interBTC Specification

Interlay

Jan 31, 2022

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

INTERBTC AT A GLANCE

The *interBTC bridge* 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.

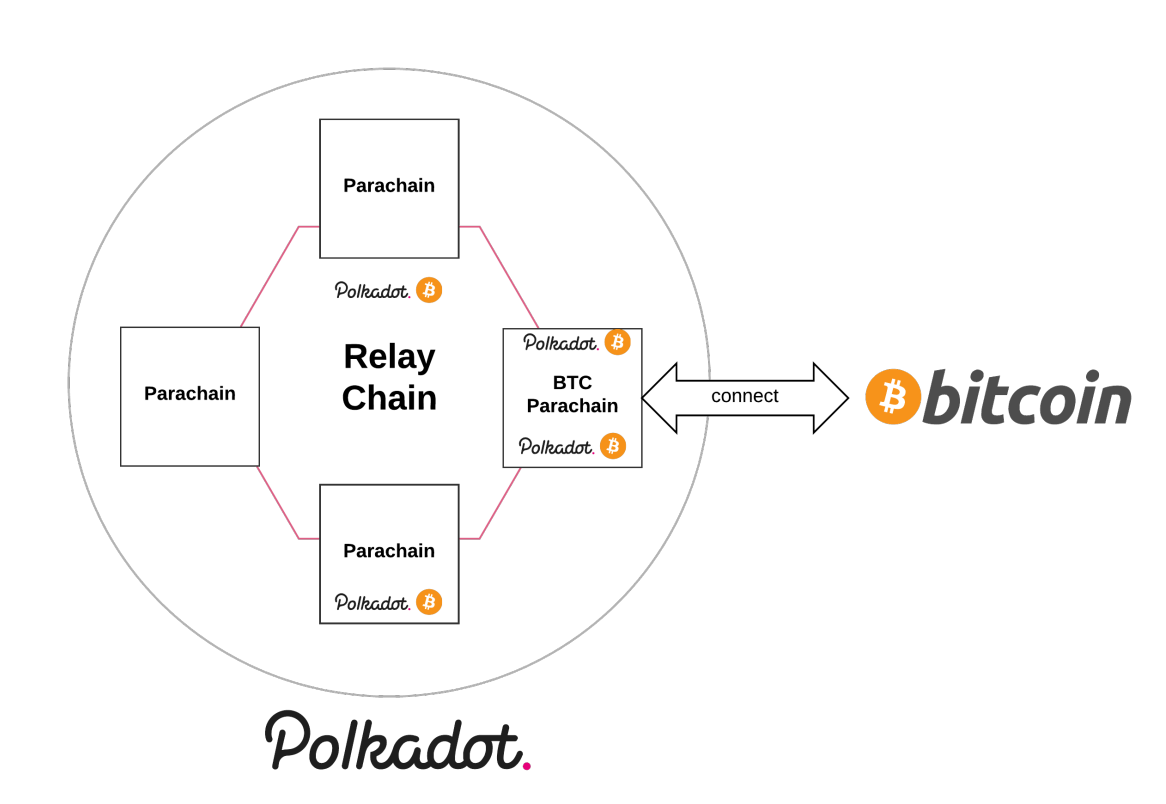


Fig. 1.1: The interBTC bridge allows the creation of collateralized 1:1 Bitcoin-backed tokens in Polkadot. These tokens can be transferred and traded within the Polkadot ecosystem.

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*.

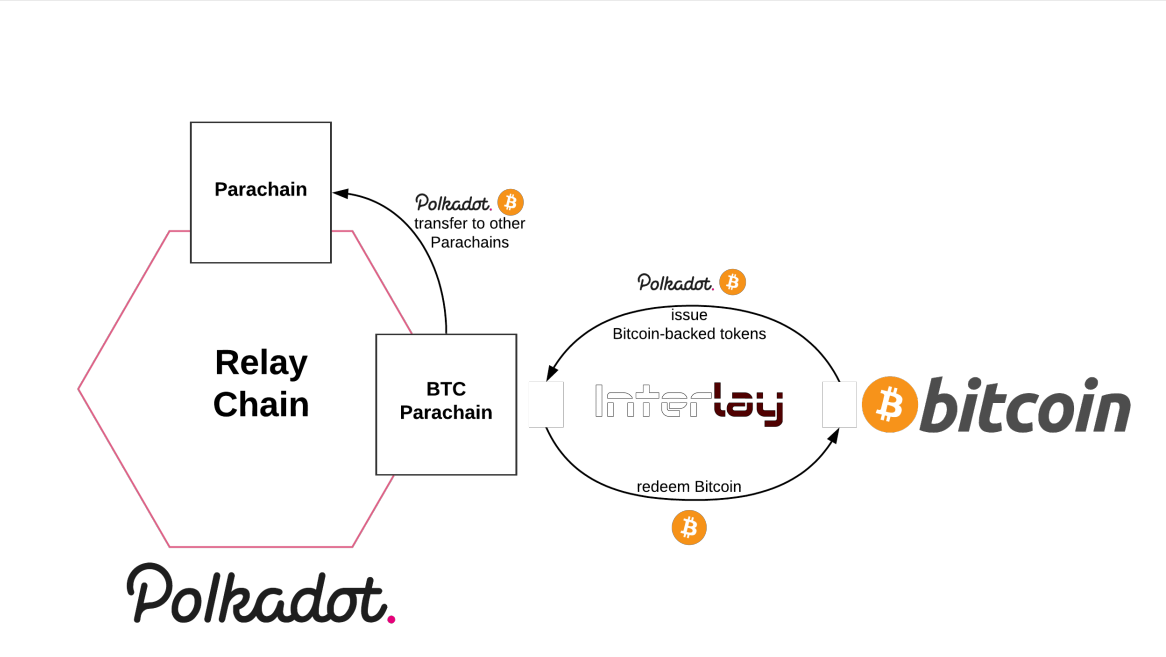


Fig. 1.2: The BTC Parachain includes a protocol to issue interBTC by locking Bitcoin and a protocol to redeem Bitcoin by burning interBTC tokens.

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.

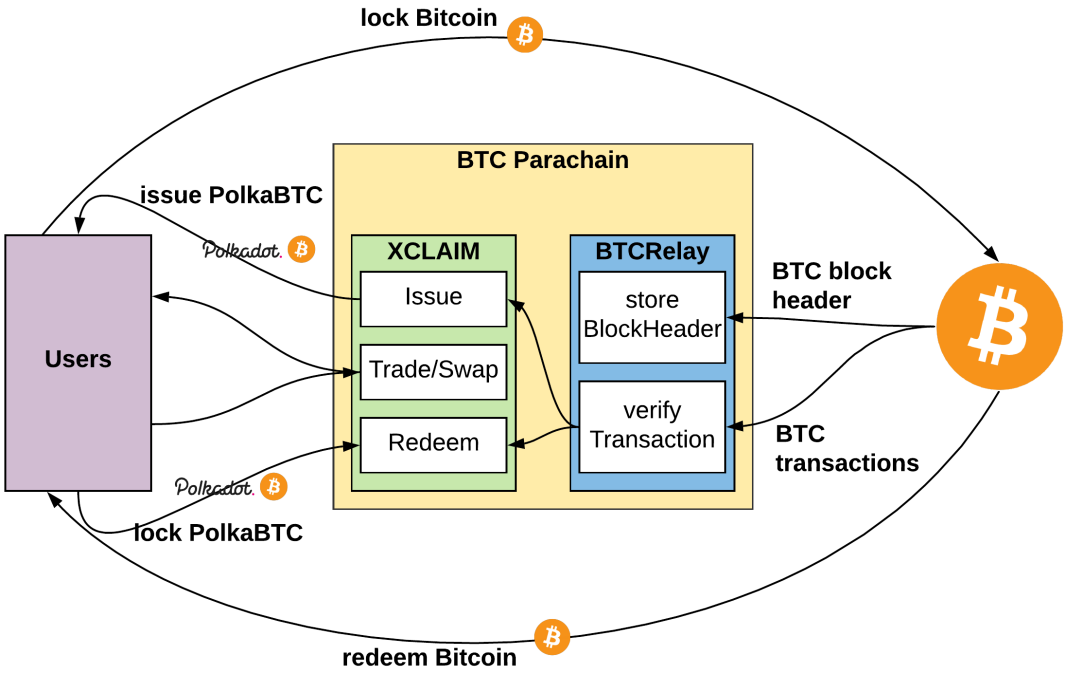


Fig. 1.3: The BTC Parachain consists of two logically different components. The XCLAIM(BTC,DOT) component (in green) maintains the accounts that own interBTC tokens. The BTC-Relay (blue) is repsonible for verifying the Bitcoin state to verify transactions. Users (in purple) are able to create new interBTC 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:

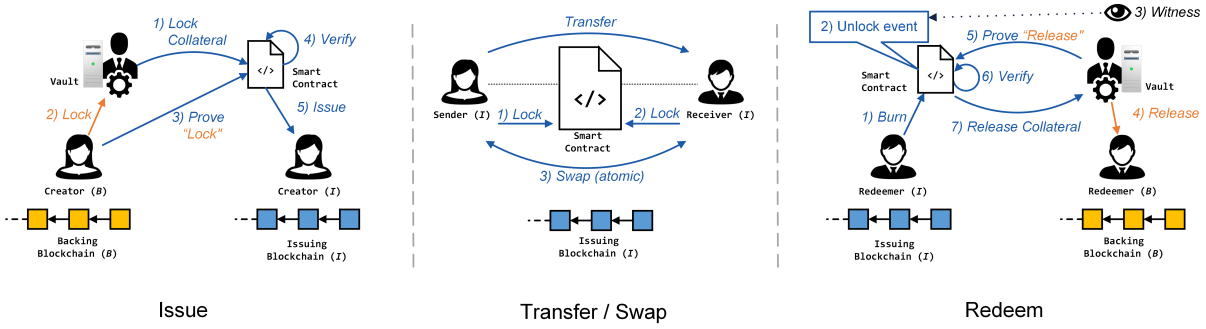


Fig. 2.1: The issue, transfer/swap, and redeem protocols in XCLAIM.

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

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.

3.1 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. 3.1 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*.
  - **Reporting**: Monitors that other Vaults do not move locked BTC on Bitcoin without prior authorization by the BTC Parachain (i.e., through one of the *Redeem*, *Replace* or *Refund* protocols).
  - **Relaying**: Submits block headers published on Bitcoin to the *BTC-Relay*.
- **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 *Currency* 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.
- **Governance Mechanism**: The Parachain Governance Mechanism monitors the correct operation of the BTC Parachain. Interacts with the *Security* module and can manually update the parameterization of all components in the BTC Parachain.

3.2 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 optimization and improvements of a module with minimal impact on other modules.

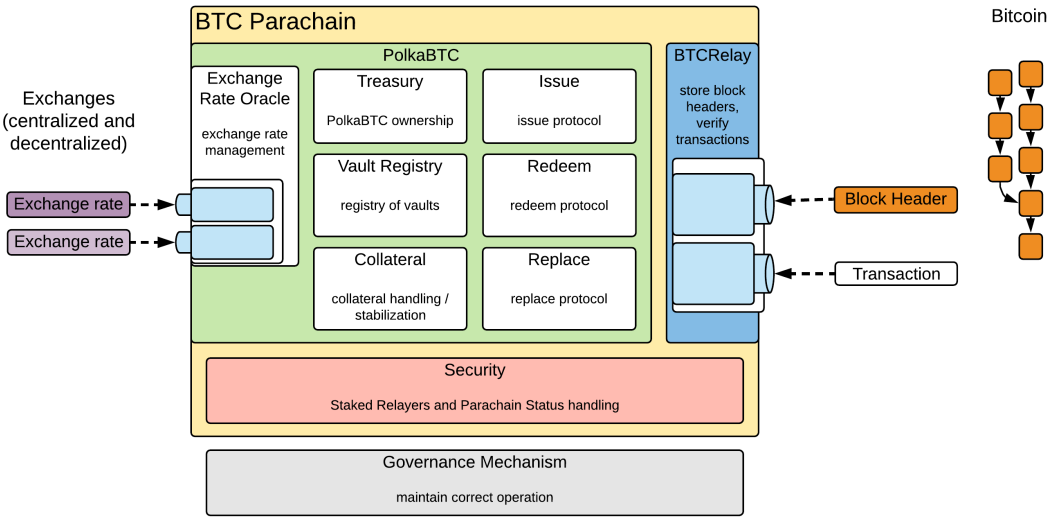


Fig. 3.1: High level overview of the BTC Parachain. interBTC consists of seven modules. The Oracle module stores the exchange rates based on the input of centralized and decentralized exchanges. The Treasury module maintains the ownership of interBTC, the VaultRegistry module stores information about the current Vaults in the system, and the Issue, Redeem and Replace modules expose functions 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.

3.2.1 BTC-Relay

BTC-Relay is a key component of the BTC Parachain on Polkadot. Its main task is to allow the Parachain to verify the state of Bitcoin and react to transactions and events. Specifically, BTC-Relay acts as a Bitcoin SPV/light client on Polkadot, storing only Bitcoin block headers and allowing users to verify transaction inclusion proofs. Further, it is able to handle forks and follows the chain with the most accumulated Proof-of-Work.

The correct operation of BTC-Relay is crucial: should BTC-Relay cease to operate, the bridge between Polkadot and Bitcoin is interrupted.

Below, we provide an overview of its components, as well as relevant actors - offering references to the full specification contained in the rest of this document.

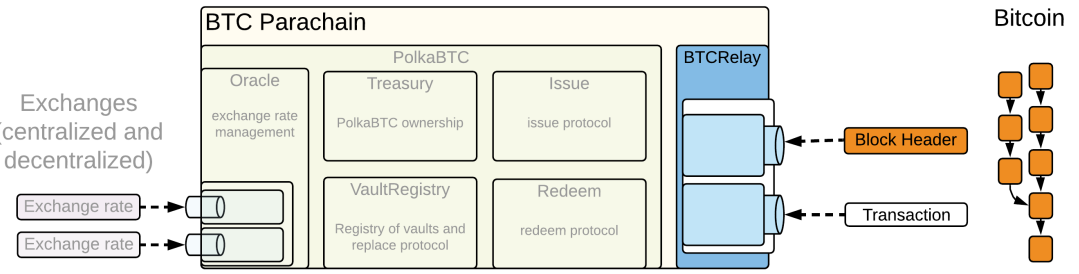


Fig. 3.2: BTC-Relay (highlighted in blue) is a key component of the BTC Parachain: it is necessary to verify and keep track of the state of Bitcoin.

