interbtc Documentation: interbtc

Interlay

Jul 02, 2021

INTRODUCTION

Note: Please note that this specification is a living document. The actual implementation might deviate from the specification. In case of deviations in the code, the code has priority over the specification.

CHAPTER

ONE

BTC PARACHAIN AT A GLANCE

The *BTC Parachain* connects the Polkadot ecosystem with Bitcoin. It allows the creation of *interbtc*, a fungible token that represents Bitcoin in the Polkadot ecosystem. interbtc is backed by Bitcoin 1:1 and allows redeeming of the equivalent amount of Bitcoins by relying on a collateralized third-party.

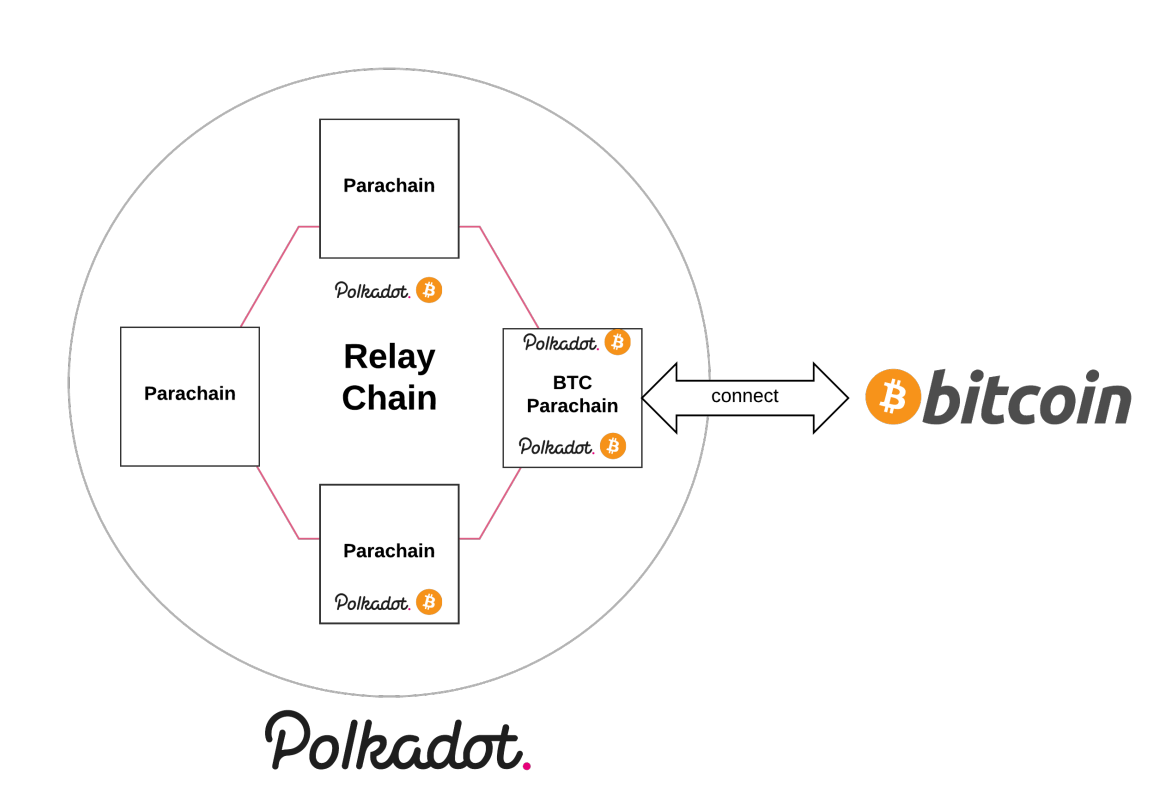


Fig. 1.1: The BTC Parachain allows the creation of collateralized 1:1 Bitcoin-backed tokens in Polkadot. These tokens can be transfered and traded within the Polkadot ecosystem.

1.1 Functionality

On a high-level, the BTC Parachain enables the issuing and redeeming of interbtc. The *issue process* allows a user to lock Bitcoin on the Bitcoin chain and, in return, issue interbtc on the BTC Parachain. Consequently, the *redeem process* allows a user to burn interbtc on the BTC Parachain and redeem previously locked Bitcoins on Bitcoin. Users can trade interbtc on the BTC Parachain and, through the Relay Chain, in other Parachains as well. The issue and redeem process can be executed by different users. Typically, this process is augmented by a collateralized realized third-party, a so-called *vault*.

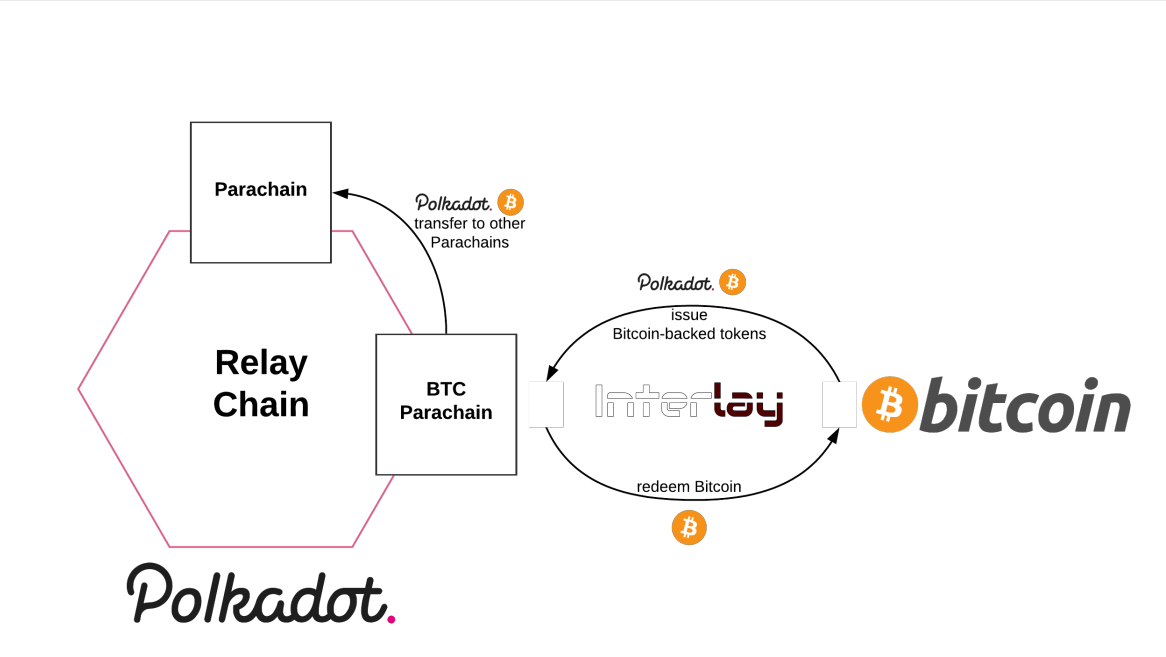


Fig. 1.2: The BTC Parachain includes a protocol to issue interbtc by locking Bitcoin and a protocol to redeem Bitcoin by burning interbtc tokens.

1.2 Components

The BTC Parachain makes use of two main components to achieve issuing and redeeming of interbtc:

- **XCLAIM(BTC,DOT)**: The XCLAIM(BTC,DOT) component implements four protocols including issue, transfer, redeem, and replace. It maintains the interbtc tokens, i.e. who owns how many tokens and manages the vaults as well as the collateral in the system.
- **BTC-Relay**: The BTC-Relay component is used to verify that certain transactions have happened on the Bitcoin blockchain. For example, when a user issues a new interbtc an equivalent amount of Bitcoins needs to be locked on the Bitcoin chain. The user can prove this to the interbtc component by verifying his transaction in the BTC-Relay component.

The figure below describes the relationships between the components in a high level. Please note that we use a simplified model here, where users are the ones augmenting the issue and redeem process. In practice, this is executed by the collateralized vaults.

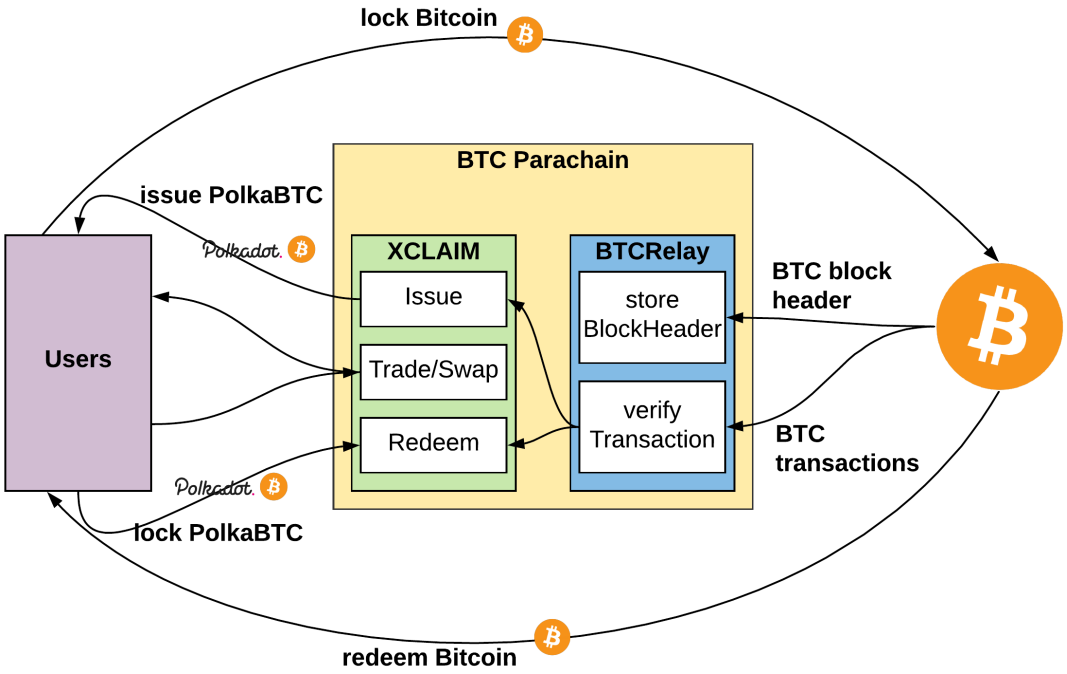


Fig. 1.3: The BTC Parachain consists of two logically different components. The XCLAIM(BTC,DOT) component (in green) maintains the accounts that own interbtc tokens. The BTC-Relay (blue) is repsonible for verifying the Bitcoin state to verify transactions. Users (in purple) are able to create new interbtc by locking BTC on the Bitcoin chain and redeeming BTC by burning interbtc. Also, users can trade interbtc on the BTC Parachain and in the wider Polkadot ecosystem.

CHAPTER

TWO

CRYPTOCURRENCY-BACKED ASSETS

Building trustless cross-blockchain trading protocols is challenging. Centralized exchanges thus remain the preferred route to executing transfers across blockchains. However, these services require trust and therefore undermine the very nature of the blockchains on which they operate. To overcome this, several decentralized exchanges have recently emerged which offer support for *commit-reveal* atomic cross-chain swaps (ACCS).

Commit-reveal ACCS, most notably based on HTCLs, enable the trustless exchange of cryptocurrencies across blockchains. To this date, this is the only mechanism to have been deployed in production. However, commit-reveal ACCS face numerous challenges:

- **Long waiting times:** Each commit-reveal ACCS requires multiple transactions to occur on all involved blockchains (commitments and revealing of secrets).
- **High costs:** Publishing multiple transaction per swap results in high fees to maintain such a system.
- **Strict online requirements:** Both parties must be online during the ACCS. Otherwise, the trade fails or, in the worst case, *loss of funds is possible*.
- **Out-of-band channels:** Secure operation requires users to exchange additional data *off-chain* (revocation commitments).
- **Race conditions:** Commit-reveal ACCS use time-locks to ensure security. Synchronizing time across blockchains, however, is challenging and opens up risks to race conditions.
- **Inefficiency:** Finally, commit-reveal ACCS are *one-time*. That is, all of the above challenges are faced with each and every trade.

Commit-reveal ACCS have been around since 2012. The practical challenges explain their limited use in practice.

2.1 Cryptocurrency-back Assets (CbA)

The idea of CbAs is that an asset is locked on a *backing blockchain* and issued 1:1 on an *issuing blockchain*. CbA that minimize trust in a third-party are based on the XCLAIM protocol. The third parties in XCLAIM are called *vaults* and are required to lock collateral as an insurance against misbehaviour.

XCLAIM introduces three protocols to achieve decentralized, transparent, consistent, atomic, and censorship resistant cross-blockchain swaps:

- **Issue**: Create Bitcoin-backed tokens, so-called *interbtc* on the BTC Parachain.
- **Transfer**: Transfer interbtc to others within the Polkadot ecosystem.
- **Redeem**: Burn Bitcoin-backed tokens on the BTC Parachain and receive 1:1 of the amount of Bitcoin in return.

The basic intuition of the protocol is as below:

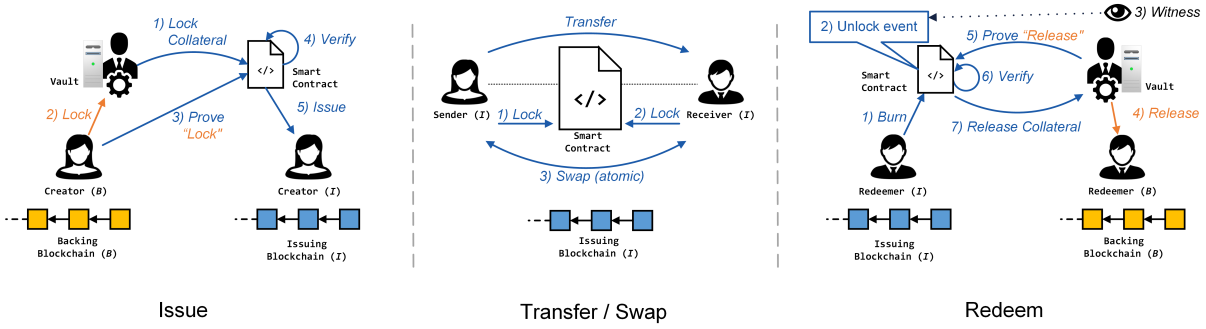


Fig. 2.1: The issue, transfer/swap, and redeem protocols in XCLAIM.

2.2 Design Principles

XCLAIM guarantees that Bitcoin-backed tokens can be redeemed for the corresponding amount of Bitcoin, or the equivalent economic value in DOT. Thereby, XCLAIM overcomes the limitations of centralized approaches through three primary techniques:

- **Secure audit logs**: Logs are constructed to record actions of all users both on Bitcoin and the BTC Parachain.
- **Transaction inclusion proofs**: Chain relays are used to prove correct behavior on Bitcoin to the BTC Parachain.
- **Proof-or-Punishment**: Instead of relying on timely fraud proofs (reactive), XCLAIM requires correct behavior to be proven proactively.
- **Over-collateralization**: Non-trusted intermediaries, i.e. vaults, are bound by collateral, with mechanisms in place to mitigate exchange rate fluctuations.

2.3 Recommended Background Reading

- **XCLAIM: Trustless, Interoperable, Cryptocurrency-backed Assets**. *IEEE Security and Privacy (S&P).* Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., & Knottenbelt, W. (2019).
- **Enabling Blockchain Innovations with Pegged Sidechains**. *Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra A., Timon J., & Wuille, P.* (2014)
- **SoK: Communication Across Distributed Ledgers**. *Cryptology ePrint Archiv, Report 2019/1128.* Zamyatin A, Al-Bassam M, Zindros D, Kokoris-Kogias E, Moreno-Sanchez P, Kiayias A, Knottenbelt WJ. (2019)
- **Proof-of-Work Sidechains**. *Workshop on Trusted Smart Contracts, Financial Cryptography* Kiayias, A., & Zindros, D. (2018)

CHAPTER

THREE

POLKADOT

Polkadot is a sharded blockchain that aims to connect multiple different blockchains together. The idea is that each shard has its custom state transition function. In Polkadot, a shard is called a Parachain. Having different shards with varying state transition functions offers to build blockchains with various cases in mind.

Each blockchain has to make trade-offs in terms of features it wishes to include. Great examples are Bitcoin which focusses on the core aspect of asset transfers with limited scripting capabilities. On the other end of the spectrum is Ethereum that features a (resource-limited) Turing complete execution environment. With Polkadot, the idea is to allow transfers between these different blockchains using a concept called Bridges.

3.1 Substrate

Polkadot is built using the Substrate framework. Substrate is a blockchain framework that allows to create custom blockchains. We refer the reader to the detailed introduction on the Substrate website.

3.2 Substrate Specifics

While this specification does not intend to give a general introduction to either Polkadot or Substrate, we want to highlight several features that are relevant to the implementation.

- **Bootstrapping**: A new Substrate node can be built either using the Substrate implementation directly or a bare Substrate node template. For a quick start, the Substrate node template is recommended.
- **Account-based model**: Substrate uses an account-based model to store user's and their balances through the Balances or Generic Asset modules.
- **DOT to Parachain**: Currently, there exists no pre-defined module to maintain DOT, Polkadot's native currency, on Substrate. This will be added in the future. For now, we assume such a module exists and model its functionality via the Generic Assets module.
- **Restricting function calls**: Functions declared in Substrate can be called by any external party. To restrict calls to specific modules, each module can have an account (`AccountId` in Substrate) assigned. Restricting a function call can then be enforced by limiting calls from pre-defined accounts (i.e. caller `Origin` must be equal to the modules `AccountId`).
- **Failure handling**: Substrate has no implicit failure handling. Errors within a function or errors raised in other function calls must be handled explicitly in the function implementation. Best practice is to (1) verify that the function conditions are met, (2) update the state, (3) emit events and return. *Note*: State can be partially updated if a transaction updates the state at a certain point and fails after the state update is executed.
- **Concurrency**: Substrate does not support concurrent state transitions at the moment.
- **Generic Rust crates**: Substrate does not include the Rust standard library due to non-deterministic behavior. However, crates can still be used and custom made if they do not depend on the Rust standard library.

CHAPTER

FOUR

ARCHITECTURE

interbtc consists of four different actors and eight modules. The component further uses two additional modules, the BTC-Relay component and the Parachain Governance mechanism.

4.1 Actors

There are four main participant roles in the system. A high-level overview of all modules and actors, as well as interactions between them, is provided in Fig. 4.1 below.

- **Vaults**: Vaults are collateralized intermediaries that are active on both the backing blockchain (Bitcoin) and the issuing blockchain to provide collateral in DOT. They receive and hold BTC from users who wish to create interbtc tokens. When a user destroys interbtc tokens, a vault releases the corresponding amount of BTC to the user's BTC address. Vaults interact with the following modules directly: *Vault Registry*, *Redeem*, and *Replace*.
- **Users**: Users interact with the BTC Parachain to create, use (trade/transfer/...), and redeem Bitcoin-backed interbtc tokens. Since the different protocol phases can be executed by different users, we introduce the following *sub-roles*:
  - **Requester**: A user that locks BTC with a vault on Bitcoin and issues interbtc on the BTC Parachain. Interacts with the *Issue* module.
  - **Sender** and **Receiver**: A user (Sender) that sends interbtc to another user (Receiver) on the BTC Parachain. Interacts with the *Treasury* module.
  - **Redeemer**: A user that destroys interbtc on the BTC Parachain to receive the corresponding amount of BTC on the Bitcoin blockchain from a Vault. Interacts with the *Redeem* module.
- **Staked Relayers**: Collateralized intermediaries which run Bitcoin full nodes and (i) monitor validity and availability of transactional data for Bitcoin blocks submitted to BTC-Relay, (ii) monitor that Vaults do not move locked BTC on Bitcoin without prior authorization by the BTC Parachain (i.e., through one of the Issue, Redeem or Replace protocols). In case either of the above errors was detected, Staked Relayers report this to the BTC Parachain. Interact with the *BTC-Relay*, *Security*, and *Vault Registry* modules.
- **Governance Mechanism**: The Parachain Governance Mechanism monitors the correct operation of the BTC Parachain, as well as the correct behaviour of Staked Relayers (and other participants if necessary). Interacts with the *Security* module when Staked Relayers misbehave and can manually interfere with the operation and parameterization of all components of the BTC Parachain.

Note: The exact composition of the Governance Mechanism is to be defined by Polkadot.

4.2 Modules

The eight modules in interbtc plus the BTC-Relay and Governance Mechanism interact with each other, but all have distinct logical functionalities. The figure below shows them.

The specification clearly separates these modules to ensure that each module can be implemented, tested, and verified in isolation. The specification follows the principle of abstracting the internal implementation away and providing a clear interface. This should allow optimisation and improvements of a module with minimal impact on other modules.

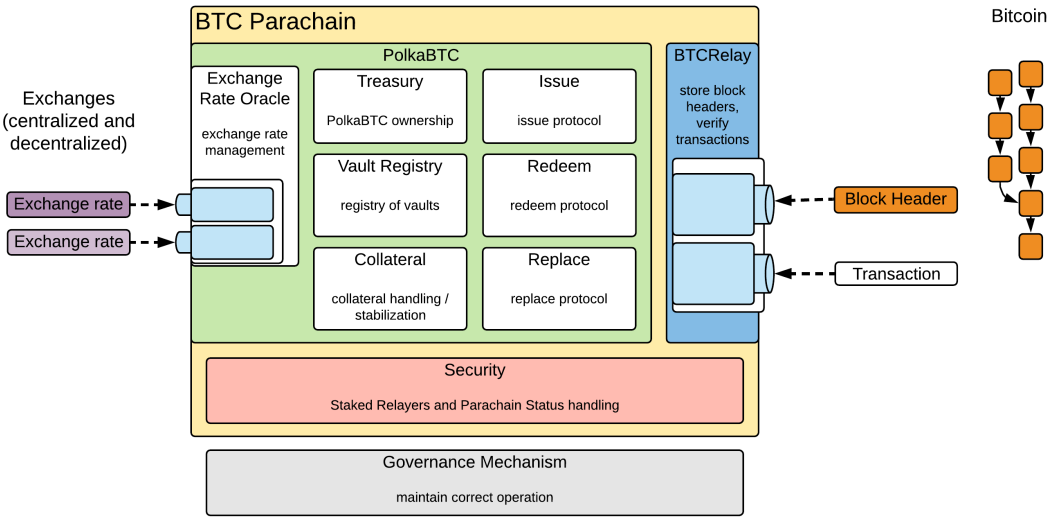


Fig. 4.1: High level overview of the BTC Parachain. interbtc consists of seven modules. The Oracle module stores the exchange rates based on the input of centralized and decentralized exchanges. The Treasury module maintains the ownership of interbtc, the VaultRegistry module stores information about the current Vaults in the system, and the Issue, Redeem and Replace modules expose funcitons and maintain data related to the respective sub protocols. The StabilizedCollateral modules handles vault collateralization, stabilization against exchange rate fluctuations and automatic liquidation. BTC-Relay tracks the Bitcoin main chain and verifies transaction inclusion. The Parachain Governance maintains correct operation of the BTC Parachain and intervenes / halts operation if necessary.

4.2.1 Exchange Rate Oracle

The Oracle module maintains the `ExchangeRate` value between the asset that is used to collateralize Vaults (DOT) and the to-be-issued asset (BTC). In the proof-of-concept, the Oracle is operated by a trusted third party to feed the current exchange rates into the system.

Note: The exchange rate oracle implementation is not part of this specification. interbtc simply expects a continuous input of exchange rate data and assumes the oracle operates correctly.

4.2.2 Treasury

The Treasury module maintains the ownership and balance of interbtc token holders. It allows respective owners of interbtc to send their tokens to other entities and to query their balance. Further, it tracks the total supply of tokens.

4.2.3 Vault Registry

The VaultRegistry module manages the Vaults in the system.It allows Managing the list of active Vaults in the system and the necessary data (e.g. BTC addresses) to execute the Issue, Redeem, and Replace protocols.

This module also handles the collateralization rates of Vaults and reacts to exchange rate fluctuations. Specifically, it:

- stores how much collateral each vault provided and how much of that collateral is allocated to interbtc.
- tracks the collateralization rate of each vault and triggers measures in case the rate declines, e.g. due to exchange rate fluctuations.
- triggers, as a last resort, automatic liquidation if a vault falls below the minimum collateralization rate.

4.2.4 Collateral

The Collateral module is the central storage for any collateral that is collected in any other module. It is allows for three simple operations: locking collateral by a party, releasing collateral back to the original party that locked this collateral, and last, slashing collateral where the collateral is relocated to a party other than the one that locked the collateral.

4.2.5 Issue

The Issue module handles the issuing process for interbtc tokens. It tracks issue requests by users, handles the collateral provided by users as griefing protection and exposes functionality for users to prove correct locking on BTC with Vaults (interacting with the endpoints in BTC-Relay).

4.2.6 Redeem

The Redeem module handles the redeem process for interbtc tokens. It tracks redeem requests by users, exposes functionality for Vaults to prove correct release of BTC to users (interacting with the endpoints in BTC-Relay), and handles the Vault's collateral in case of success (free) and failure (slash).

4.2.7 Replace

The Replace module handles the replace process for Vaults. It tracks replace requests by existing Vaults, exposes functionality for to-be-replaced Vaults to prove correct transfer of locked BTC to new vault candidates (interacting with the endpoints in BTC-Relay), and handles the collateral provided by participating Vaults as griefing protection.

4.2.8 Security

The Security module handles the Staked Relayers. Staked Relayers can register and vote, where applicable, on the status of the BTC Parachain. They can also report theft of BTC by vaults.

4.2.9 Governance Mechanism

The Governance Mechanism handles correct operation of the BTC Parachain.

Note: The Governance Mechanism is not part of this specification. The BTC Parachain simply expects continous operation of the BTC Parachain.

4.3 Interactions

We provide a detailed overview of the function calls between the different modules in Fig. 4.2.

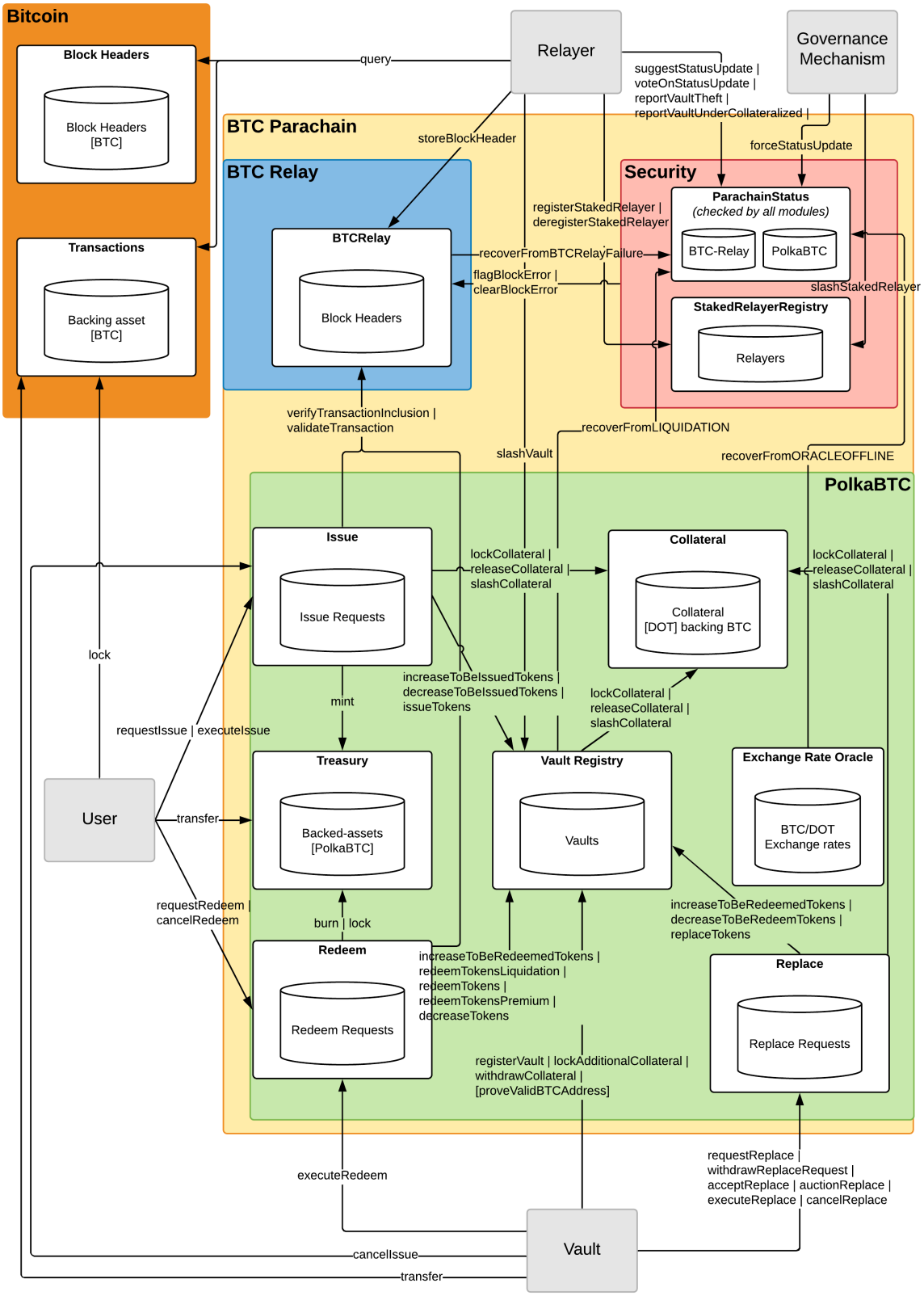


Fig. 4.2: Detailed architecture of the BTC Parachain, showing all actors, components and their interactions.

CHAPTER

FIVE

HOW TO READ THIS SPECIFICATION

This specification is a living document. The actual implementation might deviate from the specification. In case of deviations in the code, the code has priority over the specification.

5.1 Return types

Return types MAY return be wrapped in a `Result` type, in order to be able to return an error in case of failure. That is, when a return type is described as `T`, it MAY actually return a value of type `Result<T, DispatchError>`.

