Rocket Pool Documentation

Release latest

Mar 31, 2021

Contents

Rocket Pool is a decentralised Ethereum staking network built to be compatible with the Ethereum 2.0 beacon chain.

CHAPTER 1

Contents

1.1 Introduction to Rocket Pool

1.1.1 What does Rocket Pool do?

Rocket Pool is a first-of-its-kind decentralised staking network for Ethereum. An excellent guide to Rocket Pool for new users can be found in our FAQ article.

It has several objectives within the Ethereum landscape, each relevant to a different audience. It aims to:

1. Provide a network of decentralised nodes to perform proof of stake validation services for the Ethereum network;
2. Allow users who don't possess the minimum ether to become a validator - or the necessary technical skills to run a node - to participate in staking and earn rewards;
3. Allow users and businesses to run validator nodes for Eth 2.0 with only 16 ETH, to earn rewards and additional income;
4. Provide any third party businesses with seamless staking services for their users.

1.1.2 Contents

The following sections give a high-level overview of the various concepts used in Rocket Pool and how they work:

Staking

Overview

Staking is not unique to Rocket Pool, it's a feature coming to Ethereum 2.0 in the near future. Ethereum 2.0 will include a new chain called the Beacon Chain, which anyone can become a validator on by making a deposit (or "stake") of ether. Validators are responsible for ensuring the integrity of the Ethereum network, and earn interest on their deposit by doing so.

The Beacon Chain will require a large sum of ether in order to become a validator, which many may not have access to. Rocket Pool aims to solve this problem by pooling ether from multiple users together so that they can stake with less!

For Regular Stakers

Staking in Rocket Pool as a regular user is as easy as navigating to the Rocket Pool website, entering an amount of ETH to stake, and clicking Start! When you stake in Rocket Pool, you will immediately receive an amount of *rETH* with equivalent value to the ETH you deposit.

The value of rETH accumulates over time as the network earns rewards, so all you need to do to earn a profit is hold onto it. Once you're ready to exit, simply trade your rETH back in for ETH via the website (or on an exchange), and as long as the network has performed well, you'll end up with more than you put in!

For Node Operators

In order to run a node in the Rocket Pool network, you will need to install the Rocket Pool smart node software. The smart node client will allow you to create a new wallet for your node (to hold its Eth 1.0 account and Eth 2.0 validator keys) and register it with the Rocket Pool network. Once your node is registered, you can deposit ETH to create *minipools* (which will have user-deposited ETH assigned to them) and begin staking.

You may deposit either 16 or 32 ETH at a time. 16 ETH deposits create minipools which must wait until 16 user-deposited ETH is assigned to them before they begin staking. 32 ETH deposits create minipools which can begin staking and earning rewards immediately. User-deposited ETH is assigned to them later, and you are *refunded* the extra 16 ETH back to your node account.

Running a node in the Rocket Pool network is a long-term committment, as withdrawing a validator's balance will not be possible until phase 2 of the Eth 2.0 rollout. Rocket Pool provides an option for node operators to exit and gain access to their share of a validator's balance (in the form of *nETH*) before phase 2, but only after a long delay. Therefore, it is not reccommended to run a node unless you can commit to doing so over a long time period.

Reward Tokens

rETH (for regular stakers)

rETH represents a fungible, tokenized stake in the Rocket Pool network. A user stakes in Rocket Pool by depositing ETH in exchange for rETH. rETH holders can exit by simply exchanging their rETH for ETH.

The rETH token has a dynamic exchange rate for ETH, which is recorded by the Rocket Pool network and increases as the network earns rewards. This means that rETH is constantly accruing value, as long as the network is operating optimally. rETH holders can simply hold their tokens to earn a profit, and exchange them for ETH once they wish to claim their rewards.

The following example illustrates how the token is used to stake and earn rewards:

1. Alice deposits 10 ETH into Rocket Pool when the exchange rate is 1 rETH : 1 ETH. She receives 10 rETH.
2. Over the next year, the network earns rewards and increases in value by 20%. The exchange rate is now 1 rETH : 1.2 ETH.
3. Alice burns her 10 rETH via the rETH contract, and receives 12 ETH. She has exited with a 20% profit.
4. The same day, Bob deposits 10 ETH into Rocket Pool, and receives ~8.33 rETH.
5. Over the next year, the network earns rewards and increases in value by 10%. The exchange rate is now 1 rETH : 1.32 ETH.

6. Bob burns his ~8.33 rETH via the rETH contract, and receives 11 ETH. He has exited with a 10% profit.

It may take an unknown amount of time for rETH collateral to become available for exchanges, as node operators decide when to exit their validators. For this reason, Rocket Pool provides an additional pool of collateral for rETH exchanges in the form of "excess" deposit pool balance. Excess balance is defined simply as the amount of ETH in the deposit pool minus the capacity of all available minipools in the queue. In other words, as long as there is a surplus of user-deposited ETH waiting to be assigned, rETH holders can burn their tokens to claim it.

nETH (for node operators)

During Phases 0 and 1 of the Ethereum 2.0 rollout, withdrawals from the Beacon Chain will not be implemented. Rocket Pool aims to provide liquidity to node operators through the design of the nETH token, so they can access their rewards before Phase 2 is launched.

When a minipool's validator finishes staking on the Beacon Chain, the minipool is marked as withdrawable, and an amount of nETH equal to the node operator's share of the balance is minted to it. The node operator can then withdraw their nETH from the minipool and close it, but only after a significant delay. (This prevents malicious actors from attacking the Rocket Pool network by filling it with "idle" ETH.) nETH tokens are backed by Beacon Chain ether 1:1, and should trade on the open market for slightly less than 1 ETH in value.

When Phase 2 of the Ethereum 2.0 rollout is launched, users holding nETH will be able to swap it for Beacon Chain ether via the nETH contract. This effectively burns the nETH, removing it from circulation.

Nodes

Overview

Nodes are the workhorses of the Rocket Pool network. Anyone can register a node with Rocket Pool and begin staking. When a node operator wants to stake, they deposit 16 or 32 ETH into Rocket Pool, which is matched with 16 user-deposited ETH. The node performs all of the validation duties required by the Ethereum network, and earns a percentage of the rewards on the user-deposited ETH assigned to it as a commission.

The Node Commission Fee

The commission earned by a node is calculated based on network supply and demand dynamics. When the Rocket Pool network has a large pool of user-deposited ETH and limited capacity in available minipools, node demand is high. Conversely, when there is a small pool of user-deposited ETH and a lot of capacity in available minipools, node demand is low.

When a node makes a deposit to create a minipool, the node demand and commission rate are calculated at that moment and "locked in" for that minipool. High node demand results in a higher commission rate, while low demand results in a lower commission rate. The upper and lower bounds for node commission rate are recorded in Rocket Pool contracts, and will be adjustable via governance mechanics in the future.

When a node makes a deposit, they may specify a minimum commission rate they will accept for the created minipool, to account for fluctuations in the network commission rate while their transaction is mined. If the network commission rate drops below this value before the deposit transaction is mined, it is cancelled and reverts.

Watchtower Nodes

Some special nodes owned by Rocket Pool and trusted partners are designated as "watchtower" nodes. Watchtower nodes are responsible for reporting the Beacon Chain state back to the PoW chain. A majority of watchtower nodes must reach consensus on the information being reported before it takes effect.

Firstly, they report the total value of the Rocket Pool network to the Rocket Pool contracts at set intervals. This allows the dynamic rETH : ETH exchange rate to be updated in accordance with rewards earned.

Secondly, they report when a minipool's validator on the Beacon Chain is ready for *withdrawal*. The minipool is updated accordingly, so that the Rocket Pool network can track its progress. The network also mints nETH equal to the node operator's share of the validator's final balance to the minipool. After a delay, the node operator can withdraw this nETH and exchange it for ETH on the open market. After phase 2 of the Eth 2.0 rollout, it can also be burned for ETH via the nETH contract itself.

Watchtower nodes also perform some other minor tasks, such as automatically *dissolving* timed out minipools which fail to stake. This prevents user-deposited ETH in the network from sitting "idle" (not earning rewards) by returning it to the deposit pool.

Finally, watchtower nodes are granted special privileges to create "empty" minipools with 0 ETH deposited by the node operator. These minipools are assigned 32 user-deposited ETH, which the node operator can still earn a commission on the rewards for. Empty minipools are only assigned ETH if there are no regular ones waiting in the queue - they serve as a backup if there is a shortage of node operators in the network.

Minipools

Overview

Minipools are the matchmaking service at the heart of the Rocket Pool network. Whenever a node operator deposits ether into Rocket Pool, they create a minipool. This minipool contains the node operator's ether, and accepts 16 user-deposited ETH.

All minipools enter a queue and are assigned user-deposited ETH as it becomes available, in a FIFO fashion. If a node operator deposits 16 ETH, the resulting minipool must wait for its ETH assignment to begin staking. If a node operator deposits 32 ETH, the resulting minipool can begin staking immediately, and will refund the excess 16 ETH after assignment.

Minipools keep a strict record of all the funds deposited into them, and what has been withdrawn. This includes the node operator's ETH balance, and the user-deposited ETH balance. This way, the minipool knows how much to give back to each party once it has finished staking. They also record a lot of other information about the node which created them, their staking status, and more.

Staking Minipools

Once a minipool contains 32 ETH, it is ready to begin staking. The Rocket Pool smart node software will check for these minipools at set intervals, and progress them to staking. A validator keypair will be generated for each minipool, and then a transaction will be sent to forward its balance to the Eth 2.0 Validator Registration Contract. The deposit is sent to the VRC along with the validator key generated by the smartnode, and Rocket Pool's network-wide withdrawal credentials. This puts the node in charge of validation duties for the minipool, and Rocket Pool in charge of handling its withdrawal.

The smart node software will then watch for the validator being activated on the beacon chain. Once active, its validation duties will automatically be started.

Exiting & Withdrawing Minipools

Rocket Pool node operators may exit their minipools whenever they like, using commands provided by the smart node software. This sends a message to the beacon chain requesting that the validator be exited, which happens after some delay, according to the Eth 2.0 protocol. Once the validator is exited, Rocket Pool watchtower nodes will report this back to the network contracts, recording their final balance and minting nETH for the node operator.

