# Ocean Platform:

1. **Ocean Platform**
   1.1 Ocean sidechain features ................................................. 4

2. **Permissions and Policy**
   2.1 Token issuance permissions ................................................. 7
   2.2 Policy lists and transaction control .................................... 7
   2.3 Policy transactions ...................................................... 8
   2.4 Policy tokens ............................................................ 9
   2.5 User onboarding .......................................................... 10

3. **Ocean Node Setup**
   3.1 Installation ............................................................... 13
   3.2 Configuration ............................................................ 13
   3.3 Running the Ocean node .................................................. 14
   3.4 Launching with Docker ................................................... 14

4. **Sidechain configuration**
   4.1 Sample sidechain configuration ....................................... 19
   4.2 Options ................................................................. 20

5. **Block signing**
   5.1 Instructions ............................................................... 26
   5.2 Federation protocol demo ................................................. 28

6. **Ocean API reference**
   6.1 Quick reference .......................................................... 29

7. **Integration guide**
   7.1 Address Types ............................................................ 65
   7.2 Client wallet ............................................................. 66
   7.3 Fees ....................................................................... 67
   7.4 Transactions ............................................................ 67
   7.5 Serialization ............................................................. 68

8. **Ocean Wallet**
   8.1 Configuration ............................................................. 71
   8.2 Wallet servers ........................................................... 72
# Background

9.1 Attestation and Timestamping .................................................. 74

# Protocol

10.1 Single-key protocol ............................................................. 78
10.2 Federated protocol ............................................................... 81

# Mainstay Service Protocol

11.1 Overview .............................................................................. 85
11.2 Commitment Merkle Tree ...................................................... 86
11.3 Slot connection .................................................................. 88
11.4 Proof of Immutable Sequence ................................................ 89
11.5 Commitment frequency and fee policy ..................................... 90
11.6 Staychain multi-signature security .......................................... 91

# Mainstay connector service API

12.1 REST framework structure ....................................................... 93
12.2 Public Endpoints .................................................................. 93
12.3 Authenticated Endpoints ......................................................... 101

# Mainstay service client

13.1 Requirements ...................................................................... 103
13.2 Installation .......................................................................... 103
13.3 Usage .................................................................................. 103

# Sidechain guide

14.1 Initial set-up ......................................................................... 111
14.2 Attestation .......................................................................... 112
14.3 Verification .......................................................................... 112

# File repository guide

15.1 Initial set-up ......................................................................... 115
15.2 Attestation .......................................................................... 116
15.3 Verification .......................................................................... 116

# Git repository guide

16.1 Initial set-up ......................................................................... 119
16.2 Attestation .......................................................................... 120
16.3 Verification .......................................................................... 120

# Application

17.1 Prerequisites ....................................................................... 123
17.2 Instructions ......................................................................... 123
17.3 Mainstay configuration ......................................................... 124
17.4 Tools .................................................................................. 127
17.5 Initialising Mainstay .............................................................. 129
17.6 Running the service .............................................................. 131
17.7 Commitment examples .......................................................... 132

# Guardnode protocol design

18.1 Network model ..................................................................... 133
18.2 Guardnode overview ............................................................. 134
18.3 Request creation: client chain connection .............................. 135
18.4 Ticket stake auction .............................................................. 136
18.5 Service delivery and verification ............................................ 138
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 Ocean platform sidechain client and wallet, 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](https://github.com/commerceblock). Technical questions and suggestions for improvements can be raised as issues on the relevant repositories. In addition, technical questions related to our software and processes are welcomed on our Telegram group.
The open-source Ocean platform has 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 Ocean platform 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 Ocean platform model 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 via the Ocean Wallet.

Asset-backed sidechains can be configured as either private or publicly verifiable blockchains. If configured as public, the security and decentralisation of the network can be optionally enhanced through CommerceBlock’s *Guardnode*.
The Ocean sidechain client (incorporating 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.

1.1 Ocean sidechain features

1.1.1 Immutability

Ocean sidechains linked to the Mainstay service are as trustlessly immutable as Bitcoin, backed by the global decentralised Proof-of-Work consensus system. Ocean 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

Ocean sidechains operated by token issuers have the capability to incorporate user address whitelists and blacklists into the block-producing 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 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 the lightweight Ocean 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, or even remove the requirement for fees entirely. 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 block-signing nodes, and is only really limited by hardware. Ocean nodes can be launched easily on cloud infrastructure, being fully containerised (with Docker images for the major cloud providers). 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.

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 Token issuance permissions

A principal function of Ocean 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.

Permission to issue new tokens on an Ocean sidechain is controlled via the issuance tokens which is specified in the genesis block of particular sidechain. The issuance token permissioned by a specified scriptPubKey given in the chain configuration as the issuancecoinsdestination.

2.2 Policy lists and transaction control

A core functionality of the Ocean platform is the ability of the sidechain operators (the asset issuers) to control and restrict the user transaction permissions, while issued token outputs remain owned (via the control of private keys in the user wallet) 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 frezelist)
- 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 block signing nodes, and therefore be allowed to transfer ownership of a tokenised asset peer-to-peer.
The policy lists are used by the block-signing 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.

### 2.3 Policy transactions

The three policy lists (whitelist, freezelist and burnlist) are kept in the client memory, 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: so-called policy transactions. This enables addresses to be added and removed from the policy lists by remote authorised agents outside of the 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 block-signing nodes, and it is then enforced. A second policy transaction spending this previous output is then be used to remove the address from the list.
2.4 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, and form part of the sidechain genesis block.

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 can optionally be integrated directly with a KYC/ID service provider for onboarding new users of a sidechain or controlled manually by agents or
staff of the asset issuer.

2.5 User onboarding

The Ocean platform incorporates an onboarding protocol that is designed to streamline the adding of new users who have passed issuer-determined KYC checks. This protocol allows users to self-register validated whitelist addresses in a way that preserves privacy on a publicly validated sidechain.

2.5.1 Preliminaries

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

A WHITELIST asset is defined and created 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.

2.5.2 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 client using the dumpkycfile command, or from the Ocean 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.

### 2.5.3 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.

### 2.5.4 Node restart

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

### 2.5.5 Privacy

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

### 2.5.6 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.
CHAPTER 3

Ocean Node Setup

To run an Ocean node for a specific sidechain, the following is required:

- Download/install the Ocean client binaries.
- Configure the node using a configuration file and contract.

The applications required:

**OceanDaemon (oceand)**

An Ocean node that runs as a background application. It can process requests made from other applications using Remote Procedure Calls (RPC).

**OceanClient (ocean-cli)**

The client application that enables command line calls to oceand by issuing RPC commands. Commands for ocean-cli are specified in the Ocean API Reference.

### 3.1 Installation

Ocean can be downloaded from the CommerceBlock Github repository. Alternatively, executables can be compiled directly from the source code.

After installation, the node must then be configured.

### 3.2 Configuration

The oceand and ocean-cli applications use a configuration file named ocean.conf. This file defines the consensus rules of the sidechain, specify which network to connect to and can set a number of different behaviors within the application. It also defines what credentials must be provided in order to use the RPC interface. The ocean-cli application uses the configuration file to obtain the correct credentials in order to communicate with oceand using RPC.
When either of these applications are started you must provide a datadir path. The path you provide tells the applications which directory to use to:

- Obtain RPC authentication data (user, password, port).
- Store blockchain and wallet data.
- Store log files etc.

Download the sidechain specific ocean.conf file from the sidechain operator or asset issuer, and copy it to the datadir path. If there is a contract for the sidechain (terms and conditions) this will need to be copied to a terms-and-conditions directory in the datadir.

### 3.3 Running the Ocean node

Once the configuration file and contract are in place the Ocean daemon can be started.

**Linux**

Run each application from the command line within the folder you extracted them to, along with the -datadir argument. For example:

```
./oceand -datadir=path
```

And

```
./ocean-cli -datadir=path
```

Depending on your system set up, you may have to change the permissions on the files before they will run.

RPC commands can then be passed to the client via ocean-cli

To verify the genesis block hash created with the configuration, run:

```
./ocean-cli -datadir=path getblockhash 0
```

To query the synchronization status of the node, run:

```
./ocean-cli -datadir=path getblockchaininfo
```

### 3.4 Launching with Docker

Instructions for launching a full configured Ocean node with Docker.

#### 3.4.1 Requirements

Docker engine release: 18.02.0 or latest
docker-compose: 1.20.0 or latest
3.4.2 Download docker-compose.yml

Download the `docker-compose.yml` file and the contract (terms and conditions) from the sidechain operator or asset issuer.

For example (Ocean testnet):

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

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

3.4.3 Download image and start

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

3.4.4 Check status

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

3.4.5 Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Command</th>
<th>State</th>
<th>Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>ocean_node_1</td>
<td>/docker-entrypoint.sh elem ...</td>
<td>Up</td>
<td>0.0.0.0:32768-&gt;18332/tcp, 0.0.0.0:32769-&gt;7042/tcp</td>
</tr>
</tbody>
</table>

3.4.6 Check logs

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

Hit ctrl+c to stop following

3.4.7 Check if connection

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

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

3.4.8 Check block count

```
docker-compose -p ocean exec node ocean-cli -rpchost=18332 -rpcuser=ocean -rpcpassword=oceanpass getblockcount
```
Once synced, block count should be the same as in: https://cbtexplorer.com

### 3.4.9 Data persistence

```bash
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

### 3.4.10 Using docker secrets

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

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

Modify the service block to use the newly defined secrets:

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

Remove the rpc authentication arguments from the command:

```bash
-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.

### 3.4.11 Dig deeper

As root

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

As bitcoin

```bash
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
3.4.12 Execute shell commands

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

3.4.13 Scale containers

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

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

3.4.14 Stop

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

3.4.15 Remove stack

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

The Ocean platform sidechain client `oceand` can be downloaded and compiled from our Github repository. Interaction with the running configured client is performed via an RPC connection (as specified in the configuration file), or via the `ocean-cli` command-line tool.

The permissions that control an Ocean sidechain, and the unique binding to the Bitcoin blockchain via the Mainstay protocol are defined in the chain-specific configuration. Ocean nodes are configured via `oceand` command line arguments or parameters specified in the `ocean.conf` configuration file located in the `--datadir=path` argument.

The Ocean sidechain client 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.

### 4.1 Sample sidechain configuration

```plaintext
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
genesistimestamp=1568700000
freezelistcoinsdestination=76a9149d2eaa0bb68b5b9ba11250994fddfce78f41f0c0188ac
burnlistcoinsdestination=76a91415de997afac9857dc97cdd43803cf1138f3aee8788ac
whitelistcoinsdestination=76a914ff9b5c6885f87fb5519cc45c1474f301a73224a88ac
```

(continues on next page)
policycoins=2100000000000000
signblockscript=76a91464e3e58fa0a18348d94f064a09fe6ec65448ef588ac
con_mandatorycoinbase=76a9149ad28094c9ad9a772f0267b7e5bec28a8707ae688ac
issuancecoinsdestination=ae147082861e2c1b8b7e6327d4951e294be00c87
issuancecontrolscript=5221021012119c4f73a85b48f3206e33e58fa0a18348d94f064a09fe6ec65448ef588ac

4.2 Options

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

Chain

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes (but ocean defaults to test params)</td>
<td></td>
</tr>
</tbody>
</table>

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 Ocean Wallet (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

<table>
<thead>
<tr>
<th>genesis-hash critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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) if contractintx=0. 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. If contractintx=1 the hash is added to all wallet transactions, and is required by the signing nodes as mempool policy.

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

4.2.1 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

<table>
<thead>
<tr>
<th>genesis-hash critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Script determining ownership of the issuance process.

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

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

`freezelistcoinsdestination`

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

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

`burnlistcoinsdestination`

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

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

`whitelistcoinsdestination`

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`

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

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

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 (Hardware Security Module) 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`

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
Wether to enforce whitelisting rules at this node. Set to pkhwhitelist=1 for signing nodes if whitelisting is to be used.

\textbf{pkhwhitelist-scan}

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

Wether to scan the blockchain for whitelisted addresses and KYC public keys. \texttt{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.

\textbf{rescan}

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

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

\textbf{reindex}

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

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

\textbf{attestationhash}

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td></td>
<td>no (if using Mainstay)</td>
</tr>
</tbody>
</table>

Reference to a transaction ID hash in the Mainstay staychain in the Bitcoin blockchain. This should be the staychain transaction ID prior to the first sidechain attestation.

\textbf{genesisstamp}

<table>
<thead>
<tr>
<th>genesis-hash</th>
<th>critical</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>

The unix epoch timestamp included in the genesis block of the sidechain.
The Ocean platform incorporates a client/daemon that is installed and run on infrastructure controlled by the issuer to validate and sign blocks on an Ocean sidechain via a fault tolerant multiparty protocol (referred to as *Federated block signing* and the nodes with block-signing permissions as *The Federation*). The block signing permissions are defined by the public keys in the M-of-N multisig blocksigning script which is specified in the Ocean sidechain configuration (*signblockscript*). Each federation signing node coordinates with the other federation nodes to produce 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 according to a specified schedule.
Illustration of a federation of 5 signing nodes, where 3 signing keys are required to produce a valid block to extend the sidechain.

## 5.1 Instructions

The federated block-signing daemon is a pure Python application and can be downloaded from the CommerceBlock Github [here].

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`

#. For the federation run `.run_federation` or `python3 -m federation` and provide the following arguments: .. code-block:: bash

```bash
--rpcconnect $HOST --rpcport $PORT --rpcuser $USER --rpcpassword $PASS --id $NODE_ID --msgtype $MSG_TYPE --nodes $NODES_LIST
```

Federation arguments:

- `--rpcconnect`: 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**:
- **kafka**: python3 -m federation --rpconnect 127.0.0.1 --rpcport 18443 --rpcuser user --rpcpass pass --id 1 (check federation.py - defaults to 5 nodes)

### 5.1.1 Hardware Security Modules

The application has full integration for Hardware Security Modules (HSMs) that support the PKCS11 interface, for block-signing operations and key management.

**Initialisation**

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

```bash
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:

```bash
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:

### 5.1.2 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.

5.1. Instructions
5.2 Federation protocol demo

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

5.2.1 Instructions

./scripts/restart_kafka.sh python3 -m demo

5.2.2 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
  ```

5.2.3 MultiSig

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

5.2.4 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.

5.2.5 Asset Issuance

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

The Ocean client inherits all of the Remote Procedure Calls (RPCs) from Elements 0.14 (described here) which in turn inherits all RPCs from Bitcoin 0.13. Ocean specific RPCs control the advanced and extended features unique to the platform. 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 platform 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
```

As in Elements and Bitcoin, the Configuration options can be passed to oceand as command line arguments or added to the ocean.conf configuration file. For a full list of RPCs and configuration options use oceand -help.

### 6.1 Quick reference

The following RPCs are unique to the Ocean client

#### 6.1.1 Wallet

- dumpderivedkeys
- validatedderivedkeys
- dumpkycfile
- readkycfile
- createkycfile
- getderivedkeys
- getcontract
- getcontracthash
• getmappinghash
• getethaddress
• getethpeginaddress
• getethpegin
• createrawethpegin
• validateethpegin
• claimethpegin
• sendtoethmainchain
• sendanytoaddress
• createanytoaddress

6.1.2 Utility

• getutxoassetinfo
• createrawissuance
• createrawreissuance
• createrawburn
• testmempoolaccept
• createrawpolicytx
• createrawrequesttx
• getrequests
• createrawbidtx
• getrequestbids

6.1.3 Policy

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

6.1.4 Configuration options

• pkhwhitelist
• freezelist
• burnlist
• issuanceblock
• disablect
• embedcontract
• attestationhash
• embedmapping
• issuecontrolscript
• policycoins
• initialfreecoinsdestination
• freezelistcoinsdestination
• burnlistcoinsdestination
• issuancecoinsdestination
• permissioncoinsdestination
• mainchainrpchost
• mainchainrpcport
• validatepegin
• parentgenesisblockhash
• parentcontract
• fedpegaddress
• peginconfirmationdepth

6.1.5 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
### 6.1.6 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file-name</td>
<td>String</td>
<td>Required(exactly 1)</td>
<td>The name of the output file for the list of tweaked addresses and base public keys</td>
</tr>
</tbody>
</table>

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

**Example**

```
ocean-cli validatederivedkeys
```

### 6.1.7 dumpkycfile

The `dumpkycfile` RPC outputs an encrypted list of wallet tweaked public keys. A timestamp and best block hash of when the file was constructed is included.

**Parameter #1**—the filename of the output file

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file-name</td>
<td>String</td>
<td>Required(exactly 1)</td>
<td>The name of the output file for the encrypted list of tweaked public keys</td>
</tr>
</tbody>
</table>

**Parameter #2**—the public key with which to encrypt

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onboardpubkey</td>
<td>String(hex)</td>
<td>Optional (0 or 1)</td>
<td>The specific public key to be used for file encryption</td>
</tr>
</tbody>
</table>

**Result**—none if valid, errors returned if invalid inputs

**Example**

```
ocean-cli dumpkycfile dumpfile.txt 1CDXUtbf3bBtritydFMKhRbbYhxDgCF5oH
```

---

**CommerceBlock, Release 19.04**

**Example**

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

---

**CommerceBlock, Release 19.04**
6.1.8 readkycfile

The readkycfile RPC reads in an encrypted list of tweaked public keys (as produced by dumpkycfile) from a specified file, and outputs the unencrypted list of tweaked addresses along with their corresponding non-tweaked public keys.

**Parameter #1—the encrypted filename**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file-name</td>
<td>String</td>
<td>Required (exactly 1)</td>
<td>The name of the output file for the encrypted list of tweaked public keys</td>
</tr>
</tbody>
</table>

**Parameter #2—the filename of the output file**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outfile-name</td>
<td>String</td>
<td>Required (exactly 1)</td>
<td>The name of the output file for the list of tweaked public keys and corresponding addresses</td>
</tr>
</tbody>
</table>

**Result**—none if valid, errors returned if invalid inputs

**Example**

```
ocean-cli readkycfile dumpfile.txt dumpfileout.txt
```

6.1.9 createkycfile

The createkycfile RPC creates an encrypted kyc file that stores tweaked public key p2pkh and multisig address p2sh data to be whitelisted when onboarding.

**Parameter #1—the created KYC file name**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outfilename</td>
<td>String</td>
<td>Required (exactly 1)</td>
<td>Name of the KYC file</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pub-keylist</td>
<td>JSON array of JSON Objects</td>
<td>Required (1 or more)</td>
<td>Contains tweaked addresses and respective untweaked public keys</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>multisiglist</td>
<td>JSON array of JSON Objects</td>
<td>Required (1 or more)</td>
<td>Contains multisig metadata such as number of required signatures and arrays of untweaked public keys</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onboardpubkey</td>
<td>String(hex)</td>
<td>Optional (exactly 1)</td>
<td>Public key that is used for onboarding encryption</td>
</tr>
</tbody>
</table>

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

Example