5.2 Preconditions, Postconditions and Invariants

Preconditions are condition that must hold before the function is executed. Unless otherwise stated, if the precondition does not hold, the function MUST return an error. If the function is a dispatchable function (i.e. callable by users), then if the function returns an error, it MUST NOT make any changes to the storage. The postconditions describe the changes the function MAY make to the storage. Additionally, it describes the return value of the function, if any. Invariants describe conditions that must hold both before and after the execution, but the function might not check whether the invariant holds prior to execution if the code assures that it always holds.

5.3 Errors and Events

Error listed in the function specification are not necessarily exhaustive - a function MAY return errors not listed. Similarly, events listed in the function specification are not necessarily exhaustive - a function MAY emit other events.

CHAPTER

SIX

BTC-RELAY

The BTC-Relay is responsible for storing Bitcoin block headers and maintaining the current longest chain. We can use the stored block headers to verify transaction inclusion in Bitcoin. Further, BTC_Relay exposes functions to validate that the contents of a transactions are as expected.

The specification of the BTC-Relay is found here: https://interlay.gitlab.io/interbtc-spec/btcrelay-spec/.

CHAPTER

SEVEN

COLLATERAL

7.1 Overview

The Collateral module is the central storage for collateral provided by users and vaults of the system. It allows to (i) lock, (ii) release, and (iii) slash collateral of either users or vaults. It can only be accessed by other modules and not directly through external transactions.

7.1.1 Step-by-Step

The protocol has three different "sub-protocols".

- **Lock**: Store a certain amount of collateral from a single entity (user or vault).
- **Release**: Transfer a certain amount of collateral back to the entity that paid it.
- **Slash**: Transfer a certain amount of collateral to a party that was damaged by the actions of another party.

7.2 Data Model

7.2.1 Scalars

TotalCollateral

The total collateral provided.

7.2.2 Maps

CollateralBalances

Mapping from accounts to their collateral balances.

7.3 Functions

7.3.1 lockCollateral

A user or a vault locks some amount of collateral.

Specification

Function Signature

`lockCollateral(sender, amount)`

Parameters

- `sender`: The sender wishing to lock collateral.
- `amount`: The amount of collateral.

Events

- `LockCollateral(sender, amount)`: Issues an event when collateral is locked.

Precondition

- The function must be called by any of the four modules: *Issue*, *Redeem*, *Replace*, or *Vault Registry*.
- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Add the `amount` of provided collateral to the `CollateralBalances` of the `sender`.
2. Increase `TotalCollateral` by `amount`.

7.3.2 releaseCollateral

When any of the issue, redeem, or replace protocols are completed successfully the party that has initially provided collateral receives their collateral back.

Specification

Function Signature

`releaseCollateral(sender, amount)`

Parameters

- `sender`: The sender getting returned its collateral.
- `amount`: The amount of collateral.

Events

- `ReleaseCollateral(sender, amount)`: Issues an event when collateral is released.

Errors

- `ERR_INSUFFICIENT_COLLATERAL_AVAILABLE`: The `sender` has less collateral stored than the requested `amount`.

Precondition

- The function must be called by any of the four modules: *Issue*, *Redeem*, *Replace*, or *Vault Registry*.
- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the `amount` is less or equal to the `CollateralBalances` of the `sender`. If not, throw `ERR_INSUFFICIENT_COLLATERAL_AVAILABLE`.
2. Deduct the `amount` from the `sender`'s `CollateralBalances`.
3. Deduct the `amount` from the `TotalCollateral`.
4. Transfer the `amount` to the `sender`.

7.3.3 slashCollateral

When any of the issue, redeem, or replace protocols are not completed in time, the party that has initially provided collateral (`sender`) is slashed and the collateral is transferred to another party (`receiver`).

Specification

Function Signature

`slashCollateral(sender, receiver, amount)`

Parameters

- `sender`: The sender that initially provided the collateral.
- `receiver`: The receiver of the collateral.
- `amount`: The amount of collateral.

Events

- `SlashCollateral(sender, receiver, amount)`: Issues an event when collateral is slashed.

Errors

- `ERR_INSUFFICIENT_COLLATERAL_AVAILABLE`: The `sender` has less collateral stored than the requested `amount`.

Precondition

- The function must be called by any of the four modules: *Issue*, *Redeem*, *Replace*, or *Vault Registry*.
- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the `amount` is less or equal to the `CollateralBalances` of the `sender`. If not, throw `ERR_INSUFFICIENT_COLLATERAL_AVAILABLE`.
2. Deduct the `amount` from the `sender`'s `CollateralBalances`.
3. Deduct the `amount` from the `TotalCollateral`.
4. Transfer the `amount` to the `receiver`.

7.4 Events

7.4.1 LockCollateral

Emit a `LockCollateral` event when a sender locks collateral.

Event Signature

`LockCollateral(sender, amount)`

Parameters

- `sender`: The sender that provides the collateral.
- `amount`: The amount of collateral.

Function

- *lockCollateral*

7.4.2 ReleaseCollateral

Emit a `ReleaseCollateral` event when a sender releases collateral.

Event Signature

`ReleaseCollateral(sender, amount)`

Parameters

- `sender`: The sender that initially provided the collateral.
- `amount`: The amount of collateral.

Function

- *releaseCollateral*

7.4.3 SlashCollateral

Emit a `SlashCollateral` event when a sender's collateral is slashed and transferred to the receiver.

Event Signature

`SlashCollateral(sender, receiver, amount)`

Parameters

- `sender`: The sender that initially provided the collateral.
- `receiver`: The receiver of the collateral.
- `amount`: The amount of collateral.

Function

- *slashCollateral*

7.5 Errors

`ERR_INSUFFICIENT_COLLATERAL_AVAILABLE`

- **Message**: "The sender's collateral balance is below the requested amount."
- **Function**: *releaseCollateral* | *slashCollateral*
- **Cause**: the `sender` has less collateral stored than the requested `amount`.

CHAPTER

EIGHT

FEE

8.1 Overview

The fee model crate implements the fee model outlined in *Fee Model*.

8.1.1 Step-by-step

1. Fees are paid by Users (e.g., during issue and redeem requests) and forwarded to a reward pool.
2. Fees are then split between Vaults, Staked Relayers, Maintainers, and Collators.
3. Network participants can claim these rewards from the pool based on their stake.
4. Stake is determined by their participation in the network - through incentivized actions.
5. Rewards may be paid in multiple currencies.

8.2 Data Model

8.2.1 Scalars (Fee Pools)

ParachainFeePool

Tracks the balance of fees earned by the BTC-Parachain which are to be distributed across all Vault, Staked Relayer, Collator and Maintainer pools.

VaultRewards

Tracks the fee share (in %) allocated to Vaults.

- Initial value: 77%

StakedRelayerRewards

Tracks the fee share (in %) allocated to Staked Relayers.

- Initial value: 3%

CollatorRewards

Tracks the fee share (in %) allocated to Collators (excl. Parachain transaction fees).

- Initial value: 0%

MaintainerRewards

Tracks fee share (in %) allocated to Parachain maintainers.

- Initial value: 20%

8.2.2 Scalars (Fees)

IssueFee

Issue fee share (configurable parameter, as percentage) that users need to pay upon execute issuing wrapped tokens.

- Paid in wrapped tokens
- Initial value: 0.5%

IssueGriefingCollateral

Default griefing collateral as a percentage of the locked collateral of a vault a user has to lock to issue wrapped tokens.

- Paid in collateral
- Initial value: 0.005%

RedeemFee

Redeem fee share (configurable parameter, as percentage) that users need to pay upon request redeeming wrapped tokens.

- Paid in wrapped tokens
- Initial value: 0.5%

PremiumRedeemFee

Fee for users to premium redeem (as percentage). If users execute a redeem with a Vault flagged for premium redeem, they earn a premium slashed from the Vault's collateral.

- Paid in collateral
- Initial value: 5%

PunishmentFee

Fee (as percentage) that a vault has to pay if it fails to execute redeem requests (for redeem, on top of the slashed value of the request). The fee is paid in collateral based on the wrapped token amount at the current exchange rate.

- Paid in collateral
- Initial value: 10%

PunishmentDelay

Time period in which a vault cannot participate in issue, redeem or replace requests.

- Measured in Parachain blocks
- Initial value: 1 day (Parachain constant)

ReplaceGriefingCollateral

Default griefing collateral as a percentage of the to-be-locked collateral of the new vault, vault has to lock to be replaced by another vault. This collateral will be slashed and allocated to the replacing Vault if the to-be-replaced Vault does not transfer BTC on time.

- Paid in collateral
- Initial value: 0.005%

8.3 Functions

8.3.1 distributeRewards

Specifies the distribution of fees among incentivised network participants.

Specification

Function Signature

`distributeRewards()`

Function Sequence

1. Calculate the total fees for all Vaults using the *VaultRewards* percentage.
2. Calculate the total fees for all Staked Relayers using the *StakedRelayerRewards* percentage.
3. Calculate the total fees for all Collators using the *CollatorRewards* percentage.
4. Send the remaining fees to the Maintainer fund.

8.3.2 withdrawRewards

A function that allows Staked Relayers, Vaults and Collators to withdraw the fees earned.

Specification

Function Signature

`withdrawRewards(account, currency, amount)`

Parameters

- `account`: the account withdrawing rewards
- `currency`: the currency of the reward to withdraw

Events

- `WithdrawRewards(account, currency, amount)`

Function Sequence

1. Compute the rewards based on the account's stake.
2. Transfer all rewards to the account.

8.4 Events

8.4.1 WithdrawRewards

Event Signature

`WithdrawRewards(account, currency, amount)`

Parameters

- `account`: the account withdrawing rewards
- `currency`: the currency of the reward to withdraw
- `amount`: the amount withdrawn

Functions

- *withdrawRewards*

CHAPTER

NINE

EXCHANGE RATE ORACLE

Note: This exchange oracle module is a bare minimum model that relies on a single trusted oracle source. Decentralized oracles are a difficult and open research problem that is outside of the scope of this specification. However, the general interface to get the exchange rate can remain the same even with different constructions.

The Exchange Rate Oracle receives a continuous data feed on the exchange rate between BTC and DOT.

The implementation of the oracle **is not part of this specification**. interbtc assumes the oracle operates correctly and that the received data is reliable.

9.1 Data Model

9.1.1 Constants

GRANULARITY

The granularity of the exchange rate. The granularity is set to 10^{-5}.

9.1.2 Scalars

ExchangeRateBtcInDot

The BTC in DOT exchange rate. This exchange rate is used to determine how much collateral is required to issue a specific amount of interbtc.

Note: If the `ExchangeRate` is set to 1238763, it translates to 12.38763 as the last five digits are used for the floating point (as defined by the `GRANULARITY`).

SatoshiPerBytesFast

The estimated Satoshis per bytes required to get a Bitcoin transaction included in the next block.

SatoshiPerBytesMedium

The estimated Satoshis per bytes required to get a Bitcoin transaction included in the next three blocks (about 30 min).

SatoshiPerBytesSlow

The estimated Satoshis per bytes required to get a Bitcoin transaction included in the six blocks (about 1 hour).

MaxDelay

The maximum delay in seconds between incoming calls providing exchange rate data. If the Exchange Rate Oracle receives no data for more than this period, the BTC Parachain enters an `Error` state with a `ORACLE_OFFLINE` error cause.

LastExchangeRateTime

UNIX timestamp indicating when the last exchange rate data was received.

9.1.3 Enums

InclusionEstimate

The estimated time until when a BTC transaction is included based on the Satoshi per byte fee.

- `FAST: 0` - the fee to include a BTC transaction within the next block.
- `MEDIUM: 1`- the fee to include a BTC transaction within the next three blocks (~30 min)).
- `SLOW: 2` - the fee to include a BTC transaction within the six blocks (~60 min).

9.1.4 Maps

AuthorizedOracles

The account(s) of the oracle. Returns true if registered as an oracle.

9.2 Functions

9.2.1 setExchangeRate

Set the latest (aggregate) BTC/DOT exchange rate. This function invokes a check of vault collateral rates in the *Vault Registry* component.

Specification

Function Signature

`setExchangeRate(oracle, rate)`

Parameters

- `oracle`: the oracle account calling this function. Must be pre-authorized and tracked in this component!
- `rate`: the `u128` BTC/DOT exchange rate

Events

- `SetExchangeRate(oracle, rate)`: Emits the new exchange rate when it is updated by the oracle.

Errors

- `ERR_INVALID_ORACLE_SOURCE`: the caller of the function was not the authorized oracle.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the caller of the function is the `AuthorizedOracle`. If not, throw `ERR_INVALID_ORACLE_SOURCE`.
2. Update the `ExchangeRate` with the `rate`.
3. If `LastExchangeRateTime` minus the current UNIX timestamp is greater or equal to `MaxDelay`, call *recoverFromORACLEOFFLINE* to recover from an `ORACLE_OFFLINE` error (which was the case before this data submission).
4. Set `LastExchangeRateTime` to the current UNIX timestamp.
5. Emit the `SetExchangeRate` event.

9.2.2 setSatoshiPerBytes

Set the Satoshi per bytes fee

Specification

Function Signature

`setSatoshiPerBytes(fee, InclusionEstimate)`

Parameters

- `fee`: the Satoshi per byte fee.
- `InclusionEstimate`: the estimated inclusion time.

Events

- `SetSatoshiPerByte(fee, InclusionEstimate):`

Errors

- `ERR_INVALID_ORACLE_SOURCE`: the caller of the function was not the authorized oracle.

Requirements

- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- If the caller of the function is not in `AuthorizedOracles` MUST return `ERR_INVALID_ORACLE_SOURCE`.
- If the above checks passed, the function MUST update the `SatoshiPerBytes` field indicated by the `InclusionEstimate` enum.
- If the above steps passed, MUST emit the `SetSatoshiPerByte` event.

9.2.3 getExchangeRate

Returns the latest BTC/DOT exchange rate, as received from the external data sources.

Specification

Function Signature

`getExchangeRate()`

Returns

- *u128* (aggregate) exchange rate value

```
fn getExchangeRate(origin) -> Result<u128, ERR_MISSING_EXCHANGE_RATE> {...}
```

Errors

`ERR_MISSING_EXCHANGE_RATE`: the last exchange rate information exceeded the maximum delay acceptable by the oracle.

Preconditions

This function can be called by any participant to retrieve the BTC/DOT exchange rate as tracked by the BTC Parachain.

Function Sequence

1. Check if the current (UNIX) time minus the `LastExchangeRateTime` exceeds `MaxDelay`. If this is the case, return `ERR_MISSING_EXCHANGE_RATE` error.
2. Otherwise, return the `ExchangeRate` from storage.

9.2.4 getLastExchangeRateTime

Returns the UNIX timestamp of when the last BTC/DOT exchange rate was received from the external data sources.

Specification

Function Signature

`getLastExchangeRateTime()`

Returns

- *timestamp*: 32bit UNIX timestamp

```
fn getLastExchangeRateTime() -> U32 {...}
```

Function Sequence

1. Return `LastExchangeRateTime` from storage.

9.3 Events

9.3.1 SetExchangeRate

Emits the new exchange rate when it is updated by the oracle.

Event Signature

`SetExchangeRate(oracle, rate)`

Parameters

- `oracle`: the oracle account calling this function. Must be pre-authorized and tracked in this component!
- `rate`: the `u128` BTC/DOT exchange rate

Function

setExchangeRate

9.3.2 recoverFromORACLEOFFLINE

Internal function. Recovers the BTC Parachain state from a `ORACLE_OFFLINE` error and sets `ParachainStatus` to `RUNNING` if there are no other errors.

> **Attention:** Can only be called from *Exchange Rate Oracle*.

Specification

Function Signature

`recoverFromORACLEOFFLINE()`

Events

- `ExecuteStatusUpdate(newStatusCode, addErrors, removeErrors, msg)` - emits an event indicating the status change, with `newStatusCode` being the new `StatusCode`, `addErrors` the set of to-be-added `ErrorCode` entries (if the new status is `Error`), `removeErrors` the set of to-be-removed `ErrorCode` entries,, and `msg` the detailed reason for the status update.

9.4 Error Codes

`ERR_MISSING_EXCHANGE_RATE`

- **Message**: "Exchange rate not set."
- **Function**: *getExchangeRate*
- **Cause**: The last exchange rate information exceeded the maximum delay acceptable by the oracle.

`ERR_INVALID_ORACLE_SOURCE`

- **Message**: "Invalid oracle account."
- **Function**: *setExchangeRate*
- **Cause**: The caller of the function was not the authorized oracle.

CHAPTER

TEN

ISSUE

10.1 Overview

The Issue module allows as user to create new interbtc tokens. The user needs to request interbtc through the *requestIssue* function, then send BTC to a vault, and finally complete the issuing of interbtc by calling the *executeIssue* function. If the user does not complete the process in time, the vault can cancel the issue request and receive a griefing collateral from the user by invoking the *cancelIssue* function. Below is a high-level step-by-step description of the protocol.

10.1.1 Step-by-step

1. Precondition: a vault has locked collateral as described in the *Vault Registry*.
2. A user executes the *requestIssue* function to open an issue request on the BTC Parachain. The issue request includes the amount of interbtc the user wants to issue, the selected vault, and a small collateral to prevent *Griefing*.
3. A user sends the equivalent amount of BTC that he wants to issue as interbtc to the vault on the Bitcoin blockchain.
4. The user or a vault acting on behalf of the user extracts a transaction inclusion proof of that locking transaction on the Bitcoin blockchain. The user or a vault acting on behalf of the user executes the *executeIssue* function on the BTC Parachain. The issue function requires a reference to the issue request and the transaction inclusion proof of the Bitcoin locking transaction. If the function completes successfully, the user receives the requested amount of interbtc into his account.
5. Optional: If the user is not able to complete the issue request within the predetermined time frame (`IssuePeriod`), the vault is able to call the *cancelIssue* function to cancel the issue request adn will receive the griefing collateral locked by the user.

10.1.2 Security

- Unique identification of Bitcoin payments: *On-Chain Key Derivation Scheme*

10.1.3 Vault Registry

The data access and state changes to the vault registry are documented in Fig. 10.1 below.

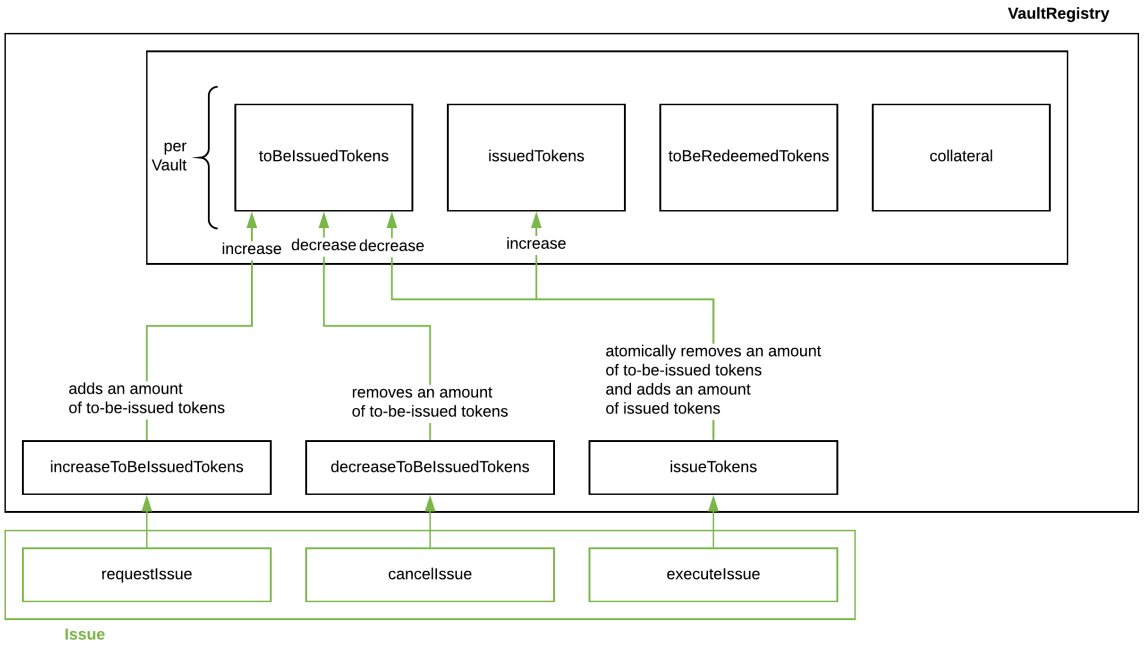


Fig. 10.1: The issue protocol interacts with three functions in the vault registry that handle updating the different token balances.

10.1.4 Fee Model

Following additions are added if the fee model is integrated.

- Issue fees are paid by users in interbtc when executing the request. The fees are transferred to the Parachain Fee Pool.
- If an issue request is executed, the user's griefing collateral is returned.
- If an issue request is canceled, the vault assigned to this issue request receives the griefing collateral.

10.2 Data Model

10.2.1 Scalars

IssuePeriod

The time difference between when an issue request is created and required completion time by a user. Concretely, this period is the amount by which *ActiveBlockCount* is allowed to increase before the issue is considered to be expired. The period has an upper limit to prevent griefing of vault collateral.

IssueGriefingCollateral

The minimum collateral (DOT) a user needs to provide as griefing protection.

Note: Serves as a measurement to disincentivize griefing attacks against a vault. A user could otherwise create an issue request, temporarily locking a vault's collateral and never execute the issue process.

10.2.2 Maps

IssueRequests

Users create issue requests to issue interbtc. This mapping provides access from a unique hash `IssueId` to a `Issue` struct. `<IssueId, Issue>`.

10.2.3 Structs

Issue

Stores the status and information about a single issue request.

Parameter	Type	Description
`vault`	Account	The BTC Parachain address of the vault responsible for this commit request.
`opentime`	u256	Block height of opening the request.
`griefingCollateral`	DOT	Collateral provided by a user.
`amount`	interbtc	Amount of interbtc to be issued.
`fee`	interbtc	Fee charged to the user for issuing.
`requester`	Account	User account receiving interbtc upon successful issuing.
`btcAddress`	bytes[20]	Base58 encoded Bitcoin public key of the Vault.
`completed`	bool	Indicates if the issue has been completed.
`cancelled`	bool	Indicates if the issue request was cancelled.

10.3 Functions

10.3.1 requestIssue

A user opens an issue request to create a specific amount of interbtc. When calling this function, a user provides her own parachain account identifier, the to be issued amount of interbtc, and the vault she wants to use in this process (parachain account identifier). Further, she provides some (small) amount of DOT collateral (`griefingCollateral`) to prevent griefing.

Specification

Function Signature

`requestIssue(requester, amount, vault, griefingCollateral)`

Parameters

- `requester`: The user's BTC Parachain account.
- `amount`: The amount of interbtc to be issued.
- `vault`: The BTC Parachain address of the vault involved in this issue request.
- `griefingCollateral`: The collateral amount provided by the user as griefing protection.

Events

- `RequestIssue(issueId, requester, amount, vault, btcAddress)`

Errors

- `ERR_VAULT_NOT_FOUND = "There exists no vault with the given account id"`: The specified vault does not exist.
- `ERR_VAULT_BANNED = "The selected vault has been temporarily banned."`: Issue requests are not possible with temporarily banned Vaults.
- `ERR_INSUFFICIENT_COLLATERAL`: The user did not provide enough griefing collateral.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Retrieve the `vault` from *Vault Registry*. Return `ERR_VAULT_NOT_FOUND` if no vault can be found.
2. Check that the `vault` is currently not banned, i.e., `vault.bannedUntil == None` or `vault.bannedUntil < current parachain block height`. Return `ERR_VAULT_BANNED` if this check fails.
3. Check if the `griefingCollateral` is greater or equal `IssueGriefingCollateral`. If this check fails, return `ERR_INSUFFICIENT_COLLATERAL`.
4. Lock the user's griefing collateral by calling the *lockCollateral* function with the `requester` as the sender and the `griefingCollateral` as the amount.
5. Call the VaultRegistry *tryIncreaseToBeIssuedTokens* function with the `amount` of tokens to be issued and the `vault` identified by its address. This function returns a unique `btcAddress` that the user should send Bitcoin to.
6. Generate an `issueId` via *generateSecureId*.
7. Store a new `Issue` struct in the `IssueRequests` mapping as `IssueRequests[issueId] = issue`, where `issue` is the `Issue` struct as:
    - `issue.vault` is the `vault`
    - `issue.opentime` is the current block number
    - `issue.griefingCollateral` is the griefing collateral provided by the user
    - `issue.amount` is the `amount` provided as input
    - `issue.requester` is the user's account
    - `issue.btcAddress` the Bitcoin address of the vault as returned in step 3

8. Issue the `RequestIssue` event with the `issueId`, the `requester` account, `amount`, `vault`, and `btcAddress`.

10.3.2 executeIssue

A user completes the issue request by sending a proof of transferring the defined amount of BTC to the vault's address.

Specification

Function Signature

`executeIssue(requester, issueId, merkleProof, rawTx)`

Parameters

- `requester`: the account of the user.
- `issueId`: the unique hash created during the `requestIssue` function.
- `merkleProof`: Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- `ExecuteIssue(issueId, requester, amount, vault)`: Emits an event with the information about the completed issue request.

Errors

- `ERR_ISSUE_ID_NOT_FOUND`: The `issueId` cannot be found.
- `ERR_COMMIT_PERIOD_EXPIRED`: The time limit as defined by the `IssuePeriod` is not met.
- `ERR_UNAUTHORIZED_USER = Unauthorized: Caller must be associated user`: The caller of this function is not the associated user, and hence not authorized to take this action.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

Note: Ideally the `SecureCollateralThreshold` in the VaultRegistry should be high enough to prevent the vault from entering into the liquidation state in-between the request and execute.

1. Checks if the `issueId` exists. Return `ERR_ISSUE_ID_NOT_FOUND` if not found. Else, loads the according issue request struct as `issue`.
2. Checks if the issue has expired by calling *hasExpired* in the Security module. If true, this throws `ERR_COMMIT_PERIOD_EXPIRED`.
3. Verify the transaction.
   a. Call *verifyTransactionInclusion* in *BTC-Relay*, providing the `txId`, and `merkleProof` as parameters. If this call returns an error, abort and return the received error.
   b. Call *validateTransaction* in *BTC-Relay*, providing `rawTx`, the amount of to-be-issued BTC (`issue.amount`), the `vault`'s Bitcoin address (`issue.btcAddress`), and the `issueId` as parameters. If this call returns an error, abort and return the received error.

4. Call the *issueTokens* with the `issue.vault` and the `amount` to decrease the `toBeIssuedTokens` and increase the `issuedTokens`.
5. Call the *mint* function in the Treasury with the `amount` and the user's address as the `receiver`.
6. Remove the `IssueRequest` from `IssueRequests`.
7. Emit an `ExecuteIssue` event with the user's address, the issueId, the amount, and the Vault's address.

10.3.3 cancelIssue

If an issue request is not completed on time, the issue request can be cancelled.

Specification

Function Signature

`cancelIssue(sender, issueId)`

Parameters

- `sender`: The sender of the cancel transaction.
- `issueId`: the unique hash of the issue request.

Events

- `CancelIssue(sender, issueId)`: Issues an event with the `issueId` that is cancelled.

Errors

- `ERR_ISSUE_ID_NOT_FOUND`: The `issueId` cannot be found.
- `ERR_TIME_NOT_EXPIRED`: Raises an error if the time limit to call `executeIssue` has not yet passed.
- `ERR_ISSUE_COMPLETED`: Raises an error if the issue is already completed.

Preconditions

- None.

Function Sequence

1. Check if an issue with id `issueId` exists. If not, throw `ERR_ISSUE_ID_NOT_FOUND`. Otherwise, load the issue request as `issue`.
2. Check if the issue has expired by calling *hasExpired* in the Security module, and throw `ERR_TIME_NOT_EXPIRED` if not.
3. Check if the `issue.completed` field is set to true. If yes, throw `ERR_ISSUE_COMPLETED`.
4. Call the *decreaseToBeIssuedTokens* function in the VaultRegistry with the `issue.vault` and the `issue.amount` to release the vault's collateral.
5. Call the *slashCollateral* function to transfer the `griefingCollateral` of the user requesting the issue to the vault assigned to this issue request with the `issue.requester` as sender, the `issue.vault` as receiver, and `issue.griefingCollateral` as amount.
6. Remove the `IssueRequest` from `IssueRequests`.
8. Emit a `CancelIssue` event with the `issueId`.

10.4 Events

10.4.1 RequestIssue

Emit a `RequestIssue` event if a user successfully open a issue request.

Event Signature

`RequestIssue(issueId, requester, amount, vault, btcAddress)`

Parameters

- `issueId`: A unique hash identifying the issue request.
- `requester`: The user's BTC Parachain account.
- `amount`: The amount of interbtc to be issued.
- `vault`: The BTC Parachain address of the vault involved in this issue request.
- `btcAddress`: The Bitcoin address of the vault.

Functions

- *requestIssue*

10.4.2 ExecuteIssue

Event Signature

`ExecuteIssue(issueId, requester, amount, vault)`

Parameters

- `issueId`: A unique hash identifying the issue request.
- `requester`: The user's BTC Parachain account.
- `amount`: The amount of interbtc to be issued.
- `vault`: The BTC Parachain address of the vault involved in this issue request.

Functions

- *executeIssue*

10.4.3 CancelIssue

Event Signature

`CancelIssue(issueId, sender)`

Parameters

- `issueId`: the unique hash of the issue request.
- `sender`: The sender of the cancel transaction.

Functions

- *cancelIssue*

10.5 Error Codes

`ERR_VAULT_NOT_FOUND`

- **Message**: "There exists no vault with the given account id."
- **Function**: *requestIssue*
- **Cause**: The specified vault does not exist.

`ERR_VAULT_BANNED`

- **Message**: "The selected vault has been temporarily banned."
- **Function**: *requestIssue*
- **Cause**: Issue requests are not possible with temporarily banned Vaults

`ERR_INSUFFICIENT_COLLATERAL`

- **Message**: "User provided collateral below limit."
- **Function**: *requestIssue*
- **Cause**: User provided griefingCollateral below `IssueGriefingCollateral`.