Until phase 2 of the Eth 2.0 rollout, node operators may only withdraw their nETH earned after a significant delay. This prevents malicious actors from attacking the Rocket Pool network by filling it with “idle” ETH. This also means that running a node is a long term committment, and they should only be run by those prepared to do so over a long time period.

Once a validator’s balance has been withdrawn by Rocket Pool (after phase 2 only), it will be split between the nETH contract and the deposit pool. The share of ETH earned by the node operator will be sent to the nETH contract, to provide collateral for nETH token exchanges. The share of ETH belonging to Rocket Pool users will be recycled to the deposit pool, either to stake again or to act as collateral for rETH token exchanges.

Refunding Node ETH

If a minipool is created with a deposit of 32 ETH, it begins staking immediately but will still receive 16 user-deposited ETH when it reaches the front of the minipool queue. When this occurs, the excess 16 ETH deposited by the node operator becomes available for refund. The smart node software provides commands to check for minipools with available refunds, and to withdraw them back to the node account.

Rocket Pool also records the time at which user-deposited ETH is assigned to the minipool. All rewards earned by the minipool *before* this point are paid exclusively to the node operator. Rewards earned after this point are shared between the node operator and Rocket Pool users (after accounting for the node commission fee).

Dissolving Minipools

If a minipool is assigned user-deposited ETH but fails to begin staking within a set timeout period, it is considered to be idle. Any ethereum account can send a transaction to an idle minipool causing it to be dissolved and the user-deposited ETH in it to be returned to the deposit pool. Once a minipool has been dissolved, the node owning it may close it to return their ETH deposit to the node account. Rocket Pool watchtower nodes will automatically dissolve idle minipools at set intervals.

1.2 The Rocket Pool Smart Contracts

1.2.1 Introduction

The Rocket Pool Smart Contracts form the foundation of the Rocket Pool network. They are the base layer of infrastructure which all other elements of the network are built on top of, including the JavaScript library, the Smart Node software stack, and all web or application interfaces.

Direct interaction with the contracts is usually not necessary, and is facilitated through the use of other software (such as the JavaScript library). This section provides a detailed description of the contract design, and information on how to build on top of Rocket Pool for developers wishing to extend it. All code examples are given as Solidity v0.6.12.

1.2.2 Contents

Contract Design & Upgradability

Architecture

The Rocket Pool network contracts are built with upgradability in mind, using a hub-and-spoke architecture. The central hub of the network is the `RocketStorage` contract, which is responsible for storing the state of the entire network. This is implemented through the use of maps for key-value storage, and getter and setter methods for reading and writing values for a key.

The `RocketStorage` contract also stores the addresses of all other network contracts (keyed by name), and restricts data modification to those contracts only. Using this architecture, the network can be upgraded by deploying new versions of an existing contract, and updating its address in storage. This gives Rocket Pool the flexibility required to fix bugs or implement new features to improve the network.

Interacting With Rocket Pool

To begin interacting with the Rocket Pool network, first create an instance of the `RocketStorage` contract using its interface:

```
import "RocketStorageInterface.sol";

contract Example {

    RocketStorageInterface rocketStorage = RocketStorageInterface(0);

    constructor(address _rocketStorageAddress) {
        rocketStorage = RocketStorageInterface(_rocketStorageAddress);
    }

}
```

The above constructor should be called with the address of the `RocketStorage` contract on the appropriate network.

Because of Rocket Pool's architecture, the addresses of other contracts should not be used directly, but retrieved from the blockchain before use. Network upgrades may have occurred since the previous interaction, resulting in outdated addresses.

Other contract instances can be created using the appropriate interface taken from the Rocket Pool repository, e.g.:

```
import "RocketStorageInterface.sol";
import "RocketDepositPoolInterface.sol";

contract Example {

    RocketStorageInterface rocketStorage = RocketStorageInterface(0);

    constructor(address _rocketStorageAddress) {
        rocketStorage = RocketStorageInterface(_rocketStorageAddress);
    }

    exampleMethod() public {
        address rocketDepositPoolAddress = rocketStorage.getAddress(keccak256(abi.
↪encodePacked("contract.address", "rocketDepositPool")));
        RocketDepositPoolInterface rocketDepositPool =␣
↪RocketDepositPoolInterface(rocketDepositPoolAddress);
        ...
    }

}
```

The Rocket Pool contracts, as defined in `RocketStorage`, are:

- `rocketRole` - Handles assignment of privileged admin roles (internal)

- `rocketVault` - Stores ETH held by network contracts (internal, not upgradeable)
- `rocketUpgrade` - Provides upgrade functionality for the network (internal)
- `rocketDepositPool` - Accepts user-deposited ETH and handles assignment to minipools
- `rocketMinipoolFactory` - Creates minipool contract instances (internal)
- `rocketMinipoolManager` - Creates & manages all minipools in the network
- `rocketMinipoolQueue` - Organises minipools into a queue for ETH assignment
- `rocketMinipoolStatus` - Handles minipool status updates from watchtower nodes
- `rocketNetworkBalances` - Handles network balance updates from watchtower nodes
- `rocketNetworkFees` - Calculates node commission rates based on network node demand
- `rocketNetworkWithdrawal` - Handles processing of beacon chain validator withdrawals
- `rocketNodeDeposit` - Handles node deposits for minipool creation
- `rocketNodeManager` - Registers & manages all nodes in the network
- `rocketDAOProtocolSettingsDeposit` - Provides network settings relating to deposits
- `rocketDAOProtocolSettingsMinipool` - Provides network settings relating to minipools
- `rocketDAOProtocolSettingsNetwork` - Provides miscellaneous network settings
- `rocketDAOProtocolSettingsNode` - Provides network settings relating to nodes
- `rocketTokenRETH` - The rETH token contract (not upgradeable)
- `rocketTokenNETH` - The nETH token contract (not upgradeable)
- `addressQueueStorage` - A utility contract (internal)
- `addressSetStorage` - A utility contract (internal)

Contracts marked as "internal" do not provide methods which are accessible to the general public, and so are generally not useful for extension. For information on specific contract methods, consult their interfaces in the Rocket Pool repository.

Deposits

Overview

The main reason for extending the Rocket Pool network is to implement custom deposit logic which funnels user deposits into the deposit pool. For example, a fund manager may wish to stake their users' ETH in Rocket Pool via their own smart contracts, and abstract the use of Rocket Pool itself away from their users.

Note: the `RocketDepositPool` contract address should *not* be hard-coded in your contracts, but retrieved from `RocketStorage` dynamically. See *Interacting With Rocket Pool* for more details.

Implementation

The following describes a basic example contract which forwards deposited ETH into Rocket Pool and minted rETH back to the caller:

```
import "RocketStorageInterface.sol";
import "RocketDepositPoolInterface.sol";
import "RocketTokenRETHInterface.sol";

contract Example {

    RocketStorageInterface rocketStorage = RocketStorageInterface(0);

    constructor(address _rocketStorageAddress) {
        rocketStorage = RocketStorageInterface(_rocketStorageAddress);
    }

    receive() external payable {
        // Check deposit amount
        require(msg.value > 0, "Invalid deposit amount");
        // Load contracts
        address rocketDepositPoolAddress = rocketStorage.getAddress(keccak256(abi.
↪encodePacked("contract.address", "rocketDepositPool")));
        RocketDepositPoolInterface rocketDepositPool =␣
↪RocketDepositPoolInterface(rocketDepositPoolAddress);
        address rocketTokenRETHAddress = rocketStorage.getAddress(keccak256(abi.
↪encodePacked("contract.address", "rocketTokenRETH")));
        RocketTokenRETHInterface rocketTokenRETH =␣
↪RocketTokenRETHInterface(rocketTokenRETHAddress);
        // Forward deposit to RP & get amount of rETH minted
        uint256 rethBalance1 = rocketTokenRETH.balanceOf(address(this));
        rocketDepositPool.deposit{value: msg.value}();
        uint256 rethBalance2 = rocketTokenRETH.balanceOf(address(this));
        require(rethBalance2 > rethBalance1, "No rETH was minted");
        uint256 rethMinted = rethBalance2 - rethBalance1;
        // Transfer rETH to caller
        require(rocketTokenRETH.transfer(msg.sender, rethMinted), "rETH was not␣
↪transferred to caller");
    }

}
```

1.3 The Rocket Pool JavaScript Library

1.3.1 Introduction

The Rocket Pool JavaScript library is the primary means of interaction with the Rocket Pool network for users and applications. It is used by applications such as the Rocket Pool website to retrieve information about the network and facilitate user interaction. It provides wrapper methods which load the Rocket Pool contracts with web3.js and abstractions to use many of their features easily.

This guide assumes familiarity with JavaScript and provides all code samples as JS.

1.3.2 Contents

Getting Started

Installation

The Rocket Pool JavaScript library can be added to your application via NPM, and requires web3.js:

```
npm install github:rocket-pool/rocketpool-js
npm install web3
```

Initialization

The library must be initialized with a web3 instance and a Truffle `RocketStorage` contract artifact:

```
import Web3 from 'web3';
import RocketPool from 'rocketpool';
import RocketStorage from './contracts/RocketStorage.json';

const web3 = new Web3('http://localhost:8545');

const rp = new RocketPool(web3, RocketStorage);
```

Usage

The Rocket Pool library is divided into several modules, each for interacting with a different aspect of the network:

- `contracts`: Handles dynamic loading of the Rocket Pool contracts
- `deposit`: Handles user deposits
- `minipool`: Manages minipools in the Rocket Pool network
- `network`: Handles miscellaneous network functionality
- `node`: Manages the nodes making up the Rocket Pool network
- `settings.deposit`: Provides information on user deposit settings
- `settings.minipool`: Provides information on minipool settings
- `network`: Provides information on network settings
- `settings.node`: Provides information on smart node settings
- `tokens.neth`: Manages nETH token interactions
- `tokens.reth`: Manages rETH token interactions

All methods typically return promises due to the asynchronous nature of working with the Ethereum network. Getters return promises which resolve to their value, while mutators (methods which send transactions) return promises which resolve to a transaction receipt. Mutators also accept a transaction options object, and an `onConfirmation` callback handler to handle specific confirmation numbers on transactions.

