated (re-mapped) object is uploaded to the S3 bucket.

15.6 Confirmer

The confirmer runs the script `redemption_confirm.py` which again requires the input of the data in the `burntassets.dat` file. The script automatically checks that the tokens are burnt on the blockchain, and verifies the changes in the re-mapped object, before adding the controller signature and uploading the fully signed and re-mapped object to the S3 bucket, completing the redemption process.