`ERR_UNAUTHORIZED_USER`

- **Message**: "Unauthorized: Caller must be associated user"
- **Function**: *executeIssue*
- **Cause**: The caller of this function is not the associated user, and hence not authorized to take this action.

`ERR_ISSUE_ID_NOT_FOUND`

- **Message**: "Requested issue id not found."
- **Function**: *executeIssue*
- **Cause**: Issue id not found in the `IssueRequests` mapping.

`ERR_COMMIT_PERIOD_EXPIRED`

- **Message**: "Time to issue interbtc expired."
- **Function**: *executeIssue*
- **Cause**: The user did not complete the issue request within the block time limit defined by the `IssuePeriod`.

`ERR_TIME_NOT_EXPIRED`

- **Message**: "Time to issue interbtc not yet expired."
- **Function**: *cancelIssue*
- **Cause**: Raises an error if the time limit to call `executeIssue` has not yet passed.

`ERR_ISSUE_COMPLETED`

- **Message**: "Issue completed and cannot be cancelled."
- **Function**: *cancelIssue*
- **Cause**: Raises an error if the issue is already completed.

CHAPTER

ELEVEN

VAULT NOMINATION

11.1 Overview

Vault Nomination is a feature aimed at increasing *interbtc* issuance capacity which introduces two actors: Nominators and Operators. Vaults who opt in to this feature take on the additional role of nomination Operators. A Nominator is anyone who locks their free collateral so that Operators they trust can issue *interbtc* backed by the nominated collateral. Nominators are rewarded a fraction of the fees generated by their collateral, while the rest of the fees is given to the Operator. Operators are assumed to be trusted by their nominators not to steal Bitcoin backed by nominated collateral.

11.1.1 Step-by-step

1. Vaults opt in to the nomination feature, becoming Operators.
2. The maximum nomination an Operator can receive is bounded by their own locked collateral.
3. Nominators select one or more Operators and lock their collateral balance onto the BTC Parachain.
4. Nominators can go offline and their nominated collateral will generate rewards passively.
5. Operator and Nominator collateral cannot be withdrawn directly. Rather, withdrawals are subject to an unbonding period.
6. In case of Operator failure, Nominators are returned any left-over collateral (after victim users are reimbursed).

11.2 Protocol

11.2.1 Security Assumptions and Considerations

1. The operating Vault is trusted by its Nominators not to steal the interbtc issued with their collateral.
2. There is no transitive trust. If a user trusts Vault A and Vault A trusts Vault B, the user does not trust Vault B.
3. Nominators are mostly-offline agents, who are slow to respond to system changes.
4. Vaults are always-online agents, who can promptly react to system updates.
5. A Nominator may expose the Vault and the other Nominators to additional economic risk by withdrawing nominated collateral during an exchange rate spike. Similarly, the Vault may expose its Nominators to additional economic risk by withdrawing excess collateral.
    - Note: in the usual case, this should be handled by having the different collateral thresholds (secure, premium redeem, liquidation). But in extreme cases (very high exchange rate volatility), it might cause concern.

11.2.2 Vault Nomination Protocol

1. Vaults can choose to opt in and out of the Nomination protocol. If they opt-in they take on the additional role of an Operator.
2. Nominators select an Operator to which they can delegate DOT balance as collateral. As a reward, they will earn a fraction of the interbtc and DOT fees generated by this collateral. The other fraction of these fees is received by the Operator.
3. Vault replacement is disallowed for Operators with nominated collateral. Otherwise, Security Assumptions 1 and 2 would be violated.
4. The nominated DOT:
    1. Is locked on the parachain
    2. Cannot be withdrawn by the operating Vault
    3. Is capped at a fraction of the Vault's deposited collateral (Max Nomination Ratio). This prevents the Operator from withdrawing its entire collateral and only exposing Nominators to economic risk, or stealing without liquidation consequences. This means that an Operator can only withdraw collateral as long as the fraction of nominated collateral does not exceed the threshold cap. Capping Nominator collateral also prevents Operators being "outnumbered" by Nominators and their relative fee earnings being marginalized.
5. Liquidation slashing is handled as follows.
    1. In case the collateral managed by the Operator falls below the liquidation threshold, the Operator and Nominators are slashed proportionally to their collateral.
    2. In case the Operator steals Bitcoin deposited at its address, its collateral is used to cover as much of the slashed amount as possible. If the Operator's collateral was not enough to cover the entire amount, the Nominators are slashed proportionally for the remaining amount.
6. Collateral withdrawals are first requested and then executed. A withdrawal request:
    1. Decreases the issuable interbtc capacity.
    2. May be cancelled if not executed. The amount in the cancelled withdrawal request becomes backing collateral again.
    3. Is subject to a window of delay (unbonding period) that allows Nominators and the Operator to react.
        1. Operator Unbonding Period. This window is longer, because Nominators are assumed to be mostly offline.
        2. Nominator Unbonding Period. This window is shorter, because the Vault Operator is assumed to always be online.
7. Collateral withdrawals are subject to the following restrictions.
    1. The remaining collateralization of an operator must not be below the secure collateral threshold.
    2. Operator withdrawals must not cause nominated collateral to exceed the Max Nomination Ratio.
8. Forced collateral withdrawal. If an operator's withdrawal would result in a violation of the Max Nomination Ratio, automatically refund excess nominated collateral to the nominators, proportionally. Both the operator withdrawal and the nominator refunds are subject to the unbonding period.
9. If an Operator issued zero interbtc, it can deregister and automatically refund Nominators their collateral.
10. When an Operator is banned, its collateralization is lowered to the secure collateral threshold by automatically refunding nominated DOT, proportionally.

11.3 Data Model

11.3.1 Scalars

NominationEnabled

Flag indicating whether this feature is enabled. As Operators may have issued `interbtc` with nominated collateral when this feature is turned off, a `False` value of this scalar only prevents the opting in of new Operators.

MaxNominatorsPerOperator

Maximum number of nominators a single operator can have.

- Initial value: 100

OperatorUnbondingPeriod

Unbonding period, measured in blocks, that Operator withdrawal requests are subject to.

- Initial value: 14400 (24 hours)

NominatorUnbondingPeriod

Unbonding period, measured in blocks, that Nominator withdrawal requests are subject to.

- Initial value: 7200 (12 hours)

11.3.2 Maps

Operators

Mapping from accounts to Operator structs.

11.3.3 Structs

Nominator

Stores the information of a Nominator.

Parameter	Type	Description
`id`	AccountId	The ID of the Nominator represented by this struct.
`collateral`	DOT	Collateral amount nominated.
`pendingWithdrawals`	BTreeMap	Mapping from the withdrawal request ID to the (maturityBlock, amount) tuple.
`collateralToBeWithdrawn`	DOT	Collateral that is not backing any interbtc and has been requested for withdrawal.

Operator

Stores the information of an Operator.

Parameter	Type	Description
`id`	AccountId	The ID of the Nominator represented by this struct.
`nominators`	BTreeMap	Mapping from the ID of a nominator to a Nominator struct.
`totalNominatedCollateral`	DOT	Total amount of collateral received as nomination.
`pendingWithdrawals`	BTreeMap	Mapping from the withdrawal request ID to the (maturityBlock, amount) tuple.
`collateralToBeWithdrawn`	DOT	Collateral that is not backing any interbtc and has been requested for withdrawal.

11.4 Functions

11.4.1 getMaxNominationRatio

Returns the maximum nomination ratio (as %), denoting the maximum `totalNominatedCollateral:operatorCollateral` value allowed.

- Example (current parameterization): `(1.5 / 1.2) - 1 = 25%`

Specification

Function Signature

`getMaxNominationRatio()`

Function Sequence

1. Return `(secureCollateralThreshold / auctionCollateralThreshold) - 1`

11.4.2 setNominationEnabled

Set the feature flag for vault nomination.

Specification

Function Signature

`setNominationEnabled(enabled)`

Parameters

- `enabled`: `True` if nomination should be enabled, `False` if it should be disabled

Function Sequence

1. Ensure the calling account is root.
2. Set the NominationEnabled scalar to the value of the `enabled` parameter

11.4.3 optInToNomination

Become an Operator in the Vault Nomination protocol

Specification

Function Signature

`optInToNomination(operatorId)`

Parameters

- `operatorId`: the id of the vault to mark as Nomination Operator.

Events

- `NominationOptIn(operatorId)`

Errors

- `VaultNominationDisabled`: the nomination feature is disabled.
- `NotAVault`: the caller of the function is not a vault.
- `VaultAlreadyOptedInToNomination`: the caller of the function is already opted in.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the nomination feature is enabled. If not, throw `VaultNominationDisabled`.
2. Check if the caller is a vault. If not, throw `NotAVault`.
3. Check if the caller is not already opted in to nomination. If not, throw `VaultAlreadyOptedInToNomination`.
4. Instantiate an `Operator` struct.
5. Add the struct to the `Operators` mapping.

11.4.4 optOutOfNomination

Deregister from being an Operator in the Vault Nomination protocol.

Specification

Function Signature

`optOutOfNomination(operatorId)`

Parameters

- `operatorId`: the id of the vault to deregister from the nomination feature.

Events

- `NominationOptOut(operatorId)`

Errors

- `VaultNotOptedInToNomination`: the caller is not an Operator.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the caller is a nomination Operator. If not, throw `VaultNotOptedInToNomination`.
2. Immediately refund all nominated collateral, bypassing the unbonding period.
3. Remove caller from the `Operators` mapping.

11.4.5 depositNominatedCollateral

Nominate collateral to a selected Operator.

Specification

Function Signature

`depositNominatedCollateral(nominatorId, operatorId, amount)`

Parameters

- `nominatorId`: the id of the user nominating collateral.
- `operatorId`: the id of the operator to receive the nomination.
- `amount`: the amount of collateral to nominate.

Events

- `IncreaseNominatedCollateral(nominatorId, operatorId, amount)`

Errors

- `VaultNominationDisabled`: the nomination feature is disabled.
- `VaultNotOptedInToNomination`: the vault is not an Operator.
- `DepositViolatesMaxNominationRatio`: the *amount* of nomination would cause the Max Nomination Ratio to be exceeded for this *operatorId*.
- `OperatorHasTooManyNominators`: the number of Nominators to the current Operator has reached *MaxNominatorsPerOperator*.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if the nomination feature is enabled. If not, throw `VaultNominationDisabled`.
2. Check if `operatorId` represents an operator. If not, throw `VaultNotOptedInToNomination`.
3. Check that the additional nominated `amount` does not cause the Max Nomination Ratio to be exceeded. If not, throw `DepositViolatesMaxNominationRatio`.
4. If the caller had no nomination to this Operator, check that the `MaxNominatorsPerOperator` would not be exceeded by receiving this nomination. If `MaxNominatorsPerOperator` would be exceeded, throw `OperatorHasTooManyNominators`.
5. Update the `Operator` object to create or update the `Nominator` entry of the caller.
6. Move collateral from `nominatorId` to the `backing_collateral` of `operatorId` in the *Vault Registry*.

11.4.6 requestOperatorCollateralWithdrawal

Request an operator collateral withdrawal, subject to an unbonding period.

Specification

Function Signature

`requestOperatorCollateralWithdrawal(operatorId, amount)`

Parameters

- `operatorId`: the id of the caller.
- `amount`: the amount to withdraw.

Events

- `RequestOperatorCollateralWithdrawal(requestId, operatorId, maturity, amount)`

Errors

- `VaultNotOptedInToNomination`: the caller is not an Operator.
- `InsufficientCollateral`: the caller has requested to withdraw more collateral than it owns.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check if `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Check if the operator has enough collateral of its own (excluding nominations). If not, throw `InsufficientCollateral`.
3. Immediately refund, proportionally, nominated collateral that would cause the Max Nomination Ratio to be exceeded.
4. Add the withdrawal request to the `pendingWithdrawals` array in the `Operator` struct.
5. Decrease the `backing_collateral` of `operatorId` in the *Vault Registry*.

11.4.7 requestNominatorCollateralWithdrawal

Request a nominator collateral withdrawal, subject to an unbonding period.

Specification

Function Signature

`requestNominatorCollateralWithdrawal(nominatorId, operatorId, amount)`

Parameters

- `nominatorId`: the id of the requester.
- `operatorId`: the id of the operator to withdraw from.
- `amount`: the amount to withdraw.

Events

- `RequestNominatorCollateralWithdrawal(requestId, nominatorId, operatorId, maturity, amount)`

Errors

- `VaultNotOptedInToNomination`: the `operatorId` is not an Operator.
- `NominatorNotFound`: the `nominatorId` is not a Nominator.
- `TooLittleNominatedCollateral`: the caller has requested to withdraw more collateral than it owns.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Check `nominatorId` is a nominator. If not, throw `NominatorNotFound`.
3. Check if the caller has at least as much nominated collateral as `amount`. If not, throw `TooLittleNominatedCollateral`.
4. Add the withdrawal request to the `pendingWithdrawals` array in the `Nominator` struct for `nominatorId`, inside the `Operator` struct of `operatorId`.
5. Decrease the `backing_collateral` of `operatorId` in the *Vault Registry*.

11.4.8 executeOperatorWithdrawal

Execute all matured (unbonded) withdrawal requests of an operator.

Specification

Function Signature

`executeOperatorWithdrawal(operatorId)`

Parameters

- `operatorId`: the id of the requester.

Events

- `ExecuteOperatorCollateralWithdrawal(operatorId, unbondedCollateral)`

Errors

- `VaultNotOptedInToNomination`: the `operatorId` is not an Operator.
- `NoMaturedCollateral`: either no collateral withdrawal has been requested, or the requests have not matured yet.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Iterate through the `withdrawalRequests` in the `Operator` struct to determine how much collateral was unbonded, removing matured requests.
3. If there is zero unbonded collateral, throw `NoMaturedCollateral`.

11.4.9 executeNominatorWithdrawal

Execute all matured (unbonded) withdrawal requests of a nominator.

Specification

Function Signature

`executeNominatorWithdrawal(nominatorId, operatorId)`

Parameters

- `nominatorId`: the id of the requester.
- `operatorId`: the id of the operator.

Events

- `ExecuteNominatorCollateralWithdrawal(nominatorId, operatorId, unbondedCollateral)`

Errors

- `VaultNotOptedInToNomination`: the `operatorId` is not an Operator.

- `NoMaturedCollateral`: either no collateral withdrawal has been requested, or the requests have not matured yet.
- `NominatorNotFound`: the `nominatorId` is not a Nominator.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Check `nominatorId` is a nominator. If not, throw `NominatorNotFound`.
3. Iterate through the `withdrawalRequests` array in the `Nominator` struct inside the `Operator` struct for `operatorId`. Determine how much collateral was unbonded, removing matured requests.
4. If there is zero unbonded collateral, throw `NoMaturedCollateral`.

11.4.10 cancelOperatorWithdrawal

Cancel an operator's withdrawal request.

Specification

Function Signature

`cancelOperatorWithdrawal(operatorId, requestId)`

Parameters

- `operatorId`: the id of the operator.
- `requestId`: the id of the withdrawal request.

Events

- `CancelOperatorCollateralWithdrawal(requestId, operatorId)`

Errors

- `VaultNotOptedInToNomination`: the `operatorId` is not an Operator.
- `WithdrawalRequestNotFound`: no withdrawal request found for the given id.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Check `requestId` corresponds to an actual withdrawal request. If not, throw `WithdrawalRequestNotFound`.
3. Remove the withdrawal request from the `withdrawalRequests` array in the `Operator` struct for `operatorId`.
4. Increase the backing collateral of `operatorId` in the *Vault Registry* by the amount in the withdrawal request.

11.4.11 cancelNominatorWithdrawal

Cancel a nominator's withdrawal request.

Specification

Function Signature

`cancelNominatorWithdrawal(nominatorId, operatorId, requestId)`

Parameters

- `nominatorId`: the id of the nominator.
- `operatorId`: the id of the operator.
- `requestId`: the id of the withdrawal request.

Events

- `CancelNominatorCollateralWithdrawal(requestId, nominatorId, operatorId)`

Errors

- `VaultNotOptedInToNomination`: the `operatorId` is not an Operator.
- `NominatorNotFound`: the `nominatorId` is not a Nominator.
- `WithdrawalRequestNotFound`: no withdrawal request found for the given id.

Preconditions

- The BTC Parachain status in the *Security* component must be set to `RUNNING:0`.

Function Sequence

1. Check `operatorId` is an operator. If not, throw `VaultNotOptedInToNomination`.
2. Check `nominatorId` is a nominator. If not, throw `NominatorNotFound`.
3. Check `requestId` corresponds to an actual withdrawal request. If not, throw `WithdrawalRequestNotFound`.
3. Remove the withdrawal request from the `withdrawalRequests` array in the `Nominator` struct inside the `Operator` struct for `operatorId`.
4. Increase the backing collateral of `operatorId` in the *Vault Registry* by the amount in the withdrawal request.

11.5 Events

11.5.1 NominationOptIn

Event Signature

`NominationOptIn(account)`

Parameters

- `account`: the id of the operator who opten in

Functions

- *optInToNomination*

11.5.2 NominationOptOut

Event Signature

`NominationOptOut(account)`

Parameters

- `account`: the id of the operator who opten out

Functions

- *optOutOfNomination*

11.5.3 IncreaseNominatedCollateral

Event Signature

`IncreaseNominatedCollateral(nominatorId, operatorId, amount)`

Parameters

- `nominatorId`: the id of the nominator who is depositing collateral
- `operatorId`: the id of the operator who receives the nomination
- `amount`: the amount of nominated collateral

Functions

- *depositNominatedCollateral*

11.5.4 RequestOperatorCollateralWithdrawal

Event Signature

`RequestOperatorCollateralWithdrawal(requestId, operatorId, maturityBlock, amount)`

Parameters

- `requestId`: the id of the request
- `operatorId`: the id of the operator withdrawing collateral
- `maturityBlock`: the block when the request can be executed
- `amount`: the amount to withdraw

Functions

- *requestOperatorCollateralWithdrawal*

11.5.5 ExecuteOperatorCollateralWithdrawal

Event Signature

`ExecuteOperatorCollateralWithdrawal(operatorId, amount)`

Parameters

- `operatorId`: the id of the operator withdrawing collateral
- `amount`: the withdrawn amount

Functions

- *executeOperatorWithdrawal*

11.5.6 CancelOperatorCollateralWithdrawal

Event Signature

`CancelOperatorCollateralWithdrawal(requestId, operatorId)`

Parameters

- `requestId`: the id of the withdrawal request to cancel
- `operatorId`: the id of the operator who requested the withdrawal

Functions

- *cancelOperatorWithdrawal*

11.5.7 RequestNominatorCollateralWithdrawal

Event Signature

`RequestNominatorCollateralWithdrawal(requestId, nominatorId, operatorId, maturityBlock, amount)`

Parameters

- `requestId`: the id of the request
- `nominatorId`: the id of the operator withdrawing collateral
- `operatorId`: the id of the operator who nominated collateral is being withdrawn
- `maturityBlock`: the block when the request can be executed
- `amount`: the amount to withdraw

Functions

- *requestNominatorCollateralWithdrawal*

11.5.8 ExecuteNominatorCollateralWithdrawal

Event Signature

`ExecuteNominatorCollateralWithdrawal(nominatorId, operatorId, amount)`

Parameters

- `nominatorId`: the id of the operator withdrawing collateral
- `operatorId`: the id of the operator who nominated collateral is being withdrawn
- `amount`: the withdrawn amount

Functions

- *executeNominatorWithdrawal*

11.5.9 CancelNominatorCollateralWithdrawal

Event Signature

`CancelNominatorCollateralWithdrawal(requestId, nominatorId, operatorId)`

Parameters

- `requestId`: the id of the withdrawal request to cancel
- `nominatorId`: the id of the nominator who requested the withdrawal
- `operatorId`: the id of the operator who nominated collateral is being withdrawn

Functions

- *cancelNominatorWithdrawal*

CHAPTER

TWELVE

REDEEM

12.1 Overview

The redeem module allows a user to receive BTC on the Bitcoin chain in return for destroying an equivalent amount of interbtc on the BTC Parachain. The process is initiated by a user requesting a redeem with a vault. The vault then needs to send BTC to the user within a given time limit. Next, the vault has to finalize the process by providing a proof to the BTC Parachain that he has send the right amount of BTC to the user. If the vault fails to deliver a valid proof within the time limit, the user can claim an equivalent amount of DOT from the vault's locked collateral to reimburse him for his loss in BTC.

Moreover, as part of the liquidation procedure, users are able to directly exchange interbtc for DOT. To this end, a user is able to execute a special liquidation redeem if one or multiple vaults have been liquidated.

12.1.1 Step-by-step

1. Precondition: A user owns interbtc.
2. A user locks an amount of interbtc by calling the *requestRedeem* function. In this function call, the user selects a vault to execute the redeem request from the list of vaults. The function creates a redeem request with a unique hash.
3. The selected vault listens for the `RequestRedeem` event emitted by the user. The vault then proceeds to transfer BTC to the address specified by the user in the *requestRedeem* function including a unique hash in the `OP_RETURN` of one output.
4. The vault executes the *executeRedeem* function by providing the Bitcoin transaction from step 3 together with the redeem request identifier within the time limit. If the function completes successfully, the locked interbtc are destroyed and the user received its BTC.
5. Optional: If the user could not receive BTC within the given time (as required in step 4), the user calls *cancelRedeem* after the redeem time limit. The user can choose either to reimburse, or to retry. In case of reimbursement, the user transfer ownership of the tokens to the vault, but receives collateral in exchange. In case of retry, the user gets back its tokens. In either case, the user is given some part of the vault's collateral as compensation for the inconvenience. In addition, some amount (depending on the vault's SLA) of collateral is transferred from the vault to the fee pool.
    a. Optional: If during a *cancelRedeem* the user selects reimbursement, and as a result the vault becomes undercollateralized, then vault does not receive the user's tokens - they are burned, and the vault's `issuedTokens` decreases. When, at some later point, it gets sufficient colalteral, it can call *mintTokensForReimbursedRedeem* to get the tokens.

12.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

12.1.3 Vault Registry

The data access and state changes to the vault registry are documented in Fig. 12.1 below.

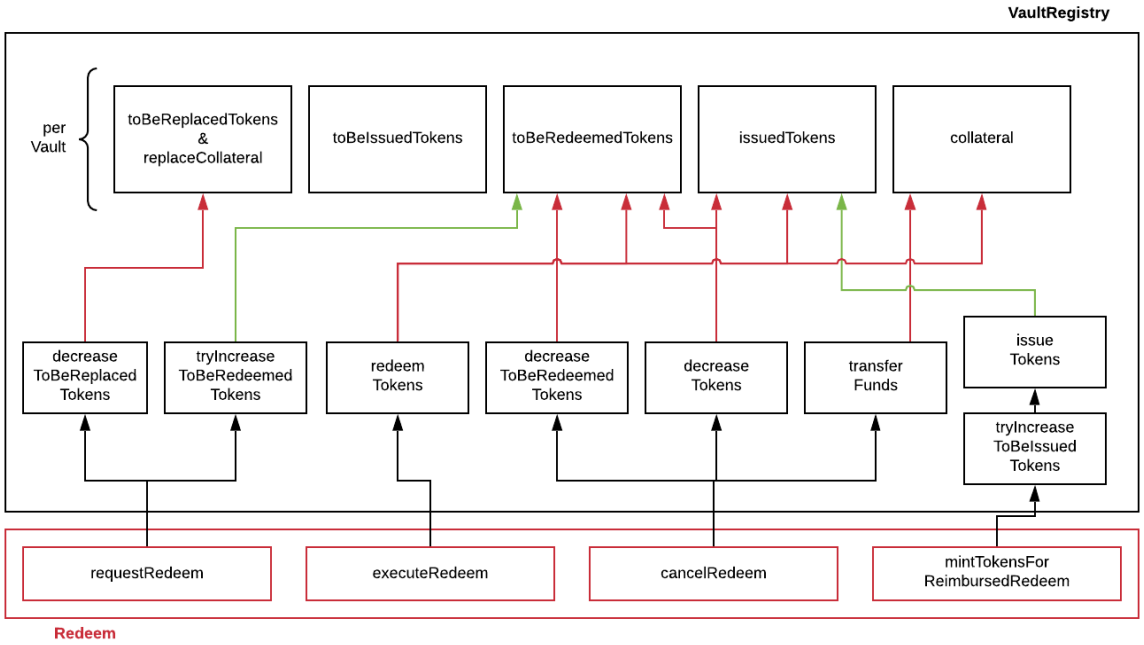


Fig. 12.1: The redeem module interacts through three different functions with the vault registry. The green arrow indicate an increase, the red arrows a decrease.

12.1.4 Fee Model

When the user makes a redeem request for a certain amount, it will actually not receive that amount of BTC. This is because there are two types of fees subtracted. First, in order to be able to pay the bitcoin transaction cost, the vault is given a budget to spend on on the bitcoin inclusion fee, based on *RedeemTransactionSize* and the inclusion fee estimates reported by the oracle. The actual amount spent on the inclusion fee is not checked. If the vault does not spend the whole budget, it can keep the surplus, although it will not be able to spend it without being liquidated for theft. It may at some point want to withdraw all of its collateral and then to move its bitcoin into a new account. The second fee that the user pays for is the parachain fee that goes to the fee pool to incentivize the various participants in the system.

The main accounting changes of a successful redeem is summarized below. See the individual functions for more details.

- `redeem.amountBTC` bitcoin is transferred to the user.
- `redeem.amountBTC + redeem.fee + redeem.transferFeeBTC` is burned from the user.
- The vault's `issuedTokens` decreases by `redeem.amountBTC + redeem.transferFeeBTC`.
- The fee pool content increases by `redeem.fee`.

12.2 Data Model

12.2.1 Scalars

RedeemPeriod

The time difference between when an redeem request is created and required completion time by a vault. Concretely, this period is the amount by which *ActiveBlockCount* is allowed to increase before the redeem is considered to be expired. The period has an upper limit to ensure the user gets his BTC in time and to potentially punish a vault for inactivity or stealing BTC. Each redeem request records the value of this field upon creation, and when checking the expiry, the maximum of the current RedeemPeriod and the value as recorded in the RedeemRequest is used. This way, users are not negatively impacted by a change in the value.

RedeemTransactionSize

The expected size in bytes of a redeem. This is used to set the bitcoin inclusion fee budget.

RedeemBtcDustValue

The minimal amount in BTc a vault can be asked to transfer to the user. Note that this is not equal to the amount requests, since an inclusion fee is deducted from that amount.

12.2.2 Maps

RedeemRequests

Users create redeem requests to receive BTC in return for interbtc. This mapping provides access from a unique hash `redeemId` to a `Redeem` struct. `<redeemId, Redeem>`.

12.2.3 Structs

Redeem

Stores the status and information about a single redeem request.

Parameter	Type	Description
`vault`	Account	The BTC Parachain address of the vault responsible for this redeem request.
`opentime`	u32	The *ActiveBlockCount* when the redeem request was made. Serves as start for the countdown until when the vault must transfer the BTC.
`period`	u32	Value of *RedeemPeriod* when the redeem request was made, in case that value changes while this redeem is pending.
`amountBTC`	BTC	Amount of BTC to be sent to the user.
`transferFeeBTC`	BTC	Budget for the vault to spend in bitcoin inclusion fees.
`fee`	interbtc	Parachain fee: amount to be transferred from the user to the fee pool upon completion of the redeem.
`premiumDOT`	DOT	Amount of DOT to be paid as a premium to this user (if the Vault's collateral rate was below `PremiumRedeemThreshold` at the time of redeeming).
`redeemer`	Account	The BTC Parachain address of the user requesting the redeem.
`btcAddress`	bytes[20]	Base58 encoded Bitcoin public key of the User.
`btcHeight`	u32	Height of newest bitcoin block in the relay at the time the request is accepted. This is used by the clients upon startup, to determine how many blocks of the bitcoin chain they need to inspect to know if a payment has been made already.
`status`	enum	The status of the redeem: `Pending`, `Completed`, `Retried` or `Reimbursed(bool)`, where bool=true indicates that the vault minted tokens for the amount that the redeemer burned

12.3 Functions

12.3.1 requestRedeem

A user requests to start the redeem procedure. This function checks the BTC Parachain status in *Security* and decides how the redeem process is to be executed. The following modes are possible:

- **Normal Redeem** - no errors detected, full BTC value is to be Redeemed.
- **Premium Redeem** - the selected Vault's collateral rate has fallen below `PremiumRedeemThreshold`. Full BTC value is to be redeemed, but the user is allocated a premium in DOT (`RedeemPremiumFee`), taken from the Vault's to-be-released collateral.

Specification

Function Signature

`requestRedeem(redeemer, amountinterbtc, btcAddress, vault)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountinterbtc`: the amount of interbtc to destroy and BTC to receive.
- `btcAddress`: the address to receive BTC.
- `vault`: the vault selected for the redeem request.

Returns

- `redeemId`: A unique hash identifying the redeem request.

Events

- `RequestRedeem(redeemId, redeemer, amount, vault, btcAddress)`

Preconditions

Let `burnedTokens` be `amountinterbtc` minus the result of the multiplication of *RedeemFee* and `amountinterbtc`. Then:

- The function call MUST be signed by *redeemer*.
- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- The selected vault MUST NOT be banned.
- The selected vault MUST NOT be liquidated.
- The redeemer MUST have at least `amountinterbtc` free tokens.
- `burnedTokens` minus the inclusion fee MUST be above the *RedeemBtcDustValue*, where the inclusion fee is the multiplication of *RedeemTransactionSize* and the fee rate estimate reported by the oracle.
- The vault's `issuedTokens` MUST be at least `vault.toBeRedeemedTokens + burnedTokens`.