When using the Rocket Pool library in your project, you may handle the promises returned in the traditional way, or use async/await syntax if supported, e.g.:

```
rp.contracts.get('rocketDepositPool')
    .then(rocketDepositPool => rocketDepositPool.methods.getBalance().call())
    .then(balance => { console.log(balance); });
```

or:

```
let rocketDepositPool = await rp.contracts.get('rocketDepositPool');
let balance = await rocketDepositPool.methods.getBalance().call();
console.log(balance);
```

Contracts

Overview

The `contracts` module loads Rocket Pool contract ABIs and addresses from `RocketStorage`, where all network contracts are registered. Contract ABIs and addresses are loaded from the chain, the ABIs are decompressed and decoded, and then web3 contract instances are created from them. This is performed dynamically because Rocket Pool contracts can be upgraded and their ABIs and addresses may change.

This module is used by other library modules internally, and generally does not need to be used directly. However, it is exposed publicly for when direct access to web3 contract instances is desired, or the library wrapper methods are insufficient.

Loading Contracts & ABIs

Network contracts can be loaded via the `contracts.get()` method, which accepts either a single contract name as a string, or a list of contract names as an array of strings. If a single contract name is passed, this method returns a promise resolving to a web3 contract instance. If a list of contract names is passed, it returns a promise resolving to an array of web3 contract instances, in the same order.

Contract addresses and ABIs can be loaded in a similar fashion via the `contracts.address()` and `contracts.abi()` methods, which accept either a single contract name, or a list of names, to retrieve data for.

Creating Contract Instances

Some network contracts, such as `RocketMinipool`, have multiple instances deployed at a number of different addresses. To create an instance of one of these contracts, use the `contracts.make(name, address)` method. It accepts the name of the contract and the address of the specific instance required, both as strings, and returns a promise resolving to a web3 contract instance.

Alternate Contract Versions

When Rocket Pool network contracts are upgraded, old versions remain on the chain and can still be accessed if required. A "contract version set", consisting of all versions of a contract by name, can be loaded with the `contracts.versions(name)` method. This method accepts the name of the contract to load, and returns a promise resolving to the version set object.

Contract version sets are primarily used for accessing old event data. They provide the following methods:

- `versionSet.current()`: Returns the current version of the contract
- `versionSet.first()`: Returns the first version of the contract deployed
- `versionSet.at(index)`: Returns the version of the contract at the specified version index (0 = first version)
- `versionSet.getPastEvents(eventName, options)`: As per web3's contract.getPastEvents, but returns a promise resolving to the events for all versions of the contract

Deposit

Overview

The `deposit` module is used to get the current deposit pool balance, make user deposits, and assign deposited ETH to minipools.

Methods

- `deposit.getBalance()`: Get the current balance of the deposit pool in wei; returns `Promise<string>`
- `deposit.deposit(options, onConfirmation)`: Make a user deposit; returns `Promise<TransactionReceipt>`
- `deposit.assignDeposits(options, onConfirmation)`: Assign deposited ETH to queued minipools; returns `Promise<TransactionReceipt>`

Minipool

Overview

The `minipool` module loads general minipool data from the chain. It also provides minipool contract functionality (which manages individual minipools and loads their data).

Data Types

`MinipoolDetails` objects contain various globally registered details of a minipool:

```
MinipoolDetails {
    address                  // The registered address of the minipool
    exists                   // Whether the minipool exists
    pubkey                   // The minipool's associated validator pubkey
    withdrawalTotalBalance   // The minipool's total validator balance at withdrawal
    withdrawalNodeBalance    // The node's share of the validator balance at withdrawal
    withdrawable             // Whether the minipool has become withdrawable yet
    withdrawalProcessed      // Whether the minipool's withdrawal has been processed␣
↪yet
}
```

`StatusDetails` objects define the current status of a minipool:

```
StatusDetails {
    status                   // The minipool's current status code
    block                    // The block at which the status was last changed
    time                     // The timestamp at which the status was last changed
}
```

`NodeDetails` objects contain details about the node owning a minipool:

```
NodeDetails {
    address                  // The address of the node owning the minipool
    fee                      // The node commission rate as a fraction of 1
    depositBalance           // The balance of ETH deposited by the node
```

(continues on next page)

(continued from previous page)

```
    refundBalance           // The balance of ETH available for refund to the node
    depositAssigned         // Whether the node's ETH deposit has been assigned
}
```

`UserDetails` objects contain details about user ETH assigned to a minipool:

```
UserDetails {
    depositBalance          // The balance of ETH from Rocket Pool user deposits
    depositAssigned         // Whether user-deposited ETH has been assigned
    depositAssignedTime     // The timestamp at which user-deposited ETH was assigned
}
```

`StakingDetails` objects contain details about a minipool's balance during staking:

```
StakingDetails {
    startBalance            // The minipool's balance when shared staking began
    endBalance              // The minipool's balance when shared staking finished
}
```

`MinipoolContract` objects wrap a web3 contract instance and provide methods for managing a minipool and retrieving its information.

Methods

Minipool Module:

- `minipool.getMinipools()`: Get the details of all minipools in the network; returns `Promise<MinipoolDetails[]>`
- `minipool.getMinipoolAddresses()`: Get the addresses of all minipools in the network; returns `Promise<string[]>`
- `minipool.getNodeMinipools(nodeAddress)`: Get the details of all minipools owned by a node; returns `Promise<MinipoolDetails[]>`
- `minipool.getNodeMinipoolAddresses(nodeAddress)`: Get the addresses of all minipools owned by a node; returns `Promise<string[]>`
- `minipool.getMinipoolDetails(address)`: Get the details of the specified minipool; returns `Promise<MinipoolDetails>`
- `minipool.getMinipoolCount()`: Get the total number of minipools in the network; returns `Promise<number>`
- `minipool.getMinipoolAt(index)`: Get the address of a minipool in the network by index; returns `Promise<string>`
- `minipool.getNodeMinipoolCount(nodeAddress)`: Get the total number of minipools owned by a node; returns `Promise<number>`
- `minipool.getNodeMinipoolAt(nodeAddress, index)`: Get the address of a minipool owned by a node by index; returns `Promise<string>`
- `minipool.getMinipoolByPubkey(validatorPubkey)`: Get the address of a minipool by its validator pubkey; returns `Promise<string>`
- `minipool.getMinipoolExists(address)`: Check whether the specified minipool exists; returns `Promise<boolean>`

- `minipool.getMinipoolPubkey(address)`: Get the specified minipool's validator pubkey; returns `Promise<string>`
- `minipool.getMinipoolWithdrawalTotalBalance(address)`: Get the specified minipool's total validator balance at withdrawal in wei; returns `Promise<string>`
- `minipool.getMinipoolWithdrawalNodeBalance(address)`: Get the node's share of the specified minipool's validator balance at withdrawal in wei; returns `Promise<string>`
- `minipool.getMinipoolWithdrawable(address)`: Get whether the specified minipool has become withdrawable yet; returns `Promise<boolean>`
- `minipool.getMinipoolWithdrawalProcessed(address)`: Get whether the specified minipool's withdrawal has been processed yet; returns `Promise<boolean>`
- `minipool.getQueueTotalLength()`: Get the total length of the minipool queue; returns `Promise<number>`
- `minipool.getQueueTotalCapacity()`: Get the total capacity of the minipool queue in wei; returns `Promise<string>`
- `minipool.getQueueEffectiveCapacity()`: Get the capacity of the minipool queue, ignoring "empty" minipools, in wei; returns `Promise<string>`
- `minipool.getQueueNextCapacity()`: Get the capacity of the next available minipool in the queue in wei; returns `Promise<string>`
- `minipool.getMinipoolNodeRewardAmount(nodeFee, userDepositBalance, startBalance, endBalance)`: Get the node reward amount for a minipool by staking details in wei; returns `Promise<string>`
- `minipool.getMinipoolContract(address)`: Get a MinipoolContract instance for the specified minipool; returns `Promise<MinipoolContract>`
- `minipool.submitMinipoolWithdrawable(minipoolAddress, stakingStartBalance, stakingEndBalance, options, onConfirmation)`: Submit a minipool's withdrawable status (watchtower nodes only); returns `Promise<TransactionReceipt>`

MinipoolContract:

- `MinipoolContract.getStatusDetails()`: Get the minipool's status details; returns `Promise<StatusDetails>`
- `MinipoolContract.getStatus()`: Get the minipool's status code; returns `Promise<number>`
- `MinipoolContract.getStatusBlock()`: Get the block at which the minipool's status was last changed; returns `Promise<number>`
- `MinipoolContract.getStatusTime()`: Get the time at which the minipool's status was last changed; returns `Promise<Date>`
- `MinipoolContract.getDepositType()`: Get the code for the type of node deposit assigned to the minipool; returns `Promise<number>`
- `MinipoolContract.getNodeDetails()`: Get the minipool's node details; returns `Promise<NodeDetails>`
- `MinipoolContract.getNodeAddress()`: Get the address of the node owning the minipool; returns `Promise<string>`
- `MinipoolContract.getNodeFee()`: Get the node commission rate for the minipool as a fraction of 1; returns `Promise<number>`

- `MinipoolContract.getNodeDepositBalance()`: Get the balance of ETH deposited to the minipool by the node in wei; returns `Promise<string>`
- `MinipoolContract.getNodeRefundBalance()`: Get the balance of ETH available for refund to the node in wei; returns `Promise<string>`
- `MinipoolContract.getNodeDepositAssigned()`: Get whether the node deposit has been assigned to the minipool; returns `Promise<boolean>`
- `MinipoolContract.getUserDetails()`: Get the minipool's user deposit details; returns `Promise<UserDetails>`
- `MinipoolContract.getUserDepositBalance()`: Get the balance of ETH deposited to the minipool by RP users in wei; returns `Promise<string>`
- `MinipoolContract.getUserDepositAssigned()`: Get whether RP user ETH has been assigned to the minipool; returns `Promise<boolean>`
- `MinipoolContract.getUserDepositAssignedTime()`: Get the time at which RP user ETH was assigned to the minipool; returns `Promise<Date>`
- `MinipoolContract.getStakingDetails()`: Get the minipool's staking details; returns `Promise<StakingDetails>`
- `MinipoolContract.getStakingStartBalance()`: Get the minipool's balance when staking begain in wei; returns `Promise<string>`
- `MinipoolContract.getStakingEndBalance()`: Get the minipool's balance when staking finished in wei; returns `Promise<string>`
- `MinipoolContract.dissolve(options, onConfirmation)`: Dissolve the prelaunch minipool and return its ETH to the node & deposit pool; returns `Promise<TransactionReceipt>`
- `MinipoolContract.refund(options, onConfirmation)`: Refund ETH owned by the node to the node account; returns `Promise<TransactionReceipt>`
- `MinipoolContract.stake(validatorPubkey, validatorSignature, depositDataRoot, options, onConfirmation)`: Stake the prelaunch minipool with the specified validator details; returns `Promise<TransactionReceipt>`
- `MinipoolContract.withdraw(options, onConfirmation)`: Withdraw the final balance & rewards from the withdrawable minipool and close it; returns `Promise<TransactionReceipt>`
- `MinipoolContract.close(options, onConfirmation)`: Close the dissolved minipool and refund its node ETH balance; returns `Promise<TransactionReceipt>`

Network

Overview

The `network` module loads miscellaneous network data from the chain. It also provides some watchtower node functionality.