```plaintext
ocean-cli createkycfile test ["{"address":2dZhhVmJkXCaWUzPnhmwQ3gBJm2NJSnrvyz,"pubkey":028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58},"nmultisig":1,"pubkeys":[028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58,0263a73eca5334af77037a1c8844b5220017bf6fb627c5a57c862dfff20ea01d99}]
```

Result:

```
028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58
```

6.1.10 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required(key pool size)</td>
<td>Base58check encoded address corresponding to the contract tweaked public key</td>
</tr>
<tr>
<td>bpubkey</td>
<td>String (hex)</td>
<td>Required(key pool size)</td>
<td>Hex encoding of the compressed untweaked public key</td>
</tr>
</tbody>
</table>

Example

```plaintext
ocean-cli getderivedkeys
```

Result:

```
{
    "address": ["2dZhhVmJkXCaWUzPnhmwQ3gBJm2NJSnrvyz", "2dC2B5loAp3q54JxW4xERsOpYdAWdH7G5nC2Os7b5"],
    "bpubkey": ["028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58", "0263a73eca5334af77037a1c8844b5220017bf6fb627c5a57c862dfff20ea01d99"]
}
```

6.1.11 getcontract

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

Parameters: none

Result—the full plain text of the current contract

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contract</td>
<td>Objects</td>
<td>Required (exactly 1)</td>
<td>A JSON object containing the plain text of the contract</td>
</tr>
</tbody>
</table>

Example
ocean-cli getcontract

Result:

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

### 6.1.12 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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block-height</td>
<td>Integer</td>
<td>Optional(0 or 1)</td>
<td>Block height to retrieve the contract hash from (Default: most recent block)</td>
</tr>
</tbody>
</table>

**Result—the contract hash**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contracthash</td>
<td>String</td>
<td>The hex-encoded hash of the contract hash</td>
</tr>
</tbody>
</table>

**Example**

```
ocean-cli getcontracthash
```

Result:

```
f4f30db53238a7529bc51fcda04ea22bd8f8b188622a6488da12281874b71f72
```

### 6.1.13 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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block-height</td>
<td>Integer</td>
<td>Optional(0 or 1)</td>
<td>Block height to retrieve the contract hash from (Default: most recent block)</td>
</tr>
</tbody>
</table>

**Result—the mapping hash**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping</td>
<td>String</td>
<td>The hex-encoded hash of the contract hash</td>
</tr>
</tbody>
</table>

**Example**
ocean-cli getmappinghash

Result:

\texttt{f4f30db53238a7529bc51fcda04ea22bd8f8b188622a6488da12281874b71f72}

6.1.14 getethaddress

The \texttt{getethaddress} RPC returns an ethereum address from an EC private key.

\textbf{Parameter \#1 — EC private key}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>private key to generate Ethereum address from</td>
</tr>
</tbody>
</table>

\textbf{Result—ethereum address if successful, RPC error if invalid data private key given}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Ethereum address for given private key</td>
</tr>
</tbody>
</table>

\textbf{Example}

\texttt{ocean-cli getethaddress}

\texttt{→ 3ecb44df2159c26e0f995712d4f39b6f6e499b40749b1cf1246c37f9516cb6a4}

Result:

\texttt{8a40bfaa73256b60764c1bf40675a99083efb075}

6.1.15 getethpeginaddress

The \texttt{getethpeginaddress} RPC returns information needed for \texttt{claimethpegin} to move coins to the sidechain. The user should send CBT coins from their eth wallet to the \texttt{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.

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

\textbf{Parameter \#1 — private key}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>private key to generate Ethereum address and claim public key from</td>
</tr>
</tbody>
</table>

\textbf{Result—JSON object containing Ethereum address and corresponding public key, or RPC error if unvalid private key given}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eth_mainchain_addresss</td>
<td>String (hex)</td>
<td>Ethereum address for given private key</td>
</tr>
<tr>
<td>eth_claim_pubkey</td>
<td>String (hex)</td>
<td>Claim Public key for given private key</td>
</tr>
</tbody>
</table>

\textbf{Example}

6.1.16 getethpegin

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

Parameter #1 — eth transaction id

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction id</td>
</tr>
</tbody>
</table>

Result—the transaction in JSON format, or RPC error if given transaction is invalid

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>JSON</td>
<td>The resulting transaction</td>
</tr>
</tbody>
</table>

Example

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

Result:

```json
{
    "eth_mainchain_address": "b6872561de5ba19d38071a7616d9d434b9e37860",
    "eth_claim_pubkey": "0397466f2b32bc3bb76d4741ae51cd1d8578b48d3f1e68da206d47321aec267ce7"
}
```
6.1.17 createrawethpegin

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

Parameter #1 — eth transaction id

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction id</td>
</tr>
</tbody>
</table>

Parameter #2 — eth transaction peg-in amount

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction amount</td>
</tr>
</tbody>
</table>

Parameter #3 — claim pubkey generated by getethpeginaddress

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim_pubkey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Claim public key</td>
</tr>
</tbody>
</table>

Result—ethereum address, or RPC error if invalid claim public key given

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (hex)</td>
<td>Ethereum address for given private key</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli createrawethpegin
˓→ 8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7 432.109
˓→ 03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
0200000001016ca60fb08c36a2e77e0810de32181b63e8250fbf9a398f9bf9e53444cbf680300000000400ffffffffff0201bfe394bdcd72be5291a04263 ...
```

6.1.18 validateethpegin

The validateethpegin RPC validates an eth ERC-20 transaction to be used from peg-in to Ocean.
Configuration options validatepegin=1, mainchainrpchost=${GETH_RPC_HOST} and mainchainrpcport=${GETH_RPC_PORT} must be set for this command.

Parameter #1 — eth transaction id

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction id</td>
</tr>
</tbody>
</table>

Parameter #2 — eth transaction peg-in amount

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction amount</td>
</tr>
</tbody>
</table>

Parameter #3 — claim pubkey generated by getethpeginaddress

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim_pubkey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Claim public key</td>
</tr>
</tbody>
</table>

Result—TRUE or FALSE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Bool</td>
<td>validation result</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli validateethpegin
  → 8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10e6e6359cc7 432.109
  → 03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
true
```

6.1.19 claimethpegin

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

Parameter #1 — eth transaction id

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction id</td>
</tr>
</tbody>
</table>

Parameter #2 — eth transaction peg-in amount

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-in transaction amount</td>
</tr>
</tbody>
</table>

Parameter #3 — claim pubkey generated by getethpeginaddress

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim_pubkey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Claim public key</td>
</tr>
</tbody>
</table>

6.1. Quick reference 39
Result—ethereum transaction ID, or RPC error if invalid inputs given

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Ethereum transaction ID</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli claimethpegin
  → 8b75539cc2b54efe15cd3a0f678545e3f154ca69ba87004d484d10eeb1359cc7 432.109
  → 03220271a8833566153dbfa52c4ba13d2e56970885e6178a4ce6fa81ecaf38c35a
```

Result:

```
bb2364284941f08cceeaf49911858125256d61f1b728e544ead6423bf06eae1e15
```

### 6.1.20 sendtoethmainchain

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

**Parameter #1 — destination address on eth mainchain**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Destination Ethereum address for funds</td>
</tr>
</tbody>
</table>

**Parameter #2 — eth amount pegged-out to eth mainchain**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Ethereum peg-out amount</td>
</tr>
</tbody>
</table>

**Parameter #3 — Fee deducted from amount being pegged-out**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtractfeefromamount</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>Whether to deduct fee from peg-out amount</td>
</tr>
</tbody>
</table>

Result—ethereum transaction ID of resulting sidechain transaction, or RPC error if invalid inputs given

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Ethereum transaction ID</td>
</tr>
</tbody>
</table>

Example

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

Result:

```
aa2364284941f08cceeaf49911858125256d61f1b728e544ead6423bf06eae1e15
```
### 6.1.21 sendanytoaddress

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

**Parameter #1 — destination address**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Destination address</td>
</tr>
</tbody>
</table>

**Parameter #2 — amount to be sent to the destination**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Total amount to be send</td>
</tr>
</tbody>
</table>

**Parameter #3 — A comment used to store what the transaction is for**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment</td>
<td>String</td>
<td>Optional (0 or 1)</td>
<td>Comment to store transaction purpose</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment_to</td>
<td>String</td>
<td>Optional (0 or 1)</td>
<td>Comment to store name of recipient</td>
</tr>
</tbody>
</table>

**Parameter #5 — Return a transaction even when a blinding attempt fails due to number of blinded inputs/outputs if this is set to true**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignoreblindfail</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>If true return a transaction even if blinding attempt fails</td>
</tr>
</tbody>
</table>

**Parameter #6 — Split a transaction that goes over the size limit into smaller transactions if this is set to true**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitlargetxs</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>If true split a transaction that goes over the size limit</td>
</tr>
</tbody>
</table>

**Parameter #7 — Choose which balances should be used first. 1 - descending, 2 - ascending**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>balanceSortType</td>
<td>Integer</td>
<td>Optional (0 or 1)</td>
<td>Specify which balance to use first</td>
</tr>
</tbody>
</table>

**Result** — txid of generated transaction, or RPC error if failure

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Constructed transaction txid</td>
</tr>
</tbody>
</table>

**Example**
6.1.22 createanytoaddress

The createanytoaddress RPC creates a transaction that sends an amount to a given address with as many non-policy assets as needed. This rpc should only used in chains that are comprised of non-policy assets which are fungible.

Parameter #1 — destination address

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>address (base58check)</td>
<td>Required (exactly 1)</td>
<td>Destination address</td>
</tr>
</tbody>
</table>

Parameter #2 — amount to be sent to the destination

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount to be send</td>
</tr>
</tbody>
</table>

Parameter #3 — Return a transaction even when a blinding attempt fails due to number of blinded inputs/outputs if this is set to true

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignoreblindfail</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>If true return a transaction even if blinding attempt fails</td>
</tr>
</tbody>
</table>

Parameter #4 — Split a transaction that goes over the size limit into smaller transactions if this is set to true

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitlargetxs</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>If true split a transaction that goes over the size limit</td>
</tr>
</tbody>
</table>

Parameter #5 — Choose which balances should be used first. 1 - descending, 2 - ascending

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>balanceSortType</td>
<td>Integer</td>
<td>Optional (0 or 1)</td>
<td>Specify which balance to use first</td>
</tr>
</tbody>
</table>

Parameter #6 — Allow the selection of watch only inputs similar to fundrawtransaction

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowwatchonly</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>Allow watch only outputs or not</td>
</tr>
</tbody>
</table>

Result—array raw generated transactions, or RPC error if failure

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txs</td>
<td>Array (hex string)</td>
<td>Raw transactions</td>
</tr>
</tbody>
</table>
### 6.1.23 getutxoassetinfo

The `getutxoassetinfo` RPC returns a summary of the total amounts of unspent (and un-burnt) assets in the UTXO set. Amounts 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contract</code></td>
<td>Object</td>
<td>An array of JSON objects of unspent amounts for each issued asset</td>
</tr>
<tr>
<td>→ <code>asset</code></td>
<td>String (hex)</td>
<td>Asset ID</td>
</tr>
<tr>
<td>→ <code>spendabletxouts</code></td>
<td>Integer</td>
<td>Number of spendable utxos for the asset</td>
</tr>
<tr>
<td>→ <code>amoutspendable</code></td>
<td>Amount</td>
<td>Total amount of the asset that is spendable</td>
</tr>
<tr>
<td>→ <code>frozentxouts</code></td>
<td>Integer</td>
<td>The number of the assets outputs that are frozen</td>
</tr>
<tr>
<td>→ <code>amoutfrozen</code></td>
<td>Amount</td>
<td>Total amount of the asset that is frozen</td>
</tr>
</tbody>
</table>

**Example**

```bash
ocean-cli getutxoassetinfo
```

**Result:**

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

### 6.1.24 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetaddress</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check issued asset address</td>
</tr>
</tbody>
</table>

Parameter #2—the amount of issued asset

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetamount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of asset to issue</td>
</tr>
</tbody>
</table>

Parameter #3—the Base58check address for the reissuance token

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenaddress</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check reissuance token address</td>
</tr>
</tbody>
</table>

Parameter #4—the amount of reissuance token

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenamount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of reissuance token to generate</td>
</tr>
</tbody>
</table>

Parameter #5—the Base58check address for the issuanceAsset change

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeaddress</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check change address</td>
</tr>
</tbody>
</table>

Parameter #6—the amount of issuanceAsset change

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeamount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of issuance asset to return</td>
</tr>
</tbody>
</table>

Parameter #7—the number of issuanceAsset outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changenum</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>number of issuance asset outputs</td>
</tr>
</tbody>
</table>

Parameter #8—input TXID

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputtxid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Domain asset input transaction id</td>
</tr>
</tbody>
</table>

Parameter #9—input transaction vout

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vout</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Domain asset input vout number</td>
</tr>
</tbody>
</table>

Result—the unsigned raw transaction in hex, or JSON RPC error

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>Hex encoded raw transaction</td>
</tr>
</tbody>
</table>
### Example

```
ocean-cli createrawissuance 2deJ6F3w6HUtXM8jY5YPC8wtaXerqFX7HA 123.0  
2ddSmTuJAb0CzDyU9hPgHcP40jAGmJRTnWk 1.23 XKSxznoA799169xt3zCm7a4krdTIKZANv3 332.  
9995 1 0.0005 40ac4e02a54ea14190e96d6e1c5c877b12522db3fb5adff58b1aed0cc11150a 0  
```

Result:
```
0200000000010a511ccd0aeb158fdd5afbb32d5217b875c1c6e6de99041a14ea6024eac4000000008000ff  
```

### 6.1.25 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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetaddress</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address to send re-issued asset to</td>
</tr>
</tbody>
</table>

**Parameter #2—the amount of re-issued asset**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetamount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of asset to re-issue</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenaddress</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check reissuance token address</td>
</tr>
</tbody>
</table>

**Parameter #4—the amount of reissuance token**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenamount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of reissuance token</td>
</tr>
</tbody>
</table>

**Parameter #5—input TXID**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputtxid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Re-issuance token input transaction ID</td>
</tr>
</tbody>
</table>

**Parameter #6—input transaction vout**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vout</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Re-issuance token input vout number</td>
</tr>
</tbody>
</table>

**Parameter #7—the asset entropy**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entropy</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Hex encoded asset entropy</td>
</tr>
</tbody>
</table>
Result—the unsigned raw transaction in hex, or JSON RPC error

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>Hex encoded raw transaction</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli createrawissuance 2deJ6F3w6HUtXM8jY5YPc8wtaXergFX7HA 0.5
→2ddSmTuJAbC2zDyU9hPghcp4oJAGmJRtnWk 1.00
→40ac4e02a64ea14190e96d66e1c5c877b12522db3fb5adffd58b1aed0cc11150a 0
→b2e15d0d7a0c94e4e2ce0fe6e8691b9e451377f6e46e8045a86f7c4b5d4f0f23
```

Result:

```
02000000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024ec400000008000...f887d004b979b054c5fa98c28701230f4f5d4b7c6fa845806ee4f67713459e1b69e8e60fcee2e4940c7a0d5de1b201000000000000c350000000000000
```

6.1.26 createrawburn

The `createrawburn` RPC creates a raw unsigned burn transaction with a single input containing the burn amount and a single OP_RETURN output.

Parameter #1—input TXID

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Input transaction ID containing the asset to be burnt</td>
</tr>
</tbody>
</table>

Parameter #2—input transaction vout

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vout</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Input transaction vout</td>
</tr>
</tbody>
</table>

Parameter #3—the asset type

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asset</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Asset ID of asset to be burnt</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Amount of asset to burn</td>
</tr>
</tbody>
</table>

Result—the unsigned raw transaction in hex

Example

```
ocean-cli createrawissuance
→40ac4e02a64ea14190e96d66e1c5c877b12522db3fb5adffd58b1aed0cc11150a
→b2e15d0d7a0c94e4e2ce0fe6e8691b9e451377f6e46e8045a86f7c4b5d4f0f23 0.342100
```
Result:

```
0200000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff0000000000000000000000000000 ... 00000000000000000000000000000000000000000100000002dd231b0001000000000754dee2e4940c7a0d5de1b201000000000000c350000000000000
```

### 6.1.27 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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rawtxs</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Hex encodings of signed transactions</td>
</tr>
</tbody>
</table>

**Parameter #2—accept large fee**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowhighfees</td>
<td>Boolean</td>
<td>Optional (0 or 1)</td>
<td>Allow large fee</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>The raw transaction ID</td>
</tr>
<tr>
<td>allowed</td>
<td>boolean</td>
<td>The determination on whether the transaction is valid/allowed</td>
</tr>
<tr>
<td>reject-reason</td>
<td>String</td>
<td>The reason if the transaction is rejected</td>
</tr>
</tbody>
</table>

**Example**