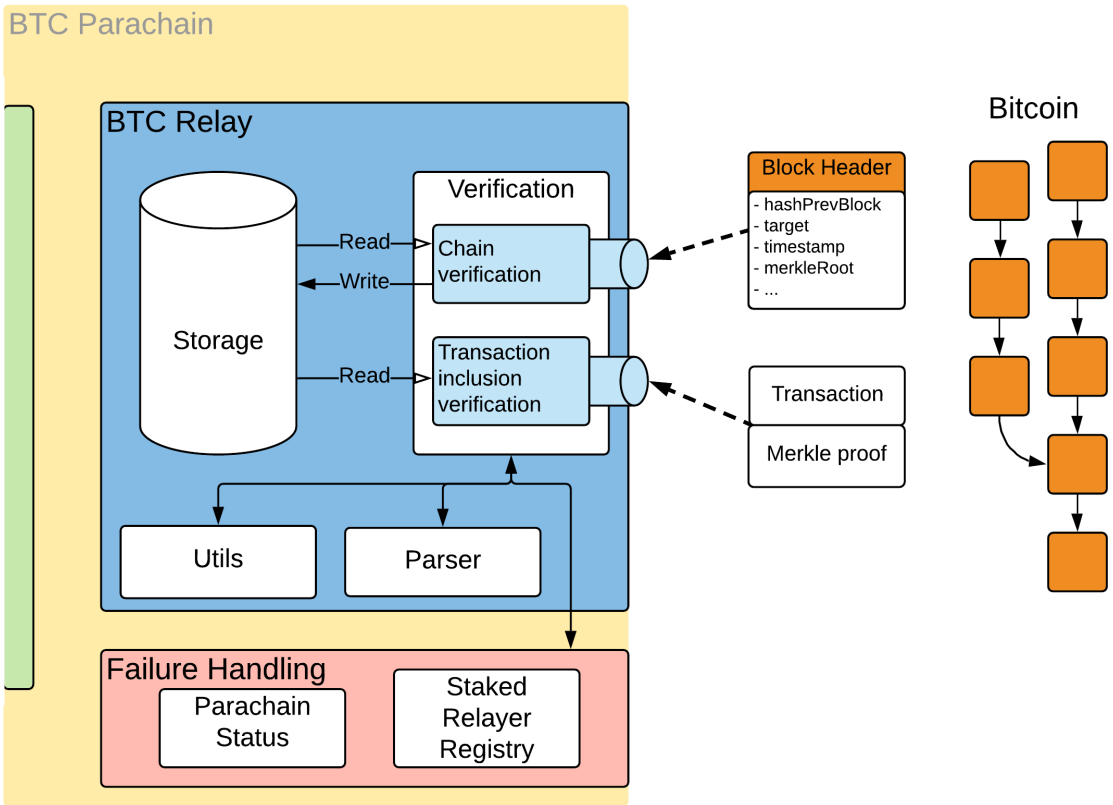


Fig. 3.3: Overview of the BTC-Relay architecture. Bitcoin block headers are submitted to the Verification Component, which interacts with the Utils, Parser and Failure Handling components, as well as the Parachain Storage.

3.2.2 Oracle

The Oracle module maintains the exchange rate value between the asset that is used to collateralize Vaults (e.g. DOT) and the wrapped asset (interBTC). Governance authorizes trusted third parties to feed the current exchange rates into the system for a nominal fee.

3.2.3 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.

3.2.4 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.
- Triggers, as a last resort, automatic liquidation if a vault falls below the minimum collateralization rate.

3.2.5 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.

3.2.6 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).

3.2.7 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).

3.2.8 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.

3.2.9 Security

The Security module is the kernel of the BTC Parachain. It is imported by most modules to ensure that the chain is running.

3.2.10 Governance Mechanism

The Governance Mechanism handles correct operation of the BTC Parachain.

3.3 Interactions

3.3.1 Dependency Graph

We provide a dependency graph of the different pallets in Fig. 3.4. Note that for clarity, dependencies that are already implied by transitivity are not displayed. That is, if `a -> b`, `b -> c` and `a -> b`, we do not show a dependency `a -> c` even when it is an explicit dependency in the implementation.

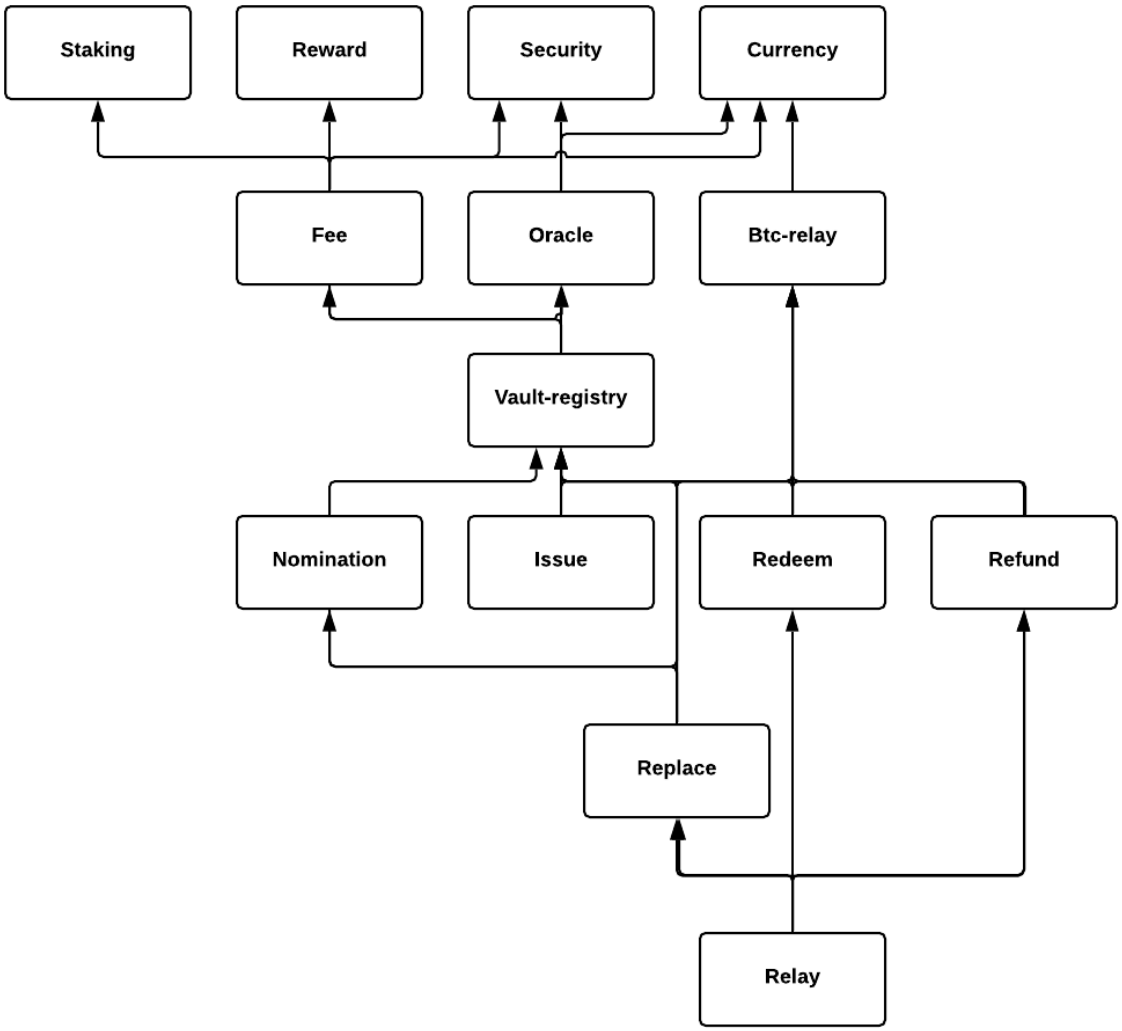


Fig. 3.4: Pallet dependency graph

3.3.2 External Interactions

We provide an overview in Fig. 3.5 of the main ways that different actors interact with the parachain. Note that we only include the function calls that have side effects, i.e., that write to storage. Also, some calls that are not central to the main protocol are omitted to keep the overview clear. The pallets are displayed in the center column, while the various actors surround it in yellow.

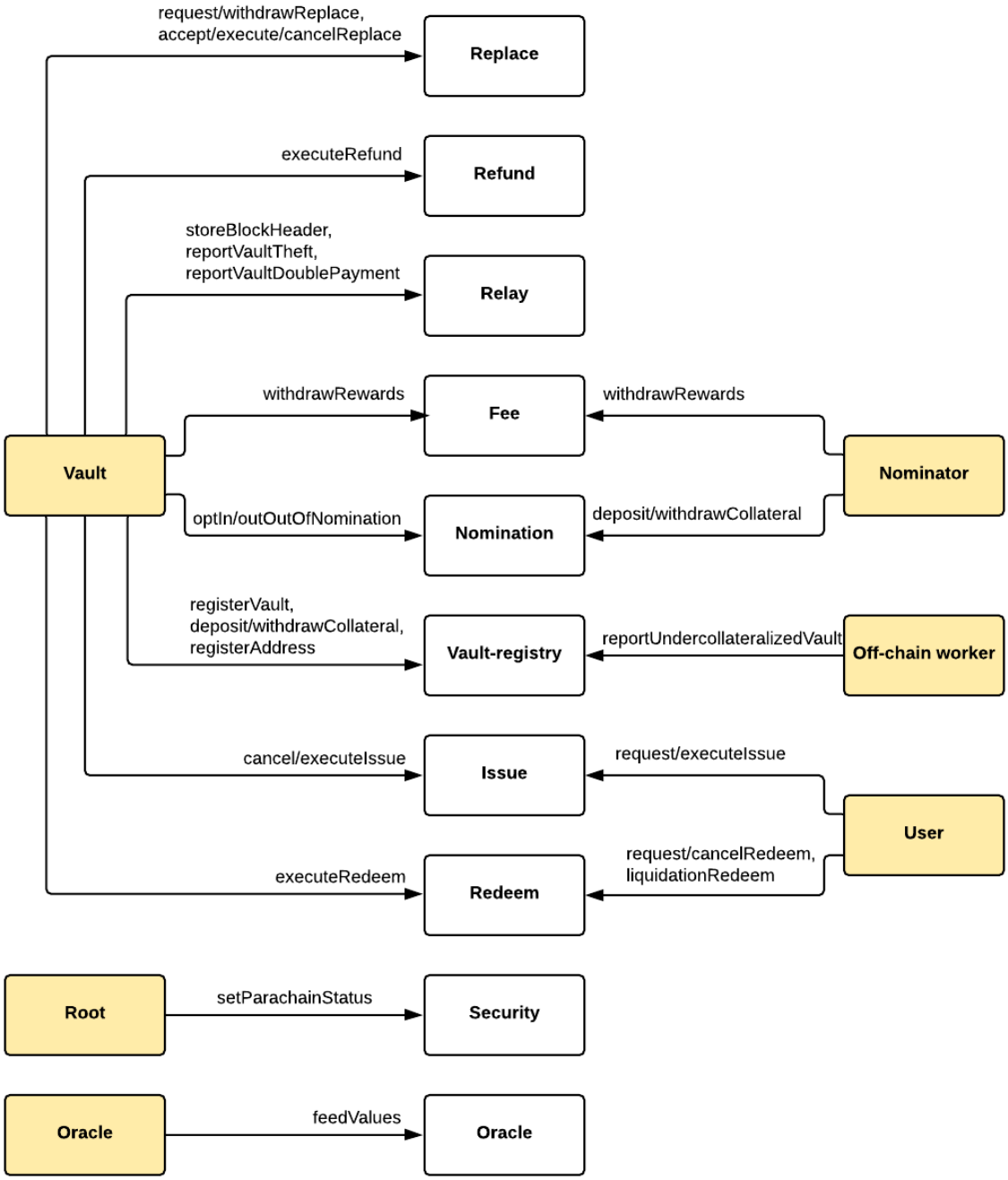


Fig. 3.5: Overview of interactions of different actors with the parachain.

CHAPTER

FOUR

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.

4.1 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.

4.2 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.

- **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, and (3) emit events and return. *Note*: It is now possible to add a `transactional` attribute to function that ensure that state is only updated if the function or any of its called functions are not resulting in an error. This prevents partial state update and must be used on all external functions.
- **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

FIVE

BITCOIN DATA MODEL

This is a high-level overview of Bitcoin's data model. For the full details, refer to https://bitcoin.org/en/developer-reference. While the serialized versions of these structs are used in the bridge's API, they are parsed by the chain into a more convenient internal representation. See *Data Model*.

5.1 Block Headers

The 80 bytes block header encodes the following information:

Note: as per bip64, blocks with a version number of less than 4 are rejected. As a consequence, blocks that were mined before December 2015 will not successfully parse in the bridge. This is acceptable, because the bridge is not expected to be initialized with such an old block as genesis.

Bytes	Parameter	Type	Description
4	`version`	i32	The block version to follow.
32	`hashPrevBlock`	char[32]	The double sha256 hash of the previous block header.
32	`merkleRoot`	char[32]	The double sha256 hash of the Merkle root of all transaction hashes in this block.
4	`timestamp`	u32	The block timestamp included by the miner.
4	`nBits`	u32	The target difficulty threshold, see also the Bitcoin documentation.
4	`nonce`	u32	The nonce chosen by the miner to meet the target difficulty threshold.

5.2 Transactions

A transaction is broadcasted in a serialized byte format (also called raw format). It consists of a variable size of bytes and has the following format. Both 'normal' transaction and transactions segregated witness data are supported.

Bytes	Parameter	Type	Description
4	`version`	i32	Transaction version number.
0/2	`flags`	Option<u8[2]>	If present, always 0001, and indicates the presence of witness data
var	`tx_in count`	uint	Number of transaction inputs.
var	`tx_in`	*Inputs*	List of transaction inputs.
var	`tx_out count`	uint	The number of transaction outputs.
var	`tx_out`	*Outputs*	List of transaction outputs.
var	`tx_witnesses`	*Witness*	A list of witnesses, one for each input; omitted if flag is omitted above.
4	`lock_time`	u32	A Unix timestamp OR block number.

Note: Bitcoin uses the term "CompactSize Unsigned Integers" to refer to variable-length integers, which are used to indicate the number of bytes representing transaction inputs and outputs. See the Developer Reference for more details.

5.3 Inputs

Bitcoin's UTXO model requires a new transaction to spend at least one existing and unspent transaction output as a transaction input. The `txIn` type consists of the following bytes. See the reference for further details.

Bytes	Parameter	Type	Description
36	`previous_output`	outpoint	The output to be spent consisting of the transaction hash (32 bytes) and the output index (4 bytes).
var	`script bytes`	uint	Number of bytes in the signature script (max 10,000 bytes).
var	`signature script`	char[]	The script satisfying the output's script.
4	`sequence`	u32	Sequence number (default `0xffffffff`).

5.4 Outputs

The transaction output has the following format according to the reference.

Bytes	Parameter	Type	Description
8	`value`	i64	Number of satoshis to be spent.
1+	`pk_script bytes`	uint	Number of bytes in the script.
var	`pk_script`	char[]	Spending condition as script.

5.5 Witness

Bytes	Parameter	Type	Description
var	`count`	uint	The number of witness stack items in this tx_witness.
var	`witness_stack`	*Witness Stack Item*	List of witness stack items making up this tx_witness.

5.6 Witness Stack Item

Bytes	Parameter	Type	Description
var	`count`	uint	The number of bytes in this witness stack item.
var	`witness_stack`	u8[]	The bytes making up the witness stack item.

CHAPTER

SIX

ACCEPTED BITCOIN TRANSACTION FORMAT

The *Functions: Parser* module of BTC-Relay can in theory be used to parse arbitrary Bitcoin transactions. However, the interBTC component of the BTC Parachain restricts the format of Bitcoin transactions to ensure consistency and prevent protocol failure due to parsing errors.

As such, Bitcoin transactions for which transaction inclusion proofs are submitted to BTC-Relay as part of the in the interBTC *Issue*, *Redeem*, and *Replace* protocols must be P2PKH or P2WPKH transactions and follow the format below.

6.1 Case 1: OP_RETURN Transactions

The OP_RETURN field can be used to store 40 bytes in a given Bitcoin transaction. The transaction output that includes the OP_RETURN is provably unspendable. We require specific information in the OP_RETURN field to prevent replay attacks in interBTC.

Many Bitcoin wallets automatically order UTXOs. We require that the transaction contains at most three outputs, but we do not require any specific ordering of the *Payment UTXO* and *Data UTXO*. The reason behind checking for the first three outputs is that wallets like Electrum might insert the UTXOs returning part of the spent input at index 1. A merge transaction is allowed to contain any number of outputs, as long as all outputs are registered in the vault's wallet.