Methods

- `network.getBalancesBlock()`: Get the block that current network balances are set for; returns `Promise<number>`

- `network.getTotalETHBalance()`: Get the total ETH value of the network in wei; returns `Promise<string>`
- `network.getStakingETHBalance()`: Get the total amount of actively staking ETH in the network in wei; returns `Promise<string>`
- `network.getTotalRETHSupply()`: Get the last recorded total rETH token supply in the network in wei; returns `Promise<string>`
- `network.getETHUtilizationRate()`: Get the proportion of ETH in the network which is actively staking, as a fraction of 1; returns `Promise<number>`
- `network.getNodeDemand()`: Get the current network node demand in wei; returns `Promise<string>`
- `network.getNodeFee()`: Get the current network node commission rate as a fraction of 1; returns `Promise<number>`
- `network.getNodeFeeByDemand(demand)`: Get the network node commission rate for a specified node demand value (in wei); returns `Promise<number>`
- `network.getWithdrawalBalance()`: Get the current ETH balance of the network withdrawal pool in wei; returns `Promise<string>`
- `network.getWithdrawalCredentials()`: Get the network-wide withdrawal credentials submitted for all validators; returns `Promise<string>`
- `network.submitBalances(block, totalEthWe, stakingEthWei, rethSupplyWei, options, onConfirmation)`: Submit network balances (watchtower nodes only); returns `Promise<TransactionReceipt>`
- `network.processWithdrawal(validatorPubkey, options, onConfirmation)`: Process a validator withdrawal (watchtower nodes only); returns `Promise<TransactionReceipt>`

Node

Overview

The `node` module manages nodes in the Rocket Pool network. It loads node data from the chain, and can be used to register new nodes and make node deposits.

Data Types

`NodeDetails` objects define the various details of a node:

```
NodeDetails {
    address             // The address of the node
    exists              // Whether the node is registered in the network
    trusted             // Whether the node is a trusted (watchtower) node
    timezoneLocation    // The node's timezone location
}
```

Methods

- `node.getNodes()`: Get the details of all nodes in the network; returns `Promise<NodeDetails[]>`
- `node.getNodeAddresses()`: Get the addresses of all nodes in the network; returns `Promise<string[]>`

- `node.getTrustedNodes()`: Get the details of all trusted nodes in the network; returns `Promise<NodeDetails[]>`
- `node.getTrustedNodeAddresses()`: Get the addresses of all trusted nodes in the network; returns `Promise<string[]>`
- `node.getNodeDetails(address)`: Get the details of the specified node; returns `Promise<NodeDetails>`
- `node.getNodeCount()`: Get the total number of nodes in the network; returns `Promise<number>`
- `node.getNodeAt(index)`: Get the address of a node in the network by index; returns `Promise<string>`
- `node.getTrustedNodeCount()`: Get the total number of trusted nodes in the network; returns `Promise<number>`
- `node.getTrustedNodeAt(index)`: Get the address of a trusted node in the network by index; returns `Promise<string>`
- `node.getNodeExists(address)`: Check whether the specified node is registered; returns `Promise<boolean>`
- `node.getNodeTrusted(address)`: Check whether the specified node is trusted; returns `Promise<boolean>`
- `node.getNodeTimezoneLocation(address)`: Get the timezone location of the specified node; returns `Promise<string>`
- `node.registerNode(timezoneLocation, options, onConfirmation)`: Register the calling address as a node in the network; returns `Promise<TransactionReceipt>`
- `node.setTimezoneLocation(timezoneLocation, options, onConfirmation)`: Update the timezone location of the calling node; returns `Promise<TransactionReceipt>`
- `node.deposit(minimumNodeFee, options, onConfirmation)`: Make a deposit to create a minipool, with a minimum acceptable commission rate; returns `Promise<TransactionReceipt>`

Settings

Overview

The `settings` module loads Rocket Pool network settings data from the chain, and is divided into 4 submodules:

- `settings.deposit`: Loads information on user deposit settings
- `settings.minipool`: Loads information on minipool settings
- `network`: Loads information on network settings
- `settings.node`: Loads information on smart node settings

Methods

Deposit Settings:

- `settings.deposit.getDepositEnabled()`: Get whether user deposits are currently enabled; returns `Promise<boolean>`
- `settings.deposit.getAssignDepositsEnabled()`: Get whether assignment of deposits to minipools is currently enabled; returns `Promise<boolean>`

- `settings.deposit.getMinimumDeposit()`: Get the minimum deposit amount in wei; returns `Promise<string>`
- `settings.deposit.getMaximumDepositPoolSize()`: Get the maximum size of the deposit pool in wei; returns `Promise<string>`
- `settings.deposit.getMaximumDepositAssignments()`: Get the maximum number of deposit assignments to perform per transaction; returns `Promise<number>`

Minipool Settings:

- `settings.minipool.getLaunchBalance()`: Get the required balance of a minipool for launch in wei; returns `Promise<string>`
- `settings.minipool.getFullDepositNodeAmount()`: Get the amount of eth in wei to be deposited by a node for a "full" deposit; returns `Promise<string>`
- `settings.minipool.getHalfDepositNodeAmount()`: Get the amount of eth in wei to be deposited by a node for a "half" deposit; returns `Promise<string>`
- `settings.minipool.getEmptyDepositNodeAmount()`: Get the amount of eth in wei to be deposited by a node for an "empty" deposit; returns `Promise<string>`
- `settings.minipool.getFullDepositUserAmount()`: Get the amount of eth in wei to be deposited by RP users for a "full" deposit; returns `Promise<string>`
- `settings.minipool.getHalfDepositUserAmount()`: Get the amount of eth in wei to be deposited by RP users for a "half" deposit; returns `Promise<string>`
- `settings.minipool.getEmptyDepositUserAmount()`: Get the amount of eth in wei to be deposited by RP users for an "empty" deposit; returns `Promise<string>`
- `settings.minipool.getSubmitWithdrawableEnabled()`: Get whether submission of minipool withdrawable status is enabled; returns `Promise<boolean>`
- `settings.minipool.getLaunchTimeout()`: Get the timeout period in blocks for minipools to launch within; returns `Promise<number>`
- `settings.minipool.getWithdrawalDelay()`: Get the delay in blocks before nodes can withdraw nETH from minipools; returns `Promise<number>`

Network Settings:

- `network.getNodeConsensusThreshold()`: Get the threshold of watchtower node submissions for consensus as a fraction of 1; returns `Promise<number>`
- `network.getSubmitBalancesEnabled()`: Get whether network balance submission is enabled; returns `Promise<boolean>`
- `network.getSubmitBalancesFrequency()`: Get the frequency at which network balances are submitted in blocks; returns `Promise<number>`
- `network.getProcessWithdrawalsEnabled()`: Get whether processing validator withdrawals is enabled; returns `Promise<boolean>`
- `network.getMinimumNodeFee()`: Get the minimum node commission rate as a fraction of 1; returns `Promise<number>`
- `network.getTargetNodeFee()`: Get the target node commission rate as a fraction of 1; returns `Promise<number>`
- `network.getMaximumNodeFee()`: Get the maximum node commission rate as a fraction of 1; returns `Promise<number>`

- `network.getNodeFeeDemandRange()`: Get the range of node demand values in wei to base fee calculations on; returns `Promise<string>`
- `network.getTargetRethCollateralRate()`: Get the target rETH contract collateral rate as a fraction of 1; returns `Promise<number>`

Node Settings:

- `settings.node.getRegistrationEnabled()`: Get whether node registrations are currently enabled; returns `Promise<boolean>`
- `settings.node.getDepositEnabled()`: Get whether node deposits are currently enabled; returns `Promise<boolean>`

Tokens

Overview

The `tokens` module manages the various Rocket Pool tokens, and is broken down into two submodules:

- `tokens.neth`: Manages rETH token interactions
- `tokens.reth`: Manages rETH token interactions

Each submodule shares common methods for interacting with its underlying ERC-20 token. Mutator methods are restricted to their respective accounts.