```
ocean-cli testmempoolaccept
˓
→ 02000000000010a1511ccd0aeb158fddf5afbb32d52127b875c1c6e6de99041a14ea6024eac400000008000ffffffff0000000000000000000000000000 ... 00000000000000000000000000000000000000000100000002dd231b0001000000000754dee2e4940c7a0d5de1b201000000000000c350000000000000
```

Result:

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

### 6.1.28 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 the policy list and the value. Spending these outputs removes the addresses from the policy lists.

**Parameter #1—Inputs**
### inputs
- **Name**: array
- **Type**: array
- **Presence**: Required (exactly 1)
- **Description**: An array of objects, each one to be used as an input to the transaction

<table>
<thead>
<tr>
<th>→ Input</th>
<th>Object</th>
<th>Required(1 or more)</th>
<th>An object describing a particular input with fields listed below</th>
</tr>
</thead>
<tbody>
<tr>
<td>→→ txid</td>
<td>String (hex)</td>
<td>Required(exactly 1)</td>
<td>The transaction ID of the output to be spent in hex encoding</td>
</tr>
<tr>
<td>→→ vout</td>
<td>Integer</td>
<td>Required(exactly 1)</td>
<td>The output index number (vout) of the output to be spent</td>
</tr>
<tr>
<td>→→ sequence</td>
<td>Integer</td>
<td>Optional (0 or 1)</td>
<td>The sequence number to use for the input</td>
</tr>
</tbody>
</table>

### Parameter #2—Addresses to add to policy lists and control keys

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy output</td>
<td>array</td>
<td>Required (exactly 1)</td>
<td>An array of objects, each one is used to add an address to a policy list</td>
</tr>
<tr>
<td>→ Input</td>
<td>Object</td>
<td>Required (1 or more)</td>
<td>An object encoding a policy output, with fields listed below</td>
</tr>
<tr>
<td>→→ pubkey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>The public key of the policy authority wallet used to spend the output</td>
</tr>
<tr>
<td>→→ value</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>The amount of policy asset to be sent to the output</td>
</tr>
<tr>
<td>→→ address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>The address to be added to the policy list</td>
</tr>
</tbody>
</table>

### Parameter #3—locktime

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locktime</td>
<td>Integer</td>
<td>Required(exactly 1)</td>
<td>Indicates the earliest time (in block height or Unix epoch time) a transaction can be added to the block chain</td>
</tr>
</tbody>
</table>

### Parameter #4—policy asset

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy asset</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>The 32 byte asset ID of the policy asset of the list being updated</td>
</tr>
</tbody>
</table>

**Result**—the unsigned raw transaction in hex. If the transaction couldn’t be generated, result will be set to JSON ‘null’ and the JSON-RPC error field may contain an error message

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>The resulting unsigned raw transaction in serialized transaction format</td>
</tr>
</tbody>
</table>

**Example**

```bash
calendar-cli createrawpolicytx "" {
    "txid": "f3b75af773c3e38fd190f6c0943d311ce2dd8a26c7e7a9e600c58f8b21e53d4",
```

(continues on next page)
"vout": 1
}"

{
"pubkey": "03d5be1ca0b06b54f6a29a8e245fdf58698164538191c5b376d3b27e6d3229b81a",
"value": 10.0
"address": "1HPkc4to3GzVcEV8Le6sS4V5AXWQceH5kZ"
}

Result:

6.1.29 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ Input</td>
<td>Object</td>
<td>Required(1 or more)</td>
<td>An object describing an input transaction</td>
</tr>
<tr>
<td>→→ txid</td>
<td>String (hex)</td>
<td>Required(exactly 1)</td>
<td>The txid of the input transaction</td>
</tr>
<tr>
<td>→→ vout</td>
<td>Integer</td>
<td>Required(exactly 1)</td>
<td>The output index number (vout) of input transaction</td>
</tr>
</tbody>
</table>

Parameter #2—Output object with request details

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Object</td>
<td>Required(1 or more)</td>
<td>An object identifying an output</td>
</tr>
<tr>
<td>→ pubkey</td>
<td>string (hex)</td>
<td>Required (exactly 1)</td>
<td>Target request public key</td>
</tr>
<tr>
<td>→‘decayConst’</td>
<td>Number</td>
<td>Required (exactly 1)</td>
<td>Decay constant determining speed at which tickets value decreases</td>
</tr>
<tr>
<td>→ endBlock-Height’</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>End block request service period</td>
</tr>
<tr>
<td>→ fee</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Percentage of fees to be paid to guardnodes</td>
</tr>
<tr>
<td>→ genesisBlock-Hash</td>
<td>String</td>
<td>Required (exactly 1)</td>
<td>Hash of client chain genesis block</td>
</tr>
<tr>
<td>→ startBlock-Height'</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Start block request service period, end block of auction period</td>
</tr>
<tr>
<td>→ startPrice</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Starting price of ticket</td>
</tr>
<tr>
<td>→ tickets</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Number of tickets to issue</td>
</tr>
<tr>
<td>→ value</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>value of input transaction</td>
</tr>
</tbody>
</table>

Result—the unsigned raw transaction in hex. If the transaction couldn’t be generated, result will be set to JSON ‘null’ and the JSON-RPC error field may contain an error message.
### 6.1.30 `getrequests`

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

#### Parameter #1—the client genesis block hash

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genesishash</td>
<td>string</td>
<td>Optional</td>
<td>The genesis block hash to filter requests by</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>array</td>
<td>array of JSON objects of details for each active request</td>
</tr>
<tr>
<td>request</td>
<td>string (hex)</td>
<td>JSON objects containing details for each active request</td>
</tr>
<tr>
<td>genesisBlock</td>
<td>string</td>
<td>Client chain genesis block hash</td>
</tr>
<tr>
<td>numTickets</td>
<td>Integer</td>
<td>Number of tickets available</td>
</tr>
<tr>
<td>decayConst</td>
<td>Number</td>
<td>Decay constant determining speed at which tickets value</td>
</tr>
<tr>
<td>startPrice</td>
<td>Amount</td>
<td>Starting price of ticket</td>
</tr>
<tr>
<td>auctionPrice</td>
<td>Amount</td>
<td>Current price of tickets</td>
</tr>
<tr>
<td>feePercentage</td>
<td>Amount</td>
<td>Percentage of fees to be paid to guardnodes</td>
</tr>
<tr>
<td>endBlockHeight</td>
<td>Integer</td>
<td>End block request service period</td>
</tr>
<tr>
<td>txid</td>
<td>String (hex)</td>
<td>Request transaction ID</td>
</tr>
</tbody>
</table>

#### Example

```bash
call `ocean-cli getrequests`
```
ocean-cli getrequests
ocean-cli getrequests 123450e138b1014173844ee0e4d557ff8a2463b14fcaeb18f6a63aa7c7e1d05

Result:

```json
[
  {
    "genesisBlock": "123450e138b1014173844ee0e4d557ff8a2463b14fcaeb18f6a63aa7c7e1d05",
    "startBlockHeight": 105,
    "numTickets": 20,
    "decayConst": 2,
    "startPrice": 5.0,
    "auctionPrice": 4.8,
    "feePercentage": 5,
    "endBlockHeight": 350,
    "txid": "666450e138b1014173844ee0e4d557ff8a2463b14fcaeb18f6a63aa7c7e1d05"
  }
]
```

6.1.31 createrawbibidtx

The `createrawbidtx` RPC creates a raw bid transaction funded by given inputs. Bids must be in the domain asset.

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

**Parameter #2**—Output object with request details

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>object</td>
<td>Required (1 or more)</td>
<td>JSON objects identifying an output</td>
</tr>
<tr>
<td>→ pubkey</td>
<td>string (hex)</td>
<td>Required (exactly 1)</td>
<td>Target stake public key</td>
</tr>
<tr>
<td>→ value</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Staked value locked in target pubkey</td>
</tr>
<tr>
<td>→ change</td>
<td>Amount</td>
<td>Optional (0 or 1)</td>
<td>Change value of transaction</td>
</tr>
<tr>
<td>→ changeAddr</td>
<td>String (base58check)</td>
<td>Optional (0 or 1)</td>
<td>Change address of transaction</td>
</tr>
<tr>
<td>→ fee</td>
<td>Amount</td>
<td>Required (exactly 1)</td>
<td>Fee value of transaction</td>
</tr>
<tr>
<td>→ endBlockHeigt</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Service end block height</td>
</tr>
<tr>
<td>→ requestTxid</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Percentage of fees to be paid to guardnodes</td>
</tr>
<tr>
<td>→ feePubKey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Public key to receive fees on client chain</td>
</tr>
</tbody>
</table>

**Result**—the unsigned raw transaction in hex. If the transaction couldn’t be generated, result will be set to JSON ‘null‘ and the JSON-RPC error field may contain an error message

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>The resulting unsigned raw transaction in serialized transaction format</td>
</tr>
</tbody>
</table>

**Example**

```json
ocean-cli createrawbidtx ''{
  
  "txid": "43bd75af773ce38fd190f6c0943d311ce2dd8a26c7e7a9e600c58f8b21e53d4",
  
  "vout": 1,
}
'''
```

(continues on next page)
6.1.32 getrequestbids

The getrequestbids RPC returns all the active bids for a given request in the blockchain.

Parameter #1—the request transaction hash

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requesthash</td>
<td>string</td>
<td>Optional</td>
<td>Hash of the request transaction to return bids for</td>
</tr>
</tbody>
</table>

Result—an array of JSON objects containing details for each active request and its corresponding bids

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>array</td>
<td>array of JSON objects of details for each active request</td>
</tr>
<tr>
<td>→ request</td>
<td>string (hex)</td>
<td>JSON objects containing details for each active request</td>
</tr>
<tr>
<td>→ genesisBlock</td>
<td>string</td>
<td>Client chain genesis block hash</td>
</tr>
<tr>
<td>→ numTickets</td>
<td>Integer</td>
<td>Number of tickets available</td>
</tr>
<tr>
<td>→ decayConst</td>
<td>Number</td>
<td>Decay constant determining speed at which tickets value</td>
</tr>
<tr>
<td>→ startPrice</td>
<td>Amount</td>
<td>Starting price of ticket</td>
</tr>
<tr>
<td>→ auctionPrice</td>
<td>Amount</td>
<td>Current price of tickets</td>
</tr>
<tr>
<td>→ feePercentage</td>
<td>Amount</td>
<td>Percentage of fees to be paid to guardnodes</td>
</tr>
<tr>
<td>→ endBlockHeight</td>
<td>Integer</td>
<td>End block request service period</td>
</tr>
<tr>
<td>→ txid</td>
<td>String (hex)</td>
<td>Request transaction ID</td>
</tr>
<tr>
<td>→ bids</td>
<td>array</td>
<td>array of objects containing bids information</td>
</tr>
<tr>
<td>→ hash</td>
<td>String</td>
<td>hash of bid transaction</td>
</tr>
<tr>
<td>→ → feePubKey</td>
<td>String (hex)</td>
<td>Public key that receives fees on client service chain</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli getrequestbids
0200000000056473950498487ef62c10e46f14aec78dce956b02eb41f7e2cce8b6d56292db40100000000ffe
```

Result:
6.1.33 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check encoded contract tweaked address</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpubkey</td>
<td>String (hex)</td>
<td>Required (exactly 1)</td>
<td>Hex encoded base public key</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addtowhitelist 2dZhhVmJkXCaWUzPmh mwQ3gBj m2NJSnr vyz...
028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58
```

6.1.34 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

---

6.1. Quick reference 53
Parameter #2—the base (un-tweaked) compressed public keys that the p2sh was created with

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basepubkeys</td>
<td>array</td>
<td>Required (1 or more)</td>
<td>Hex encoded base public keys</td>
</tr>
</tbody>
</table>

Parameter #3—the n of Multisig

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmultisig</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Number of signatures required for multisig</td>
</tr>
</tbody>
</table>

Parameter #4—the Base58 KYC address

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kycaddress</td>
<td>string (base58check)</td>
<td>Optional (0 or 1)</td>
<td>Base58check encoded KYC address</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli addmultitowhitelist 2dZhhVmJkXCaWUzPmhwQ3gBjM2NJ3nrvyz
\[028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58,
\-028f9c608ded55e89aef8ade69b90612510dbd333c8d63cbe1072de9049731bb58\] 1
```

### 6.1.35 querywhitelist

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

**Parameter #1—the Base58check encoded address**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>string (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check encoded address</td>
</tr>
</tbody>
</table>

Result—TRUE of FALSE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>boolean</td>
<td>1 if the address is present, 0 otherwise</td>
</tr>
</tbody>
</table>

**Example**

```
ocean-cli querywhitelist 2dZhhVmJkXCaWUzPmhwQ3gBjM2NJ3nrvyz
```

Result:

```
1
```
6.1.36 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 checks 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 described in `dumpderivedkeys`.

Parameter #1—the filename to read in the list

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>string</td>
<td>required (exactly 1)</td>
<td>Name of file containing the list of contract tweaked address and base public keys</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

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

6.1.37 removefromwhitelist

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

Parameter #1—the Base58check encoded address

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>string (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check encoded address</td>
</tr>
</tbody>
</table>
```

Result—none if valid, errors returned if invalid inputs

Example

```
ocean-cli removefromwhitelist 2dZhhVmJkXCaWUzPmhwwQ3gBjM2NJSnrzyz
```

6.1.38 clearwhitelist

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

Parameters: none

Result: none

Example

```
ocean-cli clearwhitelist
```

6.1.39 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
6.1.40 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tweakedaddress</td>
<td>string (base58check)</td>
<td>Required</td>
<td>Base58check encoded contract tweaked p2sh address</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basepubkeys</td>
<td>array</td>
<td>Required (1 or more)</td>
<td>Hex encoded base public keys</td>
</tr>
</tbody>
</table>

Parameter #3—the n of Multisig

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmultisig</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Number of signatures required for multisig</td>
</tr>
</tbody>
</table>

Parameter #4—the fee asset type

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feeasset</td>
<td>String</td>
<td>Optional (0 or 1)</td>
<td>Type of asset for fee payment</td>
</tr>
</tbody>
</table>

Result—transaction hex if valid, errors returned if invalid inputs or wallet error occurs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>The resulting transaction id serialized transaction format</td>
</tr>
</tbody>
</table>

Example

```plaintext
ocean-cli sendaddmultitowhitelisttx 2dZhhVmJkXCaWUzPmhwQ3gBJm2NJSnrvyz
—is-028f9c608ded55e89ae8ade69b90612510dbd333c8d63cbe1072de9049731bb58,"
->028f9c608ded55e89ae8ade69b90612510dbd333c8d63cbe1072de9049731bb58] 1
```

6.1.41 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
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>naddresses</td>
<td>Integer</td>
<td>Required (exactly 1)</td>
<td>Number of addresses to register</td>
</tr>
</tbody>
</table>

Parameter #2—the fee asset type

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feeasset</td>
<td>String</td>
<td>Optional (0 or 1)</td>
<td>Type of asset for fee</td>
</tr>
</tbody>
</table>

Result—transaction hex if valid, errors returned if invalid inputs or wallet error

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>String (hex)</td>
<td>The resulting transaction id serialized transaction format</td>
</tr>
</tbody>
</table>

Example

```
ocean-cli sendaddtowhitelisttx 100
```

6.1.42 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address to add to freeze list</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

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

6.1.43 queryfreezelist

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

Parameter #1—the Base58check encoded address

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address</td>
</tr>
</tbody>
</table>

Result—TRUE or FALSE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Boolean</td>
<td>1 if the address is present, 0 otherwise</td>
</tr>
</tbody>
</table>

Example
ocean-cli queryfreezelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz

Result:
1

6.1.44 removefromfreezelist

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

Parameter #1—the Base58check encoded address

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address to remove from freeze list</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

ocean-cli removefromfreezelist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz

6.1.45 clearfreezelist

The clearfreezelist RPC clears the mempool whitelist of all addresses.

Parameters: none

Result: none

Example

ocean-cli clearfreezelist

6.1.46 addtoburnlist

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

Parameter #1—the Base58check address

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address to add to freeze list</td>
</tr>
</tbody>
</table>

Result—none if valid, errors returned if invalid inputs

Example

ocean-cli addtoburnlist 2dZhhVmJkXCaWUzPmhmwQ3gBJm2NJSnrvyz
### 6.1.47 queryburnlist

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

**Parameter #1—the Base58check encoded address**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address</td>
</tr>
</tbody>
</table>

**Result**—TRUE of FALSE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Boolean</td>
<td>1 is the address is present, 0 otherwise</td>
</tr>
</tbody>
</table>

**Example**

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

Result:

```
1
```

### 6.1.48 removefromburnlist

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

**Parameter #1—the Base58check encoded address**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Presence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>String (base58check)</td>
<td>Required (exactly 1)</td>
<td>Base58check address to remove from burn list</td>
</tr>
</tbody>
</table>

**Result**—none if valid, errors returned if invalid inputs

**Example**

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

### 6.1.49 clearburnlist

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

**Parameters:** none

**Result:** none

**Example**

```
ocean-cli clearburnlist
```
6.1.50 pkhwhitelist

Enables node mempool address whitelisting. With this option set all addresses are blacklisted by default and must be added to the whitelist in order to transact.

*Argument—TRUE or FALSE*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable whitelisting</td>
</tr>
</tbody>
</table>

*Example—The following examples enable whitelisting*

```plaintext
oceand -pkhwhitelist
```

*In ocean.conf: pkhwhitelist=1*

6.1.51 freezelist

Enables node mempool address freezelisting.

*Argument—TRUE or FALSE*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable freeze list</td>
</tr>
</tbody>
</table>

6.1.52 burnlist

Enables node mempool address burnlisting.

*Argument—TRUE or FALSE*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable burn list</td>
</tr>
</tbody>
</table>

6.1.53 issuanceblock

Enables blocking of invalid issuance transactions from mempool by checking that asset issuance transactions have an issuanceAsset input.

*Argument—TRUE or FALSE*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable</td>
</tr>
</tbody>
</table>

6.1.54 disablect

Disables confidential transactions and addresses.

*Argument—TRUE or FALSE*
<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to disable confidential transactions</td>
</tr>
</tbody>
</table>

### 6.1.55 embedcontract

Enable hash of chain’s contract to be embedded in block header and addresses.

**Argument**—TRUE or FALSE

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable</td>
</tr>
</tbody>
</table>

**Example**—The following examples will disable contract hash embedding

```plaintext
oceand -embedcontract=0
```

**In ocean.conf**: embedcontract=0

### 6.1.56 attestationhash

Embed the given attestation hash in the block header.

**Argument**—Attestation hash

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>null</td>
<td>256-bit hex encoding of attestation hash</td>
</tr>
</tbody>
</table>

**Example**—The following will set the contract hash for embedding

```plaintext
oceand -attestationhash=aa2364284941f08cceeaf49911858125256d61f1b728e544ead6423bf06ea1e15
```

**In ocean.conf**: attestationhash=aa2364284941f08cceeaf49911858125256d61f1b728e544ead6423bf06ea1e15

### 6.1.57 embedmapping

Enable asset mapping object embedding in block header.

**Argument**—TRUE or FALSE

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable</td>
</tr>
</tbody>
</table>

### 6.1.58 issuecontrolscript

Embed the given issuance controller script in the genesis block as a coinbase transaction in an op_return script. Does nothing if set after chain initialisation.

**Argument**—issuance control script
6.1.59 policycoins

The amount of policy coins created in the genesis block.

*Argument—Number of policy coins to create*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount</td>
<td>0</td>
<td>Amount of policy coins to create</td>
</tr>
</tbody>
</table>

6.1.60 initialfreecoinsdestination

The destination of the OP_TRUE initial freecoins created in the genesis block. This functionality is used primarily for testing.

*Argument—Address to hold created coins*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Destination of created freecoins</td>
</tr>
</tbody>
</table>

**Example**