Note: Please refer to the interBTC specification for more details on the *Refund*, *Redeem* and *Replace* protocols.

Inputs	Outputs
Arbitrary number of inputs	**Index 0 to 2**: *Payment UTXO*: P2PKH / P2WPKH output to `btcAddress` Bitcoin address. *Data UTXO*: OP_RETURN containing `identifier` **Index 3-31**: Any other UTXOs that will not be considered.

The value and recipient address (`btcAddress`) of the *Payment UTXO* and the `identifier` in the *Data UTXO* (OP_RETURN) depend on the executed interBTC protocol:

- In *Refund* `btcAddress` is the Bitcoin address of the user for the refunding process and `identifier` is the `refundId` of the `RefundRequest` in `RefundRequests`.
- In *Redeem* `btcAddress` is the Bitcoin address of the user who triggered the redeem process and `identifier` is the `redeemId` of the `RedeemRequest` in `RedeemRequests`.
- In *Replace* `btcAddress` is the Bitcoin address of the new vault, which has agreed to replace the vault which triggered the replace protocol and `identifier` is the `replaceId` of the `ReplaceRequest` in `ReplaceRequests`.

6.2 Case 2: Regular P2PKH / P2WPKH / P2SH / P2WSH Transactions

We accept regular P2PKH, P2WPKH, P2SH, and P2WSH transactions. We ensure that the recipient address is unique via the On-Chain Key Derivation Scheme.

Many Bitcoin wallets automatically order UTXOs. We require that the *Payment UTXO* is included within the first three indexes (index 0 - 2). We *do not* require any specific ordering of those outputs. The reason behind checking for the first three outputs is that wallets like Electrum might insert the UTXOs returning part of the spent input at index 1.

Note: Please refer to the interBTC specification for more details on the *Issue* protocol.

Inputs	Outputs
Arbitrary number of inputs	**Index 0 to 2**: *Payment UTXO*: Output to `btcAddress` Bitcoin address. **Index 3-31**: Any other UTXOs that will not be considered.

The recipient address (`btcAddress`) of the *Payment UTXO* is a address derived from the public key the vault submitted to the BTC-Parachain.

CHAPTER

SEVEN

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.

7.1 External Functions

Public functions called by users of the platform - most calls are assumed to be signed by an account able to pay the transaction fees.

7.2 Internal Functions

Private functions called by block construction hooks or other external functions.

7.3 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 external (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.

7.4 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

EIGHT

BTC-RELAY

8.1 Overview

Below, we provide an overview of the BTC-Relay components - offering references to the full specification contained in the rest of this document.

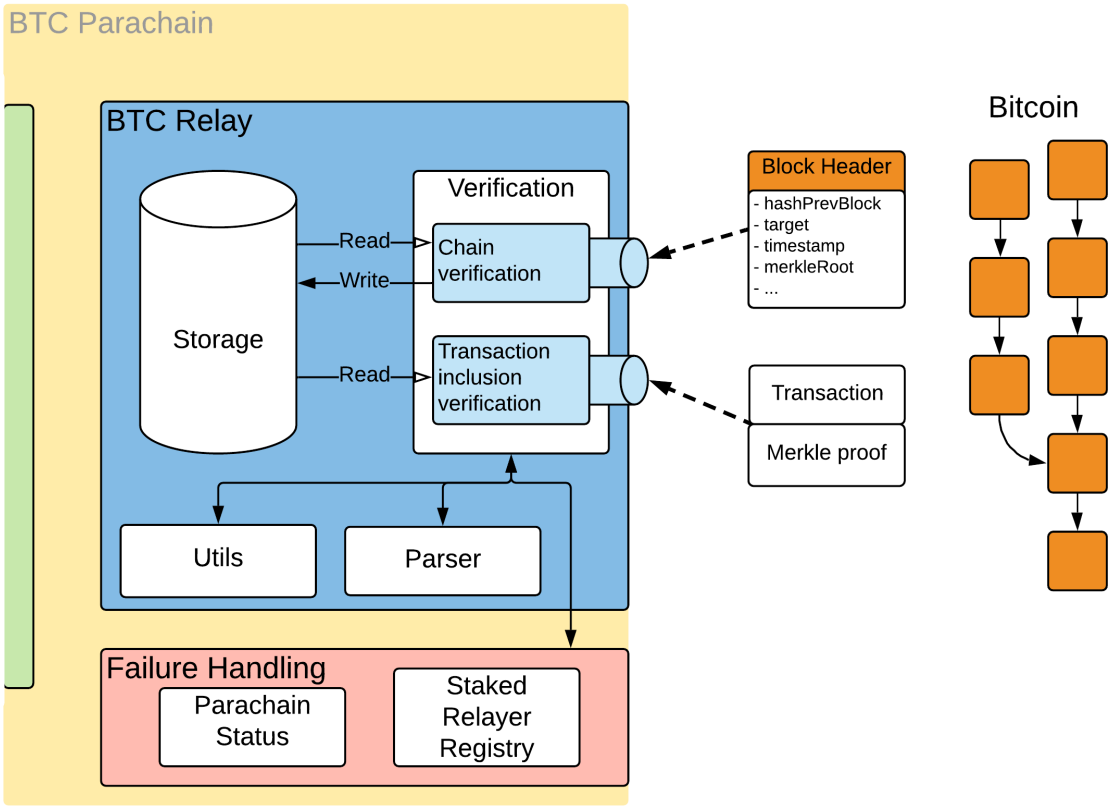


Fig. 8.1: Overview of the BTC-Relay architecture. Bitcoin block headers are submitted to the Verification Component, which interacts with the Utils, Parser and Failure Handling components, as well as the Parachain Storage.

8.1.1 Storage

This component stores the Bitcoin block headers and additional data structures, necessary for operating BTC-Relay. See *Data Model* for more details.

8.1.2 Verification

The Verification component offers functionality to verify Bitcoin block headers and transaction inclusion proofs. See *Functions: Storage and Verification* for the full function specification.

In more detail, the verification component performs the operations of a Bitcoin SPV client. See this paper (Appendix D) for a more detailed and formal discussion on the necessary functionality.

- *Difficulty Adjustment* - check and keep track of Bitcoin's difficulty adjustment mechanism, so as to be able to determine when the PoW difficulty target needs to be recomputed.
- *PoW Verification* - check that, given a 80 byte Bitcoin block header and its block hash, (i) the block header is indeed the pre-image to the hash and (ii) the PoW hash matches the difficulty target specified in the block header.
- *Chain Verification* - check that the block header references an existing block already stored in BTC-Relay.
- *Main Chain Detection / Fork Handling* - when given two conflicting Bitcoin chains, determine the *main chain*, i.e., the chain with the most accumulated PoW (longest chain in Bitcoin, though under consideration of the difficulty adjustment mechanism).
- *Transaction Inclusion Verification* - given a transaction, a reference to a block header, the transaction's index in that block and a Merkle tree path, determine whether the transaction is indeed included in the specified block header (which in turn must be already verified and stored in the Bitcoin main chain tracked by BTC-Relay).

An overview and explanation of the different classes of blockchain state verification in the context of cross-chain communication, specifically the difference between full validation of transactions and mere verification of their inclusion in the underlying blockchain, can be found in this paper (Section 5).

8.1.3 Utils

The Utils component provides "helper" functions used by the Storage and Verification components, such as the calculation of Bitcoin's double SHA256 hash, or re-construction of Merkle trees. See *Functions: Utils* for the full function specification.

8.1.4 Parser

The Parser component offers functions to parse Bitcoin's block and transaction data structures, e.g. extracting the Merkle tree root from a block header or the OP_RETURN field from a transaction output. See *Functions: Parser* for the full function specification.

8.2 Specification

8.2.1 Data Model

The BTC-Relay, as opposed to Bitcoin SPV clients, only stores a subset of information contained in block headers and does not store transactions. Specifically, only data that is absolutely necessary to perform correct verification of block headers and transaction inclusion is stored.

Note that the structs used to represent bitcoin transactions and blocks is slightly different from the *Bitcoin Data Model*. For example, no `tx_in count` is required, since this information is implicitly stored in the vector of inputs.

Types

RawBlockHeader

An 80 bytes long Bitcoin blockchain header, according to the format as specified by the Bitcoin reference.

Constants

DIFFICULTY_ADJUSTMENT_INTERVAL

The interval in number of blocks at which Bitcoin adjusts its difficulty (approx. every 2 weeks = 2016 blocks).

TARGET_TIMESPAN

Expected duration of the different adjustment interval in seconds, `1209600` seconds (two weeks) in the case of Bitcoin.

TARGET_TIMESPAN_DIVISOR

Auxiliary constant used in Bitcoin's difficulty re-target mechanism.

UNROUNDED_MAX_TARGET

The maximum difficulty target, $2^{224} - 1$ in the case of Bitcoin. For more information, see the Bitcoin Wiki.

MAIN_CHAIN_ID

Identifier of the Bitcoin main chain tracked in the `ChainsIndex` mapping. At any point in time, the `BlockChain` with this identifier is considered to be the main chain and will be used to transaction inclusion verification.

STABLE_BITCOIN_CONFIRMATIONS

Global security parameter (typically referred to as `k` in scientific literature), determining the umber of confirmations (in blocks) necessary for a transaction to be considered "stable" in Bitcoin. Stable thereby means that the probability of the transaction being excluded from the blockchain due to a fork is negligible.

STABLE_PARACHAIN_CONFIRMATIONS

Global security parameter (typically referred to as `k` in scientific literature), determining the umber of confirmations (in blocks) necessary for a transaction to be considered "stable" in the BTC Parachain. Stable thereby means that the probability of the transaction being excluded from the blockchain due to a fork is negligible.

Note: We use this to enforce a minimum delay on Bitcoin block header acceptance in the BTC-Parachain in cases where a (large) number of block headers are submitted as a batch.

Structs

BlockHeader

Representation of a Bitcoin block header, constructed by the parachain from the *RawBlockHeader*. The main differences compared to the *Block Headers* in *Bitcoin Data Model* is that this contains the unpacked `target` constructed from `nBits`, and an additional `hash` of the `BlockHeader` for convenience.

Note: Fields marked as [Optional] are not critical for the secure operation of BTC-Relay, but can be stored anyway, at the developers discretion. We omit these fields in the rest of this specification.

Parameter	Type	Description
`merkleRoot`	H256Le	Root of the Merkle tree referencing transactions included in the block.
`target`	u256	Difficulty target of this block (converted from `nBits`, see Bitcoin documentation.).
`timestamp`	timestamp	UNIX timestamp indicating when this block was mined in Bitcoin.
`hashPrevBlock`	H256Le	Block hash of the predecessor of this block.
`hash`	H256Le	Block hash of of this block.
.	.	.
`version`	i32	[Optional] Version of the submitted block.
`nonce`	u32	[Optional] Nonce used to solve the PoW of this block.

RichBlockHeader

Representation of a Bitcoin block header containing additional metadata. This struct is used to store Bitcoin block headers.

Parameter	Type	Description
`blockHeight`	u32	Height of this block in the Bitcoin main chain.
`chainRef`	u32	Pointer to the `BlockChain` struct in which this block header is contained.
`blockHeader`	BlockHeader	Associated parsed `BlockHeader` struct.
`paraHeight`	u32	The `activeBlockCount` at the time the block header was submitted to the relay. See the security pallet for more information.

BlockChain

Representation of a Bitcoin blockchain / fork.

Parameter	Type	Description
`chainId`	u32	Unique identifier for faster lookup in `ChainsIndex`
`startHeight`	u32	Lowest block number in this chain. Used to determine the forking point during chain reorganizations.
`maxHeight`	u32	Max. block height in this chain.

Transaction

Representation of a Bitcoin Transaction. It differs from the one specified in *Bitcoin Data Model* in that it does not contain in lengths of the input and output vectors, because this data is implicit in the vector. Furthermore, we use different types for the inputs and outputs. The segregated witnesses and `flags`, if any, are placed inside the inputs.

Parameter	Type	Description
`version`	i32	Transaction version number.
`inputs`	Vec<*TransactionInput*>	Vector of transaction inputs.
`output`	Vec<*TransactionOutput*>	Vector of transaction inputs.
`lockTime`	*LockTime*	A Unix timestamp OR block number.

TransactionInput

Representation of a Bitcoin transaction input. It differs from the one specified in *Bitcoin Data Model* in that it contains `flags` and the segregated witnesses. Furthermore, it contains dedicated fields for coinbase transactions.

Parameter	Type	Description
`previousHash`	H256Le,	The hash of the transaction to spend from.
`previousIndex`	u32,	The index of the output within the transaction pointed to by `previousHash` to spend from.
`coinbase`	bool,	True if the transaction input is the newly mined funds.
`height`	Option<u32>,	An optional blockheight used in the coinbase transaction. See https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki
`script`	Vec<u8>,	The script satisfying the output's script.
`sequence`	u32,	Sequence number (default `0xffffffff`).
`flags`	u8,	The flags set in `Transaction` that indicates a Segrated Witness transaction. If none were set in the transaction, this value is 0.
`witness`	Vec<Vec<u8>>,	The witness scripts of the transaction. See See https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

TransactionOutput

Representation if a Bitcoin transaction output

Parameter	Type	Description
`value`	i64,	The number of satoshis to transfer to this output.
`script`	*Script*	The spending condition of the output.

Script

Representation if a Bitcoin transaction output

Parameter	Type	Description
`bytes`	Vec<u8>,	The spending condition of the output.

Enums

LockTime

Represents either a unix timestamp OR a blocknumber. See the Bitcoin source.

Discriminant	Description
`Time(u32)`	Lock time interpreted as a unix timestamp.
`BlockHeight(u32)`	Lock time interpreted as a block number.

Data Structures

BlockHeaders

Mapping of <`blockHash, RichBlockHeader`>, storing all verified Bitcoin block headers (fork and main chain) submitted to BTC-Relay.

Chains

Level of indirection over *ChainsIndex*, i.e. the values stored in this map are keys of `ChainsIndex`. `Chains[0]` MUST always be `0`, such that `ChainsIndex[Chains[0]]` is the bitcoin *main chain*. The remaining items MUST sort the chains by height, i.e. it MUST hold that for each `0 < i < j`, `ChainsIndex[Chains[i]].maxHeight >= ChainsIndex[Chains[j]].maxHeight`. Furthermore, keys MUST be consecutive, i.e. for each `i`, if `Chains[i]` does not exist, `Chains[i+1]` MUST NOT exist either.

Note: The assumption for `Chains` is that, in the majority of cases, block headers will be appended to the *main chain* (longest chain), i.e., the `BlockChain` entry at the most significant position in the queue/heap. Similarly, transaction inclusion proofs (*verifyTransactionInclusion*) are only checked against the *main chain*. This means, in the average case lookup complexity will be O(1). Furthermore, block headers can only be appended if they (i) have a valid PoW and (ii) do not yet exist in `BlockHeaders` - hence, spamming is very costly and unlikely. Finally, blockchain forks and re-organizations occur infrequently, especially in Bitcoin. In principle, optimizing lookup costs should be prioritized, ideally O(1), while inserting of new items and re-balancing can even be O(n).

ChainsIndex

The main storage map of `BlockChain` structs, indexed by a *values* from the *Chains*. `ChainsIndex[0]` MUST always contain the main chain.

BestBlock

32 byte Bitcoin block hash (double SHA256) identifying the current blockchain tip, i.e., the `RichBlockHeader` with the highest `blockHeight` in the `BlockChain` entry, which has the most significant `height` in the `Chains` priority queue (topmost position).

Note: Bitcoin uses SHA256 (32 bytes) for its block hashes, transaction identifiers and Merkle trees. In Substrate, we hence use `H256` to represent these hashes.