Methods

All Tokens:

- `tokens.[token].balanceOf(account)`: Get the token balance of the specified account (address) in wei; returns `Promise<string>`
- `tokens.[token].allowance(account, spender)`: Get the allowance of the specified account, for the specified spender (addresses) in wei; returns `Promise<string>`
- `tokens.[token].transfer(to, amount, options, onConfirmation)`: Transfer the specified amount of tokens in wei to the 'to' address; returns `Promise<TransactionReceipt>`
- `tokens.[token].approve(spender, amount, options, onConfirmation)`: Approve an allowance of the specified amount in wei for the specified spender (address); returns `Promise<TransactionReceipt>`
- `tokens.[token].transferFrom(from, to, amount, options, onConfirmation)`: Transfer the specified amount of tokens in wei from the 'from' address to the 'to' address; returns `Promise<TransactionReceipt>`

nETH Token:

- `tokens.neth.burn(amount, options, onConfirmation)`: Burn the specified amount of nETH in wei for ETH; returns `Promise<TransactionReceipt>`

rETH Token:

- `tokens.reth.getEthValue(rethAmount)`: Get the amount of ETH in wei backing an amount of rETH in wei; returns `Promise<string>`
- `tokens.reth.getRethValue(ethAmount)`: Get the amount of rETH in wei backed by an amount of ETH in wei; returns `Promise<string>`

- `tokens.reth.getExchangeRate()`: Get the amount of ETH backing 1 rETH; returns `Promise<number>`
- `tokens.reth.getTotalCollateral()`: Get the total amount of ETH collateral available for exchage in wei; returns `Promise<string>`
- `tokens.reth.getCollateralRate()`: Get the proportion of rETH backed by ETH collateral in the contract as a fraction of 1; returns `Promise<number>`
- `tokens.reth.burn(amount, options, onConfirmation)`: Burn the specified amount of rETH in wei for ETH; returns `Promise<TransactionReceipt>`

1.4 The Rocket Pool Smart Node Stack

1.4.1 Introduction

The Rocket Pool smart node software stack provides all of the necessary infrastructure for running a smart node in the Rocket Pool network. The software stack consists of two main components:

- The smart node client, which provides a command-line interface for managing a smart node either locally or remotely (over SSH)
- The smart node service, which provides an API for client communication and performs background node tasks (such as validator duties)

The smart node service consists of a number of Docker containers, in order to ensure consistent behavior across different systems.

1.4.2 Contents

Getting Started

OS & Hardware Requirements

The smart node client is supported on Linux, MacOS and Windows. **Note that a smart node cannot be run locally on Windows at this stage; the Windows client can only be used to manage a remote server.**

The smart node service is supported on AMD64 architecture and all Unix platforms, with automatic OS dependency installation for Ubuntu, Debian, CentOS and Fedora. **OS dependencies (docker engine and docker-compose) must be installed manually on all other Unix platforms.**

Support for additional architectures (e.g. ARM) and operating systems will be added incrementally, after successful testing of the existing version.

The Smart Node service requires at least 16GB of memory and 200GB of (SSD) hard disk space in order to run. Note that a node operator must have **root** access to their node in order to install and run the smart node service.

Installation

Firstly, install the smart node client locally. For Linux & MacOS, run either the cURL or wget command depending on which utilities are installed on your system. You can check with `curl --version` and `wget --version` respectively.

Linux (64 bit):

With cURL:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-linux-amd64 --create-dirs -o ~/bin/rocketpool && chmod +x ~/bin/
↪rocketpool
```

With wget:

```
mkdir -p ~/bin && wget https://github.com/rocket-pool/smartnode-install/releases/
↪latest/download/rocketpool-cli-linux-amd64 -O ~/bin/rocketpool && chmod +x ~/bin/
↪rocketpool
```

Note: You may need to start a new shell session before the `rocketpool` **command is available to use.**

MacOS (64 bit):

With cURL:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-darwin-amd64 -o /usr/local/bin/rocketpool && chmod +x /usr/local/bin/
↪rocketpool
```

With wget:

```
wget https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-darwin-amd64 -O /usr/local/bin/rocketpool && chmod +x /usr/local/bin/
↪rocketpool
```

Windows (64 bit):

1. Download the smart node client.
2. Move it to the desired location on your system (e.g. `C:\bin\rocketpool.exe`).
3. Open the command prompt and run it via its full path (e.g. `C:\bin\rocketpool.exe`).

Secondly, install the smart node service either locally or on a remote server. To install locally, simply run `rocketpool service install`. To install remotely, provide flags for the remote host address, username, and SSH identity file, e.g.:

```
rocketpool --host example.com --user username --key /path/to/identity.pem service␣
↪install
```

If automatic dependency installation is not supported on your platform (or if you would prefer to install OS dependencies yourself), use the `-d` option to skip this step (e.g. `rocketpool service install -d`). Then, manually install docker engine and docker-compose.

The following installation options are available:

- `-r`: Verbose mode (print all output from the installation process)
- `-d`: Skip automatic installation of OS dependencies
- `-n`: Specify a network to run the smart node on (default: medalla)
- `-v`: Specify a version of the smart node service package files to use (default: latest)

Once the smart node service has been installed, you may need to start a new shell session if working locally. This is required for updated user permissions to take effect (for interacting with docker engine).

Configuration

Once the smart node service is installed, it must be configured before use. Simply run `rocketpool service config` and follow the prompts to select which Eth 1.0 and Eth 2.0 clients to run in the smart node stack.

You may use Infura rather than run a full Eth 1.0 client if desired. If you do, you will need to create an account and set up a new project to obtain a project ID. Note that Infura will limit requests after a certain threshold, so uptime is not guaranteed.

By default, the smart node will select a random Eth 2.0 client to run, in order to increase network client diversity. You may, however, select a specific client to run if you prefer.

Upgrading Your Smart Node

Rocket Pool smart nodes are not upgraded automatically, as ethereum client updates **may contain breaking changes** and prevent validators from working. As such, node operators need to be aware of Rocket Pool smart node updates, and apply them manually. Node operators are encouraged to read the changelogs of the relevant repositories.

Upgrading the Smart Node Client

The smart node client can be upgraded simply by downloading a new version of the binary and replacing the existing version with it. For Linux & MacOS, run either the cURL or wget command depending on which utilities are installed on your system. You can check with `curl --version` and `wget --version` respectively.

Linux (64 bit):

With cURL:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-linux-amd64 --create-dirs -o ~/bin/rocketpool && chmod +x ~/bin/
↪rocketpool
```

With wget:

```
mkdir -p ~/bin && wget https://github.com/rocket-pool/smartnode-install/releases/
↪latest/download/rocketpool-cli-linux-amd64 -O ~/bin/rocketpool && chmod +x ~/bin/
↪rocketpool
```

MacOS (64 bit):

With cURL:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-darwin-amd64 -o /usr/local/bin/rocketpool && chmod +x /usr/local/bin/
↪rocketpool
```

With wget:

```
wget https://github.com/rocket-pool/smartnode-install/releases/latest/download/
↪rocketpool-cli-darwin-amd64 -O /usr/local/bin/rocketpool && chmod +x /usr/local/bin/
↪rocketpool
```

Windows (64 bit):

1. Download the smart node client.
2. Overwrite your existing client executable with it (e.g. `C:\bin\rocketpool.exe`).

Upgrading the Smart Node Service

Before upgrading the smart node service, if you have made any customizations to your service configuration files, back them up:

```
cp -r ~/.rocketpool ~/.rocketpool.bak
```

Any changes to these files will be overwritten, so you will need to restore them manually after updating.

Next, pause the service before making changes to it:

```
rocketpool service pause
```

Then, upgrade the service configuration files with:

```
rocketpool service install -d
```

You may optionally specify a version of the Rocket Pool smart node service to use, e.g.:

```
rocketpool service install -d -v 0.0.3
```

Once you've upgraded the service configuration files, restore any customizations you made previously. Then, start the service back up:

```
rocketpool service start
```

Post-Upgrade Tasks

Once you have upgraded the smart node client and/or service, check you are running the correct versions with:

```
rocketpool service version
```

In some cases, you may need to rebuild your validator keystores (e.g. if the Eth 2.0 client you are using has updated wallet functionality). If in doubt, you can always do this with no risk to your existing validator keys. Once your Eth 1.0 client has finished re-syncing, rebuild your validator keystores with:

```
rocketpool wallet rebuild
```

You can check to see if your validator keys have been loaded correctly with:

```
rocketpool service logs validator
```

Always ensure that your validator container has loaded the keys for each of your minipools.

Customizing the Rocket Pool Service

This section describes how to customize the Rocket Pool service, and is intended for advanced users with custom setups. If you're happy to run your smart node as provided "out of the box" by Rocket Pool, skip ahead.

All examples given below assume you are working locally on your smart node. If you manage your node remotely, SSH into it before running any commands.

Customizing Storage Location

By default, chain data for your Eth 1.0 and Eth 2.0 clients will be stored in persistent volumes created by Docker. These volumes are managed by Docker and are usually stored on your primary partition.

If you would like to change the location at which your chain data is stored (for example, to store it on a different drive), you may instead mount local filesystem paths to your `eth1` and `eth2` containers.

Do not mount an existing chain data directory to your client container if it is still in use by a running process. **Never share a chain database between multiple processes, as this will result in a corrupted database**.

1. If the Rocket Pool service is already running, stop it with:

   ```
   rocketpool service terminate
   ```

2. Open `~/.rocketpool/docker-compose.yml`, and modify the `services.eth1` and `services.eth2` sections as follows:

   - Change `eth1clientdata:/ethclient` to `/path/to/eth1/storage:/ethclient` (example only) to set a local filesystem path for your Eth 1.0 chain database
   - Change `eth2clientdata:/ethclient` to `/path/to/eth2/storage:/ethclient` (example only) to set a local filesystem path for your Eth 2.0 chain database

3. Modify the `volumes` section at the bottom of the file as follows:

   - Remove the `eth1clientdata:` line if you set a custom path for your Eth 1.0 chain database
   - Remove the `eth2clientdata:` line if you set a custom path for your Eth 2.0 chain database
   - Remove the `volumes` section entirely if it's empty

4. Restart the Rocket Pool service with:

   ```
   rocketpool service start
   ```

Using External Eth 1.0 and Eth 2.0 Clients

By default, the Rocket Pool service will run its own Eth 1.0 (Geth) and Eth 2.0 (Lighthouse / Nimbus / Prysm / Teku) clients. However, you may already have your own clients running on your host OS which you want to configure Rocket Pool to communicate with. Note that you should still run the validator process provided by Rocket Pool, as the service performs its own key management and loads validator keys into it.