```
oceand -initialfreecoinsdestination=76a91427e17844a9dff73ace482eae458105005ad672e488ac
```

In `ocean.conf`: initialfreecoinsdestination=76a91427e17844a9dff73ace482eae458105005ad672e488ac

6.1.61 freezelistcoinsdestination

The destination of the tokens for controlling the freezelist.

*Argument—Address to hold created freeze tokens*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Destination of freeze list policy tokens</td>
</tr>
</tbody>
</table>

6.1.62 burnlistcoinsdestination

The destination of the tokens for controlling the burnlist.

*Argument—Address to hold created burn tokens*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Destination of burn list policy tokens</td>
</tr>
</tbody>
</table>

---

Chapter 6. Ocean API reference
6.1.63 issuancecoinsdestination

The destination of the tokens for controlling issuances.

*Argument—Address to hold created issuance coins*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Destination of issuance policy tokens</td>
</tr>
</tbody>
</table>

6.1.64 permissioncoinsdestination

The destination of the policy tokens for permitting request creation.

*Argument—Address to hold created permitting request tokens*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Destination of permission policy tokens</td>
</tr>
</tbody>
</table>

6.1.65 mainchainrpchost

The rpc host address which the daemon will try to connect to validate peg-ins, if enabled. By default cookie authentication is attempted.

*Argument—main chain rpc host*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>String</td>
<td>host address</td>
</tr>
</tbody>
</table>

6.1.66 mainchainrpcport

The rpc port number which the daemon will try to connect to validate peg-ins, if enabled. By default cookie authentication is attempted.

*Argument—main chain rpc port*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>Integer</td>
<td>port number</td>
</tr>
</tbody>
</table>

6.1.67 validatepegin

Enable validation of all peg-in claims.

*Argument—TRUE or FALSE*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1 to enable validation of peg-ins</td>
</tr>
</tbody>
</table>
6.1.68 parentgenesisblockhash

Set parent genesis blockhash. Ethereum mainnet is default.

*Argument—parent block hash*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String (hex)</td>
<td>d4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb</td>
<td>Hash of desired parent block</td>
</tr>
</tbody>
</table>

6.1.69 parentcontract

Set parent ERC20 contract script. Enabling causes creation of a new chain with a different genesis block.

*Argument—ERC20 contract script*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String (hex)</td>
<td>076C97e1c869072eE22ff8c91978C99B4bcB02591</td>
<td>Hex encoding of ERC20 script</td>
</tr>
</tbody>
</table>

6.1.70 fedpegaddress

Set ETH address of federated peg. Here a new chain is created with a different genesis block.

*Argument—Ethereum address*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>null</td>
<td>Ethereum address of federated peg</td>
</tr>
</tbody>
</table>

6.1.71 peginconfirmationdepth

Set required depth of network for peg-in claims to be considered valid.

*Argument—block height*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block height</td>
<td>Integer</td>
<td>8</td>
<td>block height</td>
</tr>
</tbody>
</table>
CHAPTER 7

Integration guide

The Ocean platform is based on Bitcoin, and has a superset of functionality, both in the protocol and the RPC interface. This guide describes methods and instructions for the integration of an Ocean sidechain into services, products and exchanges. For detailed instructions on installing and configuring an Ocean sidechain node, please read the ‘Ocean Node Setup’ section of the documentation.

7.1 Address Types

Ocean platform sidechains support a superset of the Bitcoin scripting language, and all redeem scripts and scriptPubKey script valid on Bitcoin are also valid on Ocean platform sidechains. Ocean sidechains therefore support the standard address formats, including P2PKH, P2SH and Segwit P2SH-P2WPKH. The Base58check address and private key (WIF compressed) prefixes used by oceand and wallets can be customised for specific asset backed sidechains, however the following defaults are used:

chain=ocean_main (mainnet)

<table>
<thead>
<tr>
<th>Type</th>
<th>Pre-fix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PKH</td>
<td>0x00</td>
<td>1EN1hdE4wVdgmn6fWJQDmyUUwXNQiRXbmGx</td>
</tr>
<tr>
<td>P2SH</td>
<td>0x05</td>
<td>3F42dAiWVPx4sGMwRVtNQ6qsftgj24aJdh</td>
</tr>
<tr>
<td>WIF</td>
<td>0x80</td>
<td>Kyp3hfVDandgQtAbn79yt9g3AAbwL9xsAN4ixULqVPAuhdWneG2</td>
</tr>
</tbody>
</table>

chain=ocean_test (testnet)

<table>
<thead>
<tr>
<th>Type</th>
<th>Pre-fix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PKH</td>
<td>0xEB</td>
<td>2dnmm69GmmxXaxxRs3zVfwySpRuwgTBVsyt</td>
</tr>
<tr>
<td>P2SH</td>
<td>0x4B</td>
<td>XQiFymagC2PQ7c818sCgLstwjCikhBvgyH</td>
</tr>
<tr>
<td>WIF</td>
<td>0xEF</td>
<td>cQB3AaV51rKwaKdTArvHMCejfPU1bnFewCWXqNvrLc3bASgWmnGU</td>
</tr>
</tbody>
</table>

chain=gold_main (DGLD)
<table>
<thead>
<tr>
<th>Type</th>
<th>Pre-fix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PKH</td>
<td>0x26</td>
<td>GXCw7kZ1vMEyrZxoELstQEpqSYBrQchASF</td>
</tr>
<tr>
<td>P2SH</td>
<td>0x61</td>
<td>gG9XDA91ozag79Bug6Xh1dtFajPWPkA6FY</td>
</tr>
</tbody>
</table>
| WIF    | 0xB4    | TfxqS4DzgnGq5nr8YPdNh4wzykcXM6FdL8kqpaVGN

New addresses from the oceand can be generated using the getnewaddress RPC. P2SH-P2WPKH addresses can be generated by then supplying a wallet address to the addwitnessaddress. A P2SH multisig address can be generated using the createmultisig RPC with a supplied array of public keys.

### 7.2 Client wallet

The oceand client has an RPC wallet that is an extension of the bitcoind wallet. This is a BIP32 HD wallet that is generated on first use, with the wallet.dat file stored in the datadir/chain directory. The only significant difference is that the oceand wallet has functionality to handle multiple token types. For example, the wallet status accessed via the getwalletinfo RPC, which for example returns:

```json
{
  "walletversion": 130000,
  "balance": {
    "21c5b781892bfa1ac7267418195be0b5eb13d4c917fbd72dec9c9d8b26270": 32.0,
    "7e679401d07144ee6b3ee19f0cb02884ca10a40407264a30c51f9b8534c89f898": 144.3454
  },
  "unconfirmed_balance": {},
  "immature_balance": {},
  "txcount": 0,
  "keypoololdest": 1571756105,
  "keypoolsize": 100,
  "paytxfee": 0.00000000
}
```

Where this wallet holds two token types (or asset IDs): 21c5b781892bfa1ac7267418195be0b5eb13d4c917fbd72dec9c9d8b26270 and 7e679401d07144ee6b3ee19f0cb02884ca10a40407264a30c51f9b8534c89f898.

Sending one of these token types to an address can be performed using the sendtoaddress RPC as with bitcoind, but with the chosen asset ID as an additional argument. For example:

```
./oceand -datadir=path sendtoaddress GXCw7kZ1vMEyrZxoELstQEpqSYBrQchASF 16.0 '' ''
```

Will send 16.0 of asset 21c5b781892bfa1ac7267418195be0b5eb13d4c917fbd72dec9c9d8b26270 to address GXCw7kZ1vMEyrZxoELstQEpqSYBrQchASF.

If you don’t care about the specific asset ID being sent (i.e. if the different tokens are considered fungible) then the sendanytoaddress RPC can be used. e.g.

```
./oceand -datadir=path sendanytoaddress GXCw7kZ1vMEyrZxoELstQEpqSYBrQchASF 60.0
```

Will select any assets summing to the value of 60.0 to the specified address, with the coin selection optimized for keeping assets together.
7.3 Fees

The oceand wallet will add transaction fees based on the parameters (minrelaytxfee) specified in the ocean.conf configuration file and the transaction size. If fixedfee is set, then a fixed transaction fee is used, independent of the transaction size - this option can be used for sidechains that employ whitelisting.

The transaction fee in Ocean transactions is specified explicitly, by paying an amount to an output with an empty scriptPubKey. This means that in Ocean transactions, the inputs amounts of each asset must equal exactly the output amounts of each asset (i.e. all assets are conserved). This is in contrast to Bitcoin where the difference between the input and output amounts is taken as the transaction fee.

By default, the transaction fee can be paid in any token type (but only one, and with a single output).

7.4 Transactions

Ocean transactions have a similar serialization format to Bitcoin transactions but with some additional flags and 32-byte asset ID labels for each output. All cryptographic algorithms (i.e. SHA256, secp256k1), signature formats (DER encoding), scriptSig and scriptPubKey formats and SigHash types and flags are identical to Bitcoin.

Raw transactions can be created using the RPCs createrawtransaction (if each asset ID is sent to a separate address) or createrawtxoutputs (if the specification of individual outputs is required). These raw transactions can then be signed using signrawtransaction with either wallet or WIF keys supplied as arguments, and then broadcast with sendrawtransaction as with bitcoind.

For a given transaction ID, the deserialized transaction can be retrieved with the following RPC:

```
./oceand -datadir=path getrawtransaction
-lc606ab1490c4111bc0cc79af559b3566eb104f0f81f783d63911087d789fae true
```

This will return the JSON encoded deserialized transaction. e.g.

```
|
| "hex": "0200000000013b60b9276d9e1d6fbeb9d580057f40a63a3d1209b663960e28c96a2898f601ca0100000054029802b10001", |
| "txid": "1c606ab1490c4111bc0cc79af559b3566eb104f0f81f783d63911087d789fae", |
| "hash": "1c606ab1490c4111bc0cc79af559b3566eb104f0f81f783d63911087d789fae", |
| "withash": "ab3393f4fb64685a4de97ac89c6903d6a0ba107c87fff33c9a981a7590bae48e", |
| "size": 341, |
| "vsize": 341, |
| "version": 2, |
| "locktime": 111, |
| "vin": [ |
| "txid": "ca01f698286ac9280e9663b609123d3a6407f0580d5b9be6f1d9e6d27b963b", |
| "vout": 1, |
| "scriptSig": { |
| "asm": "3045022100a3b1a25644a671c4cd6248f83fb3ca896778f7e68cade4539816090ba67c0c1802200735eab82ba12c3093a2f9a07437dada1a60651ef194031ffa01bba65082a593d672895949a4c90451", |
| "hex": "483045022100a3b1a25644a671c4cd6248f83fb3ca896778f7e68cade4539816090ba67c0c1802200735eab82ba12c3093a2f9a07437dada1a60651ef194031ffa01bba65082a593d672895949a4c90451", |
| "is_pEGIN": false, |
```

(continues on next page)
This is similar to a deserialized Bitcoin transaction, except that each output also has an asset field for the asset ID.

### 7.5 Serialization

CommerceBlock maintains several libraries for wallet tools that include transaction building, serialization and signing for different platforms.

These include:
- oceanjs A Javascript library for node.js and browsers.
- oceanj A Java library for APK.

The above raw transaction (hex) serialization is broken down with the following labeling:

```
02000000  # version
00  #flag
01  # number of vins
3b60b92769e1d6fbeb9d580057f40a63a3d1209b663960e28c96a2898f601ca  #vin[0] txid
01000000  #vin[0] previous index (vout)
6b        # # scriptSig
483045022100a3b1a25644a671c4cd6248f83fb3ca896778f7e68cade4539816090ba67c0c1802200735eab82b2a12c30958   ➔ # scriptSig
feffffff  # sequence
03  # number of outputs
01 230f4f5d4b7c6fa845806ee4f67713459e1b69e8e60fcee2e4940c7a0d5de1b2  #output[0] asset ID
00  # value flag
01 3b60b92769e1d6fbeb9d580057f40a63a3d1209b663960e28c96a2898f601ca  #vin[0] txid
00  # scriptPubKey
01 230f4f5d4b7c6fa845806ee4f67713459e1b69e8e60fcee2e4940c7a0d5de1b2  #output[1] asset ID
00  # value flag
00009184e72855c  #output[0] value
00  # nonce
19 76a9146668f07ad957d44271fc563c60fddda292bab9c4088ac  #scriptPubKey
01 230f4f5d4b7c6fa845806ee4f67713459e1b69e8e60fcee2e4940c7a0d5de1b2  #output[1] asset ID
00  # value flag
00001b48eb57e000  #output[1] value
00  # nonce
19 76a9146668f07ad957d44271fc563c60fddda292bab9c4088ac  #scriptPubKey
01 230f4f5d4b7c6fa845806ee4f67713459e1b69e8e60fcee2e4940c7a0d5de1b2  #output[2] asset ID
01 0000000000001aa4  #output[2] value (tx fee)
00  # nonce
00  #scriptPubKey (empty for tx fee output)
6f000000  #locktime
```

7.5. Serialization
The Ocean Wallet is a lightweight wallet client that can hold assets on Ocean platform sidechains and which manages user private keys, token balances and registration/onboarding. It is built from the popular Electrum Bitcoin wallet and incorporates all of Electrum’s main features, including multisig functionality and hardware wallet integration. It is built entirely from Python, and uses PyQt libraries for the desktop GUI and Kivy libraries for the Android GUI.

The Ocean Wallet performs independent verification of transaction confirmation in the sidechain (and on Bitcoin) using the Simple Payment Verification (SPV) protocol. SPV is a technique described in Satoshi Nakamoto’s whitepaper that allows a client to verify that a transaction is included in a blockchain, without downloading the entire blockchain. An SPV client only needs download the block headers, which are much smaller than the full blocks. To verify that a transaction is in a block, a SPV wallet requests a proof of inclusion, in the form of a Merkle branch. This offers much more security than web wallets, because the wallet does not need to trust servers to confirm payments.

In addition to client-side SPV transaction confirmation in the Ocean sidechain where an asset is issued and transacted, the Ocean Wallet also connects to the Bitcoin network (via Electrum servers) to trustlessly verify the unique single history of an Ocean sidechain against Bitcoin’s global Proof-of-Work. With this option enabled, the Ocean Wallet also downloads the Bitcoin blockchain headers and independently verifies SPV proofs of the Mainstay commitments.

The Ocean Wallet can be downloaded as Windows (.exe) or OSX (.dmg) binaries from the Github releases page. It can also be compiled from source or run in pure Python from the Ocean Wallet Github repository.

8.1 Configuration

The Ocean Wallet is configured to connect to and verify specific Ocean sidechains, via a configuration file (chain_config) which is imported when the Ocean Wallet is first launched. This configuration file contains the genesis hash of the sidechain along with block verification scripts, Mainstay staychain verification pathways and Ocean Server connections. In addition, the configuration contains asset and chain specific customisations, including fee rates, token units, token names and mapping information.
8.2 Wallet servers

The Ocean Wallet is configured to connect to Ocean Servers, which deliver the sidechain blockheaders and transaction confirmation proofs (SPV) via the Stratum protocol. The Ocean Server is forked from the ElectrumX server used to operate the Electrum network, and is designed for high-performance and scaling, providing a large number of wallets with transactions, headers and proofs. The Ocean Server connects to an Ocean sidechain full node for a specific chain, and requires a separate chain-specific configuration (which requires the sidechain genesis hash and address formats).

Ocean Servers (configured for a specific sidechain) must be run alongside Ocean platform nodes to run a wallet service.
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. 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 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: proof a single unforked version of the federated sidechain is provided by Bitcoin’s proof-of-work.

9.1 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 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 accommodate 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 OpenTimestamps 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 and provenance: any proof-of-existence (i.e. a timestamp) does not act as a proof that anything else (e.g. an alternative ordering of transactions or version of a document) does not also exist.
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 and provenance 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 creating a cryptographic proof of immutable sequence (PoIS) by initiating a fan-in-only transaction staychain within the Bitcoin blockchain that is uniquely committed to the genesis block of a sidechain, as described in the next section. The protocol does not employ OP_RETURN outputs, providing additional privacy, censorship resistance and efficiency.
The MainStay protocol employs the underlying concept of a staychain of linked transactions within the Bitcoin blockchain, where all transactions in the staychain conform to having only 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 committing 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 - however it will always remain fail secure (i.e. previously reinforced transactions are provably unique without trust).
10.1 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.

10.1.1 Initialisation

The initial step in the protocol is the creation of the base transaction $\text{TxID}[0]$, 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: $\text{Addr}[0]$ with a derivation path of $m/0$.

2. The signing entity then creates a transaction (the base transaction $\text{BaseTx}$) paying an amount of BTC (to cover at least initial transaction fees) to the base address $\text{Addr}[0]$ 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: $\text{TxID}[0]$.

4. The sidechain is then configured and linked to the Bitcoin staychain. The pointer $\text{TxID}[0]$ is embedded directly in the genesis block of the sidechain as metadata in a defined location, along with the $xpub$. 
10.1.2 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 \( \text{blockhash}[j] \).
2. The 32 byte \( \text{blockhash}[j] \) is then split into 16 2-byte parts, which are then converted into an array of 16 integers \( \text{bhint}[16] \).
3. A BIP32 derivation path (the commitment path \( \text{path}[j] \)) is formed from this integer array sequence, and prepended with \( m/0 \).
4. The commitment address \( \text{Addr}[j] \) is then derived from the xpub with \( \text{path}[j] \).

For example:

\[
\begin{align*}
\text{blockhash}_j &= 310057788c6073640dc222466d003411cd5c1cc0bf2803fc6ebbfae03ceb4451 \\
\text{path}_j &= m/0/12544/22392/35936/29540/3522/8774/27904/13329/52572/7360/48936/1020/\rightarrow 28347/64224/15595/17489
\end{align*}
\]

5. The signing entity then creates a Bitcoin transaction with one input spending the single output of transaction \( \text{TxID}[j-1] \) (initially the base transaction when \( j = 1 \)) and paying to a single P2PKH output with address \( \text{Addr}[j] \). The transaction is then signed using the private key derived from xpriv with \( \text{path}[j-1] \). The valid transaction is then broadcast to the Bitcoin network. Once it is confirmed in a block, it is referenced by transaction ID: \( \text{TxID}[j] \).
10.1.3 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 \( \text{TxID}[0] \) is retrieved from the sidechain genesis block along with the master xpub.
2. \( \text{TxID}[0] \) is retrieved from the Bitcoin blockchain.
3. The staychain is tracked until the unspent tip \( \text{TxID}[t] \), confirming each component transaction consists of only a single output:

\[
\text{TxID}[0] \rightarrow \text{TxID}[1] \rightarrow \text{TxID}[2] \rightarrow \text{TxID}[3] \rightarrow \ldots \rightarrow \text{TxID}[t]
\]
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: \( w \leftarrow w 1 \) 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.

10.1.4 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 \texttt{SIGHASH\_ANYONE\_CAN\_PAY} inputs. The base transaction will always define the commitment sequence through to the tip.

10.2 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 \( x\text{priv} \) 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.

10.2.1 Initialisation

1. Each signing node \( i \) where \( i = 1, \ldots, m \) generates a master extended private key \( x\text{priv}[i] \) and corresponding extended public \( x\text{pub}[i] \).

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 \( x\text{pub}[i] \) 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 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].

10.2.2 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: \( \text{OP}_n \) pubkey[1] pubkey[2] ... pubkey[m] \( \text{OP}_n \text{OP}_m \text{OP}_\text{CHECKMULTISIG} \)

10.2.3 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:

\[
\text{TxID}[0] \rightarrow \text{TxID}[1] \rightarrow \text{TxID}[2] \rightarrow \text{TxID}[3] \rightarrow \ldots \rightarrow \text{TxID}[t]
\]

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 \( \text{Addr}[w] \) equals \( \text{Addr}[t] \), block \( w \) on the sidechain (and all below it) are confirmed as reinforced.

8. If not true, the sidechain block height is decremented: \( w \leftarrow w + 1 \) and the check repeated.
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.

### 11.1 Overview

The primary purpose of the Mainstay scheme is to provide a cryptographic Proof of Immutable Sequence (PoIS) for a succession of changing states of some arbitrary system or process - i.e. cryptographic proof that the sequence of states has only a single, linked, verifiable history that cannot be altered (or double-spent) without trusting any authority. This PoIS is obtained via the trustless immutability and uniqueness inherent to the Bitcoin blockchain, where proof-of-work (via the permissionless mining of blocks to extend the chain) creates a practically irreversible, incorruptible and globally unique 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 and unique 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 *uniqueness* for sequential commitments in sequential Merkle Trees anchored to the successive transactions of the staychain, only one simple additional rule *must* be followed: each commitment from a particular sequence of states must always be be verifiably committed to the *same* leave position within the Merkle tree. If the commitment is always validated in the same position, then the sequence is as immutable and unique (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 Tree (CMT) 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 (referred 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.

### 11.2 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:

\[
N(\text{Parent}) = \text{SHA256}(N(L) || N(R))
\]

*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 and uniqueness 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 leaves (slots) and the depth of the tree is increased (increasing the depth of the tree will simply increase the size of the proof path).

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.

### 11.2.1 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) or the user has yet to submit the first commitment, the corresponding leaf commitment is set to zero. Once the root of the current updated tree (merkle_root) 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 merkle_root.

The slot-proof for a specific slotid provides cryptographic proof that a particular commitment was committed to a specified staychain transaction (identified by the transaction ID TxID) at that specific slot position.

Example slot-proof for a commitment in slotid = 1:

```json
{
    "txid": "38fa2c6e103673925a9985a5a6cbb6fd0bf1677c5c88e27a9e4b0229197b13",
    "commitment": "d235db29356bb02f37e16712c44a724282fd81134fbd6a61407b3009755a9e",
    "merkle_root": "f46a58a0cc796fade0c7854f169eb86a66797ac493ea35f28d0e35efee62399b",
    "ops": [
        {
            "append": false,
            "ts": 16777219
        }
    ]
}
```

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

### 11.3 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 can optionally provide a public key for authenticating a submitted commitment. In addition, the client will be provided with API access details and an access token to securely submit the commitment.

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

On the initiation of a connection, the client identifier (and optionally the client public key) is added to the active slot list (ASL) in the position corresponding to slotid. The connector service API then receives authenticated commitments from the client and if required, signatures are verified using the client public key. If the API token is valid, and commitment signatures are valid then the commitment is added to the CMT at the slotid position. The connector server updates the cached CMT root (merkle_root) each time a new slot commitment is received and verified. New verified commitments arriving for a particular slot overwrite the previous commitment.

At intervals determined by the staychain attestation frequency, the commitment server then commitments the merkle_root 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 (i.e. mined into a Bitcoin block), the service then generates slot-proofs for each of the active slots. These slot-proofs are then available to retrieve by the clients via the connector service API. The status of any commitment (i.e. whether it is pending or confirmed) can be checked at any time via the service API.

### 11.4 Proof of Immutable Sequence

Clients can retrieve slot-proofs from the Mainstay service API as they become available, or they can retrieve previous state slot proofs, or any proof sequence for a specified slot. A proof of immutable sequence (PoIS) will consist of one or more slot proofs depending on the application and the state commitments that have been made to the slot. In general, for proving a single history of a sequence of state changes, this will consist of an array of slot-proofs: a slot-proof sequence (SPS). The SPS will enable a trustless proof that only a single sequence of commitments have been made to the specified slot, that can be verified by anyone.
Formation of a slot-proof sequence from a series of consecutive state commitments.

11.5 Commitment frequency and fee policy

The service agreement with individual slot clients will specify the target staychain commitment and 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 $c_{\text{target}}$ (e.g. 1 hour) and the connector server will generate and broadcast a new staychain transaction containing the CMR every $c_{\text{target}}$ 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 $\text{maxfee}$ (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 $\text{maxfee}$ being insufficient) then the staychain transaction (updated with the latest CMR) is re-broadcast with an additional $\text{maxfee}$ for the next 1 hour period (i.e. the fee will now be 2x $\text{maxfee}$) using the replace-by-fee (RBF). This will then be repeated each $c_{\text{target}}$ until the transaction is confirmed.

The value of $\text{maxfee}$ may be increased and $c_{\text{target}}$ decreased as more clients join the service, increasing the reliability and regularity of proofs.
11.6 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).
The Mainstay connector service includes a public API to both perform attestations of state and retrieve slot proofs and sequence proofs (PoIS). To commit data to a slot requires an API token, which is provided after sign-up on mainstay.xyz an allocation of the slot position (additionally a signature can be required with the commitment, with the user public key supplied at sign-up).

12.1 REST framework structure

```python
response = json response object
response['error'] : json response error field
timestamp : timestamp in ms
allowance : time taken to respond in ns
```

12.2 Public Endpoints

12.2.1 Index

API index page.

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

response:

```json
{
  "response": "Mainstay-API-v1",
  "timestamp": 1548329067489,
  "allowance":
```

(continues on next page)
12.2.2 Latest Attestation

Provide information on latest Merkle root commitment to the staychain.

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

response:

```json
{
  "response":
  {
    "merkle_root": "f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28d7e35efee62399b",
    "txid": "38fa2c6e103673925aaec5e5a5adbcb6fd0bf1677c5c88e27a9e94b0229197b13"
  },
  "timestamp": 154839116363,
  "allowance":
  {
    "cost": 1796883
  }
}
```

12.2.3 Latest Commitment

Provide information on latest commitment for a specific slot position.

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

response:

```json
{
  "response":
  {
    "commitment": "d235db29356bb02f37e16712c4d34a724228f8d81134fcbfda61407b3009755a9e",
    "merkle_root": "f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28d7e35efee62399b",
    "txid": "38fa2c6e103673925aaec5e5a5adbcb6fd0bf1677c5c88e27a9e94b0229197b13"
  },
  "timestamp": 154839116363,
  "allowance":
  {
    "cost": 3119659
  }
}
```

12.2.4 Commitment

Fetch commitment information for a specific slot position and merkle_root.
request: https://mainstay.xyz/api/v1/commitment?position=3&merkle_root=f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28d8e35efee62399b
response:

```
{
  "response": {
    "commitment": "d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e",
    "merkle_root": "f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28d8e35efee62399b",
    "timestamp": 1548329204516,
    "allowance": {
      "cost": 1484074
    }
  }
}
```

### 12.2.5 Commitment Latest Proof

Fetch latest commitment proof for a specific slot position.

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

**response:**

```
{
  "response": {
    "txid": "38fa2c6e103673925a9c5e5aaddcb6fd0bf16775c88e27a9e4b0229197b13",
    "commitment": "d235db29356bb02f37e16712c4d34a724282fd81134fbfda61407b3009755a9e",
    "merkle_root": "f46a58a0cc796fade0c7854f169eb86a06797ac493ea35f28d8e35efee62399b",
    "timestamp": 1548330374527,
    "allowance": {
      "cost": 19732506
    }
    "ops": [
      {
        "append": false,
        "commitment": "5309053b9d4db8f86d2c7ec164645bdff1669111280e49e04c036c323b58f4709",
      },
      {
        "append": false,
        "commitment": "213e122a8e314a94f111dd8dc797814660b680f7258f1d95dec56318eabd7c",
      },
      {
        "append": true,
        "commitment": "406ab5d975ae922753fad4db83c3716ed4d21c6a0191f8336c76000962f63ba"
      }
    ]
  }
}
```
12.2.6 Commitment Verify

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

request:  
`https://mainstay.xyz/api/v1/commitment/verify?position=1&commitment=5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451e94e053`

response:

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

12.2.7 Commitment Proof

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

request:  
`https://mainstay.xyz/api/v1/commitment/proof?position=1&merkle_root=f46a58a0cc796fad0c7854f169eb86a06797ae493ea35f28dbe35efee62399b`

response:

```json
{
    "response": {
        "merkle_root": "f46a58a0cc796fad0c7854f169eb86a06797ae493ea35f28dbe35efee62399b",
        "commitment": "5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451e94e053",
        "ops": [
            {
                "append": false,
                "commitment": "21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee869830d617e4f519c"
            },
            {
                "append": true,
                "commitment": "622d1b5efe11e9031f1b25aac11587e0ff81a37e9565ded16ee82bbcc0c2fc1"
            },
            {
                "append": true,
                "commitment": "406ab5d975ae922753fad4db3c3716ed4d2d1c6a0191f8336c7600962f63ba"
            }
        ],
        "timestamp": 1548330450896,
        "allowance": {
```
12.2.8 Commitment Data

Get staychain information on a specific commitment.

**request:** https://mainstay.xyz/api/v1/commitment/commitment?commitment=5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463

**response:**

```json
{
    "response": {
        "attestation": {
            "merkle_root": "f46a58a0cc796f0ade0c7854f169eb86a06797ac493ea35f28d6e35e62399b",
            "txid": "38fa2c6e103673925aaec50e5aadd8b60bf0b1677c5c88e27a9e4b0229197b13",
            "confirmed": true,
            "inserted_at": "16:06:41 23/01/19"
        },
        "merkleproof": {
            "position": 1,
            "merkle_root": "f46a58a0cc796f0ade0c7854f169eb86a06797ac493ea35f28d6e35e62399b",
            "commitment": "5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463",
            "ops": [
                { "append": false, "commitment": "21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee0e9830d617e4f519c" },
                { "append": true, "commitment": "622d1b5e1e9031fb25aac11587e0ff81a37e9565de166ee8e02b0c02fc1" },
                { "append": true, "commitment": "406ab5d975ae922753fad4db83c3716ed4d1c6a0191f8336c76000962f63ba" }
            ]
        }
    },
    "timestamp": 1548330505898,
    "allowance": {
        "cost": 60414043
    }
}
```
12.2.9 Merle Tree

Get information on the commitments to a Merkle tree.

request: https://mainstay.xyz/api/v1/merkleroot?merkle_root=f46a58a0cc796fde0c7854f169eb86a06797ac493ea35f28dbe35efee62399b

response:

```json
{
    "response":{
        "attestation":{
            "merkle_root": "f46a58a0cc796fde0c7854f169eb86a06797ac493ea35f28dbe35efee62399b",
            "txid": "38fa2c6e103673925aaec50e5aadcb6fd0bf1677c5c88e27a9e4b0229197b13",
            "confirmed": true,
            "inserted_at": "16:06:41 23/01/19"
        },
        "merkle_commitment": [
            {
                "position": 0,
                "commitment": "21b0a66806bdc99ac4f2e697d05cb17c757ae10deb851ee869830d617e4f519c"
            },
            {
                "position": 1,
                "commitment": "5555c2bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053"
            },
            {
                "position": 2,
                "commitment": "5309053b9d4db8f86d2c7ec164645bdf166911280e49e04c9436c323b584709"
            },
            {
                "position": 3,
                "commitment": "d235db29356bb02f37e16712c4d34a724282fd81134f8f8a61407b3009755a9e"
            },
            {
                "position": 4,
                "commitment": "9b07569d4fd42ae3a19c0803b7401443e0275feb728e8103330d7d8615e9cb62"
            }
        ],
        "timestamp": 1548330553639,
        "allowance":{
            "cost": 3318936
        }
    }
}
```

12.2.10 Slot Position

Get information on a client slot position.
request: https://mainstay.xyz/api/v1/position?position=1

response:

```json
{
    "response": {
        "position": [
            {
                "position": 1,
                "merkle_root": "300ab292905c67631e46e6d014be286fe1bb6dc550ae2df8348fcb1cc21011",
                "commitment": "5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
                "ops": [
                    {
                        "append": false,
                        "commitment": "2851174cf04f206e6dfdf78a9208c90a324f3ea5e97ee5b0629d35b5a853fbcfc"
                    },
                    {
                        "append": true,
                        "commitment": "622d1b5e1fe1e9301fb1b25aac11587e0ff81a37e9565ded16ee82b60c2fcl"
                    },
                    {
                        "append": true,
                        "commitment": "406ab5d975ae922753fad4db83c3716ed4d2dc6a0191f8336c7600962f63ba"
                    }
                ]
            }
        ,
        {
            "position": 1,
            "merkle_root": "2522e16722cfb1b29d01bbbe6babe5f4eb7dd69b8bf8a00f9110328eebf4e3e",
            "commitment": "5555c29bc4ac63ad3aa4377d82d40460440a67f6249b463453ca6b451c94e053",
            "ops": [
                {
                    "append": false,
                    "commitment": "586f199e25d902706e0ebf24e2720e62f3f4343abd7b2ddc2fabc155f359a3f"
                },
                {
                    "append": true,
                    "commitment": "622d1b5e1fe1e9301fb1b25aac11587e0ff81a37e9565ded16ee82b60c2fcl"
                },
                {
                    "append": true,
                    "commitment": "406ab5d975ae922753fad4db83c3716ed4d2dc6a0191f8336c7600962f63ba"
                }
            ]
        }
    }
    ,
    "timestamp": 1548330579389,
    "allowance":
}

(continues on next page)
```
12.2.11 Attestation

Get information on an attestation.

**request:** https://mainstay.xyz/api/v1/attestation?txid=38fa2c6e103673925aaec50e5aadcb6fd0bf1677c5c88e27a9e4b0229197b13

**response:**

```json
{
   "response": {
      "attestation": {
         "merkle_root": "f46a58a0cc796fa0e0c7854f169eb86a06797ac493ea35f28d5e35e6fe62399b",
         "txid": "38fa2c6e103673925aaec50e5aadcb6fd0bf1677c5c88e27a9e4b0229197b13",
         "confirmed": true,
         "inserted_at": "16:06:41 23/01/19"
      },
      "attestationInfo": {
         "txid": "86b372fb70e0935bfff4d6ba112e78cb9a3201ca15251dcd7db7cbf135b342b5",
         "amount": 149.9999155,
         "blockhash": "3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14",
         "time": 1548253554
      }
   },
   "timestamp": 1548330644403,
   "allowance": {
      "cost": 7959634
   }
}
```

12.2.12 Block

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

**request:** https://mainstay.xyz/api/v1/blockhash?hash=3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14

**response:**

```json
{
   "response": {
      "blockhash": {
         "txid": "86b372fb70e0935bfff4d6ba112e78cb9a3201ca15251dcd7db7cbf135b342b5",
         "merkle_root": "f46a58a0cc796fa0e0c7854f169eb86a06797ac493ea35f28d5e35e6fe62399b",
         "amount": 149.9999155,
         "blockhash": "3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14",
         "time": 1548253554
      }
   },
   "timestamp": 1548330644403,
   "allowance": {
      "cost": 7959634
   }
}
```
"amount": 149.9999155,
"blockhash": "3c50145441751dfb8f01cd05f21a24d0763005334667daa734bbf4147eeabe14",
"time": "14:25:54 23/01/19"
},
"timestamp": 1548330671498,
"allowance": {
  "cost": 1543490
}
}

12.3 Authenticated Endpoints

12.3.1 Commitment Send

Node.js example

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

const url = "https://mainstay.xyz/api/v1";
const route = '/commitment/send'
const pubKey = '1CsSceq9GWhmozaky3DGa24ER6qRDgibf';
const privKey = 'bac52bbda2194e7ea1cd3da6585b66d28f1a7a3683eca9af4ba6373d323d24f';
const commitment = 'F0111111111111111111111111111110F';

let keyPair = ec.keyFromPrivate('97ddae0f3a25b92268175400149d65d6887b9cefasf28ea2c078e05edc15a3c0a');
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
    }
};
```

(continues on next page)
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":
  "eyJwb3NpdGlvbiI6MCwiY29tbWl0bWVudCI6IkYwMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMTExMEYifQ==",
  "X-MAINSTAY-SIGNATURE":
  "IjBqe50XtfZbQ1b0jr+JltswSPfZlWwZugXcpYbwYmpRl1+htq5b7wLYY9RtQ6Bw9Ym5dw0vMNRaDwR8pkea2Y=
  ",
  "X-MAINSTAY-SIGNATURE":
  "IjBqe50XtfZbQ1b0jr+JltswSPfZlWwZugXcpYbwYmpRl1+htq5b7wLYY9RtQ6Bw9Ym5dw0vMNRaDwR8pkea2Y=
"
' http://localhost:9000/api/v1/commitment/send
```

response

```
{"response":"feedback","timestamp":1541761540171,"allowance":{"cost":4832691}}
```
Mainstay service client

The Mainstay service client is a pure Python command-line tool used to manage interaction with the `mainstay.xyz` API and proof service. The tool can be used to perform state attestations, retrieve and collate proofs and to verify proofs of immutable sequence via connection to a Bitcoin full node or Bitcoin block-explorer API.

### 13.1 Requirements

**Python3**

To verify proofs against the Bitcoin blockchain, RPC or HTTP connection details to a `bitcoind` node or trusted block explorer must be provided.

### 13.2 Installation

Via PyPi:

```
pip3 install pymainstay
```

or directly from source (downloaded from Github):