BestBlockHeight

Integer representing the maximum block height (`height`) in the `Chains` priority queue. This is also the `blockHeight` of the `RichBlockHeader` entry pointed to by `BestBlock`.

ChainCounter

Integer increment-only counter used to track existing BlockChain entries. Initialized with 1 (0 is reserved for `MAIN_CHAIN_ID`).

8.2.2 Functions: Storage and Verification

initialize

Initializes BTC-Relay with the first Bitcoin block to be tracked and initializes all data structures (see *Data Model*).

Note: BTC-Relay **does not** have to be initialized with Bitcoin's genesis block! The first block to be tracked can be selected freely.

Warning: Caution when setting the first block in BTC-Relay: only succeeding blocks can be submitted and **predecessors and blocks from other chains will be rejected**! Similarly, caution is required with the initial block height argument, since if an incorrect value is used, all subsequently reported block heights will be incorrect.

Specification

Function Signature

`initialize(relayer, rawBlockHeader, blockHeight)`

Parameters

- `relayer`: the account submitting the block
- `rawBlockHeader`: 80 byte raw Bitcoin block header, see *RawBlockHeader*.
- `blockHeight`: integer Bitcoin block height of the submitted block header

Events

- `Initialized(blockHeight, blockHash, relayer)`: if the first block header was stored successfully, emit an event with the stored block's height (`blockHeight`) and the (PoW) block hash (`blockHash`).

Errors

- `ERR_ALREADY_INITIALIZED = "Already initialized"`: return error if this function is called after BTC-Relay has already been initialized.

Preconditions

- This is the first time this function is called, i.e., when BTC-Relay is being deployed.
- The blockheader MUST be parsable.
- `blockHeight` MUST match the height on the bitcoin chain. Note that the parachain can not check this - it's the caller's responsability!
- `rawBlockHeader` MUST match a block on the bitcoin main chain. Note that the parachain can not check this - it's the caller's responsability!

- `rawBlockHeader` MUST be a block mined after December 2015 - see *Block Headers*. This is NOT checked by the parachain - it's the caller's responsibility!

Postconditions

Let `blockHeader` be the parsed `rawBlockHeader`. Then:

- `ChainsIndex[0]` MUST be set to a new `BlockChain` value, where `BlockChain.chainId = 0` and``BlockChain.startHeight = BlockChain.maxHeight = blockHeight``
- A value `block` of type `RichBlockHeader` MUST be added to `BlockHeaders`, where:
  - `block.basic_block_header = blockHeader`
  - `block.chainRef = 0`
  - `block.paraHeight` is the current activeBlockCount (see the Security module)
  - `block.blockHeight = blockHeight`
- `BestBlockHeight` MUST be `ChainsIndex[0].maxHeight`
- `BestBlock` MUST be `blockHeader.hash`
- `StartBlockHeight` MUST be set to `blockHeight`

storeBlockHeader

Method to submit block headers to the BTC-Relay. This function calls *verifyBlockHeader* to check that the block header is valid. If so, from the block header and stores the hash, height and Merkle tree root of the given block header in `BlockHeaders`. If the block header extends an existing `BlockChain` entry in `Chains`, it appends the block hash to the `chains` mapping and increments the `maxHeight`. Otherwise, a new `Blockchain` entry is created.

Specification

Function Signature

`storeBlockHeader(relayer, rawBlockHeader)`

Parameters

- `relayer`: the account submitting the block
- `rawBlockHeader`: 80 byte raw Bitcoin block header, see *RawBlockHeader*.

Events

- `StoreMainChainHeader(blockHeight, blockHash, relayer)`: if the block header was successful appended to the currently longest chain (*main chain*) emit an event with the stored block's height (`blockHeight`) and the (PoW) block hash (`blockHash`).
- `StoreForkHeader(forkId, blockHeight, blockHash, relayer)`: if the block header was successful appended to a new or existing fork, emit an event with the block height (`blockHeight`) and the (PoW) block hash (`blockHash`).

Invariants

- The values in `Chains` MUST be such that for each `0 < i < j`, `ChainsIndex[Chains[i]].maxHeight >= ChainsIndex[Chains[j]].maxHeight`.
- The keys in `Chains` MUST be consecutive, i.e. for each `i`, if `Chains[i]` does not exist, `Chains[i+1]` MUST NOT exist either.
- The keys in `ChainsIndex` MUST be consecutive, i.e. for each `i`, if `ChainsIndex[i]` does not exist, `ChainsIndex[i+1]` MUST NOT exist either.
- For all `i > 0` the following MUST hold: *ChainsIndex[i].maxHeight* < *ChainsIndex[0].maxHeight* + *STABLE_BITCOIN_CONFIRMATIONS*.

- For all `i`, the following MUST hold: `ChainsIndex[i].chainRef == i`.
- `BestBlock.chainRef` MUST be 0
- `BestBlock.blockHeight` MUST be `ChainsIndex[0].maxHeight`
- `BestBlockHeight` MUST be `ChainsIndex[0].maxHeight`

Preconditions

- The BTC Parachain status MUST NOT be set to `SHUTDOWN: 3`.
- The given `rawBlockHeader` MUST parse be parsable into `blockHeader`.
- There MUST be a block header `prevHeader` stored in `BlockHeaders` with a hash equal to `blockHeader.hashPrevBlock`.
- A block chain `prevBlockchain` MUST be stored in `ChainsIndex[prevHeader.chainRef]`.
- *verifyBlockHeader* MUST return `Ok` when called with `blockHeader`, `prevHeader.blockHeight + 1` and `prevHeader`.
- **If `prevHeader` is the last element a chain (i.e. `blockHeader` does not create a new fork), then:**
  - `prevBlockChain` MUST NOT already contain a block of height `prevHeader.blockHeight + 1`.
  - If `prevBlockChain.chain_id` is _not_ zero (i.e. the block is being added to a fork rather than the main chain), and the fork is `STABLE_BITCOIN_CONFIRMATIONS` blocks ahead of the main chain, then calling *swapMainBlockchain* with this fork MUST return `Ok`.

Postconditions

- If `prevHeader` is the last element a chain (i.e. `blockHeader` does not create a new fork), then:
  - `ChainsHashes[prevBlockChain.chain_id, prevHeader.blockHeight + 1]` MUST be set to `blockHeader.hash`.
  - `ChainsIndex[prevBlockChain.chain_id].max_height` MUST be increased by 1.
  - If `prevBlockChain.chain_id` is zero (i.e. the a block is being added to the main chain), then:
    - `BestBlock` MUST be set to `blockHeader.hash`
    - `BestBlockHeight` MUST be set to `prevHeader.blockHeight + 1`
  - If `prevBlockChain.chain_id` is _not_ zero (i.e. the block is being added to a fork rather than the main chain), then:
    - If the fork is `STABLE_BITCOIN_CONFIRMATIONS` blocks ahead of the main chain, i.e. `prevHeader.blockHeight + 1 >= BestBlockHeight + STABLE_BITCOIN_CONFIRMATIONS`, then the fork is moved to the mainchain. That is, *swapMainBlockchain* MUST be called with the fork as argument.
  - A new `RichBlockHeader` MUST be stored in `BlockHeaders` that is constructed as follows:
    - `RichBlockHeader.blockHeader = blockHeader`,
    - `RichBlockHeader.blockHeight = prevBlock.blockHeight + 1`,
    - `RichBlockHeader.chainRef = prevBlockChain.chainId`,
    - `RichBlockHeader.paraHeight` is set to the current active block count - see the security module for details
- If `prevHeader` is *not* the last element a chain (i.e. `blockHeader` creates a *new* fork), then:
  - `ChainCounter` MUST be incremented. Let `newChainCounter` be the incremented value, then
  - `ChainsHashes[newChainCounter, prevHeader.blockHeight + 1]` MUST be set to `blockHeader.hash`.

  - A new blockchain MUST be inserted into `ChainsIndex`. Let `newChain` be the newly inserted chain. Then `newChain` MUST have the following values:
    * `newChain.chainId = newChainCounter`,
    * `newChain.startHeight = prevHeader.blockHeight + 1`,
    * `newChain.maxHeight = prevHeader.blockHeight + 1`,
  - A new value MUST be added to `Chains` that is equal to `newChainCounter` in a way that maintains the invariants specified above.
  - A new `RichBlockHeader` MUST be stored in `BlockHeaders` that is constructed as follows:
    * `RichBlockHeader.blockHeader = blockHeader`,
    * `RichBlockHeader.blockHeight = newChain.blockHeight + 1`,
    * `RichBlockHeader.chainRef = prevBlockChain.chainId`,
    * `RichBlockHeader.paraHeight` is set to the current active block count - see the security module for details

- `BestBlockHeight` MUST be set to `Chains[0].max_height`
- `BestBlock` MUST be set to `ChainsHashes[0, Chains[0].max_height`

Warning: The BTC-Relay does not necessarily have the same view of the Bitcoin blockchain as the user's local Bitcoin client. This can happen if (i) the BTC-Relay is under attack, (ii) the BTC-Relay is out of sync, or, similarly, (iii) if the user's local Bitcoin client is under attack or out of sync (see *Security*).

Note: The 80 bytes block header can be retrieved from the bitcoin-rpc client by calling the getBlock and setting verbosity to `0` (`getBlock <blockHash> 0`).

swapMainBlockchain

Specification

Function Signature

`swapMainBlockchain(fork)`

Parameters

- `fork`: pointer to a `BlockChain` entry in `Chains`.

Preconditions

- `fork` is `STABLE_BITCOIN_CONFIRMATIONS` blocks ahead of the main chain, i.e. `fork.maxHeight >= BestBlockHeight + STABLE_BITCOIN_CONFIRMATIONS`

Postconditions

Let `lastBlock` be the last rich block header in `fork`, i.e. the blockheader for which `lastBlock.blockHeight == fork.maxHeight` and `lastBlock.chainRef == fork.chainId` holds. Then:

- Each ancestor `a` of `lastBlock` MUST move to the main chain, i.e. `a.chainRef` MUST be set to `MAIN_CHAIN_ID`.
- `ChainsIndex[MAIN_CHAIN_ID].maxHeight` MUST be set to `lastBlock.blockHeight`.
- Each fork `fork` except the main chain that contains an ancestor of `lastBlock` MUST set `fork.startHeight` to the lowest `blockHeight` in the fork that is not an ancestor of `lastBlock`.

- Each block `b` in the mainchain that is not an acestor of `lastBlock` MUST move to `prevBlockChain`, i.e. `b.chainRef = prevBlockChain.chainId`.
- `prevBlockChain.startHeight` MUST be set to the lowest `blockHeight` of all blocks `b` that have `b.chainRef == prevBlockChain.chainId`.
- `prevBlockChain.maxHeight` MUST be set to the highest `blockHeight` of all blocks `b` that have `b.chainRef == prevBlockChain.chainId`.

The figure below ilustrates an example execution of this function.

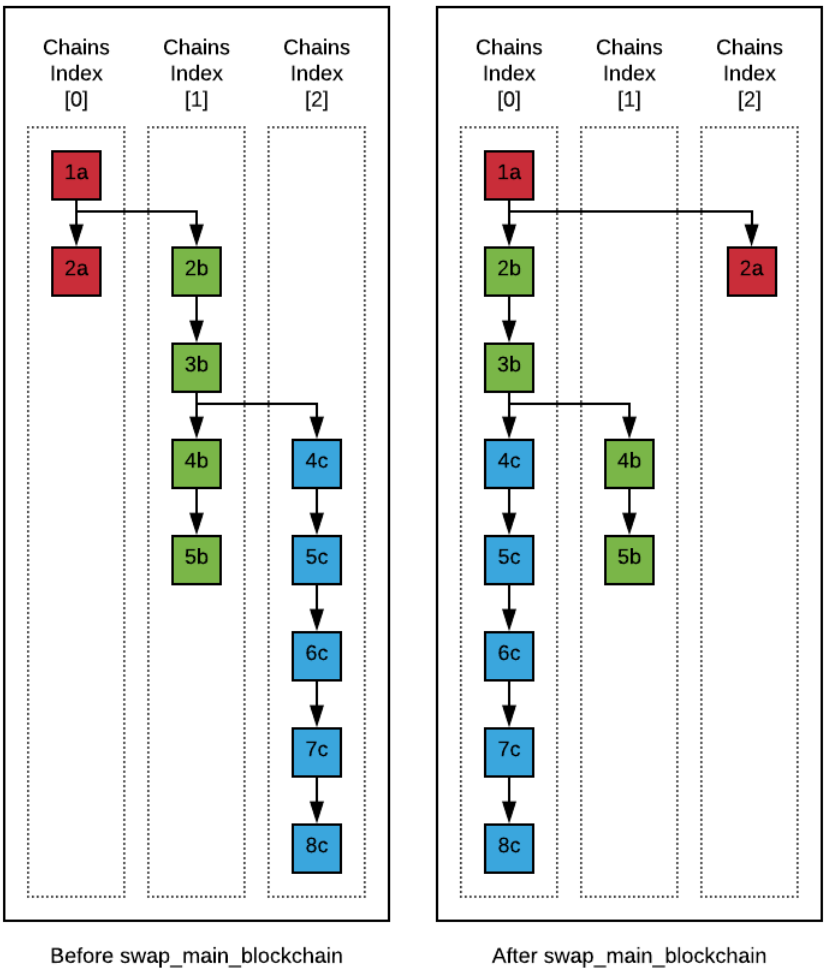


Fig. 8.2: On the left is an example of the state of `ChainsIndex` prior to calling `swapMainBlockchain`, and on the right is the corresponding state after the function returns.

In contrast the the figure about, when looking up the chains through the `Chains` map, the chains are sorted by `maxHeight`, and the same execution would look as follows:

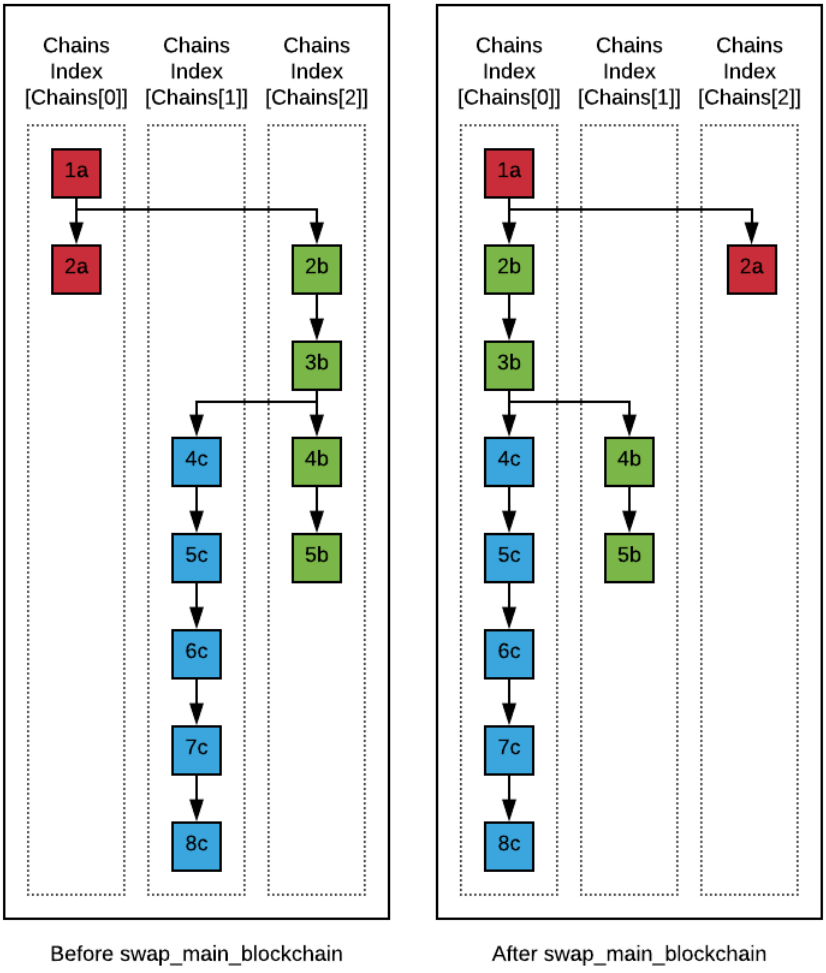


Fig. 8.3: On the left is an example of the state of `Chains` prior to calling `swapMainBlockchain`, and on the right is the corresponding state after the function returns.

verifyBlockHeader

The `verifyBlockHeader` function verifies Bitcoin block headers. It returns `Ok` if the blockheader is valid, otherwise an error.