Postconditions

Let `burnedTokens` be `amountinterbtc` minus the result of the multiplication of *RedeemFee* and `amountinterbtc`. Then:

- The vault's `toBeRedeemedTokens` MUST increase by `burnedTokens`.
- `amountinterbtc` of the redeemer's tokens MUST be locked by this transaction.
- *decreaseToBeReplacedTokens* MUST be called, supplying `vault` and `burnedTokens`. The returned `replaceCollateral` MUST be released by this function.
- **A new `RedeemRequest` MUST be added to the `RedeemRequests` map, with the following value:**
  -
  - `redeem.vault` is the requested `vault`
  - `redeem.opentime` is the current *ActiveBlockCount*
  - `redeem.fee` is *RedeemFee* multiplied by `amountinterbtc`,
  - `redeem.transferFeeBtc` is the inclusion_fee, which is the multiplication of *RedeemTransactionSize* and the fee rate estimate reported by the oracle,
  - `redeem.amount_btc` is `amountinterbtc - redeem.fee - redeem.transferFeeBtc`,
  - `redeem.period` is the current value of the *RedeemPeriod*,
  - `redeem.redeemer` is the `redeemer` argument,
  - `redeem.btc_address` is the `btcAddress` argument,
  - `redeem.btc_height` is the current height of the btc relay,
  - `redeem.status` is `Pending`,
  - If the vault's collateralization rate is above the *PremiumRedeemThreshold*, then `redeem.premium` is `0`,
  - If the vault's collateralization rate is below the *PremiumRedeemThreshold*, then `redeem.premium` is *PremiumRedeemFee* multiplied by the worth of `redeem.amount_btc`,

12.3.2 liquidationRedeem

A user executes a liquidation redeem that exchanges interbtc for DOT from the *LiquidationVault*. The 1:1 backing is being recovered, hence this function burns interbtc without releasing any BTC.

Specification

Function Signature

`liquidationRedeem(redeemer, amountinterbtc)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountinterbtc`: the amount of interbtc to destroy.

Events

- `RequestRedeem(redeemID, redeemer, redeemAmountWrapped, feeWrapped, premium, vaultID, userBtcAddress, transferFeeBtc)`

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- The function call MUST be signed.
- The redeemer MUST have at least `amountinterbtc` free tokens.

Postconditions

- `amountinterbtc` tokens MUST be burned from the user.
- *redeemTokensLiquidation* MUST be called with `redeemer` and `amountinterbtc` as arguments.

12.3.3 executeRedeem

A vault calls this function after receiving an `RequestRedeem` event with his public key. Before calling the function, the vault transfers the specific amount of BTC to the BTC address given in the original redeem request. The vault completes the redeem with this function.

Specification

Function Signature

`executeRedeem(vault, redeemId, merkleProof, rawTx)`

Parameters

- `vault`: the vault responsible for executing this redeem request.
- `redeemId`: the unique hash created during the `requestRedeem` function.
- `merkleProof`: Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- `ExecuteRedeem(redeemer, redeemId, amount, vault):`

Preconditions

- The function call MUST be signed be *someone*, i.e. not necessarily the *redeemer*.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.

- A *pending* `RedeemRequest` MUST exist with an id equal to `redeemId`.
- The request MUST NOT have expired.
- – The `rawTx` MUST decode to a valid transaction that transfers at least the amount specified in the `RedeemRequest` struct. It MUST be a transaction to the correct address, and provide the expected OP_RETURN, based on the `RedeemRequest`.
- The `merkleProof` MUST contain a valid proof of of `rawTX`.
- The bitcoin payment MUST have been submitted to the relay chain, and MUST have sufficient confirmations.

Postconditions

- `redeemRequest.amount_btc - redeemRequest.transferFeeBtc` of the tokens in the redeemer's account MUST be burned.
- `redeemRequest.fee` MUST be unlocked and transferred from the redeemer's account to the fee pool.
- *redeemTokens* MUST be called, supplying `redeemRequest.vault`, `redeemRequest.amountBtc - redeemRequest.transferFeeBtc`, `redeemRequest.premium` and `redeemRequest.redeemer` as arguments.
- `redeemRequest.status` MUST be set to `Completed`.

12.3.4 cancelRedeem

If a redeem request is not completed on time, the redeem request can be cancelled. The user that initially requested the redeem process calls this function to obtain the Vault's collateral as compensation for not refunding the BTC back to his address.

The failed vault is banned from further issue, redeem and replace requests for a pre-defined time period (`PunishmentDelay` as defined in *Vault Registry*).

The user is able to choose between reimbursement and retrying. If the user chooses the retry, it gets back the tokens, and a punishment fee is transferred from the vault to the user. If the user chooses reimbursement, then he receives the equivalent worth of the tokens in collateral, plus a punishment fee. In this case, the tokens are transferred from the user to the vault. In either case, the vault may also be slashed an additional punishment that goes to the fee pool.

With the SLA model additions, the punishment fee paid to the user stays constant (i.e., the user always receives the punishment fee of e.g. 10%). However, vaults may be slashed more than the punishment fee, as determined by the SLA. The surplus slashed collateral is routed into the Parachain Fee pool and handled like regular fee income. For example, if the vault is punished with 20%, 10% punishment fee is paid to the user and 10% is paid to the fee pool.

Specification

Function Signature

`cancelRedeem(redeemId, reimburse)`

Parameters

- `redeemId`: the unique hash of the redeem request.
- `reimburse`: boolean flag, specifying if the user wishes to be reimbursed in DOT and slash the vault, or wishes to keep the interbtc (and retry to redeem with another Vault).

Events

`CancelRedeem(redeemId, redeemer, amountBtc, fee, vault)`: Emits an event with the `redeemId` that is cancelled.

Preconditions

- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- A *pending* `RedeemRequest` MUST exist with an id equal to `redeemId`.
- The function call MUST be signed by `redeemRequest.redeemer`, i.e. this function can only be called by the account who made the redeem request.
- The request MUST be expired.

Postconditions

Let `amountIncludingParachainFee` be equal to the worth in collateral of `redeem.amount_btc + redeem.transfer_fee_btc`. Then:

- If the vault is liquidated, the redeemer MUST be transferred part of the vault's collateral: an amount of `vault.backingCollateral * ((amountIncludingParachainFee) / vault.to_be_redeemed_tokens)`.
- **If the vault is *not* liquidated, the fellowing collateral changes are made:**
  - If `reimburse` is true, the user SHOULD be reimbursed the worth of `amountIncludingParachainFee` in collateral. The transfer MUST be saturating, i.e. if the amount is not available, it should transfer whatever amount *is* available.
  - A punishment fee SHOULD be tranferred from the vault's backing collateral to the reedeemer: an amount of *PunishmentFee* times the worth of `amountIncludingParachainFee`. The transfer MUST be saturating, i.e. if the amount is not available, it should transfer whatever amount *is* available.
  - An additional punishment fee SHOULD be transferred to the fee pool: an amount ranging from *LiquidationCollateralThreshold* to *PremiumRedeemThreshold* times the worth of `amountIncludingParachainFee`, depending on the vault's SLA. The transfer MUST be saturating, i.e. if the amount is not available, it should transfer whatever amount *is* available.
- **If `reimburse` is true:**
  - `redeem.fee` MUST be transferred from the vault to the fee pool.
  - **If after the loss of collateral the vault is below the *SecureCollateralThreshold*:**
    - `amountIncludingParachainFee` of the user's tokens are *burned*.
    - *decreaseTokens* MUST be called, supplying the vault, the user, and `amountIncludingParachainFee` as arguments.
    - The `redeem.status` is set to `Reimbursed(false)`, where the `false` indicates that the vault has not yet received the tokens.
  - **If after the loss of collateral the vault remains above the *SecureCollateralThreshold*:**
    - `amountIncludingParachainFee` of the user's tokens MUST be unlocked and transferred to the vault.
    - *decreaseToBeRedeemedTokens* MUST be called, supplying the vault and `amountIncludingParachainFee` as arguments.
    - The `redeem.status` is set to `Reimbursed(true)`, where the `true` indicates that the vault has received the tokens.
- **If `reimburse` is false:**
  - All the user's tokens that were locked in *requestRedeem* MUST be unlocked, i.e. an amount of `redeem.amount_btc + redeem.fee + redeem.transfer_fee_btc`.
  - The vault's `toBeRedeemedTokens` MUST decrease by `amountIncludingParachainFee`.
- The vault MUST be banned.

12.3.5 mintTokensForReimbursedRedeem

If a redeemrequest has the status `Reimbursed(false)`, the vault was unable to back the to be received tokens at the time of the `cancelRedeem`. After gaining sufficient collateral, the vault can call this function to finally get its tokens.

Specification

Function Signature

`mintTokensForReimbursedRedeem(vault, redeemId)`

Parameters

- `redeemId`: the unique hash of the redeem request.
- `reimburse`: boolean flag, specifying if the user wishes to be reimbursed in DOT and slash the vault, or wishes to keep the interbtc (and retry to redeem with another Vault).

Events

`MintTokensForReimbursedRedeem(vaultId, redeemId, amountMinted)`

Preconditions

- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- A `RedeemRequest` MUST exist with an id equal to `redeemId`.
- `redeem.status` MUST be `Reimbursed(false)`.
- The vault MUST have sufficient collateral to remain above the *SecureCollateralThreshold* after issuing `redeem.amount_btc + redeem.transfer_fee_btc` tokens.
- The vault MUST NOT be banned.

Postconditions

- The function call MUST be signed by `redeem.vault`, i.e. this function can only be called by the the vault.
- *tryIncreaseToBeIssuedTokens* and *issueTokens* MUST be called, both with the vault and `redeem.amount_btc + redeem.transfer_fee_btc` as arguments.
- `redeem.amount_btc + redeem.transfer_fee_btc` tokens MUST be minted to the vault.

12.4 Events

12.4.1 RequestRedeem

Emit an event when a redeem request is created. This event needs to be monitored by the vault to start the redeem request.

Event Signature

- `RequestRedeem(redeemID, redeemer, redeemAmountWrapped, feeWrapped, premium, vaultID, userBtcAddress, transferFeeBtc)`

Parameters

- `redeemID`: the unique identifier of this redeem request.
- `redeemer`: address of the user triggering the redeem.
- `redeemAmountWrapped`: the amount to be received by the user.
- `feeWrapped`: the fee to be given to the foo pool.

- `premium`: the premium to be given to the user, if any.
- `vaultID`: the vault selected for the redeem request.
- `userBtcAddress`: the address the vault is to transfer the funds to.
- `transferFeeBtc`: the budget the vault has to spend on bitcoin inclusion fees, paid for by the user.

Functions

- ref:*requestRedeem*

12.4.2 LiquidationRedeem

Emit an event when a user does a liquidation redeem.

Event Signature

`LiquidationRedeem(redeemer, amountinterbtc)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountinterbtc`: the amount of interbtc to burned.

Functions

- ref:*liquidationRedeem*

12.4.3 ExecuteRedeem

Emit an event when a redeem request is successfully executed by a vault.

Event Signature

`ExecuteRedeem(redeemer, redeemId, amountinterbtc, vault)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `redeemId`: the unique hash created during the `requestRedeem` function.
- `amountinterbtc`: the amount of interbtc to destroy and BTC to receive.
- `vault`: the vault responsible for executing this redeem request.

Functions

- ref:*executeRedeem*

12.4.4 CancelRedeem

Emit an event when a user cancels a redeem request that has not been fulfilled after the `RedeemPeriod` has passed.

Event Signature

`CancelRedeem(redeemId, redeemer, amountBtc, fee, vault)`

Parameters

- `redeemId`: the unique hash of the redeem request.
- `redeemer`: The redeemer starting the redeem process.
- `amountBtc`: the amount that was to be received by the user.

- `fee`: the parachain fee that was to be added to the fee pool upon a successful redeem.
- `vault`: the vault who failed to execute the redeem.

Functions

- ref:*cancelRedeem*

12.4.5 MintTokensForReimbursedRedeem

Emit an event when a vault minted the tokens corresponding the a cancelled redeem that was reimbursed to the user, when the vault did not have sufficient collateral at the time of the cancellation to back the tokens.

Event Signature

`MintTokensForReimbursedRedeem(vaultId, redeemId, amountMinted)`

Parameters

- `vault`: if of the vault that now mints the tokens.
- `redeemId`: the unique hash of the redeem request.
- `amountMinted`: the amount that the vault just minted.

Functions

- ref:*mintTokensForReimbursedRedeem*

12.5 Error Codes

`ERR_VAULT_NOT_FOUND`

- **Message**: "There exists no vault with the given account id."
- **Function**: *requestRedeem*, *liquidationRedeem*
- **Cause**: The specified vault does not exist.

`ERR_AMOUNT_EXCEEDS_USER_BALANCE`

- **Message**: "The requested amount exceeds the user's balance."
- **Function**: *requestRedeem*, *liquidationRedeem*
- **Cause**: If the user is trying to redeem more BTC than his interbtc balance.

`ERR_VAULT_BANNED`

- **Message**: "The selected vault has been temporarily banned."
- **Function**: *requestRedeem*
- **Cause**: Redeem requests are not possible with temporarily banned Vaults

`ERR_AMOUNT_EXCEEDS_VAULT_BALANCE`

- **Message**: "The requested amount exceeds the vault's balance."
- **Function**: *requestRedeem*, *liquidationRedeem*
- **Cause**: If the user is trying to redeem from a vault that has less BTC locked than requested for redeem.

`ERR_REDEEM_ID_NOT_FOUND`

- **Message**: "The `redeemId` cannot be found."
- **Function**: *executeRedeem*
- **Cause**: The `redeemId` in the `RedeemRequests` mapping returned `None`.

`ERR_REDEEM_PERIOD_EXPIRED`

- **Message**: "The redeem period expired."
- **Function**: *executeRedeem*
- **Cause**: The time limit as defined by the `RedeemPeriod` is not met.

`ERR_UNAUTHORIZED`

- **Message**: "Caller is not authorized to call this function."
- **Function**: *cancelRedeem* | *mintTokensForReimbursedRedeem*
- **Cause**: Only the user can call *cancelRedeem*, and only the vault can call *mintTokensForReimbursedRedeem*.

`ERR_REDEEM_PERIOD_NOT_EXPIRED`

- **Message**: "The period to complete the redeem request is not yet expired."
- **Function**: *cancelRedeem*
- **Cause**: Raises an error if the time limit to call `executeRedeem` has not yet passed.

`ERR_REDEEM_CANCELLED`

- **Message**: "The redeem is in an unexpected cancelled state."
- **Function**: *cancelRedeem* | *mintTokensForReimbursedRedeem* | *executeRedeem*
- **Cause**: The status of the redeem is not as required for this call.

`ERR_REDEEM_COMPLETED`

- **Message**: "The redeem is already completed."
- **Function**: *cancelRedeem* | *executeRedeem*
- **Cause**: The status of the redeem is not as expected for this call.

CHAPTER

THIRTEEN

REFUND

13.1 Overview

The Refund module is a user failsafe mechanism. In case a user accidentally locks more Bitcoin than the actual issue request, the refund mechanism seeks to ensure that either (1) the initial issue request is increased to issue more interbtc or (2) the BTC are returned to the sending user.

13.1.1 Step-by-step

If a user falsely sends additional BTC (i.e., $|\text{BTC}| > |\text{interbtc}|$) during the issue process:

1. **Case 1: The originally selected vault has sufficient collateral locked to cover the entire BTC amount sent by the user:**

    a. Increase the issue request interbtc amount and the fee to reflect the actual BTC amount paid by the user.

    b. As before, issue the interbtc to the user and forward the fees.

    c. Emit an event that the issue amount was increased.

2. **Case 2: The originally selected vault does NOT have sufficient collateral locked to cover the additional BTC amount s**

    a. Automatically create a return request from the issue module that includes a return fee (deducted from the originial BTC payment) paid to the vault returning the BTC.

    b. The vault fulfills the return request via a transaction inclusion proof (similar to execute issue). However, this does not create a new interbtc.

Note: Only case 2 is handled in this module. Case 1 is handled directly by the issue module.

Note: Normally, enforcing actions by a vault is achieved by locking collateral of the vault and slashing the vault in case of misbehavior. In the case where a user sends too many BTC and the vault does not have enough "free" collateral left, we cannot lock more collateral. However, the original vault cannot move the additional BTC sent as this would be flagged as theft and the vault would get slashed. The vault can possibly take the overpaid BTC though if the vault would not be backing any interbtc any longer (e.g. due to redeem/replace).

13.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

CHAPTER

FOURTEEN

REPLACE

14.1 Overview

The Replace module allows a vault (*OldVault*) to be replaced by transferring the BTC it is holding locked to another vault (*NewVault*), which provides the necessary DOT collateral. As a result, the DOT collateral of the *OldVault*, corresponding to the amount of replaced BTC, is unlocked. The *OldVault* must thereby provide some amount of collateral to protect against griefing attacks, where the *OldVault* never finalizes the Replace protocol and the *NewVault* hence temporarily locked DOT collateral for nothing.

The *OldVault* is responsible for ensuring that it has sufficient BTC to pay for the transaction fees.

Conceptually, the Replace protocol resembles a SPV atomic cross-chain swap.

14.1.1 Step-by-Step

1. Precondition: a vault (*OldVault*) has locked DOT collateral in the Vault Registry and has issued interbtc tokens, i.e., holds BTC on Bitcoin.
2. *OldVault* submits a replacement request, indicating how much BTC is to be migrated by calling the *requestReplace* function.
    - *OldVault* is required to lock some amount of DOT collateral (*ReplaceGriefingCollateral*) as griefing protection, to prevent *OldVault* from holding *NewVault*'s DOT collateral locked in the BTC Parachain without ever finalizing the redeem protocol (transfer of BTC).
3. Optional: If an *OldVault* has changed its mind or can't find a *NewVault* to replace it, it can call the *withdrawReplaceRequest* function to withdraw the request of some amount. For example, if *OldVault* requested a replacement for 10 tokens, and 2 tokens have been accepted by some *NewVault*, then it can withdraw up to 8 tokens from being replaced.
4. A new candidate vault (*NewVault*), commits to accepting the replacement by locking up the necessary DOT collateral to back the to-be-transferred BTC (according to the `SecureCollateralThreshold`) by calling the *acceptReplace* function..
    - Note: from the *OldVault*'s perspective a redeem is very similar to an accepted replace. That is, its goal is to get rid of tokens, and it is not important if this is achieved by a user redeeming, or by a vault accepting the replace request. As such, when a user requests a redeem with a vault that has requested a replacement, the *OldVault*'s `toBeReplacedTokens` is decreased by the amount of tokens redeemed by the user. The reserved griefing collateral for this replace is then released.
5. Within a pre-defined delay, *OldVault* must release the BTC on Bitcoin to *NewVault*'s BTC address, and submit a valid transaction inclusion proof by calling the *executeReplace* function (call to `verifyTransactionInclusion` in *BTC-Relay*). If *OldVault* releases the BTC to *NewVault* correctly and submits the transaction inclusion proof to Replace module on time, *OldVault*'s DOT collateral is released - *NewVault* has now replaced *OldVault*.
    - Note: as with redeems, to prevent *OldVault* from trying to re-use old transactions (or other payments to *NewVaults* on Bitcoin) as fake proofs, we require *OldVault* to include a `nonce` in an OP_RETURN output of the transfer transaction on Bitcoin.

6. Optional: If *OldVault* fails to provide the correct transaction inclusion proof on time, the *NewVault*'s `collateral` is unlocked and *OldVault*'s `griefingCollateral` is sent to the *NewVault* as reimbursement for the opportunity costs of locking up DOT collateral via the *cancelReplace*.

14.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

14.1.3 Vault Registry

The data access and state changes to the vault registry are documented in Fig. 14.1 below.

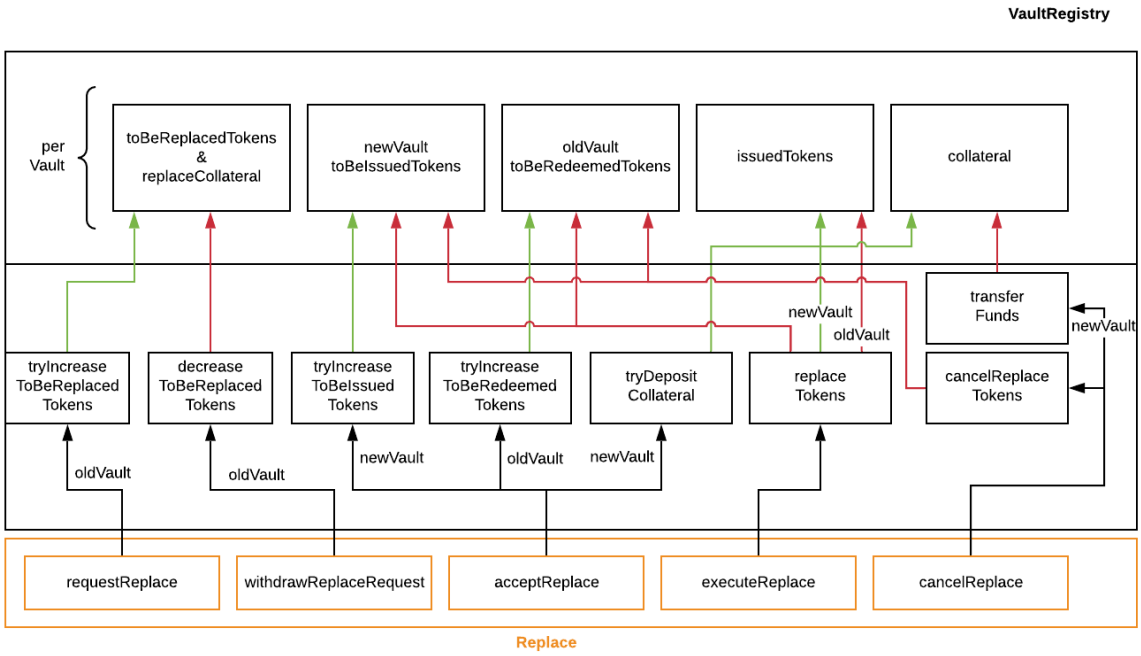


Fig. 14.1: The replace module interacts with functions in the vault registry to handle updating token balances of vaults. The green lines indicate an increase, the red lines a decrease.

14.1.4 Fee Model

Following additions are added if the fee model is integrated.

- If a replace request is canceled, the griefing collateral is transferred to the new_vault.
- If a replace request is executed, the griefing collateral is transferred to the old_vault.

14.2 Data Model

14.2.1 Scalars

ReplaceBtcDustValue

The minimum amount a *newVault* can accept - this is to ensure the *oldVault* is able to make the bitcoin transfer. Furthermore, it puts a limit on the transaction fees that the *oldVault* needs to pay.

ReplacePeriod

The time difference between a replace request is accepted by another vault and the transfer of BTC (and submission of the transaction inclusion proof) by the to-be-replaced Vault. Concretely, this period is the amount by which *ActiveBlockCount* is allowed to increase before the redeem is considered to be expired. The replace period has an upper limit to prevent griefing of vault collateral. Each accepted replace request records the value of this field upon creation, and when checking the expiry, the maximum of the current ReplacePeriod and the value as recorded in the ReplaceRequest is used. This way, vaults are not negatively impacted by a change in the value.

14.2.2 Maps

ReplaceRequests

Vaults create replace requests if they want to have (a part of) their DOT collateral to be replaced by other Vaults. This mapping provides access from a unique hash `ReplaceId` to a `ReplaceRequest` struct. `<ReplaceId, Replace>`.

14.2.3 Structs

Replace

Stores the status and information about a single replace request.

Parameter	Type	Description
`oldVault`	Account	Account of the vault that is to be replaced.
`newVault`	Account	Account of the new vault, which accepts the replace request.
`amount`	interbtc	Amount of BTC / interbtc to be replaced.
`griefingCollateral`	DOT	Griefing protection collateral locked by *oldVault*.
`collateral`	DOT	DOT collateral locked by the new Vault.
`acceptTime`	u32	The *ActiveBlockCount* when the replace request was accepted by a new Vault. Serves as start for the countdown until when the old vault must transfer the BTC.
`btcAddress`	bytes[20]	Base58 encoded Bitcoin public key of the new Vault.
`btcHeight`	u32	Height of newest bitcoin block in the relay at the time the request is accepted. This is used by the clients upon startup, to determine how many blocks of the bitcoin chain they need to inspect to know if a payment has been made already.
`period`	u32	Value of *ReplacePeriod* when the redeem request was made, in case that value changes while this replace is pending.
`status`	Enum	Status of the request: Pending, Completed or Cancelled

Note: The `btcAddress` parameter is not to be set by the new vault, but is extracted from the `Vaults` mapping in `VaultRegistry` for the account of the new Vault.

14.3 Functions

14.3.1 requestReplace

An *OldVault* (to-be-replaced Vault) submits a request to be (partially) replaced. If it requests more than it can fulfil (i.e. the sum of `toBeReplacedTokens` and `toBeRedeemedTokens` exceeds its `issuedTokens`, then the request amount is reduced such that the sum of `toBeReplacedTokens` and `toBeRedeemedTokens` is exactly equal to `issuedTokens`.

Specification

Function Signature

`requestReplace(oldVault, btcAmount, griefingCollateral)`

Parameters

- `oldVault`: Account identifier of the vault to be replaced (as tracked in `Vaults` in *Vault Registry*).
- `btcAmount`: Integer amount of BTC / interbtc to be replaced.
- `griefingCollateral`: collateral locked by the *oldVault* as griefing protection

Events

- `RequestReplace(oldVault, btcAmount, replaceId)`

Preconditions

- The function call MUST be signed by *oldVault*.
- The vault MUST be registered
- The vault MUST NOT be banned
- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- The vault MUST provide sufficient `griefingCollateral` such that the ratio of all of its `toBeReplacedTokens` and `replaceCollateral` is above *ReplaceGriefingCollateral*.
- The vault MUST request sufficient tokens to be replaced such that its total is above `ReplaceBtcDustValue`.

Postconditions

- The vault's `toBeReplaceedTokens` is increased by `tokenIncrease = min(btcAmount, vault.toBeIssuedTokens - vault.toBeRedeemedTokens)`.
- An amount of `griefingCollateral * (tokenIncrease / btcAmount)` is locked by this transaction.
- The vault's `replaceCollateral` is increased by the amount of collateral locked in this transaction.

14.3.2 withdrawReplaceRequest

The *OldVault* decreases its `toBeReplacedTokens`.

Specification

Function Signature

`withdrawReplaceRequest(oldVault, tokens)`

Parameters

- `oldVault`: Account identifier of the vault withdrawing it's replace request (as tracked in `Vaults` in *Vault Registry*)
- `tokens`: The amount of `to_be_replaced_tokens` to withdraw.

Events

`WithdrawReplaceRequest(oldVault, withdrawnTokens, withdrawnGriefingCollateral)`: emits an event stating that a vault (*oldVault*) has withdrawn some amount of `toBeReplacedTokens`.

Preconditions

- The function call MUST be signed by *oldVault*.
- The vault MUST be registered
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- The vault MUST have a non-zero amount of `toBeReplaceedTokens`.

Postconditions

- The vault's `toBeReplacedTokens` is decrease by an amount of `tokenDecrease = min(toBeReplacedTokens, tokens)`
- The vault's `replaceCollateral` is decreased by the amount `releasedCollateral = replaceCollateral * (tokenDecrease / toBeReplacedTokens)`.
- The `releasedCollateral` is unlocked.

14.3.3 acceptReplace

A *NewVault* accepts an existing replace request. It can optionally lock additional DOT collateral specifically for this replace. If the replace is cancelled, this amount will be unlocked again.

Specification

Function Signature

`acceptReplace(newVault, oldVault, btcAmount, collateral, btcAddress)`

Parameters

- `newVault`: Account identifier of the vault accepting the replace request (as tracked in `Vaults` in *Vault Registry*)
- `replaceId`: The identifier of the replace request in `ReplaceRequests`.
- `collateral`: DOT collateral provided to match the replace request (i.e., for backing the locked BTC). Can be more than the necessary amount.

Events

`AcceptReplace(replaceId, oldVault, newVault, btcAmount, collateral, btcAddress)`: emits an event with data that the *oldVault* needs to execute the replace.

Preconditions

- The function call MUST be signed by *newVault*.
- *oldVault* and *newVault* MUST be registered

- *oldVault* MUST NOT be equal to *newVault*
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- *newVault*'s free balance MUST be enough to lock `collateral`
- *newVault* MUST have lock sufficient collateral to remain above the *SecureCollateralThreshold* after accepting `btcAmount`.
- The replaced tokens MUST be at least``ReplaceBtcDustValue``.

Postconditions

The actual amount of replaced tokens is calculated to be `consumedTokens = min(oldVault.toBeReplacedTokens, btcAmount)`. The amount of griefingCollateral used is `consumedGriefingCollateral = oldVault.replaceCollateral * (consumedTokens / oldVault.toBeReplacedTokens)`.

- The *oldVault*'s `replaceCollateral` is decreased by `consumedGriefingCollateral`.
- The *oldVault*'s `toBeReplacedTokens` is decreased by `consumedTokens`.
- The *oldVault*'s `toBeRedeemedTokens` is increased by `consumedTokens`.
- The *newVault*'s `toBeIssuedTokens` is increased by `consumedTokens`.
- The *newVault* locks additional collateral; its `backingCollateral` is increased by `collateral * (consumedTokens / oldVault.toBeReplacedTokens)`.
- A new `ReplaceRequest` is added to storage. The amount is set to `consumedTokens`, `griefingCollateral` to `consumedGriefingCollateral`, `collateral` to the `collateral` argument, `accept_time` to the current active block number, `period` to the current `ReplacePeriod`, `btcAddress` to the `btcAddress` argument, `btc_height` to the current height of the btc-relay, and `status` to `pending`.

14.3.4 executeReplace

The to-be-replaced vault finalizes the replace process by submitting a proof that it transferred the correct amount of BTC to the BTC address of the new vault, as specified in the `ReplaceRequest`. This function calls *verifyAndValidateTransaction* in *BTC-Relay*.

Specification

Function Signature

`executeReplace(oldVault, replaceId, merkleProof, rawTx)`

Parameters

- `oldVault`: Account identifier of the vault making this call.
- `replaceId`: The identifier of the replace request in `ReplaceRequests`.
- `merkleProof`: Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- `ExecuteReplace(oldVault, newVault, replaceId)`: emits an event stating that the old vault (*oldVault*) has executed the BTC transfer to the new vault (*newVault*), finalizing the `ReplaceRequest` request (`requestId`).