```
python3 setup.py install
```

### 13.3 Usage

The Mainstay client interface (`msc`) can be used to fetch and verify proof sequences, synchronize and verify the immutability and uniqueness of sidechains and state sequences, perform authenticated data commitments and attestations, and generate and manage `mainstay.xyz` authentication keys. The interface is used via commands to perform different operations with specified arguments. The commands available can be listed with the `--help` argument:
usage: msc [-h] [-q] [-v]  
  (attest,a,fetch,f,verify,v,sync,s,config,c,keygen,k,info,i) ...

Mainstay client

optional arguments:
  -h, --help            show this help message and exit
  -q, --quiet           Be more quiet.
  -v, --verbose         Be more verbose. Both -v and -q may be used multiple
times.

Commands:
  Mainstay operations are performed via commands:

  (attest,a,fetch,f,verify,v,sync,s,config,c,keygen,k,info,i)
  attest (a)    Commit data to a Mainstay slot
  fetch (f)     Fetch proofs from the Mainstay service
  verify (v)    Verify Mainstay proofs against the Bitcoin blockchain
  sync (s)      Syncronise sidechain to Bitcoin via a sequence proof
  config (c)    Set configuration
  keygen (k)    Generate signing keys for attestations
  info (i)      Mainstay service status information

For each command, the possible arguments can be again listed with the --help flag. E.g.

msc attest -h

usage: msc attest [-h] [-f FILENAME | -c COMMITMENT | -g GIT | -d DIRECTORY]
  [-s SLOT] [--url SERVICE_URL] [-t API_TOKEN] [-k PRIVKEY]

optional arguments:
  -h, --help            show this help message and exit
  -f FILENAME, --file FILENAME
                        Attest the SHA256 hash of the specified file.
  -c COMMITMENT, --commit COMMITMENT
                        Hex string of the 32 bytes commitment.
  -g GIT, --git GIT     Attest the HEAD of the specified Git repository. If 0
                        use stored path.
  -d DIRECTORY, --dir DIRECTORY
                        Attest the state of the specified sequence directory.
  -s SLOT, --slot SLOT  Specify the slot position index.
  --url SERVICE_URL     URL for the Mainstay connector service. Default:
                        https://mainstay.xyz
  -t API_TOKEN, --token API_TOKEN
                        API token for the specified slot position.
  -k PRIVKEY, --privkey PRIVKEY
                        Private key for signing the commitment.

13.3.1 Configuration

The client can be used in a stateless fashion, with all configuration supplied via the command-line options, however
a configuration file (config.json) can be used, which is located in the application data directory. The current
configuration, and the location of the application data directory on a particular system, can be retrieved as follows:

msc config
All configuration is set via the same command. For connection to a particular slot, the slot position (-s), the API token (-t) and the authentication key (-k) can be set initially, so they do not have to be specified as arguments for each subsequent call.

### 13.3.2 Attestation

To perform commitments to a specified `mainstay.xyz` slot requires an API token that will have been provided on initialization of the slot. In addition, if a public key was specified on initialization, the commitment must be signed by the corresponding private key. The signature is computed by the client if the private key is provided (or is set in the config).

The client will send 32 byte commitment supplied as an argument (-c) to the specified slot, or the SHA256 hash of a specified file path (-f). For example:

```
msc attest -c 4db3dbb10b33d94389446982f022ee55be8eaefa7d8f40046054a693f23a1c85 -s 2
```

The client will return whether the commitment has been recieved by the `mainstay.xyz` successfully.

To attest the state of a file sequence directory, supply the directory path:

```
msc attest -d /Users/username/directory/ -s 2
```

To attest the latest state of a Git repository, supply the repository path:

```
msc attest -g /Users/username/gitrepo/ -s 2
```

### 13.3.3 Proof retrieval

The client can retrieve, store and update sequence proofs for a specified or configured slot position with the `fetch` command. This requires no token or authentication, as the proofs are publicly accessible. All retrieved sequence proofs are stored locally in the application data directory (the location of directory can be found with the `config` command), and can also optionally be saved to a specified file (-f) or printed to standard output (-o).

To retrieve the full sequence proof for a specified slot from when it was initialised, supply the argument `-i 0`. E.g. (for slot 2),

```
msc fetch -i 0 -s 2
```

This sequence proof will then be saved to a file named `slot_2_sequence.msp` in the application data directory.

To retrieve the sequence proof to a specific staychain transaction ID (e.g. `9eeccf2e6ca6f7257a379debccfb3e822df8658d03c95ec47fbd2267d218f03d`):

```
msc fetch -i 9eeccf2e6ca6f7257a379debccfb3e822df8658d03c95ec47fbd2267d218f03d -s 2
```

Once a sequence proof for a specified slot has been fetched, it can be updated to include all new slot proofs in the sequence up to the latest with the update `-u` argument:

```
msc fetch -u -s 2
```

### 13.3.4 Verification

The client can perform various independent and trustless verification operations on sequence proofs to confirm the immutability of specified sequences.
Full verification of a specified sequence proof is performed in two stages, as a sequence proof bridges a secondary system to the Bitcoin blockchain. So typically a user will want to independently verify two properties of a specific sequence proof:

1. That the sequence proof is attested to the unbroken sequence of staychain transactions confirmed in the Bitcoin blockchain at the specified slot position.
2. That the sequence proof corresponds to the sequence of state changes in the external system.

The client enables users to perform each verification separately according to their individual requirements. Both operations must be successfully performed to verify a unique single history.

**Bitcoin blockchain verification**

To verify a specified sequence proof against the Bitcoin blockchain, a connection to a full Bitcoin node must be provided. This is set using the \(-b\) argument, and can be either an RPC URL with authentication details or a public HTTP address (for a remote block explorer). The Bitcoin node can also be set in the client config. For example:

```
msc config \-b username:password@localhost:8332
```

or

```
msc config \-b https://api.blockcypher.com/v1/btc/main/txs/
```

If no node URL is provided, a default public Bitcoin block explorer is used (currently: api.blockcypher.com/v1/btc/main/txs).

To perform the verification of a sequence proof against Bitcoin, the proof can be supplied as a file (using the \(-f\) argument) or as a JSON object (using the \(-p\) argument). To verify the stored proof in the application data directory, use \(-p 0\). For example:

```
msc verify \-p 0 \-s 2
```

If the verification of unique sequence is successful, the client will return the staychain commitment details. For example,

```
Verified proof sequence
Start commitment in block
→00000000000000000002e347026ca276fc5035f637deea48c6386c90504f042b height 604260 at 2019-11-17T21:07:18Z
End commitment in block
→00000000000000000001589123ee33c19e5a7ac8ac8f173867c8f877a7051d16 height 604757 at 2019-11-21T10:17:52Z
```

If the staychain base transaction ID is also included (the unique identifier) in the configuration, or via the \(-i\) argument, the client will additionally verify that this TxID is part of the staychain.

**State change history verification**

To verify that a specified sequence proof corresponds a sequence of state changes, one of the additional arguments of \(-l\), \(-d\) or \(-g\) is used along with the \(-p\) or \(-f\) arguments specifying the proof. The simplest of these is the \(-l\) argument which simply verifies that the given sequence proof matches a specified list of 32-byte commitments. These commitments are supplied hex encoded and comma separated. For example:

```
msc verify \-p 0 \-s 2 \-l "c635faa8f63f80d40fcc5f764aa3cb2c6de66027682ece03efc499db2edad780, 4113d23c569dcb821bf23f0f551b4eb980999be89464ac7f424b6dbda12924, 118d182a45bffea9fd8c6eb98453b6edc19327f6df2887b10700d194c275259"
```

(continues on next page)
If the proof sequence matches the commitment list exactly and in order the client will return the verification:

```
Verified proof sequence against commitment list.
```

If the sequence does not match, the verification will fail.

```
Verification failed. Commitments not matched.
```

If all slot-proofs in the sequence proof are matched to commitments in the list in order, but there are additional commitments included in the list, then the client will return:

```
Verification failed. Additional commitments on list not in proof.
```

To verify that a specified sequence proof corresponds to a chronological sequence of files in a specified directory, the additional argument `-d` is used to specify the directory path. This directory must contain the matching sequence of files, named in an alpha-numeric order corresponding to the sequence of changes. For example:

```
msc verify -p 0 -s 2 -d /Users/username/directory/
```

If the proof sequence matches the full hash chain of files in the specified directory exactly and in sequence, then the client will return the verification:

```
Verified proof sequence against directory hash chain.
```

The client will also return a warning if additional files have been added to the directory since the last attestation has been performed.

```
WARNING: last 1 files not attested.
Last file attested: document-v0.5.txt
```

To verify that a specified sequence proof corresponds to the commit history of a Git repository, the additional argument `-g` is used to specify the directory path of the Git repository. The client also checks that the initial commit message of the repository is a staychain TxID and slot ID. For example:

```
msc verify -p 0 -s 2 -g /Users/username/gitrepo/
```

If the proof sequence matches the full hash chain of files in the specified directory exactly and in sequence, then the client will return the verification:

```
Verified proof sequence against commit history to b40a656d028618f6c1d73465c07d810078fd74e4
```

Where `b40a656d028618f6c1d73465c07d810078fd74e4` is the latest Git commit included in the sequence proof. If there have been additional commits to the Git repository since the latest attestation in the sequence proof, the client will return a warnings with the number of non-attested commits. For example:

```
WARNING: last 3 commits not attested.
```

The `verify -g` operation also verifies that the staychain base TxID in the sequence proof, and slot ID, are added as the commit message in the initial commit of the repository. If this commitment is not present, the following warning is given:

```
Staychain ID not committed to Git history
```

13.3. Usage
13.3.5 Sidechain synchronization

The client can be used to synchronize a sidechain state against a Bitcoin staychain. This is performed using the `sync` command, and requires an RPC connection to both a full Bitcoin node (or trusted block explorer) and the sidechain node. As with the Bitcoin node connection, the sidechain node connection can also be set in the client config:

```
msc config -b username1:password1@localhost:8332  # Bitcoin node
msc config -n username2:password2@localhost:8336  # Sidechain node
```

To verify that a sidechain history is unique against Bitcoin’s global state, and determine the latest attested sidechain block, the full sequence proof is retrieved, fully verified and then verified against the Bitcoin staychain the the sidechain state. For example:

```
msc sync -s 1
```

If the verification is successful, the client will return the latest sidechain verified block. For example:

```
Verified sidechain attestation sequence
Latest attested sidechain block:
→47e3d796f0ae87f2261e620018ffbe0458175e17faf2762f209a17c727a8690 height 163188
```

13.3.6 Key generation and authentication

The client can generate keys to be used for attestation authentication, and generate commitment signatures using the `keygen` command. To generate a 256 bit private key, the `-g` argument is used with optional supplied entropy. For example:

```
msc keygen -g entropy
```

This generated key is then saved in the config and automatically used to sign attestations sent the the Mainstay service URL. The generated hex-encoded private key can then be used to generate the corresponding secp256k1 compressed public key using the `-p` argument.

```
msc keygen -p c76849c6ac48c4996b2847a5b87d9ee0e9463eal1c827591a50978b1b2682804
```

The returned hex-encoded public key is supplied in the web form used to sign-up to the mainstay.xyz service, if signature based authentication is required.

13.3.7 Staychain status and information

When initializing a sidechain, Git repository or file repository, the staychain base TxID and slot position must be committed to the initial state in order to prove uniqueness. In a sidechain, this information is committed to the genesis block, and in the case of a Git repository, this information is added as the message of the initial commit. To retrieve the latest staychain TxID to perform this initialisation, the `info` command can be used.

```
msc info
```

This returns the base ID. For example:

```
Base ID: 9d049eb88c13d7c4bad6f2597417da525effebe47b2095621b8cebad7ded4cf5:2
```

The argument `-c` will also set this in the config.

To initialise a Git repository and link it to the staychain and slot position, the initial commit will be as follows:
This guide describes how to link a sidechain to the Bitcoin blockchain via the mainstay.xyz connector service, in order to obtain a trustless proof of a unique single history of transactions for the sidechain. Here, the sidechain is any linked blockchain and can in principle be public or private, permissioned or permissionless, but a defined entity will be responsible for performing state commitments to the mainstay.xyz API. Once a blockchain is connected to the service, it becomes a Bitcoin sidechain i.e. a blockchain that is linked to and dependent upon data encoded on a separate blockchain: the mainchain.

This guide assumes that the sidechain node/client instance has a Bitcoin-like API (i.e. with getbestblockhash RPC call) and that a 32+4 byte commitment can be made to the genesis block of the sidechain in a specified location. If the sidechain is based on the Ocean platform client, these commitments can be specified in the ocean.conf that defines the chain.

The guide is split into three distinct sections for setting up the linking, performing state attestations by the permissioned entity and performing public verification.

### 14.1 Initial set-up

The entity responsible for attesting the sidechain state to Bitcoin is called the administrator. The administrator is responsible for the agreement with the mainstay.xyz service for exclusive use of a connector service slot. This is obtained by signing up on the mainstay.xyz website, where the administrator will be given a slot position slot_id and an API access token to perform attestations token via email. For additional security, the administrator can optionally require that all sidechain attestations committed to the slot are signed by an administrator private key (the corresponding public key being provided on sign-up at mainstay.xyz).

To generate this key, the administrator can use the Mainstay client. For example (where entropy is any string to add additional entropy to the keygen),

```
msc keygen -g entropy
```

> Private key: c66cb6eb7cd585788b294be28c8dcd6be4e37a0a6d238236b11c0beb25833bb9

This generates a hex encoded 32-byte private key (privkey). The corresponding public key can be generated as follows:
The generated public key is then supplied at sign-up. The private key is stored in the msc config file.

Once the mainstay.xyz slot is active, the base staychain Bitcoin transaction ID can be retrieved as follows (for slot_id = 3):

```
msc info -s 3
```

If the slot has been activated (usually within an hour of sign-up completion), the base ID will be returned. E.g.

```
> Base ID: 420d083de8ab078dbba5ea37f877cb35dd621e34f231ccee977a16cd241449d51:3
```

If not active, the following message will be returned:

```
> Slot 7 not active.
```

### 14.1 Sidechain configuration

The base ID (which is the staychain base TxID and slot ID) must then be committed to the sidechain genesis block to uniquely link the sidechain to a single slot-proof sequence. In the case of an Ocean platform sidechain, this means setting the following parameters in the ocean.conf chain definition file:

```
attestationhash=420d083de8ab078dbba5ea37f877cb35dd621e34f231ccee977a16cd241449d51
mappinghash=00000000000000000000000000000000000000000000000000000000000000000003
```

The sidechain can then be launched and new blocks created at regular intervals (see the Ocean platform and block signing guide for more details).

### 14.2 Attestation

The administrator has access to a connected sidechain node. At a frequency equal to the block creation frequency of the sidechain, the administrator commits the hash of the sidechain tip to the mainstay.xyz slot:

```
$BLOCKHASH = `ocean-cli getbestblockhash`
msc attest -c $BLOCKHASH -s 3 -t token
```

This would typically be performed automatically as a cron job. Alternatively, `attestation-tool` in the main mainstay application can be used.

### 14.3 Verification

The previous steps are performed solely by the administrator. Verification can be performed by anyone who is verifying the sidechain (i.e. running a full sidechain node). In addition to having RPC access to the sidechain client (oceand), a verifier also requires the Mainstay client installed (msc) and RPC access to a full bitcoind node (alternatively a trusted block-explorer API can be used).

To verify the full proof sequence for the sidechain and determine the latest sidechain attested block, the `sync` command of the Mainstay client can be used. This requires the sidechain oceand and bitcoind RPC credentials and URLs are provided (as `−n` and `−b` respectively). To sync, simply run:
If the verification is successful, the client will return the latest sidechain verified block. For example:

```
msc sync -s 3 -n username2:password2@localhost:8336 -b
--username1:password1@localhost:8332
```

Verified sidechain attestation sequence
Latest attested sidechain block:

```
47e3d796f0ae87f2261e620018ffb1e0458175e17faf2762f209a17c727a8690 height 163188
```

The `msc` client retrieves and verifies the full sequence of staychain commitments back to the base ID transaction. This may take some time if there is substantial history. The full slot proof sequence is written to the msc data directory (the location of which can be found by running `msc config`). After the initial sync, further `sync` commands append to the saved proof sequence, taking much less time.
CHAPTER 15

File repository guide

This guide describes how to link a file repository to the Bitcoin blockchain via the mainstay.xyz connector service, in order to obtain a trustless proof of a unique single history of the sequence of files added to the repository. The repository is defined by a specified directory on the users system, and the user is able to unambiguously prove to anyone the order in which documents or other files are added to this directory, and that only these files have been added.

The file-repository protocol creates a proof of an immutable sequence for files/documents by concatenating the SHA256 hashes of an ordered set of files, and then attesting the SHA256 of this concatenation to the mainstay.xyz slot each time it is extended with a new file. This data structure enables the generation of proof both of the order of files added to the repository over time and the unquineness of the files in the repository. This proof can be used to demonstrate the immutability and uniqueness of the version history of a file or document.

The guide is split into three distinct sections for setting up, performing repository attestations by the permissioned entity and verification.

15.1 Initial set-up

The entity responsible for attesting the file repository to Bitcoin is called the administrator. The administrator is responsible for the agreement with the mainstay.xyz service for exclusive use of a connector service slot. This is obtained by signing up on the mainstay.xyz website, where the administrator will be given a slot position slot_id and an API access token to perform attestations token via email. For additional security, the administrator can optionally require that all file repository attestations committed to the slot are signed by an administrator private key (the corresponding public key being provided on sign-up at mainstay.xyz).

To generate this key, the administrator can use the Mainstay client. For example (where entropy is any string to add additional entropy to the keygen),

```
msc keygen -g entropy
```

> Private key: c66cb6eb7cd58578b294be28c8cd6be4e37a0a6d238236b11c0beb25833bb9

This generates a hex encoded 32-byte private key (privkey). The corresponding public key can be generated as follows:
The generated public key is then supplied at sign-up. The private key is stored in the `msc` config file.

Once the `mainstay.xyz` slot is active, the base staychain Bitcoin transaction ID can be retrieved as follows (e.g. for `slot_id = 3`):

\[
\text{msc info -s 3}
\]

If the slot has been activated (usually within an hour of sign-up completion), the base ID will be returned. E.g.

\[
> \text{Base ID: 420d083de8ab078dbba5ea37f877cb35dd621e34f231ccee997a16cd241449d51:3}
\]

If not active, the following message will be returned:

\[
> \text{Slot 7 not active.}
\]

If the purpose of the file repository is to prove that a published document has only a single possible version history, then the Base ID should be committed (i.e. included) to the document (e.g. either as part of the visible main document or as file info metadata).

The administrator then creates a directory for the file repository on their system with path `repo_path`. Files can then be added to this directory, but they must be added with filenames in alphanumeric order in order for the attestation and verification to operate correctly. E.g. for a single document version history, the successive versions of the document should be named in alphanumeric order (e.g. `version_1.0.doc`, `version_1.1.doc` etc.). No files should be deleted from the repository, and the repository directory should be regularly backed up - loss of any files will prevent the proof from being verified.

### 15.2 Attestation

Each time the repository is updated with a new file version, the administrator can perform an attestation to the `mainstay.xyz` slot. This requires a single call of `msc` with the path of the repository directory supplied with the `-d` argument of the `attest` command. E.g.

\[
\text{msc attest -d repo_path -s 3 -t token}
\]

### 15.3 Verification

The previous steps are performed solely by the administrator. Verification can be performed by anyone who has access to the full document history (i.e. the contents of the document repository). The first step is to copy the full repository to a directory `repo_path` on the verifiers machine. The verifier also requires the Mainstay client installed (`msc`) and RPC access to a full `bitcoind` node (alternatively a trusted block-explorer API can be used).

The initial step in the verification process is to retrieve the full proof sequence for the slot to the base ID of the repository. This is done with the `fetch` command and the `-i` argument specifying the base TxID (and the `-s` argument specifying the slot ID).

\[
\text{msc fetch -i 420d083de8ab078dbba5ea37f877cb35dd621e34f231ccee997a16cd241449d51 -s 3}
\]

Which returns information on the attestation history. E.g.:
Sequence length: 36
Start: 7 Feb 2020 11:34:41
End: 10 Feb 2020 00:21:01

This retrieves the full proof sequence from the latest attestation to the base ID in the first commit of the repository. This sequence proof is saved in the msc data directory (which can be found by running `msc config`). The sequence proof can also be saved to any other file using the additional `-f` argument.

After fetching the sequence proof, it is then verified in two stages: 1. Verification that it is fully committed to the valid (i.e. single output) Bitcoin staychain in the specified slot position. 2. That the slot commitments correspond fully (and in order) to the file repository commit sequence.

The first stage is performed with supplied `bitcoind` RPC credentials and URL (as `-b`) and the slot ID (`-s`).