Note: This function does not check whether the submitted block header extends the main chain or a fork. This check is performed in *storeBlockHeader*.

Specification

Function Signature

`verifyBlockHeader(blockHeader, blockHeight, prevBlockHeader)`

Parameters

- `blockHeader`: the *BlockHeader* to check.
- `blockHeight`: height of the block.
- `prevBlockHeader`: the *RichBlockHeader* that is the block header's predecessor.

Returns

- `Ok(())` if all checks pass successfully, otherwise an error.

Errors

- `ERR_DUPLICATE_BLOCK = "Block already stored"`: return error if the submitted block header is already stored in BTC-Relay (duplicate PoW `blockHash`).
- `ERR_LOW_DIFF = "PoW hash does not meet difficulty target of header"`: return error when the header's `blockHash` does not meet the `target` specified in the block header.
- `ERR_DIFF_TARGET_HEADER = "Incorrect difficulty target specified in block header"`: return error if the `target` specified in the block header is incorrect for its block height (difficulty re-target not executed).

Preconditions

- A block with the `blockHeader.hash` MUST NOT already have been stored.
- `blockHeader.hash` MUST be be below `BlockHeader.target`
- `blockHeader.target` MUST match the expected target, which is calculated based on previous targets and timestamps. See the Bitcoin Wiki for more information.

Postconditions

- `Ok(())` MUST be returned.

verifyTransactionInclusion

The `verifyTransactionInclusion` function is one of the core components of the BTC-Relay: this function checks if a given transaction was indeed included in a given block (as stored in `BlockHeaders` and tracked by `Chains`), by reconstructing the Merkle tree root (given a Merkle proof). Also checks if sufficient confirmations have passed since the inclusion of the transaction (considering the current state of the BTC-Relay `Chains`).

Specification

Function Signature

`verifyTransactionInclusion(txId, merkleProof, confirmations, insecure)`

Parameters

- `txId`: 32 byte hash identifier of the transaction.
- `merkleProof`: Merkle tree path (concatenated LE sha256 hashes, dynamic sized).
- `confirmations`: integer number of confirmation required.

Note: The Merkle proof for a Bitcoin transaction can be retrieved using the `bitcoin-rpc` gettxoutproof method and dropping the first 170 characters. The Merkle proof thereby consists of a list of SHA256 hashes, as well as an indicator in which order the hash concatenation is to be applied (left or right).

Returns

- `True`: if the given `txId` appears in at the position specified by `txIndex` in the transaction Merkle tree of the block at height `blockHeight` and sufficient confirmations have passed since inclusion.
- Error otherwise.

Events

- `VerifyTransaction(txId, txBlockHeight, confirmations)`: if verification was successful, emit an event specifying the `txId`, the `blockHeight` and the requested number of `confirmations`.

Errors

- `ERR_SHUTDOWN = "BTC Parachain has shut down"`: the BTC Parachain has been shutdown by a manual intervention of the Governance Mechanism.
- `ERR_MALFORMED_TXID = "Malformed transaction identifier"`: return error if the transaction identifier (`txId`) is malformed.
- `ERR_CONFIRMATIONS = "Transaction has less confirmations than requested"`: return error if the block in which the transaction specified by `txId` was included has less confirmations than requested.
- `ERR_INVALID_MERKLE_PROOF = "Invalid Merkle Proof"`: return error if the Merkle proof is malformed or fails verification (does not hash to Merkle root).
- `ERR_ONGOING_FORK = "Verification disabled due to ongoing fork"`: return error if the `mainChain` is not at least `STABLE_BITCOIN_CONFIRMATIONS` ahead of the next best fork.

Preconditions

- The BTC Parachain status must not be set to `SHUTDOWN: 3`. If `SHUTDOWN` is set, all transaction verification is disabled.

Function Sequence

1. Check that `txId` is 32 bytes long. Return `ERR_MALFORMED_TXID` error if this check fails.
2. Check that the current `BestBlockHeight` exceeds `txBlockHeight` by the requested confirmations. Return `ERR_CONFIRMATIONS` if this check fails.

a. If `insecure == True`, check against user-defined `confirmations` only

b. If `insecure == True`, check against `max(confirmations, STABLE_BITCOIN_CONFIRMATIONS)`.

3. Check if the Bitcoin block was stored for a sufficient number of blocks (on the parachain) to ensure that staked relayers had the time to flag the block as potentially invalid. Check performed against `STABLE_PARACHAIN_CONFIRMATIONS`.
4. Extract the block header from `BlockHeaders` using the `blockHash` tracked in `Chains` at the passed `txBlockHeight`.
5. Check that the first 32 bytes of `merkleProof` are equal to the `txId` and the last 32 bytes are equal to the `merkleRoot` of the specified block header. Also check that the `merkleProof` size is either exactly 32 bytes, or is 64 bytes or more and a power of 2. Return `ERR_INVALID_MERKLE_PROOF` if one of these checks fails.
6. Call *computeMerkle* passing `txId`, `txIndex` and `merkleProof` as parameters.

a. If this call returns the `merkleRoot`, emit a `VerifyTransaction(txId, txBlockHeight, confirmations)` event and return `True`.

b. Otherwise return `ERR_INVALID_MERKLE_PROOF`.

validateTransaction

Given a raw Bitcoin transaction, this function

1) Parses and extracts
    a. the value and recipient address of the *Payment UTXO*,
    b. [Optionally] the OP_RETURN value of the *Data UTXO*.
2) Validates the extracted values against the function parameters.

Note: See *Bitcoin Data Model* for more details on the transaction structure, and *Accepted Bitcoin Transaction Format* for the transaction format of Bitcoin transactions validated in this function.

Specification

Function Signature

`validateTransaction(rawTx, paymentValue, recipientBtcAddress, opReturnId)`

Parameters

- `rawTx`: raw Bitcoin transaction including the transaction inputs and outputs.
- `paymentValue`: integer value of BTC sent in the (first) *Payment UTXO* of transaction.
- `recipientBtcAddress`: 20 byte Bitcoin address of recipient of the BTC in the (first) *Payment UTXO*.
- `opReturnId`: [Optional] 32 byte hash identifier expected in OP_RETURN (see *Replay Attacks*).

Returns

- `True`: if the transaction was successfully parsed and validation of the passed values was correct.
- Error otherwise.

Events

- `ValidateTransaction(txId, paymentValue, recipientBtcAddress, opReturnId)`: if parsing and validation was successful, emit an event specifying the `txId`, the `paymentValue`, the `recipientBtcAddress` and the `opReturnId`.

Errors

- `ERR_INSUFFICIENT_VALUE = "Value of payment below requested amount"`: return error the value of the (first) *Payment UTXO* is lower than `paymentValue`.

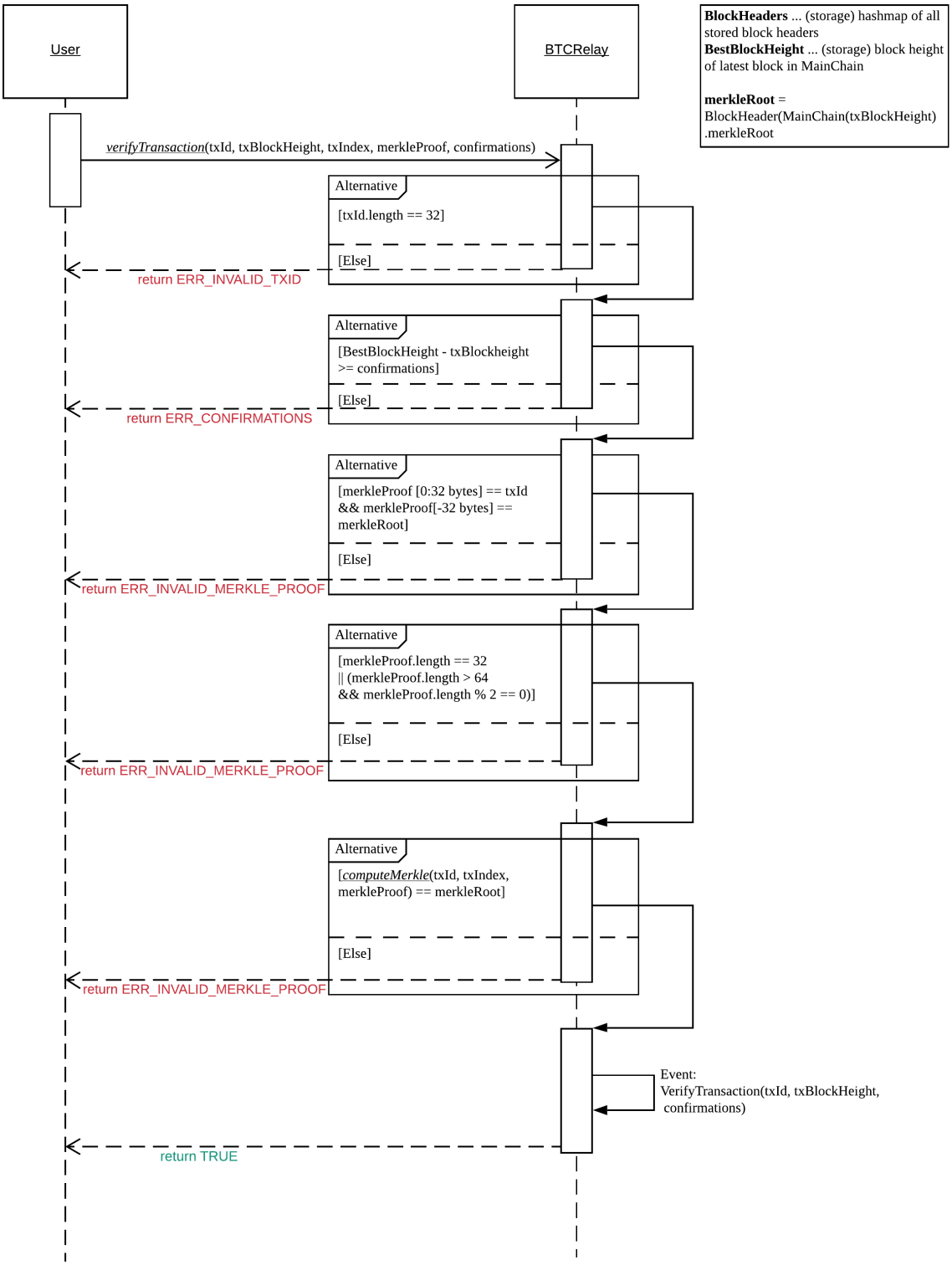


Fig. 8.4: The steps to verify a transaction in the *verifyTransactionInclusion* function.

- `ERR_TX_FORMAT = "Transaction has incorrect format"`: return error if the transaction has an incorrect format (see *Accepted Bitcoin Transaction Format*).
- `ERR_WRONG_RECIPIENT = "Incorrect recipient Bitcoin address"`: return error if the recipient specified in the (first) *Payment UTXO* does not match the given `recipientBtcAddress`.
- `ERR_INVALID_OPRETURN = "Incorrect identifier in OP_RETURN field"`: return error if the OP_RETURN field of the (second) *Data UTXO* does not match the given `opReturnId`.

Preconditions

- The BTC Parachain status must not be set to `SHUTDOWN: 3`. If `SHUTDOWN` is set, all transaction validation is disabled.

Function Sequence

See the raw Transaction Format section in the Bitcoin Developer Reference for a full specification of Bitcoin's transaction format (and how to extract inputs, outputs etc. from the raw transaction format).

1. Extract the `outputs` from `rawTx` using *extractOutputs*.

a. Check that the transaction (`rawTx`) has at least 2 outputs. One output (*Payment UTXO*) must be a P2PKH or P2WPKH output. Another output (*Data UTXO*) must be an OP_RETURN output. Raise `ERR_TX_FORMAT` if this check fails.

2. Extract the value of the *Payment UTXO* using *extractOutputValue* and check that it is equal (or greater) than `paymentValue`. Return `ERR_INSUFFICIENT_VALUE` if this check fails.
3. Extract the Bitcoin address specified as recipient in the *Payment UTXO* using *extractOutputAddress* and check that it matches `recipientBtcAddress`. Return `ERR_WRONG_RECIPIENT` if this check fails, or the error returned by *extractOutputAddress* (if the output was malformed).
4. Extract the OP_RETURN value from the *Data UTXO* using *extractOPRETURN* and check that it matches `opReturnId`. Return `ERR_INVALID_OPRETURN` error if this check fails, or the error returned by *extractOPRETURN* (if the output was malformed).

verifyAndValidateTransaction

The `verifyAndValidateTransaction` function is a wrapper around the *verifyTransactionInclusion* and the *validateTransaction* functions. It adds an additional check to verify that the validated transaction is the one included in the specified block.

Specification

Function Signature

`verifyAndValidateTransaction(merkleProof, confirmations, rawTx, paymentValue, recipientBtcAddress, opReturnId)`

Parameters

- `txId`: 32 byte hash identifier of the transaction.
- `merkleProof`: Merkle tree path (concatenated LE sha256 hashes, dynamic sized).
- `confirmations`: integer number of confirmation required.
- `rawTx`: raw Bitcoin transaction including the transaction inputs and outputs.
- `paymentValue`: integer value of BTC sent in the (first) *Payment UTXO* of transaction.
- `recipientBtcAddress`: 20 byte Bitcoin address of recipient of the BTC in the (first) *Payment UTXO*.

- `opReturnId`: [Optional] 32 byte hash identifier expected in OP_RETURN (see *Replay Attacks*).

Returns

- `True`: If the same transaction has been verified and validated.
- Error otherwise.

Function Sequence

1. Parse the `rawTx` to get the tx id.
2. Call *verifyTransactionInclusion* with the applicable parameters.
3. Call *validateTransaction* with the applicable parameters.

flagBlockError

Flags tracked Bitcoin block headers when Staked Relayers report and agree on a `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY` failure.

Attention: This function **does not** validate the Staked Relayers accusation. Instead, it is put up to a majority vote among all Staked Relayers in the form of a

Note: This function can only be called from the *Security* module of interBTC, after Staked Relayers have achieved a majority vote on a BTC Parachain status update indicating a BTC-Relay failure.

Specification

Function Signature

`flagBlockError(blockHash, errors)`

Parameters

- `blockHash`: SHA256 block hash of the block containing the error.
- `errors`: list of `ErrorCode` entries which are to be flagged for the block with the given blockHash. Can be "NO_DATA_BTC_RELAY" or "INVALID_BTC_RELAY".

Events

- `FlagBTCBlockError(blockHash, chainId, errors)` - emits an event indicating that a Bitcoin block hash (identified `blockHash`) in a `BlockChain` entry (`chainId`) was flagged with errors (`errors` list of `ErrorCode` entries).