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- *oldVault* MUST be registered as a vault

- A pending `ReplaceRequest` MUST exist with an id equal to `replaceId`.
- The request MUST NOT have expired.
- The `rawTx` MUST decode to a valid transaction that transfers at least the amount specified in the `ReplaceRequest` struct. It MUST be a transaction to the correct address, and provide the expected OP_RETURN, based on the `ReplaceRequest`.
- The `merkleProof` MUST contain a valid proof of of `rawTX`.
- The bitcoin payment MUST have been submitted to the relay chain, and MUST have sufficient confirmations.

Postconditions

- *replaceTokens* has been called, providing the `oldVault`, `newVault`, `replaceRequest.amount`, and `replaceRequest.collateral` as arguments.
- The griefing collateral as specifiied in the `ReplaceRequest` is unlocked to *oldVault*.
- `replaceRequest.status` is set to `Completed`.

14.3.5 cancelReplace

If a replace request is not executed on time, the replace can be cancelled by the new vault. Since the new vault provided additional collateral in vain, it can claim the old vault's griefing collateral.

Specification

Function Signature

`cancelReplace(newVault, replaceId)`

Parameters

- `newVault`: Account identifier of the vault accepting the replace request (as tracked in `Vaults` in *Vault Registry*)
- `replaceId`: The identifier of the replace request in `ReplaceRequests`.

Events

- `CancelReplace(replaceId, newVault, oldVault, slashedCollateral)`: emits an event stating that the old vault (*oldVault*) has not completed the replace request, that the new vault (*newVault*) cancelled the `ReplaceRequest` request (`requestId`), and that `slashedCollateral` has been slashed from *oldVault* to *newVault*.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- *oldVault* MUST be registered as a vault
- A pending `ReplaceRequest` MUST exist with an id equal to `replaceId`.
- `newVault` MUST be equal to the *newVault* specified in the `ReplaceRequest`. That is, this function can only be can only be called by the *newVault*.
- The request MUST have expired.

Postconditions

- *cancelReplaceTokens* has been called, providing the `oldVault`, `newVault`, `replaceRequest.amount`, and `replaceRequest.amount`.
- **If *newVault* is *not* liquidated:**
  - the griefing collateral is slashed from the *oldVault* to the new vault's `backingCollateral`.

  - If unlocking `replaceRequest.collateral` does not put the collaterlization rate of the *newVault* below `SecureCollateralThreshold`, the collateral is unlocked and its `backingCollateral` decreases by the same amount.

- If *newVault is* liquidated, the griefing collateral is slashed from the *oldVault* to the new vault's free balance.
- `replaceRequest.status` is set to `Cancelled`.

14.4 Events

14.4.1 RequestReplace

Emit an event when a replace request is made by an *oldVault*.

Event Signature * `RequestReplace(oldVault, btcAmount, replaceId)`

Parameters

- `oldVault`: Account identifier of the vault to be replaced (as tracked in `Vaults` in *Vault Registry*).
- `btcAmount`: Integer amount of BTC / interbtc to be replaced.
- `replaceId`: The unique identified of a replace request.

Functions

- *requestReplace*

14.4.2 WithdrawReplaceRequest

Emits an event stating that a vault (*oldVault*) has withdrawn some amount of `toBeReplacedTokens`.

Event Signature

`WithdrawReplaceRequest(oldVault, withdrawnTokens, withdrawnGriefingCollateral)`

Parameters

- `oldVault`: Account identifier of the vault requesting the replace (as tracked in `Vaults` in *Vault Registry*)
- `withdrawnTokens`: The amount by which `toBeReplacedTokens` has decreased.
- `withdrawnGriefingCollateral`: The amount of griefing collateral unlocked.

Functions

- ref:*withdrawReplaceRequest*

14.4.3 AcceptReplace

Emits an event stating which vault (*newVault*) has accepted the `ReplaceRequest` request (`requestId`), and how much collateral in DOT it provided (`collateral`).

Event Signature

`AcceptReplace(replaceId, oldVault, newVault, btcAmount, collateral, btcAddress)`

Parameters

- `replaceId`: The identifier of the replace request in `ReplaceRequests`.
- `oldVault`: Account identifier of the vault being replaced (as tracked in `Vaults` in *Vault Registry*)
- `newVault`: Account identifier of the vault that accepted the replace request (as tracked in `Vaults` in *Vault Registry*)

- `btcAmount`: Amount of tokens the *newVault* just accepted.
- `collateral`: Amount of collateral the *newVault* locked for this replace.
- `btcAddress`: The address that *oldVault* should transfer the btc to.

Functions

- ref:*acceptReplace*

14.4.4 ExecuteReplace

Emits an event stating that the old vault (*oldVault*) has executed the BTC transfer to the new vault (*newVault*), finalizing the `ReplaceRequest` request (`requestId`).

Event Signature

```
ExecuteReplace(oldVault, newVault, replaceId)
```

Parameters

- `oldVault`: Account identifier of the vault being replaced (as tracked in `Vaults` in *Vault Registry*)
- `newVault`: Account identifier of the vault that accepted the replace request (as tracked in `Vaults` in *Vault Registry*)
- `replaceId`: The identifier of the replace request in `ReplaceRequests`.

Functions

- ref:*executeReplace*

14.4.5 CancelReplace

Emits an event stating that the old vault (*oldVault*) has not completed the replace request, that the new vault (*newVault*) cancelled the `ReplaceRequest` request (`requestId`), and that `slashedCollateral` has been slashed from *oldVault* to *newVault*.

Event Signature

```
CancelReplace(replaceId, newVault, oldVault, slashedCollateral)
```

Parameters

- `replaceId`: The identifier of the replace request in `ReplaceRequests`.
- `oldVault`: Account identifier of the vault being replaced (as tracked in `Vaults` in *Vault Registry*)
- `newVault`: Account identifier of the vault that accepted the replace request (as tracked in `Vaults` in *Vault Registry*)
- `slashedCollateral`: Amount of griefingCollateral slashed to *newVault*.

Functions

- ref:*cancelReplace*

14.5 Error Codes

`ERR_UNAUTHORIZED`

- **Message**: "Unauthorized: Caller must be *newVault*."
- **Function**: *cancelReplace*
- **Cause**: The caller of this function is not the associated *newVault*, and hence not authorized to take this action.

`ERR_INSUFFICIENT_COLLATERAL`

- **Message**: "The provided collateral is too low."
- **Function**: *requestReplace*
- **Cause**: The provided collateral is insufficient to match the amount of tokens requested for replacement.

`ERR_REPLACE_PERIOD_EXPIRED`

- **Message**: "The replace period expired."
- **Function**: *executeReplace*
- **Cause**: The time limit as defined by the `ReplacePeriod` is not met.

`ERR_REPLACE_PERIOD_NOT_EXPIRED`

- **Message**: "The period to complete the replace request is not yet expired."
- **Function**: *cancelReplace*
- **Cause**: A vault tried to cancel a replace before it expired.

`ERR_AMOUNT_BELOW_BTC_DUST_VALUE`

- **Message**: "To be replaced amount is too small."
- **Function**: *requestReplace*, *acceptReplace*
- **Cause**: The vault requests or accepts an insufficient number of tokens.

`ERR_NO_PENDING_REQUEST`

- **Message**: "Could not withdraw to-be-replaced tokens because it was already zero."
- **Function**: *requestReplace* | *acceptReplace*
- **Cause**: The vault requests or accepts an insufficient number of tokens.

`ERR_REPLACE_SELF_NOT_ALLOWED`

- **Message**: "Vaults can not accept replace request created by themselves."
- **Function**: *acceptReplace*
- **Cause**: A vault tried to accept a replace that it itself had created.

`ERR_REPLACE_COMPLETED`

- **Message**: "Request is already completed."
- **Function**: *executeReplace* | *cancelReplace*
- **Cause**: A vault tried to operate on a request that already completed.

`ERR_REPLACE_CANCELLED`

- **Message**: "Request is already cancelled."
- **Function**: *executeReplace* | *cancelReplace*
- **Cause**: A vault tried to operate on a request that already cancelled.

`ERR_REPLACE_ID_NOT_FOUND`

- **Message**: “Invalid replace ID”
- **Function**: *executeReplace* | *cancelReplace*
- **Cause**: An invalid replaceID was given - it is not found in the `ReplaceRequests` map.

`ERR_VAULT_NOT_FOUND`

- **Message**: “The `vault` cannot be found.”
- **Function**: *requestReplace* | *acceptReplace* | *cancelReplace*
- **Cause**: The vault was not found in the existing `Vaults` list in `VaultRegistry`.

Note: It is possible that functions in this pallet return errors defined in other pallets.

CHAPTER

FIFTEEN

SECURITY

The Security module is responsible for (1) tracking the status of the BTC Parachain, (2) the "active" blocks of the BTC Parachain, and (3) generating secure identifiers.

1. **BTC Parachain Status**: The BTC Parachain has three distinct states: `Running`, `Error`, and `Shutdown` which determine which functions can be used.
2. **Active Blocks**: When the BTC Parachain is not in the `Running` state, certain operations are restricted. In order to prevent impact on the users and vaults for the core issue, redeem, and replace operations, the BTC Parachain only considers Active Blocks for the Issue, Redeem, and Replace Periods.
3. **Secure Identifiers**: As part of the *OP_RETURN* scheme to prevent replay attacks, the security module generates unique identifiers that are used to identify transactions.

15.1 Overview

15.1.1 Failure Modes

The BTC Parachain can enter into an ERROR and SHUTDOWN state, depending on the occurred error. An overview is provided in the figure below.

Failure handling methods calls are **restricted**, i.e., can only be called by pre-determined roles.

15.1.2 Oracle Offline

The *Exchange Rate Oracle* experienced a liveness failure (no up-to-date exchange rate available). The frequency of the oracle updates is defined in the Oracle module.

Error code: `ORACLE_OFFLINE`

15.1.3 BTC-Relay Offline

The *BTC-Relay* has less blocks stored than defined as the `STABLE_BITCOIN_CONFIRMATIONS`.

This is the initial state of the BTC-Parachain. After more than the `STABLE_BITCOIN_CONFIRMATIONS` BTC blocks have been stored in BTC-Relay, the BTC Parachain cannot decide if or not it is behind in terms of Bitcoin blocks since we make no assumption about the frequency of BTC blocks being produced.

Error code: `BTC_RELAY_OFFLINE`

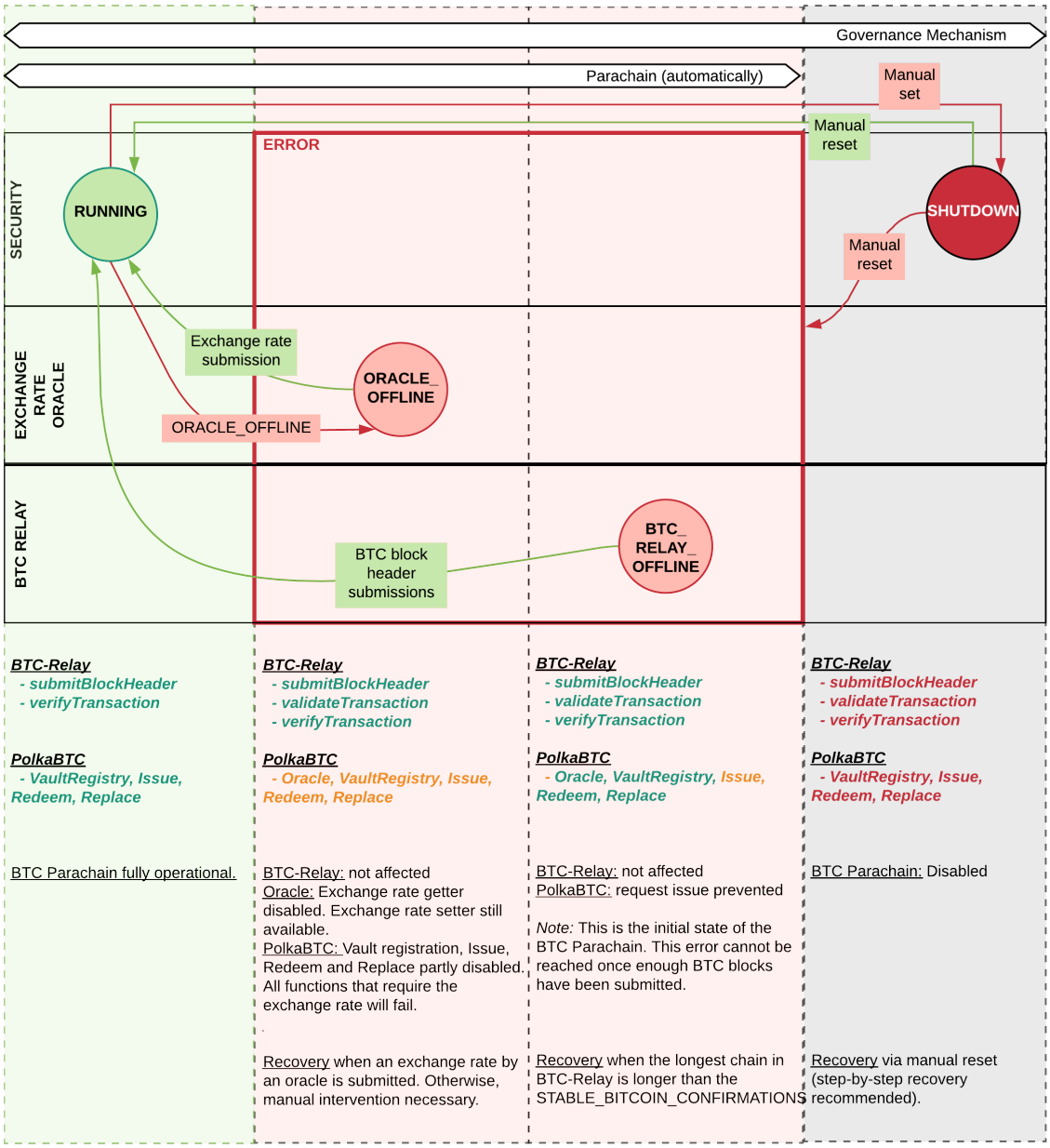


Fig. 15.1: (Informal) State machine showing the operational and failure modes and how to recover from or flag failures.

15.2 Data Model

15.2.1 Enums

StatusCode

Indicates ths status of the BTC Parachain.

- `RUNNING: 0` - BTC Parachain fully operational
- `ERROR: 1`- an error was detected in the BTC Parachain. See `Errors` for more details, i.e., the specific error codes (these determine how to react).
- `SHUTDOWN: 2` - BTC Parachain operation fully suspended. This can only be achieved via manual intervention by the Governance Mechanism.

ErrorCode

Enum specifying error codes tracked in `Errors`.

- `NONE: 0`
- `ORACLE_OFFLINE: 1`
- `BTC_RELAY_OFFLINE: 2`

15.3 Data Storage

15.3.1 Scalars

ParachainStatus

Integer/Enum (see `StatusCode` below). Defines the current state of the BTC Parachain.

Errors

Set of error codes (`ErrorCode` enums), indicating the reason for the error. The `ErrorCode` entries included in this set specify how to react to the failure.

Nonce

Integer increment-only counter, used to prevent collisions when generating identifiers for e.g., redeem or replace requests (for OP_RETURN field in Bitcoin).

ActiveBlockCount

A counter variable that increments every block where the parachain status is `RUNNING:0`. This variable is used to keep track of durations, such as issue/redeem/replace expiry. This is used instead of the block number because if the parachain status is not `RUNNING:0`, no payment proofs can be submitted, so it would not be fair towards users and vaults to continue counting down the (expiry) periods.

15.4 Functions

15.4.1 generateSecureId

Generates a unique ID using an account identifier, the `Nonce` and a random seed.

Specification

Function Signature

`generateSecureId(account)`

Parameters

- `account`: Parachain account identifier (links this identifier to the AccountId associated with the process where this secure id is to be used, e.g., the user calling *requestIssue*).

Returns

- `hash`: a cryptographic hash generated via a secure hash function.

Function Sequence

1. Increment the `Nonce`.
2. Concatenate `account`, `Nonce`, and `parent_hash()`.
3. SHA256 hash the result of step 1.
4. Return the resulting hash.

Note: The funtion `parent_hash()` is assumed to return the hash of the parachain's parent block - which precedes the block this function is called in.

15.4.2 hasExpired

Checks if the given period has expired since the given starting point. This calculation is based on the *ActiveBlockCount*.

Specification

Function Signature

`has_expired(opentime, period)`

Parameters

- `opentime`: the *ActiveBlockCount* at the time the issue/redeem/replace was opened.
- `period`: the number of blocks the user or vault has to complete the action.

Returns

- `true` if the period has expired

Function Sequence

1. Add the `opentime` and `period`.
2. Compare this against *ActiveBlockCount*.

15.4.3 setParachainStatus

Governance sets a status code for the BTC Parachain manually.

Specification

Function Signature

`setParachainStatus(StatusCode)`

Parameters

- `StatusCode`: the new StatusCode of the BTC-Parachain.

15.4.4 insertParachainError

Governance inserts an error for the BTC Parachain manually.

Specification

Function Signature

`insertParachainError(ErrorCode)`

Parameters

- `ErrorCode`: the ErrorCode to be added to the set of errors of the BTC-Parachain.

15.4.5 removeParachainError

Governance removes an error for the BTC Parachain manually.

Specification

Function Signature

`removeParachainError(ErrorCode)`

Parameters

- `ErrorCode`: the ErrorCode to be removed from the set of errors of the BTC-Parachain.

15.5 Events

15.5.1 RecoverFromErrors

Event Signature

`RecoverFromErrors(StatusCode, ErrorCode[])`

Parameters

- `StatusCode`: the new StatusCode of the BTC Parachain
- `ErrorCode[]`: the list of current errors

CHAPTER

SIXTEEN

SLA

16.1 Overview

The SLA module implements the scheme outline in the *Service Level Agreements*. Its main purpose is to compute the delta used to increase or decrease the reward stake.

We define an SLA score to be a real number between 0 and 100: SLA $= [0, 100)$. The delta is a rational number.

SLAs are used to compute the rewards earned by a participant through the performance of predefined "desired actions".

Additionally, Vaults with high SLAs avoid having their entire collateral slashed in case they fail to correctly execute a Redeem request (i.e., only the minimum amount of collateral is slashed, defined by the Liquidation-Threshold).

16.1.1 Step-by-step

1. Vault and Staked Relayers interact with the BTC-Parachain.
2. Certain actions have an impact on their SLA. If this is the case, the function updates the SLA score of the Vault or Staked Relayer accordingly.
3. The SLA is stored for each Vault and Staked Relayer to impact collateral slashing for Vaults and for fee allocation.

16.2 Data Model

16.2.1 Scalars (Vaults)

VaultSLATarget

Target value for Vault SLAs.

- Initial value: 100

FailedRedeem (Decrease)

- Initial value: -100

ExecutedIssue (Increase)

Based on volume of the issue request as compared to the average issue request size (avgIssueSizeN) of the last N issue requests. SLAIncrease = max(requestSize / avgIssueSizeN * maxSLA, maxSLA)

- Initial value: maxSLA = 4

SubmitIssueProof (Increase)

Vault submits correct Issue proof on behalf of the user.

- Initial value: 1

16.2.2 Scalars (Staked Relayers)

Staked Relayer SLA Target

Target value for Staked Relayer SLAs

- Initial value: 100

Block Submission (Increase)

- Initial value: +1

Correct Theft Report (Increase)

- Initial value: +1

16.2.3 Maps

VaultSLA

Mapping from Vault accounts to their SLA score.

StakedRelayerSLA

Mapping from Staked Relayer accounts to their SLA score.

16.3 Functions

16.3.1 SlashAmountVault

We reduce the amount of slashed collateral based on a Vaults SLA. The minimum amount slashed is given by the `LiquidationThreshold`, the maximum amount slashed by the `PremiumRedeemThreshold`. The actual slashed amount of collateral is a linear function parameter zed by the two thresholds:

MinSlashed $=$ LiquidationThreshold -100% (currently 10%) MaxSlashed $=$ PremiumRedeemThreshold -100% (currently 30%)

RealSlashed $= (\textbf{MaxSlashed} - \textbf{MinSlashed})/\textbf{SLATarget} * \textbf{SLA} \ + (\text{LiquidationThreshold} - 100\%)$

Specification

Function Signature

`SlashVault(account)`

Parameters

- `account`: The account ID of the vault.

Returns

- `rate`: The rate (in %) to-be-slashed.

Function Sequence

1. Based on the Vault's SLA, calculate the to-be-slashed percentage based on the formula above.

16.3.2 updateSLA

Updates the SLA of a Vault or Relayer.

Specification

Function Signature

`updateSLA(account, delta)`

Parameters

- `account`: the account that will be updated
- `delta`: the increase or decrease in the sla score.

Events

- `UpdateSLA`

16.4 Events

16.4.1 UpdateSLA

Event Signature

`UpdateSLA(account, total_score, delta)`

Parameters

- `account`: the account that will be updated
- `total_score`: the SLA score of the account after the update
- `delta`: the increase or decrease in the sla score.

Functions

- *updateSLA*

CHAPTER

SEVENTEEN

STAKED RELAYERS

The *Staked Relayers* module is responsible for handling the registration and staking of Staked Relayers. It also wraps functions for Staked Relayers to submit Bitcoin block headers to the *BTC-Relay*.

17.1 Overview

Staked Relayers are participants whose main role it is to run Bitcoin full nodes and:

1. Submit valid Bitcoin block headers to increase their *SLA* score.
2. Check vaults do not move BTC, unless expressly requested during *Redeem*, *Replace* or *Refund*.

In the second case, a single staked relayer report suffices - the module should check the accusation (using a Merkle proof), and liquidate the vault if valid.

Staked Relayers are overseen by the Parachain **Governance Mechanism**. The Governance Mechanism also votes on critical changes to the architecture or unexpected failures, e.g. hard forks or detected 51% attacks (if a fork exceeds the specified security parameter *k*, see Security Parameter k.).

17.2 Data Model

17.2.1 Structs

StakedRelayer

Stores the information of a Staked Relayer.

Parameter	Type	Description
`stake`	Backing	Total amount of collateral/stake provided by this Staked Relayer.

17.3 Data Storage

17.3.1 Constants

STAKED_RELAYER_STAKE

Integer denoting the minimum stake which Staked Relayers must provide when registering.

17.3.2 Maps

StakedRelayers

Mapping from accounts of StakedRelayers to their struct. `<Account, StakedRelayer>`.

TheftReports

Mapping of Bitcoin transaction identifiers (SHA256 hashes) to account identifiers of Vaults who have been caught stealing Bitcoin. Per Bitcoin transaction, multiple Vaults can be accused (multiple inputs can come from multiple Vaults). This mapping is necessary to prevent duplicate theft reports.

17.4 Functions

17.4.1 registerStakedRelayer

Registers a new Staked Relayer, locking the provided collateral, which must exceed `STAKED_RELAYER_STAKE`.

Specification

Function Signature

`registerStakedRelayer(stakedRelayer, stake)`

Parameters

- `stakedRelayer`: The account of the staked relayer to be registered.
- `stake`: to-be-locked collateral/stake.

Events

- `RegisterStakedRelayer(StakedRelayer, collateral)`: emit an event stating that a new staked relayer (`stakedRelayer`) was registered and provide information on the Staked Relayer's stake (`stake`).

Errors

- `ERR_ALREADY_REGISTERED = "This AccountId is already registered as a Staked Relayer"`: The given account identifier is already registered.
- `ERR_INSUFFICIENT_STAKE = "Insufficient stake provided"`: The provided stake was insufficient - it must be above `STAKED_RELAYER_STAKE`.

Preconditions

Function Sequence

The `registerStakedRelayer` function takes as input an AccountID and collateral amount (to be used as stake) to register a new staked relayer in the system.

1) Check that the `stakedRelayer` is not already in `StakedRelayers`. Return `ERR_ALREADY_REGISTERED` if this check fails.
2) Check that `stake > STAKED_RELAYER_STAKE` holds, i.e., the staked relayer provided sufficient collateral. Return `ERR_INSUFFICIENT_STAKE` error if this check fails.
3) Lock the stake/collateral by calling *lockCollateral* and passing `stakedRelayer` and the `stake` as parameters.

4) Store the provided information (amount of `stake`) in a new `StakedRelayer` and insert it into the `StakedRelayers` mapping using the `stakedRelayer` AccountId as key.

5) Emit a `RegisterStakedRelayer(StakedRelayer, collateral)` event.

17.4.2 deRegisterStakedRelayer

De-registers a Staked Relayer, releasing the associated stake.

Specification

Function Signature

`registerStakedRelayer(stakedRelayer)`

Parameters

- `stakedRelayer`: The account of the staked relayer to be de-registered.

Events

- `DeRegisterStakedRelayer(StakedRelayer)`: emit an event stating that a staked relayer has been de-registered (`stakedRelayer`).

Errors

- `ERR_NOT_REGISTERED = "This AccountId is not registered as a Staked Relayer"`: The given account identifier is not registered.

Preconditions

Function Sequence

1) Check if the `stakedRelayer` is indeed registered in `StakedRelayers`. Return `ERR_NOT_REGISTERED` if this check fails.

3) Release the stake/collateral of the `stakedRelayer` by calling *lockCollateral* and passing `stakedRelayer` and the `StakeRelayer.stake` (as retrieved from `StakedRelayers`) as parameters.

4) Remove the entry from `StakedRelayers` which has `stakedRelayer` as key.

5) Emit a `DeRegisterStakedRelayer(StakedRelayer)` event.

17.4.3 slashStakedRelayer

Slashes the stake/collateral of a staked relayer and removes them from the staked relayer list (mapping).

> **Warning:** This function can only be called by the Governance Mechanism.

Specification

Function Signature

`slashStakedRelayer(governanceMechanism, stakedRelayer)`

Parameters

- `governanceMechanism`: The AccountId of the Governance Mechanism.
- `stakedRelayer`: The account of the staked relayer to be slashed.

Events

- `SlashStakedRelayer(stakedRelayer)`: emits an event indicating that a given staked relayer (`stakedRelayer`) has been slashed and removed from `StakedRelayers`.

Errors

- `ERR_GOVERNANCE_ONLY = This action can only be executed by the Governance Mechanism`: Only the Governance Mechanism can slash Staked Relayers.
- `ERR_NOT_REGISTERED = "This AccountId is not registered as a Staked Relayer"`: The given account identifier is not registered.

Function Sequence

1. Check that the caller of this function is indeed the Governance Mechanism. Return `ERR_GOVERNANCE_ONLY` if this check fails.
2. Retrieve the staked relayer with the given account identifier (`stakedRelayer`) from `StakedRelayers`. Return `ERR_NOT_REGISTERED` if not staked relayer with the given identifier can be found.
3. Confiscate the Staked Relayer's collateral. For this, call *slashCollateral* providing `stakedRelayer` and `governanceMechanism` as parameters.
4. Remove `stakedRelayer` from `StakedRelayers`
5. Emit `SlashStakedRelayer(stakedRelayer)` event.

17.4.4 reportVaultTheft

A staked relayer reports misbehavior by a vault, providing a fraud proof (malicious Bitcoin transaction and the corresponding transaction inclusion proof).

A vault is not allowed to move BTC from any registered Bitcoin address (as specified by `Vault.wallet`), except in the following three cases:

1) The vault is executing a *Redeem*. In this case, we can link the transaction to a `RedeemRequest` and check the correct recipient.
2) The vault is executing a *Replace*. In this case, we can link the transaction to a `ReplaceRequest` and check the correct recipient.
3) The vault is executing a *Refund*. In this case, we can link the transaction to a `RefundRequest` and check the correct recipient.
4) [Optional] The vault is "merging" multiple UTXOs it controls into a single / multiple UTXOs it controls, e.g. for maintenance. In this case, the recipient address of all outputs (e.g. `P2PKH` / `P2WPKH`) must be the same Vault.

In all other cases, the vault is considered to have stolen the BTC.

This function checks if the vault actually misbehaved (i.e., makes sure that the provided transaction is not one of the above valid cases) and automatically liquidates the vault (i.e., triggers *Redeem*).

Specification

Function Signature

`reportVaultTheft(vault, merkleProof, rawTx)`

Parameters

- `vaultId`: the account of the accused Vault.
- `merkleProof`: Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- `ReportVaultTheft(vault)` - emits an event indicating that a vault (`vault`) has been caught displacing BTC without permission.

Errors

- `ERR_STAKED_RELAYERS_ONLY = "This action can only be executed by Staked Relayers"`: The caller of this function was not a Staked Relayer. Only Staked Relayers are allowed to suggest and vote on BTC Parachain status updates.
- `ERR_ALREADY_REPORTED = "This txId has already been logged as a theft by the given Vault"`: This transaction / vault combination has already been reported.
- `ERR_VAULT_NOT_FOUND = "There exists no vault with the given account id"`: The specified vault does not exist.
- `ERR_ALREADY_LIQUIDATED = "This vault is already being liquidated`: The specified vault is already being liquidated.
- `ERR_VALID_REDEEM = "The given transaction is a valid Redeem execution by the accused Vault"`: The given transaction is associated with a valid *Redeem*.
- `ERR_VALID_REPLACE = "The given transaction is a valid Replace execution by the accused Vault"`: The given transaction is associated with a valid *Replace*.
- `ERR_VALID_REFUND = "The given transaction is a valid Refund execution by the accused Vault"`: The given transaction is associated with a valid *Refund*.
- `ERR_VALID_MERGE_TRANSACTION = "The given transaction is a valid 'UTXO merge' transaction by the accused Vault"`: The given transaction represents an allowed "merging" of UTXOs by the accused vault (no BTC was displaced).

Function Sequence

1. Check that the caller of this function is indeed a Staked Relayer. Return `ERR_STAKED_RELAYERS_ONLY` if this check fails.
2. Check if the specified `vault` exists in `Vaults` in *Vault Registry*. Return `ERR_VAULT_NOT_FOUND` if there is no vault with the specified account identifier.
3. Check if this `vault` has already been liquidated. If this is the case, return `ERR_ALREADY_LIQUIDATED` (no point in duplicate reporting).
4. Check if the given Bitcoin transaction is already associated with an entry in `TheftReports` (calculate `txId` from `rawTx` as key for lookup). If yes, check if the specified `vault` is already listed in the associated set of Vaults. If the vault is already in the set, return `ERR_ALREADY_REPORTED`.
5. Extract the `outputs` from `rawTx` using *extractOutputs* from the BTC-Relay.
6. Check if the transaction is a "migration" of UTXOs to the same Vault. For each output, in the extracted `outputs`, extract the recipient Bitcoin address (using *extractOutputAddress* from the BTC-Relay).

a) If one of the extracted Bitcoin addresses does not match a Bitcoin address of the accused `vault` (`Vault.wallet`) **continue to step 7**.

b) If all extracted addresses match the Bitcoin addresses of the accused `vault` (`Vault.wallet`), abort and return `ERR_VALID_MERGE_TRANSACTION`.