```
msc verify -s 3 -b username1:password1@localhost:8332 -p 0
```

The argument `-p 0` specifies that the sequence proof to be verified against Bitcoin is located in the data directory. Alternatively the path of the sequence proof can be provided (`-p proof_path`). If the verification is successful, the client will return the latest sidechain verified block. For example:

```
Verified proof sequence against staychain base
→ 420d083de8ab078dbba5ea37f877cb35dd621e34f231cc4997a16cd241449d51 slot 3
Start commitment in block
→ a883c7cb269f5d5767aebdf60691eaf4056f36a84fca99ebbc1632c511626b3c height 601563 at 7
→ Feb 2020 11:34:41
End commitment in block
→ fa968812f93f5c949d13021881179e4398582ef650dcla9e975951f9d411906 height 601611 at 10
→ Feb 2020 00:21:01
End commitment txout unspent
```

In addition, the client will state whether the last (End) commitment is in an unspent Bitcoin transaction. If it is not, there may be further commitments not reflected in the repository - this may be because a new attestation has been performed since the sequence proof was fetched. If this is the case, run `msc fetch -u -s 3` to update the update the sequence proof to the latest attestation, and repeat the verification. The stored sequence proof will then be updated to include the Bitcoin block heights of each attestation.

Once the validity of the full sequence proof against the Bitcoin blockchain has been established, the second stage of verification is to confirm that the sequence of commitments to the specified slot position correspond to the repository file hashes in a single, unbroken sequence. This is performed by using the `verify` command with the `-d` argument specifying the file repository path. E.g.

```
msc verify -s 3 -d repo_path
```

If verification is successful, the client will return:

```
Verified proof sequence directory hash chain
```

If information on the individual attestations of the sequence are required, the same command can be run with a higher verbosity `-v`, e.g.

```
msc verify -s 3 -d repo_path -v
```

The for each attestation in the sequence, the following is returned:

```
Commitment 2eaa89dfca418c6bacf999ba9f54ba43203170159d85d63fafa5783e11e741384
Latest file version_1.23.doc
```

(continues on next page)
In TxID f9499ca0c2125b4eda9490f50866fd3b9d5ff4b96b3224fac4972cc23ba67685
Block height 534875

If additional files have been made to the repository directory since the latest attestation and that are not part of the verified sequence proof, a warning is given. E.g.

WARNING: last 2 files not attested.
This guide describes how to link a Git repository to the Bitcoin blockchain via the mainstay.xyz connector service, in order to obtain a trustless proof of a unique single history of Git commits.

The guide is split into three distinct sections for setting up the linking, performing repository attestations by the permissioned entity and verification.

16.1 Initial set-up

The entity responsible for attesting the Git repository state to Bitcoin is called the administrator. The administrator is responsible for the agreement with the mainstay.xyz service for exclusive use of a connector service slot. This is obtained by signing up on the mainstay.xyz website, where the administrator will be given a slot position slot_id and an API access token to perform attestations token via email. For additional security, the administrator can optionally require that all repository attestations committed to the slot are signed by an administrator private key (the corresponding public key being provided on sign-up at mainstay.xyz).

To generate this key, the administrator can use the Mainstay client. For example (where entropy is any string to add additional entropy to the keygen),

```
msc keygen -g entropy
> Private key: c66cb6eb7cd585788b294be28c8dc6be4e37a0a6d238236b11c0beb25833bb9
```

This generates a hex encoded 32-byte private key (privkey). The corresponding public key can be generated as follows:

```
msc keygen -g c66cb6eb7cd585788b294be28c8dc6be4e37a0a6d238236b11c0beb25833bb9
> Public key: 03fe0c17d00e5d5fc879dd1c2d4f6e3d61e6d851ce3cd31173219aa72e13fcd9
```

The generated public key is then supplied at sign-up. The private key is stored in the msc config file.

Once the mainstay.xyz slot is active, the base staychain Bitcoin transaction ID can be retrieved as follows (e.g. for slot_id = 3):
If the slot has been activated (usually within an hour of sign-up completion), the base ID will be returned. E.g.

> Base ID: 420d083de8ab078dbba5ea37f877cb35dd621e34f231cce997a16cd241449d51:3

If not active, the following message will be returned:

> Slot 7 not active.

### 16.1.1 Git repository configuration

The base ID (which is the staychain base TxID and slot ID) must then be committed to the start of the Git repository, to verifiably and uniquely link it to the slot. This is performed by performing the initial commit of the repository with the base ID as the comment.

First initialise a new Git repository in the chosen directory `repo_path`:

```
git init
```

Then perform the initial commit, with the base ID as comment. E.g.:  

```
git add .
git commit -m '420d083de8ab078dbba5ea37f877cb35dd621e34f231cce997a16cd241449d51:3'
```

The repository can then be shared, or uploaded to a remote service like [github.com](https://github.com).

### 16.2 Attestation

Each time the repository is updated with a new commit, the administrator can perform an attestation to the `mainstay.xyz` slot (this may require pulling the latest changes from a remote master). This requires a single call of `msc` with the path of the repository supplied with the `-g` argument of the `attest` command. E.g.

```
msc attest -g repo_path -s 3 -t token
```

### 16.3 Verification

The previous steps are performed solely by the administrator. Verification can be performed by anyone who has access to the repository. The first step is to clone the repository to a directory `repo_path` on the verifier's machine. The verifier also requires the Mainstay client installed (`msc`) and RPC access to a full `bitcoind` node (alternatively a trusted block-explorer API can be used).

The initial step in the verification process is to retrieve the full proof sequence for the slot to the base ID of the repository. This is done with the `fetch` command and the `-g` argument specifying the repository (and the `-s` argument specifying the slot ID).

```
msc fetch -g repo_path -s 3
```

Which returns information on the attestation history. E.g.:
This will retrieve the full proof sequence from the latest attestation to the base ID in the first commit of the repository. This sequence proof is saved in the msc data directory (which can be found by running `msc config`). The sequence proof can also be saved to any other file using the additional `-f` argument.

After fetching the sequence proof, it is then verified in two stages: 1. Verification that it is fully committed to the valid (i.e. single output) Bitcoin staychain in the specified slot position. 2. That the slot commitments correspond fully (and in order) to the Git repository commit sequence.

The first stage is performed with supplied `bitcoind` RPC credentials and URL (as `-b`) and the slot ID (`-s`).

```
msc verify -s 3 -b username1:password1@localhost:8332 -p 0
```

The argument `-p 0` specifies that the sequence proof to be verified against Bitcoin is located in the data directory. Alternatively the path of the sequence proof can be provided (`-p proof_path`). If the verification is successful, the client will return the latest sidechain verified block. For example:

```
Verified proof sequence against staychain base
→ 420d083de8ab078d8ba5ea37f877cb35dd621e34f231ccej997a16cd241449d51:3 slot 3
Start commitment in block
→ a883c7cb269fd577ebebf6069eaf4056f36a84fca99eb131632c511626b3c height 601563 at Feb 2020 11:34:41
End commitment in block
→ fa968812f93f5c949b13d21812179f4398582ef650dca9e975951f9d411906 height 601611 at 10 Feb 2020 00:21:01
End commitment txout unspent
```

In addition, the client will state whether the last (End) commitment is in an unspent Bitcoin transaction. If it is not, there may be further commitments not reflected in the repository - this may be because a new attestation has been performed since the sequence proof was fetched. If this is the case, run `msc fetch -u -s 3` to update the sequence proof to the latest attestation, and repeat the verification. The stored sequence proof will then be updated to include the Bitcoin block heights of each attestation.

Once the validity of the full sequence proof against the Bitcoin blockchain has been established, the second stage of verification is to confirm that the sequence of commitments to the specified slot position correspond to the Git repository in a single, unbroken sequence. This is performed by using the `verify` command with the `-g` argument specifying the repository path. E.g.

```
msc verify -s 3 -g repo_path
```

If verification is successful, the client will return:

```
Verified proof sequence against commit history to
→ acb15cc7004ca5db3069e57072454bca62b0a
Verified Git commit history unique
Base txid: 420d083de8ab078d8ba5ea37f877cb35dd621e34f231ccej997a16cd241449d51 slot: 3
```

The first line gives the latest repository commit that has been attested. The second line confirms that the base ID and slot position was committed to the beginning of the repository (i.e. it is uniquely bound to the slot and staychain).

If additional commits have been made to the repository that are not part of the verified sequence proof, a warning is given. E.g.

```
In addition, the client will state whether the last (End) commitment is in an unspent Bitcoin transaction. If it is not, there may be further commitments not reflected in the repository - this may be because a new attestation has been performed since the sequence proof was fetched. If this is the case, run `msc fetch -u -s 3` to update the sequence proof to the latest attestation, and repeat the verification. The stored sequence proof will then be updated to include the Bitcoin block heights of each attestation.

Once the validity of the full sequence proof against the Bitcoin blockchain has been established, the second stage of verification is to confirm that the sequence of commitments to the specified slot position correspond to the Git repository in a single, unbroken sequence. This is performed by using the `verify` command with the `-g` argument specifying the repository path. E.g.

```
msc verify -s 3 -g repo_path
```

If verification is successful, the client will return:

```
Verified proof sequence against commit history to
→ acb15cc7004ca5db3069e57072454bca62b0a
Verified Git commit history unique
Base txid: 420d083de8ab078d8ba5ea37f877cb35dd621e34f231ccej997a16cd241449d51 slot: 3
```

The first line gives the latest repository commit that has been attested. The second line confirms that the base ID and slot position was committed to the beginning of the repository (i.e. it is uniquely bound to the slot and staychain).

If additional commits have been made to the repository that are not part of the verified sequence proof, a warning is given. E.g.
WARNING: last 2 commits not attested.
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.

17.1 Prerequisites

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

17.2 Instructions

17.2.1 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

### 17.3 Mainstay configuration

#### 17.3.1 Sample Config

```json
{
  "staychain": {
    "initTx": "87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1d362ac9e1d83cd840e1",
    "initScript": "51210381324c14a482646e9ad7cf82372021e5eb9a7e1b67ee168dddf1e97d7afe40af210376c091fae66bb37eb40af21037758a5cb1e60ab38f02e279c352ae",
    "initChaincodes": "0a090f710e47968aee906804f211cf10cde9a7e19408ca0f78cc55dd90ceaa",
    "topupAddress": "2MxBi6eodnuoVCw8McGrf1nuoVhastq0BXS",
    "topupScript": "51210381324c14a482646e9ad7cf92372021e5eb9a7e1b67ee168dddf1e97d7afe40af210376c091fae66bb37eb40af21037758a5cb1e60ab38f02e279c352ae",
    "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": "clientchain"
  }
}
```

(continues on next page)
17.3.2 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()`.

17.4 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:

17.4.1 Transaction Signing Tool

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

```go
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.

17.4.2 Client Signup Tool

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

```go
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`. 

17.4. Tools
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

17.4.3 Token Generator Tool

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

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

17.4.4 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:

```bash
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.

17.4.5 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

17.4.6 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=17073944010873801765385810419928396462990277618041305321615621854651469413039809327511867032326801302280634421130221500147
  -keysY=47581302232976976259016428448176075334749443379722569322944728779216384721, 1122700187475866687235948284541357909717856537392660494591205788179681685365
• go run $GOPATH/src/mainstay/cmd/multisigtool/multisigtool.go
  -chain=testnet -nKeys=2 -nSigs=1 -keys=03e52cf15e0a5cf6612314f077bb65cf9a6596b76c0fcb34b682f673a8314c7b33
• go run $GOPATH/src/mainstay/cmd/multisigtool/multisigtool.go
  -chain=regtest
For example use cases go to docs.

17.5 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 committed through the main service.

17.5. Initialising Mainstay
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.

### 17.5.1 Initial attestation

#### Generate 2 addresses

```bash
$ bitcoin-cli -datadir=testnetbtc-datadir/ getnewaddress
2MwCUjtecBAFcc7SWhEu8NyT1bLSCrTN6J

$ bitcoin-cli -datadir=testnetbtc-datadir/ getnewaddress
2N4FJ6xpbGdUvC8RjfMq6bzwXwEfWCFcYF
```

#### Generate multisig 1 of 2 address

```bash
$ bitcoin-cli -datadir=testnetbtc-datadir/ addmultisigaddress 1 "["2NFBB5okotyGFLmceXK7q18ufuv11NmefUJ","2NE8WKRRuj53udVsuyj5GbVfyUNN6ZSE4ia"]" "" -legacy
{
  "address": "2N5ckx6eXY5vx3DLwBwSZaNVShiZ6k6mSGd",
  "redeemScript": "512103d11753d31309988c323142a0171e5b2319a8651479835afa4ab8ebcb6442141b921034f0538871c910019b8e15a3b"
}
```

#### Dump priv keys

```bash
$ bitcoin-cli -datadir=testnetbtc-datadir/ dumpprivkey
"2NFBB5okotyGFLmceXK7q18ufuv11NmefUJ",
"cTgsB8DjF2vJhFRtCPopvkmmNWP6CTQeb1Sd9zvXxRF5qH9p9V4ct

$ bitcoin-cli -datadir=testnetbtc-datadir/ dumpprivkey
"2NE8WKRRuj53udVsuyj5GbVfyUNN6ZSE4ia",
"cNGEfurnx9oL6z8UXuiugPXXoxs5cmMxfwnDbAa258StQ4AQTH8P
```

#### Import generated multisig address

```bash
$ bitcoin-cli -datadir=testnetbtc-datadir/ importaddress
"2N5ckx6eXY5vx3DLwBwSZaNVShiZ6k6mSGd " false
```

#### Send funds to generated multisig address

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

bitcoin-cli -datadir=testnetbtc-datadir/ sendtoaddress
"2N5ckx6eXY5vx3DLwBwSZaNVShiZ6k6mSGd 0.07926900 " true
```

(continues on next page)
87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1dafa62ac9e1d83cd840e1
bitcoin-cli -datadir=testnetbtc-datadir/ sendrawtransaction
˓
87e56bda501ba6a022f12e178e9f1ac03fb2c07f04e1dafa62ac9e1d83cd840e1

17.5.2 Topup information

Generate 2 addresses

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ getnewaddress
2MtEZ7J82XoieL7iHyUQw9ITzplEcVQTAyK

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ getnewaddress
2MwvCUjtecBAPcc7SWhe8NyT1bLsCRtN6J

Generate multisig 1 of 2 address

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ addmultisigaddress 1 "[\n˓
"2MtEZ7J82XoieL7iHyUQw9ITzplEcVQTAyK","2MwvCUjtecBAPcc7SWhe8NyT1bLsCRtN6J"]" "" advertised as legacy

{ "address": "2NBYUyyMpeLCb67bykLBMHyulidSRGsim", "redeemScript": "512102a2411030da6082a32d0166fc19f03e264a62ca138f83a2912d0b59969670792103d11753d31309988c323142a0..." }

Dump priv keys

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ dumpprivkey
˓
cPLfW9BRrJJzNWnWHRz689XEmSTRPsHYYFRTAQChULT5nUn8K

$ bitcoin-cli -datadir=/Users/nikolaos/testnetbtc-datadir2/ dumpprivkey
˓
cTgsB8DjF2vjjhFrtCOpvkmnMwp6CTQeblSd9zvXxF5qH9V4ct

17.6 Running the service

go build && go install && mainstay

17.6.1 Running the signing tools

• signer 1
go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go
-pk cTgsB8DjF2vjhFrtCPopvkmnNWP6CTQeb1Sd9zvXxRF5qHp9V4ct -pkTopup
cPlfW9BRoRj2NwNHwz6B5XEmsTHRFsHYyFRTAQCULt5nUn8FkW -host *:5001
  • signer 2

go run $GOPATH/src/mainstay/cmd/txsigningtool/txsigningtool.go
-pk cNGEfurnx9oL6z8XUuiGPXxoxs5cmMxfewDnAh258StQ4AQTH8P -pkTopup
cTgsB8DjF2vjhFrtCPopvkmnNWP6CTQeb1Sd9zvXxRF5qHp9V4ct -host *:5002

## 17.7 Commitment examples

### 17.7.1 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:

  ![Commitment Example Image](mainstay-tools/images/commitment.png)

- 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:

### 17.7.2 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=879d36232614b868a52549cf6961caaa4c8f09d3ecf6b3714d

To get more detailed information on the commitment use:

https://mainstay.xyz/api/v1/commitment/commitment?commitment=879d36232614b868a52549cf6961caaa4c8f09d3ecf6b3714d40d60

More information on the Mainstay API can be found on [github](https://github.com).
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.