To configure Rocket Pool to use external Eth 1.0 and/or Eth 2.0 clients:

1. Configure your router's DHCP settings to lease a static IP address to your machine
2. Reconnect to your network, then find your machine's local IP address with `ifconfig`
3. Ensure your Eth 1.0 and/or Eth 2.0 clients are listening on the address `0.0.0.0`:

   - Geth: `--http --http.addr 0.0.0.0 --http.port 8545 --http.vhosts *`
   - Lighthouse: `--http --http-address 0.0.0.0 --http-port 5052`
   - Nimbus: `--rpc --rpc-address 0.0.0.0 --rpc-port 5052`
   - Prysm: `--rpc-host 0.0.0.0 --rpc-port 5052`
   - Teku: `--rest-api-enabled --rest-api-interface 0.0.0.0 --rest-api-port 5052`

4. If the Rocket Pool service is already running, pause it with:

```
rocketpool service stop
```

5. Open `~/.rocketpool/docker-compose.yml`, and modify the `services` section as follows:
   - If you want to use your own Geth instance, remove the `eth1` section, then remove all `- eth1` entries under `depends_on:` sections
   - If you want to use your own Lighthouse, Nimbus, Prysm or Teku instance, remove the `eth2` section, then remove all `- eth2` entries under `depends_on:` sections
   - Remove any `depends_on:` sections which are empty
6. Open `~/.rocketpool/config.yml`, and make the following changes:
   - To use your own Geth instance, update `chains.eth1.provider` to `http://XXX.XXX.XXX.XXX:8545`, where `XXX.XXX.XXX.XXX` is your machine's local IP address
   - To use your own Lighthouse, Nimbus, Prysm or Teku instance, update `chains.eth2.provider` to `XXX.XXX.XXX.XXX:5052`, where `XXX.XXX.XXX.XXX` is your machine's local IP address
7. Configure the Rocket Pool service, selecting Geth for your Eth 1.0 client, and the appropriate Eth 2.0 client:

   ```
   rocketpool service config
   ```

8. Restart the Rocket Pool service with:

   ```
   rocketpool service start
   ```

Customizing Eth 1.0 and Eth 2.0 Client Options

The Eth 1.0 and Eth 2.0 clients are bootstrapped via shell scripts at the following locations:

- Eth 1.0 Client: `~/.rocketpool/chains/eth1/start-node.sh`
- Eth 2.0 Beacon Chain: `~/.rocketpool/chains/eth2/start-beacon.sh`
- Eth 2.0 Validator: `~/.rocketpool/chains/eth2/start-validator.sh`

To customize the command-line options passed to the clients:

1. If the Rocket Pool service is already running, pause it with:

   ```
   rocketpool service stop
   ```

2. Modify the above files as desired
3. Restart the Rocket Pool service with:

   ```
   rocketpool service start
   ```

Please consult the documentation for specific Eth 1.0 and Eth 2.0 clients for a full list of command-line options.

Running the Rocket Pool Service Outside Docker

Users with advanced setups may wish to run the Rocket Pool service on their host OS, outside of a Docker environment. Note that when using this method, users must install and manage their own Eth 1.0 and Eth 2.0 beacon & validator clients. The Rocket Pool service daemons must also be manually registered with the OS service manager (e.g. systemd).

The following commands are unavailable when running the Rocket Pool service outside of Docker:

- `rocketpool service status`
- `rocketpool service start`
- `rocketpool service pause`
- `rocketpool service stop`
- `rocketpool service terminate`
- `rocketpool service logs`
- `rocketpool service stats`

To run the Rocket Pool service on your host OS, follow these steps:

1. Install Go
2. Clone the Rocket Pool smartnode repository and checkout the tag for the desired version:

   ```
   git clone https://github.com/rocket-pool/smartnode.git
   cd smartnode
   git checkout vX.X.X
   ```

3. Build the Rocket Pool CLI client from source and install it:

   ```
   cd rocketpool-cli
   go build rocketpool-cli.go
   sudo mv rocketpool-cli /usr/local/bin/rocketpool
   ```

4. Build the Rocket Pool service daemon from source and install it:

   ```
   cd ../rocketpool
   go build rocketpool.go
   sudo mv rocketpool /usr/local/bin/rocketpoold
   ```

5. Install the smart node service files for the desired version, skipping OS dependency (Docker) installation:

   ```
   rocketpool service install -v X.X.X -d
   ```

6. Optionally, delete the following unused files (for Docker setups only):
   - `~/.rocketpool/docker-compose.yml`
   - `~/.rocketpool/chains/*`
7. Make the following modifications to your Rocket Pool config file (`~/.rocketpool/config.yml`):
   - Update `smartnode.passwordPath` to e.g. `/home/[USERNAME]/.rocketpool/data/password`
   - Update `smartnode.walletPath` to e.g. `/home/[USERNAME]/.rocketpool/data/wallet`
   - Update `smartnode.validatorKeychainPath` to e.g. `/home/[USERNAME]/.rocketpool/data/validators`
   - Update `chains.eth1.provider` to e.g. `http://127.0.0.1:8545`
   - Update `chains.eth2.provider` to e.g. `127.0.0.1:5052` (if using Prysm, use the port for gRPC)
8. Configure the Rocket Pool service, selecting Geth for your Eth 1.0 client, and the appropriate Eth 2.0 client:

   ```
   rocketpool service config
   ```

9. Register the following services with your operating system (example systemd units for some clients are provided below):
   - Geth
   - Lighthouse / Nimbus / Prysm / Teku beacon chain
   - Lighthouse / Prysm / Teku validator
   - `/usr/local/bin/rocketpoold node`
   - `/usr/local/bin/rocketpoold watchtower`
10. Add the following alias to your `.profile` (or `.bash_profile` / `.zprofile` as appropriate) and start a new shell session:
    - `alias rp="rocketpool -d /usr/local/bin/rocketpoold"`
11. Use the alias `rp` to interact with the Rocket Pool service, e.g.: `rp node status`

Example systemd Units for Rocket Pool Services

Geth:

```
[Unit]
Description=Geth
After=network.target

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/path/to/geth --goerli --http --http.addr 127.0.0.1 --http.port 8545 --http.
↪api eth,net,personal,web3 --http.vhosts *

[Install]
WantedBy=multi-user.target
```

Lighthouse - Beacon Chain:

```
[Unit]
Description=Lighthouse Beacon
After=geth.service

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/path/to/lighthouse beacon --testnet medalla --eth1 --eth1-endpoint http://
↪127.0.0.1:8545 --http --http-address 127.0.0.1 --http-port 5052

[Install]
WantedBy=multi-user.target
```

Lighthouse - Validator:

```
[Unit]
Description=Lighthouse Validator
After=lighthouse-beacon.service
```

(continues on next page)

(continued from previous page)

```
[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/path/to/lighthouse validator --testnet medalla --datadir /home/[USERNAME]/.
↪rocketpool/data/validators/lighthouse --init-slashing-protection --delete-lockfiles␣
↪--beacon-node http://127.0.0.1:5052

[Install]
WantedBy=multi-user.target
```

Prysm - Beacon Chain:

```
[Unit]
Description=Prysm Beacon
After=geth.service

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/path/to/prysm/beacon-chain --accept-terms-of-use --medalla --http-
↪web3provider http://127.0.0.1:8545 --rpc-host 127.0.0.1 --rpc-port 5052

[Install]
WantedBy=multi-user.target
```

Prysm - Validator:

```
[Unit]
Description=Prysm Validator
After=prysm-beacon.service

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/path/to/prysm/validator --accept-terms-of-use --medalla --wallet-dir /home/
↪[USERNAME]/.rocketpool/data/validators/prysm-non-hd --wallet-password-file /home/
↪[USERNAME]/.rocketpool/data/password --beacon-rpc-provider 127.0.0.1:5052

[Install]
WantedBy=multi-user.target
```

Rocket Pool Node Daemon:

```
[Unit]
Description=Rocketpool Node
After=geth.service

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/usr/local/bin/rocketpoold --config /home/[USERNAME]/.rocketpool/config.yml␣
↪--settings /home/[USERNAME]/.rocketpool/settings.yml node
```

(continues on next page)

(continued from previous page)

```

[Install]
WantedBy=multi-user.target
```

Rocket Pool Watchtower Daemon:

```
[Unit]
Description=Rocketpool Watchtower
After=geth.service

[Service]
Type=simple
Restart=always
RestartSec=5
ExecStart=/usr/local/bin/rocketpoold --config /home/[USERNAME]/.rocketpool/config.yml␣
↪--settings /home/[USERNAME]/.rocketpool/settings.yml watchtower

[Install]
WantedBy=multi-user.target
```

The Rocket Pool Service

Starting the Service

Start the Rocket Pool service by running:

```
rocketpool service start
```

This will "build up" the smart node stack, which runs it and also ensures that it stays running. If any of the running containers crash or you restart your node, Docker will start them back up to ensure that no uptime is lost.

You can check that the containers are running correctly with:

```
rocketpool service status
```

You should see (in any order) containers with the following names:

- `rocketpool_eth1`
- `rocketpool_eth2`
- `rocketpool_validator`
- `rocketpool_api`
- `rocketpool_node`
- `rocketpool_watchtower`

Pausing the Service

If you want to pause the Rocket Pool service for any reason, run:

```
rocketpool service stop
```

This will stop all running containers, suspending their execution, but leave them intact. Note that this will stop validators from performing their validation duties, so use this command with caution. The service can be started up again with `rocketpool service start`.

Stopping the Service

If you have finished interacting with the Rocket Pool network and want to stop the service entirely, run:

```
rocketpool service terminate
```

This will "tear down" the smart node stack, stopping and removing all running containers, and deleting their state. Not only will validators stop performing validation duties, but all Ethereum clients will need to re-sync if the service is restarted. It is advised to use this command only if the node has no active minipools.

As a security measure, node data at `~/.rocketpool` (including the node wallet) will be preserved, and must be manually deleted if desired (this is not recommended).

Reconfiguring the Service

If you want to make any configuration changes to the Rocket Pool service, run:

```
rocketpool service config
```

This will repeat the configuration process performed after installation, and will overwrite your node's configuration file accordingly. For the changes to take effect, restart the Rocket Pool service with `rocketpool service start`.