7. Check if the transaction is part of a valid *Redeem*, *Replace* or *Refund* process.

a) Extract the OP_RETURN value using *extractOPRETURN* from the BTC-Relay. If this call returns an error (no valid OP_RETURN output, hence not valid *Redeem*, *Replace* or *Refund* process), **continue to step 8**.

c) Check if the extracted OP_RETURN value matches any `redeemId` in `RedeemRequest` (in `RedeemRequests` in *Redeem*), any `replaceId` in `ReplaceRequest` (in `RedeemRequests` in *Redeem*) or any `refundId` in `RefundRequest` (in `RefundRequests` in *Refund*) entries *associated with this Vault*. If no match is found, **continue to step 8**.

d) Otherwise, if an associated `RedeemRequest`, `ReplaceRequest` or `RefundRequest` was found: extract the value (using *extractOutputValue* from the BTC-Relay) and recipient Bitcoin address (using *extractOutputAddress* from the BTC-Relay). Next, check:

i) if the value is equal (or greater) than `paymentValue` in the `RedeemRequest`, `ReplaceRequest` or `RefundRequest`.

ii) if the recipient Bitcoin address matches the recipient specified in the `RedeemRequest`, `ReplaceRequest` or `RefundRequest`.

iii) if the change Bitcoin address(es) are registered to the accused `vault` (`Vault.wallet`).

If all checks are successful, abort and return `ERR_VALID_REDEEM`, `ERR_VALID_REPLACE` or `ERR_VALID_REFUND`. Otherwise, **continue to step 8**.

8. The vault misbehaved (displaced BTC).

a) Call *liquidateVault*, liquidating the vault and transferring all of its balances and collateral to the `LiquidationVault` for failure and reimbursement handling;

b) emit `ReportVaultTheft(vaultId)`

9. Return

17.5 Events

17.5.1 RegisterStakedRelayer

Emit an event stating that a new staked relayer was registered and provide information on the Staked Relayer's stake

Event Signature

`RegisterStakedRelayer(StakedRelayer, collateral)`

Parameters

- `stakedRelayer`: newly registered staked Relayer
- `stake`: stake provided by the staked relayer upon registration

Functions

- *registerStakedRelayer*

17.5.2 DeRegisterStakedRelayer

Emit an event stating that a staked relayer has been de-registered

Event Signature

`DeRegisterStakedRelayer(StakedRelayer)`

Parameters

- `stakedRelayer`: account identifier of de-registered Staked Relayer

Functions

- *deRegisterStakedRelayer*

17.5.3 SlashStakedRelayer

Emits an event indicating that a staked relayer has been slashed.

Event Signature

`SlashStakedRelayer(stakedRelayer)`

Parameters

- `stakedRelayer`: account identifier of the slashed staked relayer.

Functions

- *slashStakedRelayer*

17.5.4 ReportVaultTheft

Emits an event when a vault has been accused of theft.

Event Signature

`ReportVaultTheft(vault)`

Parameters

- `vault`: account identifier of the vault accused of theft.

Functions

- *reportVaultTheft*

17.6 Errors

`ERR_NOT_REGISTERED`

- **Message**: "This AccountId is not registered as a Staked Relayer."
- **Function**: *deRegisterStakedRelayer*, *slashStakedRelayer*
- **Cause**: The given account identifier is not registered.

`ERR_GOVERNANCE_ONLY`

- **Message**: "This action can only be executed by the Governance Mechanism"
- **Function**: *slashStakedRelayer*
- **Cause**: The suggested status (`SHUTDOWN`) can only be triggered by the Governance Mechanism but the caller of the function is not part of the Governance Mechanism.

`ERR_STAKED_RELAYERS_ONLY`

- **Message**: "This action can only be executed by Staked Relayers"
- **Function**: *reportVaultTheft*
- **Cause**: The caller of this function was not a Staked Relayer. Only Staked Relayers are allowed to suggest and vote on BTC Parachain status updates.

`ERR_ALREADY_REPORTED`

- **Message**: "This txId has already been logged as a theft by the given Vault"
- **Function**: *reportVaultTheft*
- **Cause**: This transaction / vault combination has already been reported.

`ERR_VAULT_NOT_FOUND`

- **Message**: "There exists no vault with the given account id"
- **Function**: *reportVaultTheft*
- **Cause**: The specified vault does not exist.

`ERR_ALREADY_LIQUIDATED`

- **Message**: "This vault is already being liquidated"
- **Function**: *reportVaultTheft*
- **Cause**: The specified vault is already being liquidated.

`ERR_VALID_REDEEM`

- **Message**: "The given transaction is a valid Redeem execution by the accused Vault"
- **Function**: *reportVaultTheft*
- **Cause**: The given transaction is associated with a valid *Redeem*.

`ERR_VALID_REPLACE`

- **Message**: "The given transaction is a valid Replace execution by the accused Vault"
- **Function**: *reportVaultTheft*
- **Cause**: The given transaction is associated with a valid *Replace*.

`ERR_VALID_REFUND`

- **Message**: "The given transaction is a valid Refund execution by the accused Vault"
- **Function**: *reportVaultTheft*
- **Cause**: The given transaction is associated with a valid *Refund*.

`ERR_VALID_MERGE_TRANSACTION`

- **Message**: "The given transaction is a valid 'UTXO merge' transaction by the accused Vault"
- **Function**: *reportVaultTheft*
- **Cause**: The given transaction represents an allowed "merging" of UTXOs by the accused vault (no BTC was displaced).

CHAPTER

EIGHTEEN

TREASURY

18.1 Overview

The treasury serves as the central storage for all interbtc. It exposes the *transfer* function to any user. With the transfer functions users can send interbtc to and from each other. Further, the treasury exposes three internal functions for the *Issue* and the *Redeem*.

18.1.1 Step-by-step

- **Transfer**: A user sends an amount of interbtc to another user by calling the *transfer* function.
- **Issue**: The issue module calls into the treasury when an issue request is completed and the user has provided a valid proof that he transferred the required amount of BTC to the correct vault. The issue module calls the *mint* function to grant the user the interbtc token.
- **Redeem**: The redeem protocol requires two calls to the treasury module. First, a user requests a redeem via the *requestRedeem* function. This invokes a call to the *lock* function that locks the requested amount of tokens for this user. Second, when a redeem request is completed and the vault has provided a valid proof that it transferred the required amount of BTC to the correct user, the redeem module calls the *burn* function to destroy the previously locked interbtc.

18.2 Data Model

18.2.1 Constants

- `NAME`: `interbtc`
- `SYMBOL`: `pBTC`

18.2.2 Scalars

TotalSupply

The total supply of interbtc.

18.2.3 Maps

Balances

Mapping from accounts to their balance.

Locked Balances

Mapping from accounts to their balance of locked tokens. Locked tokens serve two purposes:

1. Locked tokens cannot be transferred. Once a user locks the token, the token needs to be unlocked to become spendable.
2. Locked tokens are the only tokens that can be burned in the redeem procedure.

18.3 Functions

18.3.1 transfer

Transfers a specified amount of interbtc from a Sender to a Receiver on the BTC Parachain.

Specification

Function Signature

`transfer(sender, receiver, amount)`

Parameters

- `sender`: An account with enough funds to send the `amount` of interbtc to the `receiver`.
- `receiver`: Account receiving an amount of interbtc.
- `amount`: The number of interbtc being sent in the transaction.

Events

- `Transfer(sender, receiver, amount)`: Issues an event when a transfer of funds was successful.

Errors

- `ERR_INSUFFICIENT_FUNDS`: The sender does not have a high enough balance to send an `amount` of interbtc.

```
fn transfer(origin, receiver: AccountId, amount: Balance) -> Result {...}
```

Function Sequence

The `transfer` function takes as input the sender, the receiver, and an amount. The function executes the following steps:

1. Check that the `sender` is authorised to send the transaction by verifying the signature attached to the transaction.
2. Check that the `sender`'s balance is above the `amount`. If `Balances[sender] < amount` (in Substrate `free_balance`), raise `ERR_INSUFFICIENT_FUNDS`.
3. Subtract the sender's balance by `amount`, i.e. `Balances[sender] -= amount` and add `amount` to the receiver's balance, i.e. `Balances[receiver] += amount`.
4. Emit the `Transfer(sender, receiver, amount)` event.

18.3.2 mint

In the BTC Parachain new interbtc can be created by leveraging the *Issue*. However, to separate concerns and access to data, the Issue module has to call the `mint` function to complete the issue process in the interbtc component. The function increases the `totalSupply` of interbtc.

> **Warning:** This function can *only* be called from the Issue module.

Specification

Function Signature

`mint(requester, amount)`

Parameters

- `requester`: The account of the requester of interbtc.
- `amount`: The amount of interbtc to be added to an account.

Events

- `Mint(requester, amount)`: Issue an event when new interbtc are minted.

```
fn mint(requester:  AccountId, amount:  Balance) -> Result {...}
```

Preconditions

This is an internal function and can only be called by the *Issue module*.

Function Sequence

1. Increase the `requester` Balance by `amount`, i.e. `Balances[requester] += amount`.
2. Emit the `Mint(requester, amount)` event.

18.3.3 lock

During the redeem process, a user needs to be able to lock interbtc. Locking transfers coins from the `Balances` mapping to the `LockedBalances` mapping to prevent users from transferring the coins.

Specification

Function Signature

`lock(redeemer, amount)`

Parameters

- `redeemer`: The Redeemer wishing to lock a certain amount of interbtc.
- `amount`: The amount of interbtc that should be locked.

Events

- `Lock(redeemer, amount)`: Emits newly locked amount of interbtc by a user.

Errors

- `ERR_INSUFFICIENT_FUNDS`: User has not enough interbtc to lock coins.

Precondition

- Can only be called by the redeem module.

Function Sequence

1. Checks if the user has a balance higher than or equal to the requested amount, i.e. `Balances[redeemer] >= amount`. Return `ERR_INSUFFICIENT_FUNDS` if the user's balance is too low.
2. Decreases the user's token balance by the amount and increases the locked tokens balance by amount, i.e. `Balances[redeemer] -= amount` and `LockedBalances[redeemer] += amount`.
3. Emit the `Lock` event.

18.3.4 burn

During the *Redeem*, users first lock and then "burn" (i.e. destroy) their interbtc to receive BTC. Users can only burn tokens once they are locked to prevent transaction ordering dependencies. This means a user first needs to move his tokens from the `Balances` to the `LockedBalances` mapping via the *lock* function.

> **Warning:** This function is only internally callable by the Redeem module.

Specification

Function Signature

`burn(redeemer, amount)`

Parameters

- `redeemer`: The Redeemer wishing to burn a certain amount of interbtc.
- `amount`: The amount of interbtc that should be destroyed.

Events

- `Burn(redeemer, amount)`: Issue an event when the amount of interbtc is successfully destroyed.

Errors

- `ERR_INSUFFICIENT_LOCKED_FUNDS`: If the user has insufficient funds locked, i.e. her locked balance is lower than the amount.

```
fn burn(redeemer: AccountId, amount: Balance) -> Result {...}
```

Preconditions

This is an internal function and can only be called by the *Redeem module*.

Function Sequence

1. Check that the `redeemer`'s locked balance is above the `amount`. If `LockedBalance[redeemer] < amount` (in Substrate `free_balance`), raise `ERR_INSUFFICIENT_LOCKED_FUNDS`.
2. Subtract the Redeemer's locked balance by `amount`, i.e. `LockedBalances[redeemer] -= amount`.
3. Emit the `Burn(redeemer, amount)` event.

18.4 Events

18.4.1 Transfer

Issues an event when a transfer of funds was successful.

Event Signature

`Transfer(sender, receiver, amount)`

Parameters

- `sender`: An account with enough funds to send the `amount` of interbtc to the `receiver`.
- `receiver`: Account receiving an amount of interbtc.
- `amount`: The number of interbtc being sent in the transaction.

Function

- *transfer*

18.4.2 Mint

Issue an event when new interbtc are minted.

Event Signature

`Mint(requester, amount)`

Parameters

- `requester`: The account of the requester of interbtc.
- `amount`: The amount of interbtc to be added to an account.

Function

- *mint*

18.4.3 Lock

Emits newly locked amount of interbtc by a user.

Event Signature

`Lock(redeemer, amount)`

Parameters

- `redeemer`: The Redeemer wishing to lock a certain amount of interbtc.
- `amount`: The amount of interbtc that should be locked.

Function

- *lock*

18.4.4 Burn

Issue an event when the amount of interbtc is successfully destroyed.

Event Signature

`Burn(redeemer, amount)`

Parameters

- `redeemer`: The Redeemer wishing to burn a certain amount of interbtc.
- `amount`: The amount of interbtc that should be burned.

Function

- *burn*

18.5 Errors

`ERR_INSUFFICIENT_FUNDS`

- **Message**: "The balance of this account is insufficient to complete the transaction."
- **Functions**: *transfer* | *lock*
- **Cause**: The balance of the user of available tokens (i.e. `Balances`) is below a certain amount to either transfer or lock tokens.

`ERR_INSUFFICIENT_LOCKED_FUNDS`

- **Message**: "The locked token balance of this account is insufficient to burn the tokens."
- **Function**: *burn*
- **Cause**: The user has locked too little tokens in the `LockedBalances` to execute the burn function.

CHAPTER

NINETEEN

VAULT REGISTRY

19.1 Overview

The vault registry is the central place to manage vaults. Vaults can register themselves here, update their collateral, or can be liquidated. Similarly, the issue, redeem, refund, and replace protocols call this module to assign vaults during issue, redeem, refund, and replace procedures. Morever, vaults use the registry to register public key for the *On-Chain Key Derivation Scheme* and register addresses for the *OP_RETURN* scheme.

19.2 Data Model

19.2.1 Constants

19.2.2 Scalars

MinimumCollateralVault

The minimum collateral a vault needs to provide to participate in the issue process.

Note: This is a protection against spamming the protocol with very small collateral amounts. Vaults are still able to withdraw the collateral after registration, but at least it requires an additional transaction fee, and it provides protection against accidental registration with very low amounts of collateral.

PunishmentDelay

If a Vault fails to execute a correct redeem or replace, it is *temporarily* banned from further issue, redeem or replace requests. This value configures the duration of this ban (in number of blocks) .

SecureCollateralThreshold

Determines the over-collateralization rate for collateral locked by Vaults, necessary for issuing tokens. This threshold should be greater than the LiquidationCollateralThreshold, and typically it should be greater than the PremiumRedeemThreshold as well.

The vault can take on issue requests depending on the collateral it provides and under consideration of the `SecureCollateralThreshold`. The maximum amount of interbtc a vault is able to support during the issue process is based on the following equation: $\text{max(interbtc)} = \text{collateral} * \text{ExchangeRate}/\text{SecureCollateralThreshold}$.

Note: As an example, assume we use `DOT` as collateral, we issue `interbtc` and lock `BTC` on the Bitcoin side. Let's assume the `BTC/DOT` exchange rate is `80`, i.e. one has to pay 80 `DOT` to receive 1 `BTC`. Further, the

`SecureCollateralThreshold` is 200%, i.e. a vault has to provide two-times the amount of collateral to back an issue request. Now let's say the vault deposits 400 `DOT` as collateral. Then this vault can back at most 2.5 interbtc as: $400 * (1/80)/2 = 2.5$.

PremiumRedeemThreshold

Determines the rate for the collateral rate of Vaults, at which users receive a premium, allocated from the Vault's collateral, when performing a *Redeem* with this Vault. This threshold should be greater than the LiquidationCollateralThreshold. Typically this value should be greater than the LiquidationCollateralThreshold.

LiquidationCollateralThreshold

Determines the lower bound for the collateral rate in issued tokens. If a Vault's collateral rate drops below this, automatic liquidation is triggered.

LiquidationVault

Account identifier of an artificial vault maintained by the VaultRegistry to handle interbtc balances and DOT collateral of liquidated Vaults. That is, when a vault is liquidated, its balances are transferred to `LiquidationVault` and claims are later handled via the `LiquidationVault`.

Note: A Vault's token balances and DOT collateral are transferred to the `LiquidationVault` as a result of automated liquidations and *reportVaultTheft*.

19.2.3 Maps

Vaults

Mapping from accounts of Vaults to their struct. `<Account, Vault>`.

19.2.4 Structs

Vault

Stores the information of a Vault.

Parameter	Type	Description
`toBeIssuedTokens`	interbtc	Number of interbtc tokens currently requested as part of an uncompleted issue request.
`issuedTokens`	interbtc	Number of interbtc tokens actively issued by this Vault.
`toBeRedeemedTokens`	interbtc	Number of interbtc tokens reserved by pending redeem and replace requests.
`collateral`	DOT	Total amount of collateral provided by this vault (note: "free" collateral is calculated on the fly and updated each time new exchange rate data is received).
`toBeReplacedTokens`	interbtc	Number of interbtc tokens requested for replacement.
`replaceCollateral`	DOT	Griefing collateral to be used for accepted replace requests.
`backingCollateral`	DOT	The total amount of collateral the vault uses as insurance for the issued tokens.
`wallet`	Wallet<BtcAddress>	A set of Bitcoin address(es) of this vault, used for theft detection. Additionally, it contains the btcPublicKey used for generating deposit addresses in the issue process.
`bannedUntil`	u256	Block height until which this vault is banned from being used for Issue, Redeem (except during automatic liquidation) and Replace .
`status`	VaultStatus	Current status of the vault (Active, Liquidated, CommittedTheft)

Note: This specification currently assumes for simplicity that a vault will reuse the same BTC address, even after multiple redeem requests. **[Future Extension]**: For better security, Vaults may desire to generate new BTC addresses each time they execute a redeem request. This can be handled by pre-generating multiple BTC addresses and storing these in a list for each Vault. Caution is necessary for users which execute issue requests with "old" vault addresses - these BTC must be moved to the latest address by Vaults.

19.3 Dispatchable Functions

19.3.1 registerVault

Registers a new Vault. The vault locks up some amount of collateral, and provides a public key which is used for the *On-Chain Key Derivation Scheme*.

Specification

Function Signature

`registerVault(vault, collateral, btcPublicKey)`

Parameters

- `vault`: The account of the vault to be registered.
- `collateral`: to-be-locked collateral.
- `btcPublicKey`: public key used to derive deposit keys with the *On-Chain Key Derivation Scheme*.

Events

- `RegisterVault(Vault, collateral)`: emit an event stating that a new vault (`vault`) was registered and provide information on the Vault's collateral (`collateral`).

Preconditions

- The function call MUST be signed by `vaultId`.
- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The vault MUST NOT be registered yet
- The vault MUST have sufficient funds to lock the collateral
- `collateral > MinimumCollateralVault`, i.e., the vault MUST provide sufficient collateral (above the spam protection threshold).

Postconditions

- The provided `collateral` is locked.
- A new vault is added to `Vaults`, with `backing_collateral` set to `collateral`, and with `btcPublicKey` as the public key in the wallet.
- The status is set to `Active(true)`, meaning the new vault accepts new issues.
- The rest of the variables (`issuedTokens`, `toBeIssuedTokens`, etc) are zero-initialized.

19.3.2 registerAddress

Add a new BTC address to the vault's wallet. Typically this function is called by the vault client to register a return-to-self address, prior to making redeem/replace payments. If a vault makes a payment to an address that is not registered, nor is a valid redeem/replace payment, it will be marked as theft.

Specification

Function Signature

`registerAddress(vaultId, address)`

Parameters

- `vaultId`: the account of the vault.
- `address`: a valid BTC address.

Events

- `RegisterAddress(vaultId, address)`

Precondition

- The function call MUST be signed by `vaultId`.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST NOT be registered.

Postconditions

- `address` is added to the vault's wallet.

19.3.3 updatePublicKey

Changes a vault's public key that is used for the *On-Chain Key Derivation Scheme*.

Specification

Function Signature

`updatePublicKey(vaultId, publicKey)`

Parameters

- `vaultId`: the account of the vault.
- `publicKey`: the new BTC public key of the vault.

Events

- `UpdatePublicKey(vaultId, publicKey)`

Preconditions

- The function call MUST be signed by `vaultId`.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.

Postconditions

- The vault's public key is set to `publicKey`.

19.3.4 depositCollateral

The vault locks additional collateral as a security against stealing the Bitcoin locked with it.

Specification

Function Signature

`lockCollateral(vaultId, collateral)`

Parameters

- `vaultId`: The account of the vault locking collateral.
- `collateral`: to-be-locked collateral.

Events

- `DepositCollateral(vaultId, newCollateral, totalCollateral, freeCollateral)`: emit an event stating how much new (`newCollateral`), total collateral (`totalCollateral`) and freely available collateral (`freeCollateral`) the vault calling this function has locked.

Precondition

- The function call MUST be signed by `vaultId`.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- The vault MUST have sufficient unlocked collateral to lock.

Postconditions

- The vault's `backingCollateral` is increased by the amount `collateral`.

19.3.5 withdrawCollateral

A vault can withdraw its *free* collateral at any time, as long as the collateralization ratio remains above the `SecureCollateralThreshold`. Collateral that is currently being used to back issued interbtc remains locked until the vault is used for a redeem request (full release can take multiple redeem requests).

Specification

Function Signature

`withdrawCollateral(vaultId, withdrawAmount)`

Parameters

- `vaultId`: The account of the vault withdrawing collateral.
- `withdrawAmount`: To-be-withdrawn collateral.

Events

- `WithdrawCollateral(vaultId, withdrawAmount, totalCollateral)`: emit emit an event stating how much collateral was withdrawn by the vault and total collateral a vault has left.

Preconditions

- The function call MUST be signed by `vaultId`.
- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- A vault with id `vaultId` MUST be registered.
- The collatalization rate of the vault MUST remain above `SecureCollateralThreshold` after the withdrawal of `withdrawAmount`.

Postconditions * An amount of `withdrawAmount` is unlocked.

19.4 Functions called from other pallets

19.4.1 tryIncreaseToBeIssuedTokens

During an issue request function (*requestIssue*), a user must be able to assign a vault to the issue request. As a vault can be assigned to multiple issue requests, race conditions may occur. To prevent race conditions, a Vault's collateral is *reserved* when an `IssueRequest` is created - `toBeIssuedTokens` specifies how much interbtc is to be issued (and the reserved collateral is then calculated based on *getExchangeRate*).

Specification

Function Signature

`tryIncreaseToBeIssuedTokens(vaultId, tokens)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be locked.

Events

- `IncreaseToBeIssuedTokens(vaultId, tokens)`

Preconditions

- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- A vault with id `vaultId` MUST be registered.
- The vault MUST have sufficient collateral to remain above the `SecureCollateralThreshold` after issuing `tokens`.
- The vault status MUST be *Active(true)*
- The vault MUST NOT be banned

Postconditions

- The vault's `toBeIssuedTokens` is increased by an amount of `tokens`.

19.4.2 decreaseToBeIssuedTokens

A Vault's committed tokens are unreserved when an issue request (*cancelIssue*) is cancelled due to a timeout (failure!). If the vault has been liquidated, the tokens are instead unreserved on the liquidation vault.

Specification

Function Signature

`decreaseToBeIssuedTokens(vaultId, tokens)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be unreserved.

Events

- `DecreaseToBeIssuedTokens(vaultId, tokens)`

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- If the vault is not liquidated, it MUST have at least `tokens` `toBeIssuedTokens`.
- If the vault *is* liquidated, it MUST have at least `tokens` `toBeIssuedTokens`.

Postconditions

- If the vault is *not* liquidated, its `toBeIssuedTokens` is decreased by an amount of `tokens`.
- If the vault *is* liquidated, the liquidation vault's `toBeIssuedTokens` is decreased by an amount of `tokens`.

19.4.3 issueTokens

The issue process completes when a user calls the *executeIssue* function and provides a valid proof for sending BTC to the vault. At this point, the `toBeIssuedTokens` assigned to a vault are decreased and the `issuedTokens` balance is increased by the `amount` of issued tokens.

Specification

Function Signature

`issueTokens(vaultId, amount)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc that were just issued.

Events

- `IssueTokens(vaultId, tokens)`: Emit an event when an issue request is executed.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- If the vault is *not* liquidated, its `toBeIssuedTokens` MUST be greater than or equal to `tokens`.
- If the vault *is* liquidated, the `toBeIssuedTokens` of the liquidation vault MUST be greater than or equal to `tokens`.

Postconditions

- If the vault is *not* liquidated, its `toBeIssuedTokens` is decreased by `tokens`, while its `issuedTokens` is increased by `tokens`.
- If the vault *is* liquidated, the `toBeIssuedTokens` of the liquidation vault is decreased by `tokens`, while its `issuedTokens` is increased by `tokens`.

19.4.4 tryIncreaseToBeRedeemedTokens

Add an amount of tokens to the `toBeRedeemedTokens` balance of a vault. This function serves as a prevention against race conditions in the redeem and replace procedures. If, for example, a vault would receive two redeem requests at the same time that have a higher amount of tokens to be issued than his `issuedTokens` balance, one of the two redeem requests should be rejected.

Specification

Function Signature

`tryIncreaseToBeRedeemedTokens(vaultId, tokens)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be redeemed.

Events

- `IncreaseToBeRedeemedTokens(vaultId, tokens)`: Emit an event when a redeem request is requested.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- The vault MUST NOT be liquidated.
- The vault MUST have sufficient tokens to reserve, i.e. `tokens` must be less than or equal to `issuedTokens - toBeRedeemedTokens`.

Postconditions

- The vault's `toBeRedeemedTokens` is increased by `tokens`.

19.4.5 decreaseToBeRedeemedTokens

Subtract an amount tokens from the `toBeRedeemedTokens` balance of a vault. This function is called from *cancelRedeem*.

Specification

Function Signature

`decreaseToBeRedeemedTokens(vaultId, tokens)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc not to be redeemed.

Events

- `DecreaseToBeRedeemedTokens(vaultId, tokens)`: Emit an event when a redeem request is cancelled.

Preconditions

- The BTC Parachain status in the *Security* component must not be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- If the vault is *not* liquidated, its `toBeRedeemedTokens` MUST be greater than or equal to `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` of the liquidation vault MUST be greater than or equal to `tokens`.

Postconditions

- If the vault is *not* liquidated, its `toBeRedeemedTokens` is decreased by `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` of the liquidation vault is decreased by `tokens`.

19.4.6 decreaseTokens

Decreases both the `toBeRedeemed` and `issued` tokens, effectively burning the tokens. This is called from *cancelRedeem*.

Specification

Function Signature

`decreaseTokens(vaultId, user, tokens)`

Parameters

- `vaultId`: The BTC Parachain address of the Vault.
- `userId`: The BTC Parachain address of the user that made the redeem request.
- `tokens`: The amount of interbtc that were not redeemed.

Events

- `DecreaseTokens(vaultId, userId, tokens)`: Emit an event if a redeem request cannot be fulfilled.

Preconditions

- The BTC Parachain status in the *Security* component must not be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- If the vault is *not* liquidated, its `toBeRedeemedTokens` and `issuedTokens` MUST be greater than or equal to `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` and `issuedTokens` of the liquidation vault MUST be greater than or equal to `tokens`.

Postconditions

- If the vault is *not* liquidated, its `toBeRedeemedTokens` and `issuedTokens` are decreased by `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` and `issuedTokens` of the liquidation vault are decreased by `tokens`.

19.4.7 redeemTokens

Reduces the to-be-redeemed tokens when a redeem request completes

Specification

Function Signature

`redeemTokens(vaultId, tokens, premium, redeemerId)`

Parameters

- `vaultId`: the id of the vault from which to redeem tokens
- `tokens`: the amount of tokens to be decreased
- `premium`: amount of collateral to be rewarded to the redeemer if the vault is not liquidated yet
- `redeemerId`: the id of the redeemer

Events

One of:

- `RedeemTokens(vault, tokens)`: Emit an event when a redeem request successfully completes.
- `RedeemTokensPremium(vaultId, tokens, premium, redeemerId)`: Emit an event when a redeem event with a non-zero premium completes.

- `RedeemTokensLiquidatedVault(vaultId, tokens, amount)`: Emit an event when a redeem is executed on a liquidated vault.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- **If the vault is *not* liquidated:**
  - The vault's `toBeRedeemedTokens` must be greater than or equal to `tokens`.
  - If `premium > 0`, then the vault's `backingCollateral` must be greater than or equal to `premium`.
- If the vault *is* liquidated, then the liquidation vault's `toBeRedeemedTokens` must be greater than or equal to `tokens`

Postconditions

- **If the vault is *not* liquidated:**
  - The vault's `toBeRedeemedTokens` is decreased by `tokens`, and its `issuedTokens` is increased by the same amount.
  - If `premium = 0`, then the `RedeemTokens` event is emitted
  - If `premium > 0`, then `premium` is transferred from the vault's collateral to the redeemer. The `RedeemTokensPremium` event is emitted.
- If the vault *is* liquidated, then the liquidation vault's `toBeRedeemedTokens` is decreased by `tokens`, and its `issuedTokens` is increased by the same amount. The `RedeemTokensLiquidatedVault` event is emitted.

19.4.8 redeemTokensLiquidation

Handles redeem requests which are executed against the LiquidationVault. Reduces the issued token of the LiquidationVault and slashes the corresponding amount of collateral.

Specification

Function Signature

`redeemTokensLiquidation(redeemerId, tokens)`

Parameters

- `redeemerId` : The account of the user redeeming interbtc.
- `tokens`: The amount of interbtc to be burned, in exchange for collateral.

Events

- `RedeemTokensLiquidation(redeemerId, tokens, reward)`: Emit an event when a liquidation redeem is executed.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- The liquidation vault MUST have sufficient tokens, i.e. `tokens` MUST be less than or equal to its `issuedTokens - toBeRedeemedTokens`.