Errors

- `ERR_UNKNOWN_ERRORCODE = "The reported error code is unknown"`: The reported `ErrorCode` can only be `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY`.
- `ERR_BLOCK_NOT_FOUND = "No Bitcoin block header found with the given block hash"`: No `RichBlockHeader` entry exists with the given block hash.
- `ERR_ALREADY_REPORTED = "This error has already been reported for the given block hash and is pending confirmation"`: The error reported for the given block hash is currently pending a vote by Staked Relayers.

Function Sequence

1. Check if `errors` contains `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY`. If neither match, return `ERR_UNKNOWN_ERRORCODE`.
2. Retrieve the `RichBlockHeader` entry from `BlockHeaders` using `blockHash`. Return `ERR_BLOCK_NOT_FOUND` if no block header can be found.
3. Retrieve the `BlockChain` entry for the given `RichBlockHeader` using `ChainsIndex` for lookup with the block header's `chainRef` as key.
4. Flag errors in the `BlockChain` entry:
    a. If `errors` contains `NO_DATA_BTC_RELAY`, append the `RichBlockHeader.blockHeight` to `BlockChain.noData`
    b. If `errors` contains `INVALID_BTC_RELAY`, append the `RichBlockHeader.blockHeight` to `BlockChain.invalid` .
5. Emit `FlagBTCBlockError(blockHash, chainId, errors)` event, with the given `blockHash`, the `chainId` of the flagged `BlockChain` entry and the given `errors` as parameters.
6. Return

clearBlockError

Clears `ErrorCode` entries given as parameters from the status of a `RichBlockHeader`. Can be `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY` failure.

Note: This function can only be called from the *Security* module of interBTC, after Staked Relayers have achieved a majority vote on a BTC Parachain status update indicating that a `RichBlockHeader` entry no longer has the specified errors.

Specification

Function Signature

`flagBlockError(blockHash, errors)`

Parameters

- `blockHash`: SHA256 block hash of the block containing the error.
- `errors`: list of `ErrorCode` entries which are to be **cleared** from the block with the given blockHash. Can be `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY`.

Events

- `ClearBlockError(blockHash, chainId, errors)` - emits an event indicating that a Bitcoin block hash (identified `blockHash`) in a `BlockChain` entry (`chainId`) was cleared from the given errors (`errors` list of `ErrorCode` entries).

Errors

- `ERR_UNKNOWN_ERRORCODE = "The reported error code is unknown"`: The reported `ErrorCode` can only be `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY`.
- `ERR_BLOCK_NOT_FOUND = "No Bitcoin block header found with the given block hash"`: No `RichBlockHeader` entry exists with the given block hash.
- `ERR_ALREADY_REPORTED = "This error has already been reported for the given block hash and is pending confirmation"`: The error reported for the given block hash is currently pending a vote by Staked Relayers.

Function Sequence

1. Check if `errors` contains `NO_DATA_BTC_RELAY` or `INVALID_BTC_RELAY`. If neither match, return `ERR_UNKNOWN_ERRORCODE`.
2. Retrieve the `RichBlockHeader` entry from `BlockHeaders` using `blockHash`. Return `ERR_BLOCK_NOT_FOUND` if no block header can be found.
3. Retrieve the `BlockChain` entry for the given `RichBlockHeader` using `ChainsIndex` for lookup with the block header's `chainRef` as key.
4. Un-flag error codes in the `BlockChain` entry.
    a. If `errors` contains `NO_DATA_BTC_RELAY`: remove `RichBlockHeader.blockHeight` from `BlockChain.noData`
    b. If `errors` contains `INVALID_BTC_RELAY`: remove `RichBlockHeader.blockHeight` from `BlockChain.invalid`
5. Emit `ClearBlockError(blockHash, chainId, errors)` event, with the given `blockHash`, the `chainId` of the flagged `BlockChain` entry and the given `errors` as parameters.
6. Return

8.2.3 Functions: Parser

List of functions used to extract data from Bitcoin block headers and transactions. See the Bitcoin Developer Reference for details on the block header and transaction format.

Note: When comparing byte values, use the hash (e.g. SHA256) to avoid errors.

Block Header

extractHashPrevBlock

Extracts the `hashPrevBlock` (reference to previous block) from a Bitcoin block header.

Function Signature

`extractHashPrevBlock(blockHeaderBytes)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header.

Returns

- `hashPrevBlock`: the 32 byte block hash reference to the previous block.

Function Sequence

1. Return 32 bytes starting at index 4 of `blockHeaderBytes`

extractMerkleRoot

Extracts the `merkleRoot` from a Bitcoin block header.

Function Signature

`extractMerkleRoot(blockHeaderBytes)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header

Returns

- `merkleRoot`: the 32 byte Merkle tree root of the block header

Function Sequence

1. Return 32 bytes starting at index 36 of `blockHeaderBytes`.

extractTimestamp

Extracts the timestamp from the block header.

Function Signature

`extractTimestamp(blockHeaderBytes)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header

Returns

- `timestamp`: timestamp representation of the 4 byte timestamp field of the block header

Function Sequence

1. Return 32 bytes starting at index 68 of `blockHeaderBytes`.

extractNBits

Extracts the `nBits` from a Bitcoin block header. This field is necessary to compute that `target` in `nBitsToTarget`.

Function Signature

`extractNBits(blockHeaderBytes)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header

Returns

- `nBits`: the 4 byte nBits field of the block header

Function Sequence

1. Return 4 bytes starting at index 72 of `blockHeaderBytes`.

parseBlockHeader

Parses a 80 bytes raw Bitcoin block header and, if successful, returns a `RichBlockHeader` struct.

Function Signature

`parseBlockHeader(blockHeaderBytes)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header

Returns

- `BlockHeader`: the parsed Bitcoin block header

Errors

- `ERR_INVALID_HEADER_SIZE = "Invalid block header size"`: return error if the submitted block header is not exactly 80 bytes long.

Function Sequence

1. Check that the `blockHeaderBytes` is 80 bytes long. Return `ERR_INVALID_HEADER_SIZE` exception and abort otherwise.
2. Create a new `BlockHeader` (`BlockHeader`) struct and initialize as follows:

- `BlockHeader.merkleRoot` =*extractMerkleRoot* (`blockHeaderBytes`)
- `BlockHeader.target` = *nBitsToTarget* (*extractNBits* (`blockHeaderBytes`))
- `BlockHeader.timestamp` = *extractTimestamp* (`blockHeaderBytes`)
- `BlockHeader.hashPrevBlock = :ref:`extractHashPrevBlock` (``blockHeaderBytes`)

3. Return `BlockHeader`

Transactions

extractOutputs

Extracts the outputs from the given (raw) transaction (`rawTransaction`).

Specification

Function Signature

`extractOutputs(rawTransaction) -> u64`

Parameters

- `rawTransaction`: A variable byte size encoded transaction.

Returns

- `outputs`: A list of variable byte size encoded outputs of the given transaction.

Function Sequence

1. Determine the start of the output list in the transaction using *getOutputStartIndex*.
2. Determine the number of outputs (determine VarInt size using *determineVarIntDataLength* and extract bytes indicating the number of outputs accordingly).
3. Loop over the output size, determining the output length for each output (determine VarInt size using *determineVarIntDataLength* and extract bytes indicating the output size accordingly). Extract the bytes for each output and append them to the `outputs` list.
4. Return `outputs`.

Note: Optionally, check the output type here and add flag to return list (use tuple of flag and output bytes then).

---

getOutputStartIndex

Extracts the starting index of the outputs in a transaction (i.e., skips over the variable size list of inputs).

Function Signature

`getOutputStartIndex(rawTransaction -> u64)`

Parameters

- `rawTransaction`: A variable byte size encoded transaction.

Returns

- `outputIndex`: integer index indicating the starting point of the list of outputs in the raw transaction.

Errors

- `ERR_INVALID_TX_VERSION = "Invalid transaction version"`: The version of the given transaction is not 1 or 2.

Note: Currently, the transaction version can be 1 or 2. See transaction format details in the Bitcoin Developer Reference.

Function Sequence

See the Bitcoin transaction format in the Bitcoin Developer Reference.

1. Init position counter `pos = 0`.
2. Check the `version` bytes of the transaction (must be 1 or 2). Then skip over: `pos = pos + 4`.
3. Check if the transaction is a SegWit transaction. If yes, `pos = pos + 2`.
4. Parse the VarInt size (:ref:`determineVarIntDataLength`) and extract the bytes indicating the number of inputs accordingly. Increment `pos` accordingly.
5. Iterate over the number of inputs and skip over (incrementing `pos`). Note: it is necessary to determine the length of the `scriptSig` using *determineVarIntDataLength*.
6. Return `pos` indicating the start of the output list in the raw transaction.

determineVarIntDataLength

Determines the length of the Bitcoin CompactSize Unsigned Integers (other term for *VarInt*) in bytes. See CompactSize Unsigned Integers for details.

Function Signature

`getOutputStartIndex(varIntFlag -> u64)`

Parameters

- `varIntFlag`: 1 byte flag indicating size of Bitcoin's VarInt

Returns

- `varInt`: integer length of the VarInt (excluding flag).

Function Sequence

1. Check flag and return accordingly:

- If `0xff` return `8`,
- Else if `0xfe` return 4,
- Else if `0xfd` return 2,
- Otherwise return `0`

extractOPRETURN

Extracts the OP_RETURN of a given transaction. The OP_RETURN field can be used to store 80 bytes in a given Bitcoin transaction. The transaction output that includes the OP_RETURN is provably unspendable.

Note: The OP_RETURN field is used to include replay protection data in the interBTC *Issue*, *Redeem*, and *Replace* protocols.

Function Signature

`extractOPRETURN()`

Parameters

- `rawOutput`: raw encoded output

Returns

- `opreturn`: value of the OP_RETURN data.

Errors

- `ERR_NOT_OP_RETURN = "Expecting OP_RETURN output, but got another type.`: The given output was not an OP_RETURN output.

Function Sequence

1. Check that the output is indeed an OP_RETURN output: `pk_script[0] == 0x6a`. Return `ERR_NOT_OP_RETURN` error if this check fails. Note: the `pk_script` starts at index `9` of the output (nevertheless, make sure to check the length of VarInt indicating the output size using *determineVarIntDataLength*).
2. Determine the length of the OP_RETURN field (`pk_script[10]`) and return the OP_RETURN value (excluding the flag and size, i.e., starting at index `11`).

extractOutputValue

Extracts the value of the given output.

Note: Needs conversion to Big Endian when converting to integer.

Function Signature

`extractOutputValue(rawOutput)`

Parameters

- `rawOutput`: raw encoded output

Returns

- `value`: value of the output.

Function Sequence

1. Return the first 8 bytes of `output`, converted from LE to BE.

extractOutputAddress

Extracts the value of the given output.

Note: Please refer to the Bitcoin Developer Reference on Transactions when implementing this function.

Function Signature

`extractOutputAddress(rawOutput)`

Parameters

- `rawOutput`: raw encoded output

Returns

- `value`: value of the output.

Errors

- `ERR_INVALID_OUTPUT_SCRIPT = "Invalid or malformed output script"`: The script of the given output is invalid or malformed.

Function Sequence

1. Check if output is a SegWit output: `output[9] == 0`.
    a. If SegWit output (P2WPKH or P2WSH), check that `output[10]` equals the length of the output script (extract from``output[8]``). If this check fails, return `ERR_INVALID_OUTPUT_SCRIPT`.
    b. Return the number of characters specified in `output[8]` (length of the output script), starting with `output[11]`. This will be 20 bytes for P2WPKH and 32 bytes for P2WSH.
2. Otherwise, extract the `tag` indicating the output type: 3 bytes starting at index `8` in `output`.
    a. If P2PKH output (`tag == [0x19, 0x76, 0xa9]`). Check that `output[11] == [0x14]` or the last two bytes are equal to `[0x88, 0xac]. If this check fails, return ``ERR_INVALID_OUTPUT_SCRIPT`. Otherwise, return 20 bytes starting with `output[12]`.
    b. If P2WSH output (`tag == [0x17, 0xa9, 0x14]`). Check that the last byte is equal to `[0x87]`. If this check fails, return `ERR_INVALID_OUTPUT_SCRIPT`. Otherwise, return 32 bytes starting with `output[12]`.

8.2.4 Functions: Utils

There are several helper methods available that abstract Bitcoin internals away in the main function implementation.

sha256d

Bitcoin uses a double SHA256 hash to protect against "length-extension" attacks.

Note: Bitcoin uses little endian representations when sending hashes across the network and for storing values internally. For more details, see the documentation. The output of the SHA256 function is big endian by default.

Function Signature

`sha256d(data)`

Parameters

- `data`: bytes encoded input.

Returns

- `hash`: the double SHA256 hash encodes as a bytes from `data`.

Function Sequence

1. Hash `data` with sha256.
2. Hash the result of step 1 with sha256.
3. Return `hash`.

concatSha256d

A function that computes a parent hash from two child nodes. This function is used in the reconstruction of the Merkle tree.

Function Signature

`concatSha256d(left, right)`

Parameters

- `left`: 32 bytes of input data that are added first.
- `right`: 32 bytes of input data that are added second.

Returns

- `hash`: the double sha256 hash encoded as a bytes from `left` and `right`.

Function Sequence

1. Concatenate `left` and `right` into a 64 bytes.
2. Call the *sha256d* function to hash the concatenated bytes.
3. Return `hash`.

nBitsToTarget

This function calculates the PoW difficulty target from a compressed nBits representation. See the Bitcoin documentation for further details. The computation for the difficulty is as follows:

$$\text{target} = \text{significand} * \text{base}^{(\text{exponent}-3)}$$

Function Signature

`nBitsToTarget(nBits)`

Parameters

- `nBits`: 4 bytes compressed PoW target representation.

Returns

- `target`: PoW difficulty target computed from nBits.

Function Sequence

1. Extract the *exponent* by shifting the `nBits` to the right by 24.
2. Extract the *significand* by taking the first three bytes of `nBits`.
3. Calculate the `target` via the equation above and using 2 as the *base* (as we use the U256 type).
4. Return `target`.

checkCorrectTarget

Verifies the currently submitted block header has the correct difficulty target.

Function Signature

`checkCorrectTarget(hashPrevBlock, blockHeight, target)`

Parameters

- `hashPrevBlock`: 32 bytes previous block hash (necessary to retrieve previous target).
- `blockHeight`: height of the current block submission.
- `target`: PoW difficulty target computed from nBits.

Returns

- `True`: if the difficulty target is set correctly.
- `False`: otherwise.

Function Sequence

1. Retrieve the previous block header with the `hashPrevBlock` from the `BlockHeaders` storage and the difficulty target (`prevTarget`) of this (previous) block.
2. Check if the `prevTarget` difficulty should be adjusted at this `blockHeight`.
    a. If the difficulty should not be adjusted, check if the `target` of the submitted block matches the `prevTarget` of the previous block and check that `prevTarget``is not ``0`. Return false if either of these checks fails.
    b. The difficulty should be adjusted. Calculate the new expected target by calling the *computeNewTarget* function and passing the timestamp of the previous block (get using `hashPrevBlock` key in `BlockHeaders`), the timestamp of the last re-target (get block hash from `Chains` using `blockHeight - 2016` as key, then query `BlockHeaders`) and the target of the previous block (get using `hashPrevBlock` key in `BlockHeaders`) as parameters. Check that the new target matches the `target` of the current block (i.e., the block's target was set correctly).
        i. If the newly calculated target difficulty matches `target`, return `True`.
        ii. Otherwise, return `False`.

computeNewTarget

Computes the new difficulty target based on the given parameters, as implemented in the Bitcoin core client.