18.1 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 choose 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 an 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 \( n_p \). 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.

### 18.2 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
18.3 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 $c_{\text{Gen}}$
2. The service period start time $s_{\text{Start}}$
3. The target number of tickets $n_p$
4. The auction price decay constant $d_c$
5. The percentage of client chain transaction fees paid to guardnodes $F_p$
6. The guardnode services required

The time-locked $p_{\text{Token}}$ output is set as spendable after a time $s_{\text{End}}$ (set via OP_CHECKLOCKTIMEVERIFY). The target number of nodes ($n_p$) 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.

Fig. 2: Schematic of the creation of a request via a request transaction.

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 $c_{\text{Address}}$). 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 $p_{\text{Token}}$. 3. That the client genesis hash matches a known client chain and 4. That $s_{\text{End}} > s_{\text{Start}} + 1$ hour $>$ current time + 2 hour.

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

### 18.4 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 $< s_{\text{Start}} - 1$ 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.

![Fig. 3: Schematic of the creation of a bid transaction and the refund of the excess CBT at the end of the auction.](image)

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.
18.5 Service delivery and verification

The service interval commences at time $s_{\text{Start}}$. 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.

18.5.1 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 CHALLENGE asset transaction to the relevant client chain and measures the time taken for the guardnode to construct a 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).

18.5.2 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 sends 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.
Fig. 4: Illustration of the coordinator and guardnode interface connectivity and architecture.

### 18.6 Service fee payments

At the end of the service period (as specified in the request) 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)
Proposed design for Guardnode functionality in the Ocean client.

19.1 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
```

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:

<table>
<thead>
<tr>
<th>Pubkey</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>target pubkey permission asset is locked at</td>
</tr>
<tr>
<td>key2</td>
<td>&lt;02 pubkey prefix&gt; &lt;32 bytes client chain genesis block hash&gt;</td>
</tr>
<tr>
<td>key3</td>
<td>&lt;03 pubkey prefix&gt; &lt;4 bytes service period start time (block height)&gt; &lt;4 bytes target number of tickets&gt; &lt;4 bytes auction price decay constant&gt; &lt;4 bytes fee percentage&gt; &lt;15 bytes empty&gt;</td>
</tr>
</tbody>
</table>

The request transaction is generated with the `createrawrequesttx` RPC, which takes the following arguments:

1. Txid of input transaction
2. Vout (output index number) of input
3. Target request public key to lock funds into
4. Auction decay constant
5. End time of service period
6. Percentage of fees rewarded to guardnodes
7. Client chain genesis block hash
8. Start time of service period
9. Number of tickets in auction
10. Value of input transaction

The RPC will then create a new transaction with one input (txid and vout given) and two outputs. The first output will pay to the target request public key address and have a time-lock set to the end time (block height). Request outputs are in CBT, or the domain asset of the service chain if not in the Commerce Block network. The first output script can be constructed as:

```plaintext
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 broadcast.

## 19.2 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. Filtering by genesisBlock can be done by providing a block hash as the first argument.

e.g.

```plaintext
[
    {
        "startBlockHeight": 40060,
        "endBlockHeight": 90000,
        "confirmedBlockHeight": 30000,
        "genesisBlock": "fa1c7d059dab70cddb8cb3cc7b8971d385eecea4d68bd86c5cb6d75949789bal",
        "numTickets": 100,
        "startPrice": 100000,
        "auctionPrice": 80000,
        "decayConst": 100000,
        "feePercentage": 100,
        "txid": "cc1c7d059dab70cddb8cb3cc7b8971d385eecea4d68bd86c5cb6d75949789bal"
    }
]
```
This RPC will be called by the Guardnode operators/interface to get current requests. If blockheight < startBlockHeight then the auction is potentially still active, to return only requests that are in auction the second argument is set to true. (This RPC can be modelled on existing functions like gettxoutsetinfo).

### 19.3 Decay function

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

```cpp
CAmount CRRequest::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).

![Ticket price decay function](attachment:image.png)

Ticket price decay function with startPrice = 100000 CBT and decayConst = 1000000.

### 19.4 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 received (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 received against the request. As valid bids are received, the transaction IDs are added to this vector (along with the bid block height) up to a max of numTickets. A valid bid is described below, and are added to the vector by the UpdateRequestList function.

So the table will look like this:

```json
[
  {
    "requestTxID": "a22fe01032a5f33f37d3feb94df941a6c90d8d0c3113548e0776f3413f33346",
    "confirmedBlockHeight": 30000,
    "startBlockHeight": 40060,
    "endBlockHeight": 90000,
    "genesisBlock": "falc7d059ad70cdd8c3cc78b971d385eecead68bd86cb6d75949789ba1",
    "numTickets": 100,
    "startPrice": 100000,
    "auctionPrice": 80000,
    "decayConst": 1000000,
    "feePercentage": 100,
    "bids": [
      { "txid": "65eacf082247aaf0b1624539a0d7e3bb667b73211269907b0504a38888ab0a22",
        "feePubKey": "0300adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656",
        "bidBlockHeight": 30000,
        "bidTxId": "af3d49ff538a9a2b7b8b924aa27f102fb391811c387e7b5b06fc0345d6d4d8",
        "feePubKey": "0311adf7a8f55f92f8be6a5ed7619d1821c5bc9901f5592badea04677043b83656",
        "txid": "64c78af082247aaf0b1624539a0d7e3bb667b73211269907b0504a38888ab0a22",
    }
  }
]
```

The getrequestbids RPC outputs this vector of bids (with txids and block heights) for a given request transaction ID (by querying the in-memory table).

## 19.5 Bid transactions

Bid transactions will be created with a new RPC createrawbidtx. This will take as arguments:

1. Txid of input transaction
2. Vout (output index number) of input
3. The asset of the input, as a tag string or a hex value
4. Target stake public key to lock funds into (the address to which the stake will be paid back at the end of the service period)
5. Staked amount to lock in target pubkey
6. Change amount
7. Change address
8. Transaction fee amount
9. Service end block height
10. Request transaction ID
11. Public key 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 “changeAddress” fields in the output object. These are optional and should only be included when the input amount exceeds the bid amount.

19.6 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.

**19.7 Bid output policy**

The request bid set is used for two purposes:

1. Enable the coordinator to pay client chain fees to the winning bidders

2. 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.
The Coordinator daemon is responsible for verifying the operation of Guardnodes in the Commerceblock Covalence system. It is run by the authority controlling the service chain with the following functionality:

- Send guardnode challenges (request for specific piece of data from client chain to prove guardnode maintains a local copy of the client blockchain)
- Verify challenge responses (Verify that the response data is correct and is returned within a small enough period of time from when the challenge was sent)
- Send guardnode fee payments at the end of each request service period
- Query the connection status of the guardnode client chain node and obtain the current peer list

**20.1 Running**

Information on running a Coordinator daemon can be found in the Github repository.

**20.2 Requirements**

To run the coordinator daemon we require:

- Commerce Block service chain connectivity to receive active client requests
- Client chain connectivity to generate guardnode challenges
- A DB instance to store requests and responses
- Listener HTTP POST endpoint to receive guardnode responses
- A public RPC API to offer information on requests and guardnode response performance
### 20.3 Configuration

Connection information for all services that the coordinator needs to interact with are set via environment variables:

<table>
<thead>
<tr>
<th>Env variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO_LISTENER_HOST</td>
<td>Host address to receive guardnode responses (default: localhost:80)</td>
</tr>
<tr>
<td>CO_CHALLENGE_FREQUENCY</td>
<td>Frequency in number of blocks that new challenges are created (default: 1)</td>
</tr>
<tr>
<td>CO_CHALLENGE_DURATION</td>
<td>Challenge duration in seconds (default: 60)</td>
</tr>
<tr>
<td>CO_BLOCK_TIME</td>
<td>Service chain block time in seconds (default: 60)</td>
</tr>
<tr>
<td>CO_LOG_LEVEL</td>
<td>Environment logger log level (default: “coordinator”)</td>
</tr>
<tr>
<td>CO_API_HOST</td>
<td>RPC api host address</td>
</tr>
<tr>
<td>CO_API_USER</td>
<td>RPC api username</td>
</tr>
<tr>
<td>CO_API_PASS</td>
<td>RPC api password</td>
</tr>
<tr>
<td>CO_STORAGE_HOST</td>
<td>DB storage host address (default: “localhost:27017”)</td>
</tr>
<tr>
<td>CO_STORAGE_USER</td>
<td>DB storage username</td>
</tr>
<tr>
<td>CO_STORAGE_PASS</td>
<td>DB storage password</td>
</tr>
<tr>
<td>CO_STORAGE_NAME</td>
<td>DB storage database name (default: “coordinator”)</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_HOST</td>
<td>Client chain rpc host address</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_USER</td>
<td>Client chain rpc username</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_PASS</td>
<td>Client chain rpc password</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_GENESIS_HASH</td>
<td>Client chain genesis hash</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_BLOCK_TIME</td>
<td>Client chain block find time interval in seconds (default: 60)</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_ASSET_KEY</td>
<td>Client chain challenge asset key</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_ASSET</td>
<td>client chain challenge asset/assetlabel (default: “CHALLENGE”)</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_CHAIN</td>
<td>client chain chain name</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_PAYMENT_ASSET</td>
<td>client chain fee payment asset</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_PAYMENT_KEY</td>
<td>client chain fee payment key (optional; for payment mode only)</td>
</tr>
<tr>
<td>CO_CLIENTCHAIN_PAYMENT_ADDR</td>
<td>client chain address for payment funds (optional; for payment mode only)</td>
</tr>
<tr>
<td>CO_SERVICE_HOST</td>
<td>service chain host address</td>
</tr>
<tr>
<td>CO_SERVICE_USER</td>
<td>service chain username</td>
</tr>
<tr>
<td>CO_SERVICE_PASS</td>
<td>service chain password</td>
</tr>
</tbody>
</table>

### 20.4 Demo

A demo is included to test the coordinator locally. The demo.sh will spin up a fresh chain and create a request along with 2 bids. The alias commands at the beginning of the script must be directed to your ocean/src/ directory. Using this demo chain the coordinator demo creates two guardnodes and challenges them for information about the client chain. When they respond the coordinator output shows them being verified and stored ready for fee payments to be made later.

To run the demo:

```
./scripts/demo.sh && cargo run --example demo
```

This script is also used in a tutorial showing guardnode functionality and interaction with Coordinator.

After the script is complete the coordinator demo creates two guardnodes and randomly challenges them by randomly including CHALLENGE asset transactions in blocks. When the guardnodes respond the coordinator output shows them being verified and stored ready for fee payments to be made later.
20.5 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 {"page": <int, optional>}']): fetches up to 10 requests for the client. more can be requested with the “page” parameter
- `getrequest {"txid": "hash"}']: fetches the specific request
- `getrequestresponse {"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": "getrequestresponse", "params" : {"txid": "5eba0bf305ac8963225d68195fa7eb8b79667ad9c5fa6e9dce0185ad4a046"}, "id":1 }' userApi:passwordApi@localhost:3333

20.6 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":"0404040404040404040404040404040404040404040404004","sig": "30450221009dd76bcdc19a283654727214757b9e33ded38f00951b4f4a074e6fbeb17a6f2ef02205702424facf6333cfed3"}
```
Guardnode daemon responding to client chain coordinator challenges and generating alerts for misbehavior on the chain.

21.1 Running

Find instructions for running the guardnode daemon in the Guardnode repo.

21.2 Viewing and Spending locked outputs

When a successful bid is made your guardnode will lock up your CBT for the duration of the service period. These outputs are not recognised by some RPCs such as `getBalance`. To include your locked funds in a call to `getBalance` include all addresses and watch only addresses with the following arguments:

```
getbalance "*" 0 true
```

These funds can be respent but only using RPCs that allow watch only addresses in their coin selection, therefore RPCs such as `sendanytoaddress` will not recognise the previously locked funds. Using `createanytoaddress` with the `allowwatchonly` argument will include previously locked outputs in its inputs, e.g:

```
createanytoaddress "$ADDRT" $AMOUNT true true 1 true
```

Then sign and send the transaction using `signrawtransaction` and `sendrawtransaction`.

21.3 Demo

The following is a demo of the Guardnode in action responding to a challenge. First run the demo script which generates a request and two bids for that request on a mock service chain.
Next, in a separate terminal window execute the following script and watch the guardnode recognise the request and submit a bid.

```bash
./run_guardnode --rpcuser user1 --rpcpass password1 --rpchost 127.0.0.1:5555 --servicerpcuser user1 --servicerpcpass password1 --servicerpchost 127.0.0.1:5555 --nodelogfile $HOME/co-client-dir/ocean_test/debug.log --challengehost 127.0.0.1:5555 --bidlimit 50 --serviceblocktime 5
```

We can now mine some blocks to bring our request into its service period. Then we can send a CHALLENGE asset transaction and watch the guardnode react. In the first terminal window execute:

```bash
alias ocn='/$HOME/ocean/src/ocean-cli -datadir=$HOME/co-client-dir'
ocn generate 5
ocn sendtoaddress $(ocn getnewaddress) 1 "" false "CHALLENGE"
ocn generate 1
```

As there is no connection to a coordinator we get an error message but the would-be response message is displayed. Guardnode sends their bid txid to identify themselves, the challenge tx hash and a signature to coordinator as a response to the challenge and thus prove their active watching of the client chain. The coordinator uses this to determine whether to send payment to each guardnode at the end of the service period depending on whether they responded correctly to each challenge or not.

Further guardnode functionality such as invalid block and consensus anomaly detection will be implemented soon.

### 21.4 Demo with coordinator

We can repeat the same demo with connection to a coordinator and observe the process of coordinator generating challenges, guardnodes sending responses and coordinator verifying them.

Run a coordinator daemon and execute the following:

```bash
./run_guardnode --rpcuser user1 --rpcpass password1 --rpchost 127.0.0.1:5555 --servicerpcuser user1 --servicerpcpass password1 --servicerpchost 127.0.0.1:5555 --nodelogfile $HOME/co-client-dir/ocean_test/debug.log --challengehost 127.0.0.1:5555 --bidlimit 50 --serviceblocktime 5
```

### 21.5 Configuration

The full list of arguments are given below:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--rpchost</td>
<td>Client chain RPC host address</td>
</tr>
<tr>
<td>--rpcuser</td>
<td>Client chain RPC username</td>
</tr>
<tr>
<td>--rpcpass</td>
<td>Client chain RPC password</td>
</tr>
<tr>
<td>--servicerpchost</td>
<td>Service chain RPC host address</td>
</tr>
<tr>
<td>--servicerpcuser</td>
<td>Service chain RPC username</td>
</tr>
<tr>
<td>--servicerpcpass</td>
<td>Service chain RPC password</td>
</tr>
<tr>
<td>--serviceblocktime</td>
<td>Service chain block time</td>
</tr>
<tr>
<td>--nodelogfile</td>
<td>Client chain log file destination</td>
</tr>
<tr>
<td>--bidpubkey</td>
<td>(Optional) Pre-made bid public key</td>
</tr>
<tr>
<td>--bidlimit</td>
<td>Upper limit in bid size</td>
</tr>
<tr>
<td>--challengehost</td>
<td>Challenger host address</td>
</tr>
<tr>
<td>--uniquebidpubkeys</td>
<td>Flag to indicate new bid pubkey generation for each bid</td>
</tr>
</tbody>
</table>
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 guarding the client chain.

In order to run the guardnode service navigate to the config directory of the corresponding client shown below, download the docker-compose file and follow the README instructions on how to run the node using data persistence. The current active client services are:

- DGLD mainnet chain config
- CB testnet chain config

The service 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. For more info on this check erc20 peg

To start the service chain node, client chain node and guardnode run:

```
docker-compose up -d
```

The guardnode will automatically look for requests in the specific client chain and bid for them at the current auction price. This will only work if the service chain node wallet has been funded with CBT. Guardnode owners can limit the amount of CBT they are willing to bid with by overriding the `--bidlimit` option in the guardnode container in the compose file downloaded above.

Guardnode operators should monitor the logs and alert CB about any issues in the client chain by:

```
docker-compose logs -f guardnode
```
A guide for creating service requests in the CommerceBlock chain.

### 23.1 1. Running the service chain full node

Download the docker-compose file from [ocean github](https://github.com) and follow the [docs](https://commerceblock.com) 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.

### 23.2 2. Create a request

The `create_request.sh` script can be used to create a request. It takes the following arguments:

- Client chain genesis hash
- Start price
- End price
- Auction duration
- Service period duration
- Number of tickets
- Fee percentage to reward guardnodes
- Private key of address containing permission asset (optional)
- txid of previous request transaction to fund new request (optional)
- vout of previous request transaction to fund new request (optional)
Node information is gathered from ENV variables:

<table>
<thead>
<tr>
<th>Env variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC_CONNECT</td>
<td>RPC address</td>
</tr>
<tr>
<td>RPC_PORT</td>
<td>RPC port</td>
</tr>
<tr>
<td>RPC_USER</td>
<td>RPC username</td>
</tr>
<tr>
<td>RPC_PASS</td>
<td>RPC password</td>
</tr>
</tbody>
</table>

The script checks for currently active requests for the given client chain genesis hash. If none are currently active then a new request is created and published, therefore allowing for automatic request generation when the script is run as a cron job.

### 23.3  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.
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.

**Step 1**
Export the private key of the ethereum address that owns the CBT tokens. This should be in hex format, e.g.

```
"0xcb850d9db23b54ebbeae09995f7192af83646f9ea232645bb5a71699e5c15a6e"
```

**Step 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"
```

**Step 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. The CB chain can only handle a precision of up to 8 decimal points.

**Step 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
```

159
The amount should be exactly the same as the Ethereum transaction amount and if more than 8 decimal points were used then they should be discarded.

**Step 5**

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

**Step 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
```

### 24.1 Issues with unconfirmed peg-ins

Failing to follow the above steps correctly or attempting to claim the peg-in before the minimum number of confirmations has been reached will cause the transaction getting stuck in the user’s mempool. The node will periodically relay the transaction and that will work for transactions missing confirmations but not on any other case.

To resolve issues with stuck transactions follow the instructions below:

- Stop the CB node
- Delete the “mempool.dat” file
- Add `-zapwallettxes=1` to the node configuration
- Restart the CB node
- Re-submit `claimethpegin` with the correct information

An alternative option is to backup the “wallet.dat” file and create a new node, importing the previous wallet.

To display network rejection messages for transactions the config option `-debug=net` can be added to the CB node. That should give some hints on why the claim peg-in transaction is rejected.