Postconditions

- The liquidation vault's `issuedTokens` is reduced by `tokens`.

- The redeemer has received an amount of collateral equal to `(tokens / liquidationVault.issuedTokens) * liquidationVault.backingCollateral`.

19.4.9 tryIncreaseToBeReplacedTokens

Increases the toBeReplaced tokens of a vault, which indicates how many tokens other vaults can replace in total.

Specification

Function Signature

`tryIncreaseToBeReplacedTokens(oldVault, tokens, collateral)`

Parameters

- `vaultId`: Account identifier of the vault to be replaced.
- `tokens`: The amount of interbtc replaced.
- `collateral`: The extra collateral provided by the new vault as griefing collateral for potential accepted replaces.

Returns

- A tupple of the new total `toBeReplacedTokens` and `replaceCollateral`.

Events

- `IncreaseToBeReplacedTokens(vaultId, tokens)`: Emit an event when a replacement is requested for additional tokens.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.
- The vault MUST NOT be liquidated.
- The vault's increased `toBeReplaceedTokens` MUST NOT exceed `issuedTokens - toBeRedeemedTokens`.

Postconditions

- The vault's `toBeReplaceTokens` is increased by `tokens`.
- The vault's `replaceCollateral` is increased by `collateral`.

19.4.10 decreaseToBeReplacedTokens

Decreases the toBeReplaced tokens of a vault, which indicates how many tokens other vaults can replace in total.

Specification

Function Signature

`decreaseToBeReplacedTokens(oldVault, tokens)`

Parameters

- `vaultId`: Account identifier of the vault to be replaced.
- `tokens`: The amount of interbtc replaced.

Returns

- A tupple of the new total `toBeReplacedTokens` and `replaceCollateral`.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `vaultId` MUST be registered.

Postconditions

- The vault's `replaceCollateral` is decreased by `(min(tokens, toBeReplacedTokens) / toBeReplacedTokens) * replaceCollateral`.
- The vault's `toBeReplaceTokens` is decreased by `min(tokens, toBeReplacedTokens)`.

Note: the `replaceCollateral` is not actually unlocked - this is the responsibility of the caller. It is implemented this way, because in *requestRedeem* it needs to be unlocked, whereas in *requestReplace* it must remain locked.

19.4.11 replaceTokens

When a replace request successfully completes, the `toBeRedeemedTokens` and the `issuedToken` balance must be reduced to reflect that removal of interbtc from the `oldVault`.Consequently, the `issuedTokens` of the `newVault` need to be increased by the same amount.

Specification

Function Signature

`replaceTokens(oldVault, newVault, tokens, collateral)`

Parameters

- `oldVault`: Account identifier of the vault to be replaced.
- `newVault`: Account identifier of the vault accepting the replace request.
- `tokens`: The amount of interbtc replaced.
- `collateral`: The collateral provided by the new vault.

Events

- `ReplaceTokens(oldVault, newVault, tokens, collateral)`: Emit an event when a replace requests is successfully executed.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `oldVault` MUST be registered.
- A vault with id `newVault` MUST be registered.
- If `oldVault` is *not* liquidated, its `toBeRedeemedTokens` and `issuedTokens` MUST be greater than or equal to `tokens`.
- If `oldVault` *is* liquidated, the liquidation vault's `toBeRedeemedTokens` and `issuedTokens` MUST be greater than or equal to `tokens`.
- If `newVault` is *not* liquidated, its `toBeIssuedTokens` MUST be greater than or equal to `tokens`.
- If `newVault` *is* liquidated, the liquidation vault's `toBeIssuedTokens` MUST be greater than or equal to `tokens`.

Postconditions

- **If `oldVault` is *not* liquidated:**
  - The `oldVault`'s `toBeRedeemedTokens` and `issuedTokens` are decreased by the amount `tokens`.
  - Some of the `oldVault`'s collateral is unlocked: an amount of `tokens / toBeRedeemed`.
- If `oldVault` *is* liquidated, the liquidation vault's `toBeRedeemedTokens` and `issuedTokens` are decrease by the amount `tokens`.
- If `newVault` is *not* liquidated, its `toBeIssuedTokens` is decreased by `tokens`, while its `issuedTokens` is increased by the same amount.
- If `newVault` *is* liquidated, the liquidation vault's `toBeIssuedTokens` is decreased by `tokens`, while its `issuedTokens` is increased by the same amount.

19.4.12 cancelReplaceTokens

Cancels a replace: decrease the old-vault's to-be-redeemed tokens, and the new-vault's to-be-issued tokens. If one or both of the vaults has been liquidated, the change is forwarded to the liquidation vault.

Specification

Function Signature

`cancelReplaceTokens(oldVault, newVault, tokens)`

Parameters

- `oldVault`: Account identifier of the vault to be replaced.
- `newVault`: Account identifier of the vault accepting the replace request.
- `tokens`: The amount of interbtc replaced.

Events

- `CancelReplaceTokens(oldVault, newVault, tokens, collateral)`: Emit an event when a replace requests is cancelled.

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN: 2`.
- A vault with id `oldVault` MUST be registered.
- A vault with id `newVault` MUST be registered.
- If `oldVault` is *not* liquidated, its `toBeRedeemedTokens` MUST be greater than or equal to `tokens`.
- If `oldVault` *is* liquidated, the liquidation vault's `toBeRedeemedTokens` MUST be greater than or equal to `tokens`.
- If `newVault` is *not* liquidated, its `toBeIssuedTokens` MUST be greater than or equal to `tokens`.
- If `newVault` *is* liquidated, the liquidation vault's `toBeIssuedTokens` MUST be greater than or equal to `tokens`.

Postconditions

- If `oldVault` is *not* liquidated, its `toBeRedeemedTokens` is decreased by `tokens`.
- If `oldVault` *is* liquidated, the liquidation vault's `toBeRedeemedTokens` is decreased by `tokens`.
- If `newVault` is *not* liquidated, its `toBeIssuedTokens` is decreased by `tokens`.
- If `newVault` *is* liquidated, the liquidation vault's `toBeIssuedTokens` is decreased by `tokens`.

19.4.13 liquidateVault

Liquidates a vault, transferring token balances to the `LiquidationVault`, as well as collateral.

Specification

Function Signature

`liquidateVault(vault)`

Parameters

- `vault`: Account identifier of the vault to be liquidated.

Events

- `LiquidateVault(vault)`: Emit an event indicating that the vault with `vault` account identifier has been liquidated.

Preconditions

Postconditions

- The liquidation vault's `issuedTokens`, `toBeIssuedTokens` and `toBeRedeemedTokens` are increased by the respective amounts in the vault.
- The vault's `issuedTokens` and `toBeIssuedTokens` are set to 0.
- Collateral is moved from the vault to the liquidation vault: an amount of `confiscatedCollateral - confiscatedCollateral * (toBeRedeemedTokens / (toBeIssuedTokens + issuedTokens))` is moved, where `confiscatedCollateral` is the minimum of the `backingCollateral` and `SecureCollateralThreshold` times the equivalent worth of the amount of tokens it is backing.

Note: If a vault successfully executes a replace after having been liquidated, it receives some of its confiscated collateral back.

19.5 Events

19.5.1 RegisterVault

Emit an event stating that a new vault (`vault`) was registered and provide information on the Vault's collateral (`collateral`).

Event Signature

`RegisterVault(vault, collateral)`

Parameters

- `vault`: The account of the vault to be registered.
- `collateral`: to-be-locked collateral in DOT.

Functions

- *registerVault*

19.5.2 DepositCollateral

Emit an event stating how much new (`newCollateral`), total collateral (`totalCollateral`) and freely available collateral (`freeCollateral`) the vault calling this function has locked.

Event Signature

`DepositCollateral(Vault, newCollateral, totalCollateral, freeCollateral)`

Parameters

- `Vault`: The account of the vault locking collateral.
- `newCollateral`: to-be-locked collateral in DOT.
- `totalCollateral`: total collateral in DOT.
- `freeCollateral`: collateral not "occupied" with interbtc in DOT.

Functions

- *depositCollateral*

19.5.3 WithdrawCollateral

Emit emit an event stating how much collateral was withdrawn by the vault and total collateral a vault has left.

Event Signature

`WithdrawCollateral(Vault, withdrawAmount, totalCollateral)`

Parameters

- `Vault`: The account of the vault locking collateral.
- `withdrawAmount`: To-be-withdrawn collateral in DOT.
- `totalCollateral`: total collateral in DOT.

Functions

- *withdrawCollateral*

19.5.4 RegisterAddress

Emit an event stating that a vault (`vault`) registered a new address (`address`).

Event Signature

`RegisterAddress(vault, address)`

Parameters

- `vault`: The account of the vault to be registered.
- `address`: The added address

Functions

- *registerAddress*

19.5.5 UpdatePublicKey

Emit an event stating that a vault (`vault`) registered a new address (`address`).

Event Signature

`UpdatePublicKey(vaultId, publicKey)`

Parameters

- `vaultId`: the account of the vault.
- `publicKey`: the new BTC public key of the vault.

Functions

- *updatePublicKey*

19.5.6 IncreaseToBeIssuedTokens

Emit

Event Signature

`IncreaseToBeIssuedTokens(vaultId, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be locked.

Functions

- *tryIncreaseToBeIssuedTokens*

19.5.7 DecreaseToBeIssuedTokens

Emit

Event Signature

`DecreaseToBeIssuedTokens(vaultId, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be unreserved.

Functions

- *decreaseToBeIssuedTokens*

19.5.8 IssueTokens

Emit an event when an issue request is executed.

Event Signature

`IssueTokens(vault, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc that were just issued.

Functions

- *issueTokens*

19.5.9 IncreaseToBeRedeemedTokens

Emit an event when a redeem request is requested.

Event Signature

`IncreaseToBeRedeemedTokens(vault, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc to be redeemed.

Functions

- *tryIncreaseToBeRedeemedTokens*

19.5.10 DecreaseToBeRedeemedTokens

Emit an event when a replace request cannot be completed because the vault has too little tokens committed.

Event Signature

`DecreaseToBeRedeemedTokens(vault, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc not to be replaced.

Functions

- *decreaseToBeRedeemedTokens*

19.5.11 DecreaseTokens

Emit an event if a redeem request cannot be fulfilled.

Event Signature

`DecreaseTokens(vault, user, tokens, collateral)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `user`: The BTC Parachain address of the user that made the redeem request.
- `tokens`: The amount of interbtc that were not redeemed.
- `collateral`: The amount of collateral assigned to this request.

Functions

- *decreaseTokens*

19.5.12 RedeemTokens

Emit an event when a redeem request successfully completes.

Event Signature

`RedeemTokens(vault, tokens)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc redeemed.

Functions

- *redeemTokens*

19.5.13 RedeemTokensPremium

Emit an event when a user is executing a redeem request that includes a premium.

Event Signature

`RedeemTokensPremium(vault, tokens, premiumDOT, redeemer)`

Parameters

- `vault`: The BTC Parachain address of the Vault.
- `tokens`: The amount of interbtc redeemed.
- `premiumDOT`: The amount of DOT to be paid to the user as a premium using the Vault's released collateral.
- `redeemer`: The user that redeems at a premium.

Functions

- *redeemTokens*

19.5.14 RedeemTokensLiquidation

Emit an event when a redeem is executed under the `LIQUIDATION` status.

Event Signature

`RedeemTokensLiquidation(redeemer, redeemDOTinBTC)`

Parameters

- `redeemer` : The account of the user redeeming interbtc.
- `redeemDOTinBTC`: The amount of interbtc to be redeemed in DOT with the `LiquidationVault`, denominated in BTC.

Functions

- *redeemTokensLiquidation*

19.5.15 RedeemTokensLiquidatedVault

Emit an event when a redeem is executed on a liquidated vault.

Event Signature

`RedeemTokensLiquidation(redeemer, tokens, unlockedCollateral)`

Parameters

- `redeemer` : The account of the user redeeming interbtc.
- `tokens`: The amount of interbtc that have been refeemed.
- `unlockedCollateral`: The amount of collateral that has been unlocked for the vault for this redeem.

Functions

- *redeemTokens*

19.5.16 ReplaceTokens

Emit an event when a replace requests is successfully executed.

Event Signature

`ReplaceTokens(oldVault, newVault, tokens, collateral)`

Parameters

- `oldVault`: Account identifier of the vault to be replaced.
- `newVault`: Account identifier of the vault accepting the replace request.
- `tokens`: The amount of interbtc replaced.
- `collateral`: The collateral provided by the new vault.

Functions

- *replaceTokens*

19.5.17 LiquidateVault

Emit an event indicating that the vault with `vault` account identifier has been liquidated.

Event Signature

`LiquidateVault(vault)`

Parameters

- `vault`: Account identifier of the vault to be liquidated.

Functions

- *liquidateVault*

19.6 Error Codes

`InsufficientVaultCollateralAmount`

- **Message**: "The provided collateral was insufficient - it must be above `MinimumCollateralVault`."
- **Cause**: The vault provided too little collateral, i.e. below the MinimumCollateralVault limit.

`VaultNotFound`

- **Message**: "The specified vault does not exist."
- **Cause**: vault could not be found in `Vaults` mapping.

`ERR_INSUFFICIENT_FREE_COLLATERAL`

- **Message**: "Not enough free collateral available."
- **Cause**: The vault is trying to withdraw more collateral than is currently free.

`ERR_EXCEEDING_VAULT_LIMIT`

- **Message**: "Issue request exceeds vault collateral limit."
- **Cause**: The collateral provided by the vault combined with the exchange rate forms an upper limit on how much interbtc can be issued. The requested amount exceeds this limit.

`ERR_INSUFFICIENT_TOKENS_COMMITTED`

- **Message**: "The requested amount of `tokens` exceeds the amount available to vault."
- **Cause**: A user requests a redeem with an amount exceeding the vault's tokens, or the vault is requesting replacement for more tokens than it has available.

`ERR_VAULT_BANNED`

- **Message**: "Action not allowed on banned vault."
- **Cause**: An illegal operation is attempted on a banned vault, e.g. an issue or redeem request.

`ERR_ALREADY_REGISTERED`

- **Message**: "A vault with the given accountId is already registered."
- **Cause**: A vault tries to register a vault that is already registered.

`ERR_RESERVED_DEPOSIT_ADDRESS`

- **Message**: "Deposit address is already registered."
- **Cause**: A vault tries to register a deposit address that is already in the system.

`ERR_VAULT_NOT_BELOW_LIQUIDATION_THRESHOLD`

- **Message**: "Attempted to liquidate a vault that is not undercollateralized."
- **Cause**: A vault has been reported for being undercollateralized, but at the moment of execution, it is no longer undercollateralized.

`ERR_INVALID_PUBLIC_KEY`

- **Message**: "Deposit address could not be generated with the given public key."
- **Cause**: An error occurred while attempting to generate a new deposit address for an issue request.

Note: These are the errors defined in this pallet. It is possible that functions in this pallet return errors defined in other pallets.

CHAPTER

TWENTY

VAULT LIQUIDATIONS

Vaults are collateralized entities in the system responsible for keeping BTC in custody. If Vaults fail to behave according to protocol rules, they face punishment through slashing of collateral. There are two types of failures: **safety failures** and **crash failures**.

20.1 Safety Failures

A safety failure occurs in two cases:

1. **Theft**: a Vault is considered to have committed theft if it moves/spends BTC unauthorized by the interbtc bridge. Theft is detected and reported by Relayers via an SPV proof.
2. **Severe Undercollteralization**: a Vaults drops below the `110%` liquidation collateral threshold.

In both cases, the Vault's entire BTC holdings are liquidated and its DOT collateral is slashed - up to 150% (secure collateral threshold) of the liquidated BTC value.

Consequently, the bridge offers users to burn ("Burn Event") their tokens to restore the 1:1 balance between the issued (e.g., interbtc) and locked asset (e.g., BTC).

20.2 Crash Failures

If Vaults go offline and fail to execute redeem, they are:

- **Penalized** (punishment fee slashed) and
- **Temporarily banned for 24 hours** from accepting further issue, redeem, and replace requests.

The punishment fee is calculated based on the Vault's SLA (Service Level Agreement) level, which is a value between 0 and 100. The higher the Vault's SLA, the lower the punishment for a failed redeem.

In detail, the punishment fee is calculated as follows:

- **Minimum Punishment Fee**: 10% of the failed redeem value.
- **Maximum Punishment Fee**: 30% of the failed redeem value.
- **Punishment Fee**: calculated based on the Vaults SLA value as defined in the *SlashAmountVault*.

20.3 Liquidations (Safety Failures)

When a Vault is liquidated, its `issued` and `toBeIssued` tokens are *moved* to the Liquidation Vault. In contrast, the Vault's `toBeRedeemed` tokens are *copied* over. The Vault loses access to at least part of its backing collateral:

- The Vault loses `confiscatedCollateral = min(SECURE_THRESHOLD * (issued + toBeIssued), backingCollateral)`, and any leftover amount is released to its free balance.
- Of the confiscated collateral, an amount of `confiscatedCollateral * (toBeRedeemed / (issued + toBeIssued))` stays locked in the Vault, and the rest is moved to the Liquidation Vault. This is in anticipation of vaults being able to complete ongoing redeem and replace requests. When these requests succeed, the liquidated Vault's collateral is returned. When the requests fail (i.e., the `cancel` calls are being made), the remaining collateral is slashed to the Liquidation Vault.

When the Liquidation Vault contains tokens, users can do a liquidation_redeem ("burn event"). Users can call this function to burn interbtc and receive DOT in return.

- The user receives `liquidationVault.collateral * (burnedTokens / (issued + toBeIssued)` in its free balance.
- At most `liquidationVault.issued - liquidationVault.toBeRedeemed` tokens can be burned.

Vault liquidation affects Vault interactions is the following ways:

- Operations that increase `toBeIssued` or `toBeRedeemed` are disallowed. This means that no new issue/redeem/replace request can be made.
- Any operation that would decrease `toBeIssued` or change `issued` on a user Vault instead changes it on the Liquidation Vault
- Any operation that would decrease `toBeRedeemed` tokens on a user Vault *additionally* decreases it on the Liquidation Vault

20.3.1 Issue

- **`requestIssue`**
  - disallowed
- **`executeIssue`**
  - Overpayment protection is disabled; if a user transfers too many BTC, the user loses it.
  - SLA of Vault is not increased
- **`cancelIssue`**
  - User's griefing collateral is released back to the user, rather than slashed to the Vault.

20.3.2 Redeem

- **`requestRedeem`**
  - disallowed
- **`executeRedeem`**
  - Part of the Vault's collateral is released. Amount: `Vault.backingCollateral * (redeem.amount / Vault.toBeRedeemed)`, where `toBeRedeemed` is read before it is decreased
  - The premium, if any, is not transferred to the user.
- **`cancelRedeem`**

  - Calculates `slashedCollateral = Vault.backingCollateral * (redeem.amount / Vault.toBeRedeemed)`, where `toBeRedeemed` is read *before* it is decreased, and then:
  - **If reimburse:**
    * transfers `slashedCollateral` to user.
  - **Else if not reimburse:**
    * transfers `slashedCollateral` to Liquidation Vault.
  - Fee pool does not receive anything.

20.3.3 Replace

- **`requestReplace`, `acceptReplace`, `withdrawReplace`**
  - disallowed
- **`executeReplace`**
  - **if `oldVault` is liquidated**
    * `oldVVault`'s collateral is released as in `executeRedeem` above
  - **if `newVault` is liquidated**
    * `newVault`'s remaining collateral is slashed as in `executeIssue` above
- **`cancelReplace`**
  - **if `oldVault` is liquidated**
    * collateral is slashed to Liquidation Vault, as in `cancelRedeem` above
  - **if `newVault` is liquidated**
    * griefing collateral is slashed to `newVault`'s free balance rather than to its backing collateral

20.3.4 Implementation Notes

- In `cancelIssue`, when the griefing collateral is slashed, it is forwarded to the fee pool.
- In `cancelReplace`, when the griefing collateral is slashed, it is forwarded to the backing collateral to the Vault. In case the Vault is liquidated, it is forwarded to the free balance of the Vault.
- In `premiumRedeem`, the griefing collateral is set as 0.
- In `executeReplace`, the `oldVault`'s griefing collateral is released, regardless of whether or not it is liquidated.

CHAPTER

TWENTYONE

SECURITY ANALYSIS

21.1 Replay Attacks

Without adequate protection, inclusion proofs for transactions on Bitcoin can be **replayed** by: (i) the user to trick interbtc component into issuing duplicate interbtc tokens and (ii) the vault to reuse a single transaction on Bitcoin to falsely prove multiple redeem, replace, and refund requests. We employ two different mechanisms to achieve this:

1. *Identification via OP_RETURN*: When sending a Bitcoin transaction, the BTC-Parachain requires that a unique identifier is included as one of the outputs in the transaction.
2. *Unique Addresses via On-Chain Key Derivation*: The BTC-Parachain generates a new and unique address that Bitcoin can be transferred to.

The details of the transaction format can be found at the accepted Bitcoin transaction format.

21.1.1 OP_RETURN

Applied in the following protocols:

- *Redeem*
- *Replace*
- *Refund*

A simple and practical mitigation is to introduce unique identifiers for each protocol execution and require transactions on Bitcoin submitted to the BTC-Relay of these protocols to contain the corresponding identifier.

In this specification, we achieve this by requiring that vaults prepare a transaction with at least two outputs. One output is an OP_RETURN with a unique hash created in the *Security* module. Vaults are using Bitcoin full-nodes to send transactions and can easily and programmatically create transactions with an OP_RETURN output.

UX Issues with OP_RETURN

However, OP_RETURN has severe UX problems. Most Bitcoin wallets do not support OP_RETURN. That is, a user cannot use the UI to easily create an OP_RETURN transaction. As of this writing, the only wallet that supports this out of the box is Electrum. Other wallets, such as Samurai, exist but only support mainnet transactions (hence, have not yet been tested).

In addition, while Bitcoin's URI format (BIP21) generally supports OP_RETURN, none of the existing wallets have implemented an interpreter for this "upgraded" URI structure - this would have to be implemented manually by wallet providers. An alternative solution is to pre-generate the Bitcoin transaction for the user. The problem with this is that - again - most Bitcoin wallets do not support parsing of raw Bitcoin transactions. That is, a user cannot easily verify that the raw Bitcoin transaction string provided by interbtc indeed does what it should do (and does not steal the user's funds). This approach works with hardware wallets, such as Ledger - but again, not all users will use interbtc from hardware wallets.

21.1.2 Unique Addresses via On-Chain Key Derivation

Applied in the following protocol:

- *Issue*

To avoid the use of OP_RETURN during the issue process, and the significant usability drawbacks incurred by this approach, we employ the use of an On-chain Key Derivation scheme (OKD) for Bitcoin's ECDSA (secp256k1 curve). The BTC-Parachain maintains a BTC 'master' public key for each registered vault and generates a unique, ephemeral 'deposit' public key (and RIPEMD-160 address) for each issue request, utilizing the unique issue identifier for replay protection.

This way, each issue request can be linked to a distinct Bitcoin transaction via the receiving ('deposit') address, making it impossible for vaults/users to execute replay attacks. The use of OKD thereby allows to keep the issue process non-interactive, ensuring vaults cannot censor issue requests.

On-Chain Key Derivation Scheme

We define the full OKD scheme as follows (additive notation):

Preliminaries

A Vault has a private/public keypair (v, V), where $V = vG$ and G is the base point of the secp256k1 curve. Upon registration, the Vault submits public key V to the BTC-Parachain storage.

Issue protocol via new OKD scheme

1. **When a user creates an issue request, the BTC-Parachain**
    a. Computes $c = H(V||id)$, where id is the unique issue identifier, generated on-chain by the BTC-Parachain using the user's AccountId and an internal auto-incrementing nonce as input.
    b. Generates a new public key ("deposit public key") $D = Vc$ and then the corresponding BTC RIPEMD-160 hash-based address $addr(D)$ ('deposit' address) using D as input.
    c. Stores D and $addr(D)$ alongside the id of the Issue request.
2. The user deposits the amount of to-be-issued BTC to $addr(D)$ and submits the Bitcoin transaction inclusion proof, alongside the raw Bitcoin transaction, to BTC-Relay.
3. The BTC-Relay verifies that the destination address of the Bitcoin transaction is indeed $addr(D)$ (and the amount, etc.) and mints new interbtc to the user's AccountId.
4. The Vault knows that the private key of D is cv, where $c = H(V||id)$ is publicly known (can be computed by the Vault off-chain, or stored on-chain for convenience). The Vault can now import the private key cv into its Bitcoin wallet to gain access to the deposited BTC (required for redeem).

21.2 Counterfeiting

A vault which receives lock transaction from a user during *Issue* could use these coins to re-execute the issue itself, creating counterfeit interbtc. This would result in interbtc being issued for the same amount of lock transaction breaking **consistency**, i.e., $|locked_BTC| < |interbtc|$. To this end, the interbtc component forbids vaults to move locked funds lock transaction received during *Issue* and considers such cases as theft. This theft is observable by any user. However, we used the specific roles of Staked Relayers to report theft of BTC. To restore **Consistency**, the interbtc component slashes the vault's entire collateral and executes automatic liquidation, yielding negative utility for the vault. To allow economically rational vaults to move funds on the BTC Parachain we use the *Replace*, a non-interactive atomic cross-chain swap (ACCS) protocol based on cross-chain state verification.

21.3 Permanent Blockchain Splits

Permanent chain splits or *hard forks* occur where consensus rules are loosened or conflicting rules are introduced, resulting in multiple instances of the same blockchain. Thereby, a mechanism to differentiate between the two resulting chains *replay protection* is necessary for secure operation.

21.3.1 Backing Chain

If replay protection is provided after a permanent split of Bitcoin, the BTC-Relay must be updated to verify the latter for Bitcoin (or Bitcoin' respectively). If no replay protection is implemented, BTC-Relay will behave according to the protocol rules of Bitcoin for selecting the "main" chain. For example, it will follow the chain with most accumulated PoW under Nakamoto consensus.

21.3.2 Issuing Chain

A permanent fork on the issuing blockchain results in two chains I and I' with two instances of the interbtc component identified by the same public keys. To prevent an adversary exploiting this to execute replay attacks, both users and vaults must be required to include a unique identifier (or a digest thereof) in the transactions published on Bitcoin as part of *Issue* and *Redeem* (in addition to the identifiers introduces in Replay Attacks).

Next, we identify two possibilities to synchronize Bitcoin balances on I and I': (i) deploy a chain relay for I on I' and vice-versa to continuously synchronize the interbtc components or (ii) redeploy the interbtc component on both chains and require users and vaults to re-issue Bitcoin, explicitly selecting I or I'.

21.4 Denial-of-Service Attacks

interbtc is decentralized by design, thus making denial-of-service (DoS) attacks difficult. Given that any user with access to Bitcoin and BTC Parachain can become a vault, an adversary would have to target all vaults simultaneously. Where there are a large number of vaults, this attack would be impractical and expensive to perform. Alternatively, an attacker may try to target the interbtc component. However, performing a DoS attack against the interbtc component is equivalent to a DoS attack against the entire issuing blockchain or network, which conflicts with our assumptions of a resource bounded adversary and the security models of Bitcoin and BTC Parachain. Moreover, should an adversary perform a Sybil attack and register as a large number of vaults and ignore service requests to perform a DoS attack, the adversary would be required to lock up a large amount of collateral to be effective. This would lead to the collateral being slashed by the interbtc component, making this attack expensive and irrational.

21.5 Fee Model Security: Sybil Attacks and Extortion

While the exact design of the fee model lies beyond the scope of this paper, we outline the following two restrictions, necessary to protect against attacks by malicious vaults.

21.5.1 Sybil Attacks

To prevent financial gains from Sybil attacks, where a single adversary creates multiple low collateralized vaults, the interbtc component can enforce (i) a minimum necessary collateral amount and (ii) a fee model based on issued volume, rather than "pay-per-issue". In practice, users can in principle easily filter out low-collateral vaults.

21.5.2 Extortion

Without adequate restrictions, vaults could set extreme fees for executing *Redeem*, making redeeming of Bitcoin unfeasible. To this end, the interbtc component must enforce that either (i) no fees can be charged for executing *Redeem* or (ii) fees for redeeming must be pre-agreed upon during issue.

21.6 Griefing

Griefing describes the act of blocking a vaults collateral by creating "bogus" requests. There are two cases:

1. A user can create an issue request without the intention to issue tokens. The user "blocks" the vault's collateral for a specific amount of time. if enough users execute this, a legitimate user could possibly not find a vault with free collateral to start an issue request.
2. A vault can request to be replaced without the intention to be replaced. When another vault accepts the replace request, that vault needs to lock additional collateral. The requesting vault, however, could never complete the replace request to e.g. ensure that it will be able to serve more issue requests.

For both cases, we require the requesting parties to lock up a (small) amount of griefing collateral. This makes such attacks costly for the attacker.

21.7 Concurrency

We need to ensure that concurrrent issue, redeem, and replace requests are handled.

21.7.1 Concurrent redeem

We need to make sure that a vault cannot be used in multiple redeem requests in parallel if that would exceed his amount of locked BTC. **Example**: If the vault has 5 BTC locked and receives two redeem requests for 5 interbtc/BTC, he can only fulfil one and would lose his collateral with the other.

21.7.2 Concurrent issue and redeem

A vault can be used in parallel for issue and redeem requests. In the issue procedure, the vault's `issuedTokens` are already increased when the issue request is created. However, this is before (!) the BTC is sent to the vault. If we used these `issuedTokens` as a basis for redeem requests, we might end up in a case where the vault does not have enough BTC. **Example**: The vault already has 3 BTC in custody from previous successful issue procedures. A user creates an issue request for 2 interbtc. At this point, the `issuedTokens` by this vault are 5. However, his BTC balance is only 3. Now, a user could create a redeem request of 5 interbtc and the vault would have to fulfill those. The user could then cancel the issue request over 2 interbtc. The vault could only send 3 BTC to the user and would lose his deposit. Or the vault just loses his deposit without sending any BTC.

21.7.3 Solution

We use seperate token balances to handle issue, replace, and redeem requests in the *Vault Registry*.