Function Signature

`computeNewTarget(prevTime, startTime, prevTarget)`

Parameters

- `prevTime`: timestamp of previous block.
- `startTime`: timestamp of last re-target.
- `prevTarget`: PoW difficulty target of the previous block.

Returns

- `newTarget`: PoW difficulty target of the current block.

Function Sequence

1. Compute the actual time span between `prevTime` and `startTime`.
2. Compare if the actual time span is smaller than the target interval divided by 4 (default target interval in Bitcoin is two weeks). If true, set the actual time span to the target interval divided by 4.
3. Compare if the actual time span is greater than the target interval multiplied by 4. If true, set the actual time span to the target interval multiplied by 4.
4. Calculate the `newTarget` by multiplying the actual time span with the `prevTarget` and dividing by the target time span (2 weeks for Bitcoin).
5. If the `newTarget` is greater than the maximum target in Bitcoin, set the `newTarget` to the maximum target (Bitcoin maximum target is $2^{224} - 1$).
6. Return the `newTarget`.

computeMerkle

The computeMerkle function calculates the root of the Merkle tree of transactions in a Bitcoin block. Further details are included in the Bitcoin developer reference.

Function Signature

`computeMerkle(txId, txIndex, merkleProof)`

Parameters

- `txId`: the hash identifier of the transaction.
- `txIndex`: index of transaction in the block's transaction Merkle tree.
- `merkleProof`: Merkle tree path (concatenated LE sha256 hashes).

Returns

- `merkleRoot`: the hash of the Merkle root.

Errors

- `ERR_INVALID_MERKLE_PROOF = "Invalid Merkle Proof structure"`: raise an exception if the Merkle proof is malformed.

Function Sequence

1. Check if the length of the Merkle proof is 32 bytes long.
    a. If true, only the coinbase transaction is included in the block and the Merkle proof is the `merkleRoot`. Return the `merkleRoot`.
    b. If false, continue function execution.
2. Check if the length of the Merkle proof is greater or equal to 64 and if it is a power of 2.
    a. If true, continue function execution.
    b. If false, raise `ERR_INVALID_MERKLE_PROOF`.
3. Calculate the `merkleRoot`. For each 32 bytes long hash in the Merkle proof:
    a. Determine the position of transaction hash (or the last resulting hash) at either `0` or `1`.
    b. Slice the next 32 bytes from the Merkle proof.
    c. Concatenate the transaction hash (or last resulting hash) with the 32 bytes of the Merkle proof in the right order (depending on the transaction/last calculated hash position).
    d. Calculate the double SHA256 hash of the concatenated input with the *concatSha256d* function.

e. Repeat until there are no more hashes in the `merkleProof`.

4. The last resulting hash from step 3 is the `merkleRoot`. Return `merkleRoot`.

Example

Assume we have the following input:

- txId: `330dbbc15169c538583073fd0a7708d8de2d3dc155d75b361cbf5c24b73f3586`
- txIndex: `0`
- merkleProof: `86353fb7245cbf1c365bd755c13d2dded808770afd73305838c56951c1bb0d33b635f586cf6c4763f3fc9`

The `computeMerkle` function would go past step 1 as our proof is longer than 32 bytes. Next, step 2 would also be passed as the proof length is equal to 64 bytes and a power of 2. Last, we calculate the Merkle root in step 3 as shown below.

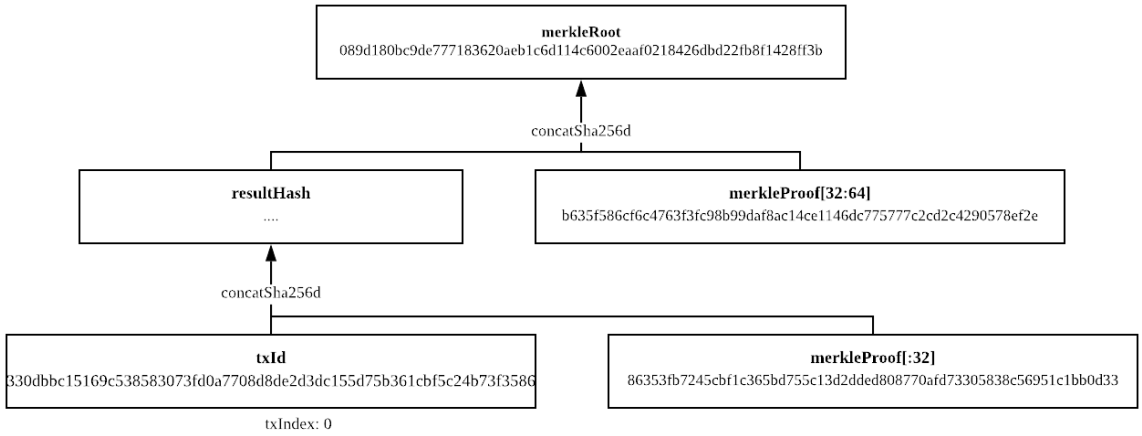


Fig. 8.5: An example of the `computeMerkle` function with a transaction from a block that contains two transactions in total.

calculateDifficulty

Given the `target`, calculates the Proof-of-Work `difficulty` value, as defined in the Bitcoin wiki.

Function Signature

`calculateDifficulty(target)`

Parameters

- `target`: target as specified in a Bitcoin block header.

Returns

- `difficulty`: difficulty calculated from given `target`.

Function Sequence

1. Return `0xffff0000000000000000000000000000000000000000000000000000` (max. possible target, also referred to as "difficulty 1") divided by `target`.

getForkIdByBlockHash

Helper function allowing to query the list of tracked forks (`Forks`) for the identifier of a fork given its last submitted ("highest") block hash.

Specification

Function Signature

`getForkIdByBlockHash(blockHash)`

Parameters

- `blockHash`: block hash of the last submitted block to a fork.

Returns

- `forkId`: if there exists a fork with `blockHash` as latest submitted block in `forkHashes`.
- `ERR_FORK_ID_NOT_FOUND`: otherwise.

Errors

- `ERR_FORK_ID_NOT_FOUND = Fork ID not found for specified block hash.`": return this error if there exists no `forkId` for the given `blockHash`.

Function Sequence

1. Loop over all entries in `Forks` and check if `forkHashes[forkHashes.length -1] == blockhash`
    a. If `True`: return the corresponding `forkId`.
2. Return `ERR_FORK_ID_NOT_FOUND` otherwise.

incrementChainCounter

Increments the current `ChainCounter` and returns the new value.

Specification

Function Signature

`incrementChainsCounter()`

Returns

- `chainCounter`: the new integer value of the `ChainCounter`.

Function Sequence

1. `ChainCounter++`
2. Return `ChainCounter`

8.2.5 Events

Initialized

If the first block header was stored successfully, emit an event with the stored block's height and the (PoW) block hash.

Event Signature

`Initialized(blockHeight, blockHash)`

Parameters

- `blockHeight`: height of the current block submission.
- `blockHash`: hash of the current block submission.

Functions

- *initialize*

StoreMainChainHeader

If the block header was stored successfully, emit an event with the stored block's height and the (PoW) block hash.

Event Signature

`StoreMainChainHeader(blockHeight, blockHash)`

Parameters

- `blockHeight`: height of the current block submission.
- `blockHash`: hash of the current block submission.

Functions

- *storeBlockHeader*

StoreForkHeader

If the submitted block header is on a fork, emit an event with the fork's id, block height and the (PoW) block hash.

Event Signature

`StoreForkHeader(forkId, blockHeight, blockHash)`

Parameters

- `forkId`: unique identifier of the tracked fork.
- `blockHeight`: height of the current block submission.
- `blockHash`: hash of the current block submission.

Functions

- *storeBlockHeader*

ChainReorg

If the submitted block header on a fork results in a reorganization (fork longer than current main chain), emit an event with the block hash of the new highest block, the new maximum block height and the depth of the fork

Event Signature

`ChainReorg(newChainTip, blockHeight, forkDepth)`

Parameters

- `newChainTip`: hash of the new highest block.
- `blockHeight`: new maximum block height (block height of fork tip).
- `forkDepth`: depth of the fork (number of block after diverging from previous main chain).

Functions

- *storeBlockHeader*

VerifyTransactionInclusion

If the verification of the transaction inclusion proof was successful, emit an event for the given transaction identifier (`txId`), block height (`txBlockHeight`), and the specified number of `confirmations`.

Event Signature

`VerifyTransaction(txId, blockHeight, confirmations)`

Parameters

- `txId`: the hash of the transaction.
- `txBlockHeight`: height of block of the transaction.
- `confirmations`: number of confirmations requested for the transaction verification.

Functions

- *verifyTransactionInclusion*

ValidateTransaction

If parsing and validation of the given raw transaction was successful, emit an event specifying the `txId`, the `paymentValue`, the `recipientBtcAddress` and the `opReturnId`.

Event Signature

`ValidateTransaction(txId, paymentValue, recipientBtcAddress, opReturnId)`

Parameters

- `txId`: the hash of the transaction.
- `paymentValue`: integer value of BTC sent in the transaction.
- `recipientBtcAddress`: Bitcoin address (hash) of recipient.
- `opReturnId`: [Optional] 32 byte hash identifier expected in OP_RETURN (replay protection).

Functions

- *validateTransaction*

8.2.6 Error Codes

A summary of error codes raised in exceptions by BTC-Relay, and their meanings, are provided below.

`ERR_ALREADY_INITIALIZED`

- **Message:** "Already initialized."
- **Function:** *initialize*
- **Cause**: Raised if the `initialize` function is called when BTC-Relay has already been initialized.

`ERR_NOT_MAIN_CHAIN`

- **Message:** "Main chain submission indicated, but submitted block is on a fork"
- **Function:** *storeBlockHeader*
- **Cause**: Raised if the block header submission indicates that it is extending the current longest chain, but is actually on a (new) fork.

`ERR_FORK_PREV_BLOCK`

- **Message:** "Previous block hash does not match last block in fork submission"
- **Function:** *storeBlockHeader*
- **Cause**: Raised if the block header does not reference the highest block in the fork specified by `forkId` (via `prevBlockHash`).

`ERR_NOT_FORK`

- **Message**: "Indicated fork submission, but block is in main chain"
- **Function**: *storeBlockHeader*
- **Cause**: Raised if raise exception if the submitted block header is actually extending the current longest chain tracked by BTC-Relay (`Chains`), instead of a fork.

`ERR_INVALID_FORK_ID`

- **Message**: "Incorrect fork identifier."
- **Function**: *storeBlockHeader*
- **Cause**: Raised if a non-existent fork identifier is passed.

`ERR_INVALID_HEADER_SIZE`

- **Message**: "Invalid block header size":
- **Function**: *parseBlockHeader*
- **Cause**: Raised if the submitted block header is not exactly 80 bytes long.

`ERR_DUPLICATE_BLOCK`

- **Message**: "Block already stored"
- **Function**: *verifyBlockHeader*
- **Cause**: Raised if the submitted block header is already stored in the BTC-Relay (duplicate PoW `blockHash`).

`ERR_PREV_BLOCK`

- **Message**: "Previous block hash not found"
- **Function**: *verifyBlockHeader*
- **Cause**: Raised if the submitted block does not reference an already stored block header as predecessor (via `prevBlockHash`).

`ERR_LOW_DIFF`

- **Message**:"PoW hash does not meet difficulty target of header"

- **Function**: *verifyBlockHeader*
- **Cause**: Raised if the header's `blockHash` does not meet the `target` specified in the block header.

`ERR_DIFF_TARGET_HEADER`

- **Message**: "Incorrect difficulty target specified in block header"
- **Function**: *verifyBlockHeader*
- **Cause**: Raised if the `target` specified in the block header is incorrect for its block height (difficulty re-target not executed).

`ERR_MALFORMED_TXID`

- **Message**: "Malformed transaction identifier"
- **Function**: *verifyTransactionInclusion*
- **Cause**: Raised if the transaction id (`txId`) is malformed.

`ERR_CONFIRMATIONS`

- **Message**: "Transaction has less confirmations than requested"
- **Function**: *verifyTransactionInclusion*
- **Cause**: Raised if the number of confirmations is less than required.

`ERR_INVALID_MERKLE_PROOF`

- **Message**: "Invalid Merkle Proof"
- **Function**: *verifyTransactionInclusion*
- **Cause**: Exception raised in `verifyTransactionInclusion` when the Merkle proof is malformed.

`ERR_FORK_ID_NOT_FOUND`

- **Message**: "Fork ID not found for specified block hash"
- **Function**: *getForkIdByBlockHash*
- **Cause**: Return this error if there exists no `forkId` for the given `blockHash`.

`ERR_NO_DATA`

- **Message**: "BTC-Relay has a NO_DATA failure and the requested block cannot be verified reliably"
- **Function**: *verifyTransactionInclusion*
- **Cause**: The BTC Parachain has been partially deactivated for all blocks with a higher block height than the lowest blocked flagged with `NO_DATA_BTC_RELAY`.

`ERR_INVALID`

- **Message**: "BTC-Relay has detected an invalid block in the current main chain, and has been halted"
- **Function**: *verifyTransactionInclusion*
- **Cause**: The BTC Parachain has been halted because Staked Relayers reported an invalid block.

`ERR_SHUTDOWN`

- **Message**: "BTC Parachain has shut down"
- **Function**: *verifyTransactionInclusion* | *storeBlockHeader* | *storeBlockHeader*
- **Cause**: The BTC Parachain has been shutdown by a manual intervention of the Governance Mechanism.

`ERR_INVALID_TXID`

- **Message**: "Transaction hash does not match given txid"
- **Function**: *validateTransaction*
- **Cause**: The transaction identifier (`txId`) does not match the actual hash of the transaction.

`ERR_INSUFFICIENT_VALUE`:

- **Message**: "Value of payment below requested amount"
- **Function**: *validateTransaction*
- **Cause**: The value of the (first) *Payment UTXO* in the validated transaction is lower than the specified `paymentValue`.

`ERR_TX_FORMAT`:

- **Message**: "Transaction has incorrect format"
- **Function**: *validateTransaction*
- **Cause**: The parsed transaction has an incorrect format (see *Accepted Bitcoin Transaction Format*).

`ERR_WRONG_RECIPIENT`

- **Message**: "Incorrect recipient Bitcoin address"
- **Function**: *validateTransaction*
- **Cause**: The recipient specified in the (first) *Payment UTXO* of the validated transaction does not match the specified `recipientBtcAddress`.

`ERR_INVALID_OPRETURN`

- **Message**: "Incorrect identifier in OP_RETURN field"
- **Function**: *validateTransaction*
- **Cause**: The OP_RETURN field of the (second) *Data UTXO* of the validated transaction does not match the specified `opReturnId`.

`ERR_INVALID_TX_VERSION`

- **Message**: "Invalid transaction version"
- **Function**: *getOutputStartIndex*
- **Cause**: : The version of the given transaction is not 1 or 2. See transaction format details in the Bitcoin Developer Reference.

`ERR_NOT_OP_RETURN`

- **Message**: "Expecting OP_RETURN output, but got another type."
- **Function**: *extractOPRETURN*
- **Cause**: The given output was not an OP_RETURN output.

`ERR_ONGOING_FORK`

- **Message**: "Verification disabled due to ongoing fork"
- **Function**: *verifyTransactionInclusion*
- **Cause**: The `mainChain` is not at least `STABLE_BITCOIN_CONFIRMATIONS` ahead of the next best fork.

CHAPTER

NINE

COLLATERAL

9.1 Overview

There are two different kinds of collateral in use in the bridge. The first is the backing collateral that vaults use as insurance to issued wrapped tokens. Multiple backing collaterals are supported, see *Multi-Collateral*, but similarly to MakerDAO, each vault uses a single currency. If vault operators want to use multiple currencies, they have to register multiple vaults. It is possible to use key derivation to run multiple vaults using a single mnemonic. When a vault is registered, they have to explicitly choose the used currency. In contrast, when interacting with vaults, the used collateral is implicit. For example, when a vault fails to execute a redeem request, the user will receive some amount of the vault's backing collateral. As such, the user might want to select a vault that uses their preferred currency.