Viewing Service Information

You can check the version of the CLI client and the service with:

```
rocketpool service version
```

You can check the current status of the service (its running containers) with:

```
rocketpool service status
```

You can view the logs for all running containers in real-time with:

```
rocketpool service logs
```

To view the logs for a single container, add its name at the end, e.g.:

```
rocketpool service logs eth2
```

Press Ctrl-C to stop.

You can also view the hardware usage for each container with:

```
rocketpool service stats
```

Press Ctrl-C to stop.

Forwarding Service Ports

Optionally, you can forward ports for Eth 1.0 and Eth 2.0 client peer discovery to the Rocket Pool service containers. This may allow you to connect to more peers, and can potentially increase sync times. The method for forwarding ports depends on your local network setup and is not covered in this document. The port ranges to forward are:

- **Eth 1.0**: `30303 (TCP & UDP)`
- **Eth 2.0**: `9001 (TCP & UDP)`

Node Setup, Registration & Management

Initializing Your Node Wallet

With the Rocket Pool service running, the first thing you'll need to do is initialize your node wallet:

```
rocketpool wallet init
```

This will prompt you to enter a node password, and will then save it to disk. You won't need to enter your node password again, it will simply be used by the smart node to unlock your wallet.

Next, a HD wallet will be generated to store your node account and all validator keys for your minipools. You will be shown a mnemonic phrase to recover your wallet in case of hardware failure, and prompted to record it and enter it to confirm it is correct. You can make sure your wallet was created successfully with:

```
rocketpool wallet status
```

Your node password is stored at `~/.rocketpool/data/password`, while your wallet is stored at `~/.rocketpool/data/wallet`. You can export the contents of both of these files and display them on screen with:

```
rocketpool wallet export
```

Feel free to back these up in a safe and secure storage area which can't be accessed by anyone else. You do not need to back your wallet up repeatedly, even after creating new minipools.

Recovering Your Node Wallet

If you lose your node wallet, you can recover it with:

```
rocketpool wallet recover
```

This will prompt you to enter the recovery mnemonic phrase for your wallet. If successful, the wallet will be restored along with your node account and all validator keys for created minipools.

Seeding Your Node Account

Next, you'll need to load your node account up with ETH and RPL to deposit into Rocket Pool. Find your node address with:

```
rocketpool node status
```

If you're participating in a testnet beta, you can obtain GoETH from one of the following faucets:

- ethstaker discord

- faucet.goerli.mudit.blog

Once you have some GoETH, you can also request RPL from a faucet directly from the CLI:

```
rocketpool faucet withdraw-rpl
```

Note that this requires a 0.5 GoETH fee to prevent abuse.

After requesting your GoETH and RPL, check your node status again to ensure your balances have increased. Note that the RPL you received will be referred to as "old RPL", which can be swapped for the new RPL token.

Registering Your Node

Once you have some ETH, register with:

```
rocketpool node register
```

You will be prompted to either detect your timezone location automatically, or enter it manually. This information is not used for KYC purposes, but is sent to Rocket Pool during registration in order to display accurate node information to users. You may abstain by manually entering a location such as `Hidden/Hidden`.

Once you've registered successfully, you can check your status with:

```
rocketpool node status
```

This should now display additional information like: `The node is registered with Rocket Pool with a timezone location of Australia/Brisbane`.

Updating Your Registration

If you want to set a withdrawal address which all node rewards & refunds will be sent to, run:

```
rocketpool node set-withdrawal-address [address]
```

If you want to update the timezone your node is registered in, run:

```
rocketpool node set-timezone
```

This will repeat the prompts run during registration, and update your node's information in the network.

Sending From Your Node Account

If you want to send ETH or tokens from your node account to another Ethereum address at any time, use:

```
rocketpool node send [amount] [token] [to-address]
```

This will send the specified amount of ETH or nETH from the node account to the specified address.

Staking RPL

Swapping old RPL for new RPL

The balances displayed when you view your node's status include any amounts of "old RPL" under the node account. This old RPL can be swapped for the new RPL token, which can be staked with Rocket Pool, allowing you to create minipools:

```
rocketpool node swap-rpl
```

You can either swap your entire old RPL balance, or a portion of it if you prefer. Note that you can also skip this step, and will be prompted to swap any old RPL when you try to stake it.

Checking the RPL Price

Before staking RPL with Rocket Pool, you may wish to view the current RPL price recorded by the Rocket Pool network contracts. The RPL price is reported by oracle nodes at set intervals, using data aggregated from a number of decentralized exchanges. Check the current RPL price with:

```
rocketpool network rpl-price
```

This will display the current recorded RPL price, along with the block it was last updated at. The price will affect the minimum RPL stake, and maximum "effective stake", per minipool.

Staking RPL with Rocket Pool

Staking RPL with Rocket Pool allows you to create minipools and earn both eth2 & RP staking rewards. Nodes must stake a minimum amount of RPL for each minipool they wish to run. You can stake as much RPL as you like, but your "effective stake" will be limited based on the number of minipools you are running. In both cases, the amounts vary depending on the current price of the RPL token recorded by the Rocket Pool network contracts.

Stake RPL with:

```
rocketpool node stake-rpl
```

You will be prompted to select one of the following options:

- Stake the minimum amount of RPL required to run one minipool
- Stake the maximum amount of RPL that will count towards your "effective stake" for one minipool
- Stake your entire RPL balance
- Stake a custom amount of RPL

Withdrawing Staked RPL

You can withdraw your staked RPL after a cooldown period has passed (since you last staked any amount):

```
rocketpool node withdraw-rpl
```

You will be prompted to select one of the following options:

- Withdraw the maximum amount of RPL possible
- Withdraw a custom amount of RPL

Note that you must leave a minimum amount of RPL staked to cover any running minipools you have. As such, you won't be able to withdraw your entire RPL balance unless you have no minipools left.

Making Deposits

Checking the Node Commission Rate

Before making a deposit, you may wish to view the current Rocket Pool network node commission rate. The commission rate varies depending on network node supply & demand dynamics, and changes as user deposits are made and minipools are created. Check the current node commission rate with:

```
rocketpool network node-fee
```

This will display the current rate, along with the minimum and maximum rates possible. If you're happy with the current rate, you can make a deposit to create a minipool and start validating.

Making a Deposit

You can make a deposit with:

```
rocketpool node deposit
```

You will then be prompted to select an amount of ETH to deposit. 16 ETH deposits create minipools which must wait for user-deposited ETH to be assigned to them before they begin staking. 32 ETH deposits create minipools which can begin staking immediately, and will have the excess 16 ETH refunded once they are assigned to.

Next, you will be shown the current network node commission rate and prompted to enter a minimum commission rate you will accept. You may either use the suggested value based on the data provided, or enter a custom one. If the network node commission rate drops below this threshold before your deposit transaction is mined, the deposit will be cancelled.

If the deposit is made successfully, the address of the newly created minipool will be displayed.

Managing Minipools

Checking Minipool Status

Once you have made one or more deposits from your node, you can view the status of your created minipools with:

```
rocketpool minipool status
```

This will list various properties of each minipool created by your node, including:

- Its address
- Its current status, and the time & block number it was last updated at
- The node commission rate on rewards earned by it
- The amount of ETH deposited by the node operator
- The amount of user-deposited ETH assigned, and the time it was assigned at
- The associated validator's public key

You will also be notified if any of your minipools have ETH available for refund or withdrawal.

Refunding From Minipools

If you have made any deposits of 32 ETH, the created minipools will have 16 ETH available for refund once user-deposited ETH is assigned to them. You can refund this ETH to your node account with:

```
rocketpool minipool refund
```

This will display a list of all eligible minipools, and prompt you to select one or all of them to refund your ETH from. Once refunded, you should see their balances reflected in your node account.

Exiting Minipools

Once you're ready to finish staking, you can exit your minipool validators from the beacon chain with:

```
rocketpool minipool exit
```

This will display a list of all eligible minipools, and prompt you to select one or all of them to exit. When you successfully exit a minipool, it can take several hours for its status to be reflected by your node. It can also take longer for it to be marked as withdrawable by the Rocket Pool network and for nETH to be minted to it for withdrawal.

Withdrawing From Minipools

If any of your minipools have exited and been marked as withdrawable by the Rocket Pool network, you can withdraw your deposit & rewards from them with:

```
rocketpool minipool withdraw
```

This will display a list of all eligible minipools, and prompt you to select one or all of them to withdraw from. Once withdrawn, the minipool/s will be destroyed, and you should see their balances reflected in your node account.

Note that before phase 2 of the Eth 2.0 rollout, rewards can only be withdrawn from exited minipools after a significant delay.

Dissolving Minipools

If you create a minipool and decide you want to back out before it begins staking, you can do so with:

```
rocketpool minipool dissolve
```

This will display a list of all minipools which do not yet have user-deposited ETH assigned to them. You will be prompted to select one or all of them to dissolve, returning your ETH deposit to your node account. Once dissolved, the minipool/s will be destroyed, and you should see their balances reflected in your node account.

If you create a minipool and it fails to stake within a set time period after user-deposited ETH is assigned to it, it may be dissolved by another party. This returns the user-deposited ETH to the deposit pool to be reassigned. If this occurs, you can close the dissolved minipools with:

```
rocketpool minipool close
```

This will display a list of all eligible minipools, and prompt you to select one or all of them to close. Once closed, the minipool/s will be destroyed, and you should see their balances reflected in your node account.