CHAPTER

TWENTYTWO

PERFORMANCE ANALYSIS

CHAPTER

TWENTYTHREE

ECONOMIC INCENTIVES

Incentives are the core of decentralized systems. Fundamentally, actors in decentralized systems participate in a game where each actor attempts to maximize its utility. Designs of such decentralized systems need to encode a mechanism that provides clear incentives for actors to adhere to protocol rules while discouraging undesired behavior. Specifically, actors make risk-based decisions: payoffs associated with the execution of certain actions are compared against the risk incurred by the action. The BTC Parachain, being an open system with multiple distinct stakeholders, must hence offer a mechanism to assure honest participation outweighs subversive strategies.

The overall objective of the incentive mechanism is an optimization problem with private information in a dynamic setting. Users need to pay fees to Vaults in return for their service. On the one hand, user fees should be low enough to allow them to profit from having interbtc (e.g., if a user stands to gain from earning interest in a stablecoin system using interbtc, then the fee for issuing interbtc should not outweigh the interest gain). On the other hand, fees need to be high enough to encourage Vaults and Staked Relayers to lock their DOT in the system and operate Vault/Staked Relayer clients. This problem is amplified as the BTC Parachain does not exist in isolation and Vaults/Staked Relayers can choose to participate in other protocols (e.g., staking, stablecoin issuance) as well. In the following we outline the constraints we see, a minimal viable incentive model, and pointers to further research questions we plan to solve by getting feedback from potential Vaults and Staked Relayers as well as quantitative modeling.

23.1 Roles

We can classify four groups of users, or agents, in the BTC Parachain system. This is mainly based on their prior cryptocurrency holdings - namely BTC and DOT.

23.1.1 Users

- **Protocol role** Users lock BTC with Vaults to create interbtc. They hold and/or use interbtc for payments, lending, or investment in financial products. At some point, users redeem interbtc for BTC by destroying the backed assets.
- **Economics** A user holds BTC and has exposure to an exchange rate from BTC to other assets. A user's incentives are based on the services (and their rewards) available when issuing interbtc.
- **Risks** A user gives up custody over their BTC to a Vault. The Vault is over-collateralized in DOT (i.e., compared to the USD they will lose when taking away the user's BTC), however, in a market crisis with significant price drops and liquidity shortages, Vaults might choose to keep the BTC. Users will be reimbursed with DOT in that case - not the currency they initially started out with.

23.1.2 Vaults

- **Protocol role** Vaults lock up DOT collateral in the BTC Parachain and hold users' BTC (i.e., receive custody). When users wish to redeem interbtc for BTC, Vaults release BTC to users according to the events received from the BTC Parachain.
- **Economics** Vaults hold DOT and thus have exposure to the DOT price against other assets. Vaults inherently make a bet that DOT will increase in value against other assets – otherwise they would simply exchange DOT against their preferred asset(s). This is a simplified view of the underlying problem. In reality, we need to additionally consider nominated vaults as well as vault pooling. Moreover, the inflation of DOT will play a major role in selection of the asset that fees should be paid in.
- **Risks** A Vault backs a set of interbtc with DOT collateral. If the exchange rate of the DOT/BTC pair drops the Vault stands at risk to not be able to keep the required level of over-collateralization. This risk can be elevated by a shortage of liquidity.

23.1.3 Staked Relayers

- **Protocol role** Staked Relayers run Bitcoin full nodes and submit block headers to BTC-Relay, ensuring it remains up to date with Bitcoin's state. They also report failures occurring on Bitcoin (missing transactional data or invalid blocks) and report misbehaving Vaults who have allegedly stolen BTC (move BTC outside of BTC Parachain constraints). Staked Relayers lock DOT as collateral to disincentivize false flagging on Vaults and Bitcoin failures.
- **Economics** Staked Relayers are exposed to similar mechanics as Vaults, since they also hold DOT. However, they have no direct exposure to the BTC/DOT exchange rate, since they (typically, at least as part of the BTC Parachain) do not hold BTC. As such, Staked Relayers can purely be motivated to earn interest on DOT, but can also have the option to earn interest in interbtc and optimize their holdings depending on the best possible return at any given time.
- **Risks** Staked Relayers need to keep an up-to-date Bitcoin full node running to receive the latest blocks and be able to verify transaction availability and validity. They might risk voting on wrong status update proposals for the BTC Parachain if their node is being attacked, e.g. eclipse or DoS attacks.

23.1.4 Collators

- **Protocol role** Collators are full nodes on both a parachain and the Relay Chain. They collect parachain transactions and produce state transition proofs for the validators on the Relay Chain. They can also send and receive messages from other parachains using XCMP.
- More on collators can be found in the Polkadot wiki: https://wiki.polkadot.network/docs/en/learn-collator#docsNav

23.2 Processes

We will now explain how each of the four agent types above profits from participating in the BTC Parachain. Specifically, we sketch a typical interaction flow with the BTC Parachain and explain how each agent type behaves.

23.2.1 Issue process

The first step is to issue interbtc and give users access to other protocols.

1. A Vault locks an amount of DOT in the BTC Parachain.
2. A user requests to issue a certain amount of interbtc. A user can directly select a Vault to issue with. If the user does not select a Vault, a Vault is automatically selected with preference given to Vaults with higher SLA rating. In the first iteration of the protocol this selection is deterministic.
3. The user transfers the equivalent amount of BTC that he wants to issue to the Vault. Additionally, the user provides a fee in BTC that is locked with the Vault as well.
4. The user proves the transfer of BTC to the BTC Parachain and receives the requested amount of newly issued interbtc.
5. The fees paid by the users are issued as interbtc as well. They are forwarded to a general fee pool and distributed according to a configurable distribution to all Vaults, Staked Relayers, Maintainers, and Collators. This ensures that all participants earn on new issue requests, independent if their current collateral is already reserved or not.
6. The user can then freely use the issued interbtc to participate in any other protocol deployed on the BTC Parachain and connected Parachains.

23.2.2 Redeem process

The BTC Parachain is intended to primarily incentivize users to issue interbtc and minimize friction to redeem BTC. Hence, the redeem process is structured in a simple way with providing the same incentives to all participating Vaults. Moreover, Vaults are punished for not fulfilling a redeem request in time.

A user can retry to redeem with other Vaults in case a redeem request is not fulfilled. In this case, the non-fulfilling Vault will be punished not by the entire BTC amount but rather by a smaller amount.

1. A user requests to redeem interbtc for BTC with a Vault and locks the equivalent amount of interbtc.
2. The Vault sends the BTC minus the globally defined fee to the user.
3. The fee is kept in interbtc and, equally to the issue process, paid into the fee pool to be distributed among all participants.
4. The Vault proves correct redeem with the BTC Parachain and unlocks the DOT collateral in return.
5. The Vault can decide to keep the DOT collateral in the BTC Parachain to participate in issue requests or withdraw the collateral.

23.2.3 interbtc interest process

Fees paid in interbtc (on Issue, Redeem, and Replace) are forwarded to a fee pool. The fee pool then distributes the interbtc fees to all Vaults, Staked Relayers, Maintainers, and Collators according to a configurable distribution, and, if implemented, depending on the SLA score. All participants are able to withdraw their accumulated fees at any time.

23.2.4 DOT interest process

Fees paid in DOT are forwarded to a fee pool. The fee pool then distributes the interbtc fees to all Vaults, Staked Relayers, Maintainers, and Collators according to a configurable distribution, and, if implemented, depending on the SLA score. All participants are able to withdraw their accumulated fees at any time.

23.2.5 Arbitrage

After the issue process is completed a user can access any protocol deployed on Polkadot using interbtc. Not everyone that wants to obtain interbtc has to take this route. We imagine that liquidity providers issue interbtc and exchange these for other assets in the Polkadot ecosystem. The price of interbtc and BTC will hence be decoupled.

Price decoupling of BTC and interbtc, in turn, can be used by arbitrage traders. If interbtc trades relatively higher than BTC, arbitrage traders will seek to issue new interbtc with their existing BTC to sell interbtc at a higher market price. In case BTC trades above interbtc, arbitrageurs seek to redeem interbtc for BTC and trade these at a higher market price.

23.3 Constraints

We sketched above how each agent can be motivated to participate based on their incentive. However, determining the fee model, including how much a user should pay in BTC fees or the interest earned in DOT or interbtc by Vaults and Staked Relayers, requires careful consideration. These numbers depend on certain constraints than can be roughly categorized in two parts:

1. **Inherent risks**: Each agent takes on different risks that include, for example, giving up custody of their BTC, exchange rate risk on the DOT/BTC pair, costs to maintain the infrastructure to operate Vault and Staked Relayer clients, as well as trusting the BTC Parachain to operate correctly and as designed.
2. **Opportunity costs**: Each agent might decide to take an alternative path to receive the desired incentives. For example, users might pick a different platform or bridge to utilize their BTC. Also Vaults, Staked Relayers, and Keepers might pick other protocols to earn interest on their DOT holdings.

We provide an overview of the risks and alternatives for the agents in Table 1. When an agent is exposed to a high risk and has several alternatives, the agent needs to receive an accordingly high reward in return: if the risks and alternatives outweigh the incentives for an agent, the agent will not join the BTC Parachain. As seen in already deployed protocols including wBTC and pTokens, experiencing – to this date – insignificant volume, the balance of risks, alternatives, and incentives need to motivate agents to join.

Table 1: A subjective rating of the risks and alternatives for each agent. Risk ratings are from low to high. Alternatives ratings are also from low to high, where “high” indicates the existence of numerous viable alternatives, while “low“ indicates that the BTC Parachain is the dominant option on the market.

Agent	Risk rating	Risks	Opportunity cost	Alternatives
User	high	Counterparty (Vault, Staked Relayer), Technical risk (BTC Parachain), Market risks (DOT/BTC volatility and liquidity through Vault)	medium	wBTC, tBTC, RenVM, ChainX
Vault	high	Counterparty (Staked Relayer), Technical risk (BTC Parachain, Vault client), Market risks (DOT/BTC volatility and liquidity)	high	Staking (relay chain, Parachains), Lending (Acala), Trading (Laminar)
Staked Relayer	low	Technical risk (BTC Parachain, relayer client, Bitcoin client)	high	Staking (relay chain, Parachains), Lending (Acala), Trading (Laminar)
Keeper	high	Counterparty (Staked Relayer), Technical risk (BTC Parachain, Vault and Keeper client), Market risks (DOT/BTC volatility and liquidity)	high	Staking (relay chain, Parachains), Lending (Acala), Trading (Laminar)

CHAPTER

TWENTYFOUR

FEE MODEL

We assume Vaults and Staked Relayer to be economically driven, i.e., following a strategy to maximize profits over time. While there may be altruistic actors, who follow protocol rules independent of the economic impact, we do not consider these here.

Byzantine actors aim to manipulate the system and cause damage to users - independent of the economic impact this has on them. To showcase how misbehavior impacts the fees earned by participants, we also consider byzantine behavior of Vaults and Staked Relayers in this analysis.

24.1 Currencies

The BTC-Parachain features three assets:

- *BTC* - the backing-asset (locked on Bitcoin)
- *interbtc* - the issued cryptocurrency-backed asset (on Polkadot)
- *DOT* - the currency used as collateral (*DOT* used initially but may later be replaced with a stablecoin, currency-set, or similar)

24.2 Actors: Income and Real/Opportunity Costs

The main question when designing the fee model for interbtc is: When are fees paid, by whom, and how much?

Below, we hence overview the income and cost sources for each actor/stakeholder in the BTC-Parachain. Thereby, we differentiate between the following cost types:

- **Internal costs**: costs associated directly with the BTC-Parachain (i.e., inflow or internal flow of funds)
- **External costs**: costs associated with external factors, such as node operation, engineering costs etc. (i.e., outflow of funds)
- **Opportunity costs**: lost revenue, if e.g. locked up collateral was to be used in other applications (e.g. to stake on the Relay chain)

24.2.1 Users

- **Income**
  - Slashed collateral
  - Use of interbtc in applications
- **Internal Cost**
  - Issue and redeem fees
  - BTC-Relay fees

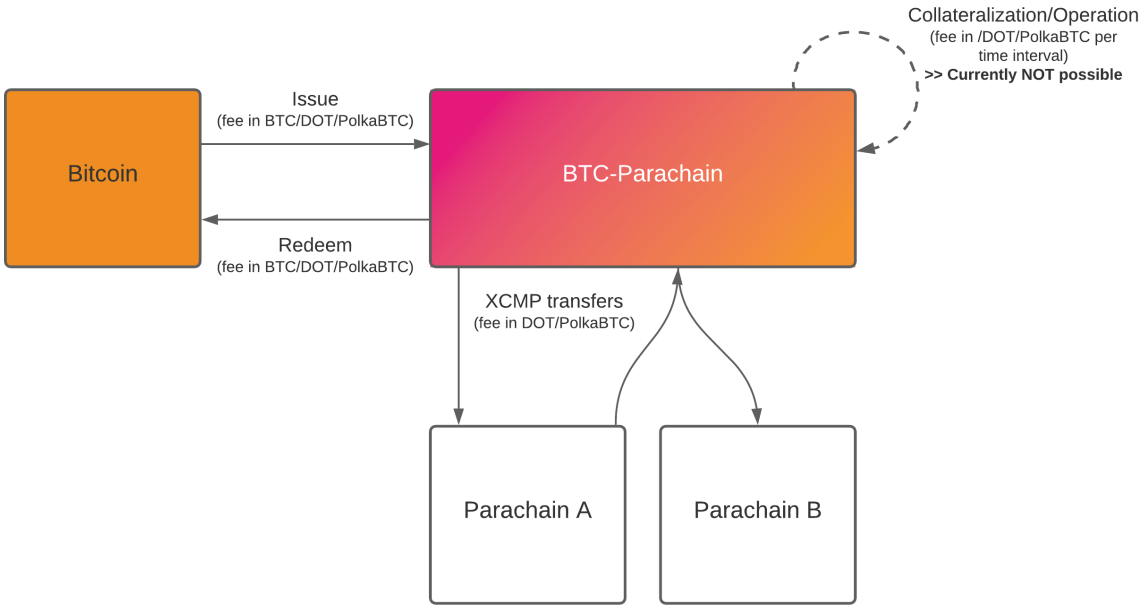


Fig. 24.1: High-level overview of fee accrual in the BTC-Parachain (external sources only).

  - Parachain transaction fees
- **External Costs**
  - *None*
- **Opportunity Cost**
  - BTC lockup

24.2.2 Vaults

- **Income**
  - Issue and redeem fees
  - SLA-based subsidy
- **Internal Cost**
  - BTC-Relay fee
  - Parachain transaction fees
  - (Upon failure: Slashed collateral)
- **External Costs**
  - *None*
- **Opportunity Cost**
  - Backing-collateral lockup

24.2.3 Staked Relayers

- **Income**
  - SLA-based subsidy
  - BTC-Relay fees
- **Internal Cost**
  - Parachain transaction fees (offset against BTC-Relay fees)
  - (Upon failure: Slashed collateral)
- **External Costs**
  - Parachain node operation/maintenance costs
  - Bitcoin full node operation/maintenance costs
- **Opportunity Cost**
  - Staking-collateral lockup

24.2.4 Collators

- **Income**
  - Transaction fees
  - Parachain subsidy (SLA-based?)
- **Internal Cost**
  - *None*
- **External Costs**
  - Parachain node operation/maintenance costs
- **Opportunity Cost**
  - Staking-collateral lockup

24.2.5 Maintainers

- **Income**
  - Parachain subsidy (revenue share)
  - Upgrade/extension/maintenance fee
- **Internal Cost**
  - *None*
- **External Costs**
  - Server maintenance (UI, etc.)
  - Operational/developer costs
- **Opportunity Cost**
  - Other bridges

24.3 Payment flows

We detail the payment flows in the figure below:

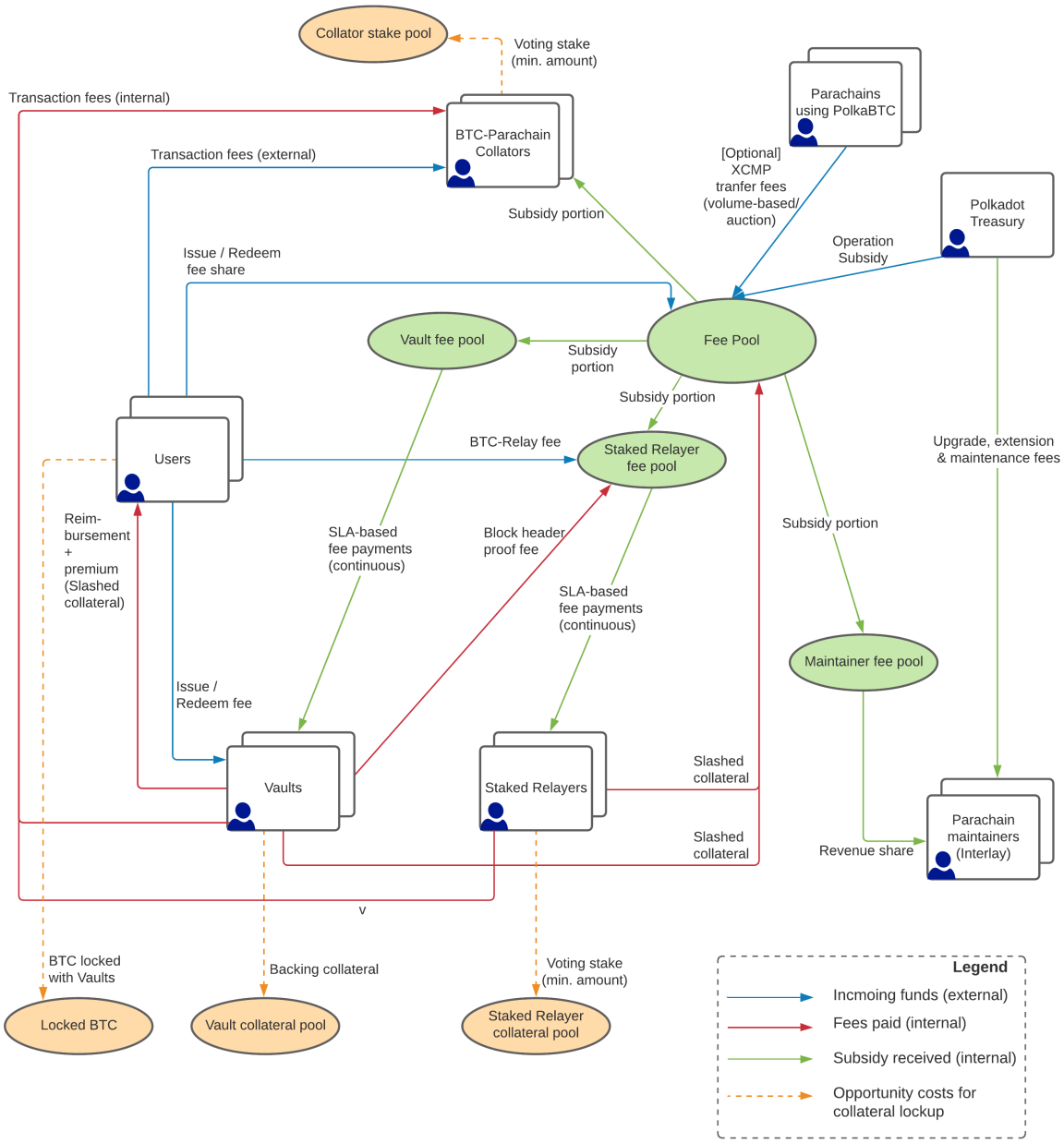


Fig. 24.2: Detailed overview of fee accrual in the BTC-Parachain, showing external and internal payment flows, as well as opportunity costs.

24.4 Challenges Around Economic Efficiency

To ensure security of interbtc, i.e., that users never face financial damage, XCLAIM relies on collateral. However, in the current design, this leads to the following economic challenges:

- **Over-collateralization**. Vaults must lock up significantly (e.g. 200%) more collateral than minted interbtc to ensure security against exchange rate fluctuations. Dynamically modifying the exchange rate could only marginally reduce this requirement, at a high computational overhead. As such, to issue 1 interbtc, one must lock up 1 BTC, as well as the 2 BTC worth of collateral (e.g. in DOT), resulting in a 300% collateralization.
- **Non-deterministic Collateral Lockup**. When a Vault locks collateral to secure interbtc, it does not know for how long this collateral will remain locked up. As such, it nearly impossible to determine a fair price for the premium charged to the user, without putting either the user or the Vault at a disadvantage.
- **Limited Chargeable Events**. The Vault only has two events during which it can charge fees: (1) fulfillment of and issue request and (2) fulfillment of a redeem request. Thereby, the fees charged for the redeem request must be **upper-bounded** for security reasons (to prevent extortion by the Vault via sky-rocketing redeem fees).

As such, an open research question is:

What is the value of a Vault's locked collateral at any given point in time, considering the value of the collateral currency, the value of locked BTC, the value of interbtc (if different from BTC), as well as the projected earning from fees over time?

24.5 Subsidizing Vault Collateral Costs

- **Higher user fees for issue/redeem** to ensure sufficiently good economic performance of Vaults to incentivize participation. Ideally, this would be combined with a supply/demand-based market for interbtc, driven by Parachains/applications on Polkadot (see below). The risk for (both) this model is that high fees may impede adoption if users revert to cheaper, yet centralized solutions.
- **XCMP fees from other Parachains**. Charge Parachains additional fees for getting access to interbtc, creating an supply/demand-based market for interbtc access. The more demand for interbtc, the higher the market price, the more BTC will be locked to mint interbtc. However, this (i) impedes adoption by other Parachains and (ii) results in clear price deviations between interbtc and BTC in times of interbtc shortage. The latter may not be a bad thing per se, yet may have an unexpected effect for applications using interbtc.
- **Polkadot treasury subsidy** to Vaults (and Staked Relayers) on a on a continuous basis, subject to correct operation / collateral usage, to account for the opportunity costs of the Vault accrued through locking up collateral.
- **Governance token model**, where tokens are allocated to Vaults on a continuous basis, subject to correct operation / collateral usage. The token model, however, needs careful consideration and a clear use case (in addition to voting).
- **On-demand collateral model via XCLAIM-Commit**, where Vaults lock up collateral only for short, deterministic periods and can hence compute an accurate fee model. In addition, users can request additional collateralization for specific periods and pay for collateral on demand. However, XCLAIM-Commit is still WIP and incurs stricter liveness requirements and a significantly more involved process for maintaining the secure 1:1 backing for Vaults.

Any of the above solutions can be implemented by themselves, or in combination - most likely, a mix of all will lead to the most well-balanced model.

24.6 Other considerations

- **Vault-User BTC Call Options / Perpetuals**: When a user locks BTC with the Vault, he implicitly sells a BTC call option to the Vault. The Vault can, at any point in time, decide to exercise this option by "stealing" the user's BTC. The price for this option is determined by *spot_price* + *punishment_fee* (*punishment_fee* is essentially the option premium). The main issue here is that we do not know how to price this option, because it has no expiry date - so this deal between the User and the Vault essentially becomes a **BTC perpetual that can be physically exercised at any point in time (American-style)**.

CHAPTER

TWENTYFIVE

SERVICE LEVEL AGREEMENTS

Vaults and Staked Relayers take up critical roles in the BTC-Parachain. Both provide collateral, have clearly defined tasks and face punishment in case of misbehavior. However, slashing collateral for each minor protocol deviation would result in too high risk profiles for Vaults and Staked Relayers, yielding these roles unattractive to users.

As a result, we introduce Service Level Agreements for Vaults and Staked Relayers: being online and following protocol rules increases the SLA, while non-critical failures reduces the rating. Higher SLAs result in higher rewards and preferred treatment where applicable in the Issue and Redeem protocols. If the SLA of a Vault or Staked Relayer falls below a certain threshold, a punishment will be incurred, ranging from a mere collateral penalty up to full collateral confiscation and a system ban.

25.1 SLA Value

The SLA value is a number between 0 and 100. When a Vault or Staked Relayer registers with BTC-Parachain, it starts with an SLA of 0.

25.2 SLA Actions

We list below several actions that Vaults and Staked Relayers can execute in the protocol that have an impact on their SLA.

25.2.1 Vaults

Desired Actions

- **Execute Issue**: execute redeem, on time with the correct amount.
- **Submit Issue Proof**: Vault submits correct Issue proof on behalf of the user.
- **Forward Additional BTC**: Vault submits correct issue or return proof where the vault is the forwarding vault.

Undesired Actions

- **Fail Redeem**: redeem not executed on time (or at all) or with the incorrect amount (more specific: fail to provide inclusion proof for BTC payment to BTC-Relay on time)

25.2.2 Staked Relayers

Desired Actions

- **Submit BTC block header**: submit a valid Bitcoin block header, that later becomes (**TODO:**define delay to not punish "good" fork submissions) part of the main chain. - [Optional]: even if the block header already is stored, an additional confirmation is treated as beneficial action. This needs to be **time-bounded**. Otherwise, resubmitting old blocks allows to improve SLA, while adding no security and spamming the Parachain)
- **Correctly report theft**: correctly report a Vault for moving BTC outside of the protocol rules (i.e., viewed as theft attempt). - Note: TX inclusion proof must pass (TODO: check how this is currently implemented).

Undesired Actions

No actions with SLA impact.

25.3 Non-SLA Actions

There are several other actions that do not impact the SLA scores at the moment. For completeness, we list them here. The SLA model might be revised and the below actions may be considered to impact the SLA in the future.

25.3.1 Vaults

Desired Actions

- **Execute Redeem**: execute redeem, on time with the correct amount.
- **Collateralization**: Maintain a collateralization rate above the *Secure Collateral Threshold*.
- **Execute Replace**: if requested replace, transfer the correct amount of BTC to the new Vault on time.

Undesired Actions

- **Fail Replace**: replace protocol (BTC transfer) not executed on time (or at all) or with the incorrect amount.
- **Undercollateralization**: Collateralization rate below *Secure Collateral Threshold*.
- **Strong Undercollateralization**: Collateralization rate below *Premium Collateral Threshold*.
- **Liquidation**: Collateralization rate below *Liquidation Collateral Threshold*, which triggers liquidation of the Vault.
- **Theft**: the Vault transfers BTC from its UTXO(s) outside of the protocol rules. There is a dedicated check for this in the BTC-Parachain: only redeem, replace and registered migration of assets are allowed and these are clearly defined.
- **Repeated Failed Redeem**: repeated failed redeem requests can incur a higher SLA deduction#
- **Repeated Failed Replace**: repeated failed replace requests can incur a higher SLA deduction

25.3.2 Staked Relayers

Desired Actions

- **Correctly report NO_DATA**: report/vote a block as NO_DATA in case of a majority vote passed
- **Correctly report INVALID**: report/vote a block as INVALID in case of a majority vote passed
- **Correct report LIQUIDATION**: report a Vault for being below the *Liquidation Collateral Threshold* and trigger automatic liquidation.
- **Correctly report ORACLE_OFFLINE**: correctly report that the/an oracle has not reported data for a pre-defined amount of time (i.e., considered offline).
- **Majority on status update vote**: participate in a status update vote on the **majority** side. - Exception: NO_DATA votes are rewarded no matter how the vote was cast. Reason: since NO_DATA does not incur slashing of minority votes, being on the "majority" side must not yield additional benefits here, otherwise this incentivizes "herd" behavior without actually performing checks.

Undesired Actions

- **Ignore vote**: do not participate in a status update vote.
- **Ignore NO_DATA**: do note vote in a NO_DATA vote at all.
- **Ignore INVALID**: do note vote in an INVALID vote at all.
- **Wrong INVALID report/vote**: report or vote on the **minority** (and persumably wrong side) of an INVALID vote
- **Governance punishment**: the governance mechanism can reduce the SLA of a Vault (e.g. if majority did not vote INVALID, but there was indeed an invalid block, i.e. an attack)
- [Optional] **Minority on status update vote**: vote on the **minority** side. - Since this will also slash collateral in most cases, e.g. INVALID votes (exception: NO_DATA), there may be no need for this extra SLA reduction.
- [Optional] **Offline**: do not perform **any** of the desired actions within a certain time frame, while being registered. Time needs to be defined.
- [Optional] **Wrong theft report**: report Vault theft but the BTC transaction turns out to be valid / according to protocol rules. - If a such wrong call will automatically fail in the parachain, then there is probably no need for SLA reduction here.
- [Optional] **Wrong ORCALE_OFFLINE report**: oracle reported offline but was online. A such wrong call will fail in the parachain, so there is probably no need for SLA reduction here.
- [Optional]: **Wrong LIQUIDATION report**: wrongly report a Vault for being below the *Liquidation Collateral Threshold*. A such wrong call will fail in the parachain, so there is probably no need for SLA reduction here.

CHAPTER

TWENTYSIX

LICENSE

CHAPTER

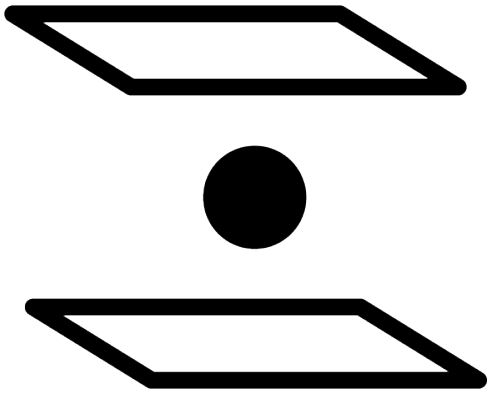
TWENTYSEVEN

INTERLAY

Interlay envisions a future where permissioned and permissionless blockchains, regardless of design and purpose, can seamlessly connect and interact. From DeFi loans to decentralized exchanges, from layer-2 protocols to application-specific ledgers: anyone should use any digital currency on any blockchain platform.

Interlay is co-founded by Imperial College London researchers Alexei Zamyatin and Dominik Harz, who have been contributing cutting edge research to the blockchain space for multiple years: from identifying centralization issues in merged mining, over off-chain computations and cross-chain bribing, to attacks against DeFi protocols.

Since the invention of XCLAIM in 2018, the team has been busy making the framework even more secure via more robust cryptographic primitives, scalable via payment channels and usable by reducing collateral requirements.



INTERLAY