Command Reference

Service Commands

- `rocketpool service install`: Install the Rocket Pool service either locally or to a remote server
- `rocketpool service config`: Configure the Rocket Pool service and select Eth 1.0 and Eth 2.0 clients
- `rocketpool service status`: Display the current status of the Rocket Pool service
- `rocketpool service start`: Start the Rocket Pool service to begin running a smart node
- `rocketpool service pause`: Pause the Rocket Pool service temporarily
- `rocketpool service stop`: Pause the Rocket Pool service temporarily
- `rocketpool service terminate`: Terminate the Rocket Pool service and remove all associated docker containers & volumes
- `rocketpool service logs [services...]`: View the logs for one or more services running as part of the docker stack
- `rocketpool service stats`: Display resource usage statistics for the Rocket Pool service
- `rocketpool service version`: Display version information for the Rocket Pool client & service

Wallet Commands

- `rocketpool wallet status`: Display the current status of the node's wallet
- `rocketpool wallet init`: Initialize the node's password and wallet
- `rocketpool wallet recover`: Recover a node wallet from a mnemonic phrase
- `rocketpool wallet rebuild`: Rebuild validator keystores from derived keys
- `rocketpool wallet export`: Display the node password and wallet file contents

Node Commands

- `rocketpool node status`: Display the current status of the node
- `rocketpool node register`: Register the node with the Rocket Pool network
- `rocketpool node set-withdrawal-address [address]`: Set the address which node rewards & refunds are sent to
- `rocketpool node set-timezone`: Update the node's timezone location
- `rocketpool node swap-rpl`: Swap old RPL tokens for new RPL
- `rocketpool node stake-rpl`: Stake RPL against the node to collateralize minipools
- `rocketpool node withdraw-rpl`: Withdraw RPL staked against the node
- `rocketpool node deposit`: Make a deposit to create a minipool and begin staking
- `rocketpool node send [amount] [token] [to]`: Send an amount of ETH or tokens to an address
- `rocketpool node burn [amount] [token]`: Burn reward tokens for ETH

Minipool Commands

- `rocketpool minipool status`: Display the current status of all minipools run by the node
- `rocketpool minipool refund`: Refund ETH from minipools which have had user-deposited ETH assigned to them
- `rocketpool minipool dissolve`: Dissolve initialized minipools and recover deposited ETH from them
- `rocketpool minipool exit`: Exit active minipool validators from the beacon chain
- `rocketpool minipool withdraw`: Withdraw rewards from minipools which have finished staking and close them
- `rocketpool minipool close`: Close minipools which have timed out and been dissolved

Auction Commands

- `rocketpool auction status`: Display the current status of the RPL auction contract and lots
- `rocketpool auction lots`: Display the details of all RPL lots
- `rocketpool auction create-lot`: Create a new RPL lot from RPL in the auction contract
- `rocketpool auction bid-lot`: Bid ETH on an active RPL lot
- `rocketpool auction claim-lot`: Clean RPL from a cleared lot you bid on
- `rocketpool auction recover-lot`: Recover unclaimed RPL from a cleared lot back to the auction contract

Oracle DAO Commands

- `rocketpool odao status`: Display the current status of the oracle DAO
- `rocketpool odao members`: Display the details of all oracle DAO members
- `rocketpool odao proposals`: Display the details of all oracle DAO proposals
- `rocketpool odao propose-invite [address] [id] [email]`: Invite a member to join the oracle DAO
- `rocketpool odao propose-leave`: Propose leaving the oracle DAO
- `rocketpool odao propose-replace [address] [id] [email]`: Propose replacing your position in the oracle DAO with a new member
- `rocketpool odao propose-kick`: Propose kicking a member from the oracle DAO
- `rocketpool odao cancel-proposal`: Cancel a proposal you created
- `rocketpool odao vote-proposal`: Vote on a proposal
- `rocketpool odao execute-proposal`: Execute a passed proposal
- `rocketpool odao join`: Join the oracle DAO (requires an executed invite proposal)
- `rocketpool odao leave`: Leave the oracle DAO (requires an executed leave proposal)
- `rocketpool odao replace`: Replace your position in the oracle DAO (requires an executed replace proposal)

Network Commands

- `rocketpool network node-fee`: Display the current network node commission rate for new minipools
- `rocketpool network rpl-price`: Display the current network RPL price information

Deposit Queue Commands

- `rocketpool queue status`: Display the current status of the deposit pool
- `rocketpool queue process`: Process the deposit pool by assigning user-deposited ETH to available minipools

API Reference

The Smart Node API

The Rocket Pool smart node service includes an API accessible via the `api` container. API commands can be invoked via `docker exec` on the machine running the service:

```
docker exec rocketpool_api /go/bin/rocketpool api [command] [subcommand] [args...]
```

The API is consumed by the smart node client, which offers a CLI interface for all provided functionality. API endpoints are provided for extension by applications wishing to interact with a running smart node.

All arguments for ETH or token amounts are given in wei. All endpoints return data in JSON format, always including `status` and `error` properties.

Wallet Commands

- `wallet status`: Get the current status of the password & wallet
- `wallet set-password [password]`: Set the node password to the specified string
- `wallet init`: Initialize the node wallet
- `wallet recover [mnemonic]`: Recover the node wallet from a mnemonic phrase (must be quoted)
- `wallet rebuild`: Rebuild validator keystores from derived keys
- `wallet export`: Get the node password & wallet file contents

Node Commands

- `node status`: Get the current status of the node
- `node can-register`: Check whether the node can be registered with Rocket Pool
- `node register [timezone-location]`: Register the node with Rocket Pool
- `node set-withdrawal-address [address]`: Set the node's withdrawal address
- `node set-timezone [timezone-location]`: Set the node's timezone location
- `node can-swap-rpl [amount]`: Check whether the node can swap an amount of old RPL for new RPL
- `node swap-rpl [amount]`: Swap an amount of old RPL for new RPL

- `node can-stake-rpl [amount]`: Check whether the node can stake an amount of RPL
- `node stake-rpl [amount]`: Stake an amount of RPL
- `node can-withdraw-rpl [amount]`: Check whether the node can withdraw an amount of staked RPL
- `node withdraw-rpl [amount]`: Withdraw an amount of staked RPL
- `node can-deposit [amount]`: Check whether the node can deposit the specified amount of ETH
- `node deposit [amount] [min-fee]`: Deposit the specified amount of ETH with a minimum commission rate
- `node can-send [amount] [token]`: Check whether the node can send an amount of tokens
- `node send [amount] [token] [to-address]`: Send the specified amount of tokens to an address
- `node can-burn [amount] [token]`: Check whether the node can burn an amount of tokens
- `node burn [amount] [token]`: Burn the specified amount of tokens for ETH

Minipool Commands

- `minipool status`: Get the current status of all minipools owned by the node
- `minipool can-refund [minipool-address]`: Check whether the specified minipool has a refund available
- `minipool refund [minipool-address]`: Refund ETH from the specified minipool
- `minipool can-dissolve [minipool-address]`: Check whether the specified minipool can be dissolved
- `minipool dissolve [minipool-address]`: Dissolve the specified minipool
- `minipool can-exit [minipool-address]`: Check whether the specified minipool can be exited from the beacon chain
- `minipool exit [minipool-address]`: Exit the specified minipool from the beacon chain
- `minipool can-withdraw [minipool-address]`: Check whether the specified minipool can be withdrawn from
- `minipool withdraw [minipool-address]`: Withdraw deposit & rewards from the specified minipool
- `minipool can-close [minipool-address]`: Check whether the specified minipool can be closed
- `minipool close [minipool-address]`: Close the specified minipool

Auction Commands

- `rocketpool auction status`: Get the current status of the RPL auction contract and lots
- `rocketpool auction lots`: Get the details of all RPL lots
- `rocketpool auction can-create-lot`: Check whether the node can create a new lot
- `rocketpool auction create-lot`: Create a new RPL lot from RPL in the auction contract
- `rocketpool auction can-bid-lot [lot-id]`: Check whether the node can bid on a lot
- `rocketpool auction bid-lot [lot-id] [amount]`: Bid an amount of ETH on an active RPL lot

- `rocketpool auction can-claim-lot [lot-id]`: Check whether the node can claim RPL from a lot
- `rocketpool auction claim-lot [lot-id]`: Clean RPL from a cleared lot you bid on
- `rocketpool auction can-recover-lot [lot-id]`: Check whether the node can recover unclaimed RPL from a lot
- `rocketpool auction recover-lot [lot-id]`: Recover unclaimed RPL from a cleared lot back to the auction contract

Oracle DAO Commands

- `rocketpool odao status`: Get the current status of the oracle DAO
- `rocketpool odao members`: Get the details of all oracle DAO members
- `rocketpool odao proposals`: Get the details of all oracle DAO proposals
- `rocketpool odao can-propose-invite [address]`: Check whether the node can invite a member to join the oracle DAO
- `rocketpool odao propose-invite [address] [id] [email]`: Invite a member to join the oracle DAO
- `rocketpool odao can-propose-leave`: Check whether the node can propose leaving the oracle DAO
- `rocketpool odao propose-leave`: Propose leaving the oracle DAO
- `rocketpool odao can-propose-replace [address]`: Check whether the node can propose replacing its position in the oracle DAO
- `rocketpool odao propose-replace [address] [id] [email]`: Propose replacing your position in the oracle DAO with a new member
- `rocketpool odao can-propose-kick [address] [fine-amount]`: Check whether the node can propose kicking a member from the oracle DAO
- `rocketpool odao propose-kick [address] [fine-amount]`: Propose kicking a member from the oracle DAO
- `rocketpool odao can-cancel-proposal [proposal-id]`: Check whether the node can cancel a created proposal
- `rocketpool odao cancel-proposal [proposal-id]`: Cancel a proposal you created
- `rocketpool odao can-vote-proposal [proposal-id]`: Check whether the node can vote on a proposal
- `rocketpool odao vote-proposal [proposal-id] [support]`: Vote on a proposal
- `rocketpool odao can-execute-proposal [proposal-id]`: Check whether the node can execute a proposal
- `rocketpool odao execute-proposal [proposal-id]`: Execute a passed proposal
- `rocketpool odao can-join`: Check whether the node can join the oracle DAO
- `rocketpool odao join`: Join the oracle DAO (requires an executed invite proposal)
- `rocketpool odao can-leave`: Check whether the node can leave the oracle DAO
- `rocketpool odao leave [bond-refund-address]`: Leave the oracle DAO (requires an executed leave proposal)

- `rocketpool odao can-replace`: Check whether the node can replace its position in the oracle DAO
- `rocketpool odao replace`: Replace your position in the oracle DAO (requires an executed replace proposal)

Network Commands

- `network node-fee`: Get the current network node commission rate
- `network rpl-price`: Get the current network RPL price information

Deposit Queue Commands

- `queue status`: Get the current status of the deposit pool and minipool queue
- `queue can-process`: Check whether the deposit pool can be processed
- `queue process`: Process the deposit pool