The second type of collateral is griefing collateral. The currency used for this type of collateral is fixed and depends on the used network. This is the currency that is also used to pay transaction fees. For example, in Kusama transaction fees are by default paid in KINT and on Polkadot transaction fees are paid in INTR.

While collateral management is logically distinct from treasury management, they are both implemented using the same *Currency* pallet. This pallet is used 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.

9.1.1 Step-by-Step

The protocol has three different "sub-protocols".

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

CHAPTER

TEN

CURRENCY

10.1 Overview

This currency pallet provides an interface for the other pallets to manage balances of different currencies. Accounts have three balances per currency: they have a `free`, `reserved`, and `frozen` amount. Users are able to freely transfer `free - frozen` balances, but only the parachain pallets are able to operate on `reserved` amounts. `Frozen` is used to implement temptorary locks of free balances like vesting schedules.

The external API for dispatchable and RPC functions use 'thin' amount types, meaning that the used currency depends on the context. For example, the currency used in *deposit_collateral* depends on the vault's `currencyId`. Sometimes, as is for example the case for *register_vault*, the function takes an additional `currencyId` argument to specify the currency to use. In contrast, internally in the parachain amounts are often represented by the `Amount` type defined in this pallet, which in addition to the amount, also contains the used currency. The benefit of this type is two-fold. First, we can guarantee that operations only work on compatible amounts. For example, it prevents adding DOT amounts to KSM amounts. Second, it allows for a more convenient api.

10.2 Data Model

10.2.1 Structs

Amount

Stores an amount and the used currency.

Parameter	Type	Description
`balance`	Balance	The amount.
`currency`	CurrencyId	The used currency.

10.3 Functions

10.3.1 from_signed_fixed_point

Constructs an `Amount` from a signed fixed point number and a `currencyId`. The fixed point number is truncated. E.g., a value of 2.5 would return 2.

Specification

Function Signature

`from_signed_fixed_point(amount, currencyId)`

Parameters

- `amount`: The amount as fixed point.
- `currencyId`: The currency.

Preconditions

- `amount` MUST be representable as a 128 bit unsigned number.

Postconditions

- An `Amount` MUST be returned where `Amount.amount` is the truncated `amount` argument, and `Amount.currencyId` is the `currencyId` argument.

10.3.2 to_signed_fixed_point

Converts an `Amount` struct into a fixed-point number.

Specification

Function Signature

`to_signed_fixed_point(amount)`

Parameters

- `amount`: The amount struct.

Preconditions

- `amount` MUST be representable by the signed fixed point type.

Postconditions

- `amount.amount` MUST be returned as a fixed point number.

10.3.3 convert_to

Converts the given `amount` into the given currency.

Specification

Function Signature

`convert_to(amount, currencyId)`

Parameters

- `amount`: The amount struct.
- `currencyId`: The currency to convert to.

Preconditions

- *convert* when called with `amount` and `currencyId` MUST return successfully.

Postconditions

- *convert* MUST be called with `amount` and `currencyId` as arguments.

10.3.4 checked_add

Adds two amounts.

Specification

Function Signature

`checked_add(amount1, amount2)`

Parameters

- `amount1`: the first amount.
- `amount2`: the second amount.

Preconditions

- `amount1.currencyId` MUST be equal to `amount2.currencyId`

Postconditions

- MUST return the sum of both amounts.

10.3.5 checked_sub

Subtracts two amounts.

Specification

Function Signature

`checked_sub(amount1, amount2)`

Parameters

- `amount1`: the first amount.
- `amount2`: the second amount.

Preconditions

- `amount1.currencyId` MUST be equal to `amount2.currencyId`

Postconditions

- MUST return `amount1 - amount2`.

10.3.6 saturating_sub

Subtracts two amounts, or zero if the result would be negative.

Specification

Function Signature

`saturating_sub(amount1, amount2)`

Parameters

- `amount1`: the first amount.
- `amount2`: the second amount.

Preconditions

- `amount1.currencyId` MUST be equal to `amount2.currencyId`

Postconditions

- if `amount2 <= amount1`, then this function MUST return `amount1 - amount2`.
- if `amount2 > amount1`, then this function MUST return zero.

10.3.7 checked_fixed_point_mul

Multiplies an amount by a fixed point scalar. The result is rounded down.

Specification

Function Signature

`checked_fixed_point_mul(amount, scalar)`

Parameters

- `amount`: the Amount struct.
- `scalar`: the fixed point scalar.

Preconditions

- The multiplied amount MUST be representable by a 128 bit unsigned integer.

Postconditions

- MUST return a copy of `amount` that is multiplied by the scalar. The result MUST be rounded down.

10.3.8 checked_fixed_point_mul_rounded_up

Like *checked_fixed_point_mul*, but with a rounded-up result.

Specification

Function Signature

`checked_fixed_point_mul_rounded_up(amount, scalar)`

Parameters

- `amount`: the Amount struct.
- `scalar`: the fixed point scalar.

Preconditions

- The multiplied amount MUST be representable by a 128 bit unsigned integer.

Postconditions

- MUST return a copy of `amount` that is multiplied by the scalar. The result MUST be rounded up.

10.3.9 rounded_mul

Like *checked_fixed_point_mul*, but with a rounded result.

Specification

Function Signature

`rounded_mul(amount, scalar)`

Parameters

- `amount`: the Amount struct.
- `scalar`: the fixed point scalar.

Preconditions

- The multiplied amount MUST be representable by a 128 bit unsigned integer.

Postconditions

- MUST return a copy of `amount` that is multiplied by the scalar. The result MUST be rounded to the nearest integer.

10.3.10 checked_div

Divides an amount by a fixed point scalar. The result is rounded down.

Specification

Function Signature

`checked_div(amount, scalar)`

Parameters

- `amount`: the Amount struct.
- `scalar`: the fixed point scalar.

Preconditions

- The multiplied amount MUST be representable by a 128 bit unsigned integer.

Postconditions

- MUST return a copy of `amount` that is divided by the scalar.

10.3.11 ratio

Returns the fixed point ratio between two amounts.

Specification

Function Signature

`ratio(amount1, amount2)`

Parameters

- `amount1`: the first Amount struct.
- `amount2`: the second Amount struct.

Preconditions

- `amount1.currencyId` MUST be equal to `amount2.currencyId`
- The ratio MUST be representable by the fixed point type.

Postconditions

- MUST return the ratio between the two amounts.

10.3.12 Comparisons: lt, le, eq, ge, gt

Compares two amounts

Specification

Function Signature

`[lt|le|eq|ge|gt](amount1, amount2)`

Parameters

- `amount1`: the first Amount struct.
- `amount2`: the second Amount struct.

Preconditions

- `amount1.currencyId` MUST be equal to `amount2.currencyId`

Postconditions

- MUST return true when the comparison holds.

10.3.13 transfer

Transfers the amount between the given accounts.

Specification

Function Signature

`transfer(amount, source, destination)`

Parameters

- `amount`: the Amount struct.
- `source`: the account to transfer from.
- `destination`: the account to transfer to.

Preconditions

- `source` MUST have sufficient unlocked funds in the given currency

Postconditions

- The free balance of `source` MUST decrease by `amount.amount` (in the currency determined by `amount.currencyId`)
- The free balance of `destination` MUST increase by `amount.amount` (in the currency determined by `amount.currencyId`)

10.3.14 lock_on

Locks the amount on the given account.

Specification

Function Signature

`lock_on(amount, accountId)`

Parameters

- `amount`: the Amount struct.
- `accountId`: the account to lock the amount on.

Preconditions

- The given account MUST have sufficient unlocked funds in the given currency.

Postconditions

- The free balance of `accountId` MUST decrease by `amount.amount` (in the currency determined by `amount.currencyId`)
- The locked balance of `accountId` MUST increase by `amount.amount` (in the currency determined by `amount.currencyId`)

10.3.15 unlock_on

Unlocks the amount on the given account.

Specification

Function Signature

`unlock_on(amount, accountId)`

Parameters

- `amount`: the Amount struct.
- `accountId`: the account to unlock the amount on.

Preconditions

- The given account MUST have sufficient locked funds in the given currency.

Postconditions

- The locked balance of `accountId` MUST decrease by `amount.amount` (in the currency determined by `amount.currencyId`)
- The free balance of `accountId` MUST increase by `amount.amount` (in the currency determined by `amount.currencyId`)

10.3.16 burn_from

Burns the amount on the given account.

Specification

Function Signature

`burn_from(amount, accountId)`

Parameters

- `amount`: the Amount struct.
- `accountId`: the account to lock the amount on.

Preconditions

- The given account MUST have sufficient locked funds in the given currency.

Postconditions

- The locked balance of `accountId` MUST decrease by `amount.amount` (in the currency determined by `amount.currencyId`)

10.3.17 mint_to

Mints the amount on the given account.

Specification

Function Signature

`mint_to(amount, accountId)`

Parameters

- `amount`: the Amount struct.
- `accountId`: the account to mint the amount on.

Postconditions

- The `free` balance of `accountId` MUST increase by `amount.amount` (in the currency determined by `amount.currencyId`)

CHAPTER

ELEVEN

FEE

11.1 Overview

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

11.1.1 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 incentivised network participants (i.e. Vaults).
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 are paid in `interBTC`.

11.2 Data Model

11.2.1 Scalars (Fees)

IssueFee

Issue fee share (configurable parameter, as percentage) that users need to pay upon issuing `interBTC`.

- Paid in `interBTC`
- Initial value: 0.5%

IssueGriefingCollateral

Issue griefing collateral as a percentage of the locked collateral of a Vault a user has to lock to issue `interBTC`.

- Paid in collateral
- Initial value: 0.005%

RefundFee

Refund fee (configurable parameter, as percentage) that users need to pay to refund overpaid `interBTC`.

- Paid in `interBTC`
- Initial value: 0.5%

RedeemFee

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

- Paid in `interBTC`
- 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 `interBTC` amount at the current exchange rate.

- Paid in collateral
- Initial value: 10%

TheftFee

Fee (as percentage) that a reporter receives if another Vault commits theft. The fee is paid in collateral taken from the liquidated Vault.

- Paid in collateral
- Initial value: 5%

TheftFeeMax

Upper bound to the reward that can be payed to a reporter on success. This is expressed in Bitcoin to ensure consistency between assets.

- Initial value: 0.1 BTC

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%

11.3 Functions

11.3.1 distributeRewards

Distributes fees among incentivised network participants.

Specification

Function Signature

`distributeRewards(amount)`

Preconditions

- There MUST be at least one registered Vault OR a treasury account.

Postconditions

- If there are no registered funds, rewards MUST be sent to the treasury account.
- Otherwise, rewards MUST be distributed according to *distributeReward*.

11.3.2 withdrawRewards

A function that allows incentivised network participants to withdraw all earned rewards.

Specification

Function Signature

`withdrawRewards(accountId, vaultId)`

Parameters

- `accountId`: the account withdrawing `interBTC` rewards.
- `vaultId`: the vault that generated `interBTC` rewards.

Events

- *WithdrawRewards*

Preconditions

- The function call MUST be signed by `accountId`.
- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The `accountId` MUST have available rewards for `interBTC`.

Postconditions

- The account's balance MUST increase by the available rewards.

- The account's withdrawable rewards MUST decrease by the withdrawn rewards.

11.4 Events

11.4.1 WithdrawRewards

Event Signature

`WithdrawRewards(account, amount)`

Parameters

- `account`: the account withdrawing rewards
- `amount`: the amount of rewards withdrawn

Functions

- *withdrawRewards*

CHAPTER

TWELVE

ORACLE

Note: This oracle model relies on trusted oracle sources. 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 Oracle receives a continuous data feed from off-chain oracles, with information in exchange rates or bitcoin inclusion estimates. Multiple oracles can be authorized, in which case the 'median' of all unexpired values is used as the actual value. It is not technically the median - when an even number of oracles have submitted values, it does not average the middle two values. Instead, it arbitrarily picks one of them. This is done because this can be done in O(n) rather than in O(n log n).

In the implementation, the *feed_values* function does not directly update the aggregate - this is done in the *on_initialize* hook, in order to keep the *feed_values* function weight independent of the number of oracles. Furthermore, for oracle offline detection and for updating the aggregate when a value becomes outdated, the *on_initialize* hook was necessary anyway.

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

12.1 Data Model

12.1.1 Enums

OracleKey

Key to indicate a specific value.

Discriminant	Description
`ExchangeRate(CurrencyId)`	Exchange rate against Bitcoin, in e.g. planck per satoshi.
`FeeEstimation`	Estimate of the Bitcoin inclusion fee, in satoshis per byte.

12.1.2 Scalars

MaxDelay

The time after which a reported value will no longer be considered valid.

12.1.3 Maps

Aggregate

Maps `oracle_key` to the median of all unexpired values reported by oracles for that key.

AuthorizedOracles

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

ValidUntil

Maps oracle_keys to a timestamp that indicates when one of the values expires, at which time a new aggregate needs to be calculated.

RawValues

Maps oracle_keys and account ids to raw timestamped values.

RawValuesUpdated

Maps oracle_key to a boolean value that indicates that a new value has been received that has not yet been included in the aggregate.

AuthorizedOracles

Maps oracle `accountId` to the oracle's name. The presence of an account id in this map indicates that the account is authorized to feed values.

12.2 Functions

12.2.1 feed_values

The dispatchable function that oracles call to feed new price data into the system.

Specification

Function Signature

`feed_values(oracle_id, Vec<oracle_key, value>)`

Parameters

- `oracle_id`: the oracle account calling this function.
- `oracle_key`: indicated which value is being set
- `value`: the value being set

Events

- *FeedValues*

Preconditions

- The function call MUST be signed by `oracle_id`.

- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The oracle MUST be authorized.

Postconditions

For each `(oracle_key, value)` pair,

- `RawValuesUpdated[oracle_key]` MUST be set to true
- `RawValues[oracle_key]` MUST be set to a `TimeStamped` values, where,
  - `TimeStamped.timestamp` MUST be the current time,
  - `TimeStamped.value` MUST be `value`.

12.2.2 get_price

Returns the latest medianized value for the given key, as calculated from the received external data sources.

Specification

Function Signature

`get_price(oracle_key)`

Parameters

- `oracle_key`: the key for which the value should be returned

Preconditions

- `ExchangeRate[oracle_key]` MUST NOT be `None`. That is, sufficient oracles must have submitted unexpired values.

Postconditions

- MUST return the fixed point value for the given key.

12.2.3 convert

Converts the given amount to the given currency.

Specification

Function Signature

`convert(amount, currencyId)`

Parameters

- `amount`: the amount to convert
- `currencyId`: the currency to convert to

Preconditions

- Exactly one of `amount.currencyId` and the `currencyId` argument MUST be the wrapped currency.
- Exactly one of `amount.currencyId` and the `currencyId` argument MUST be a collateral currency.

Postconditions

- MUST return `amount` converted to `currencyId`.

12.2.4 on_initialize

This function is called at the start of every block. When new values have been submitted, or when old values expire, this function update the aggregate value.

Specification

Function Signature

`on_initialize()`

Postconditions

- If `RawValuesUpdated` is empty, i.e., *feed_values* was not yet called since the initialization of the parachain, then the `OracleOffline` MUST be set in the *Security* pallet.
- For each `(oracle_key, updated)` in `RawValuesUpdated`, if `updated` is true, or the current time is greater than `ValidUntil[oracle]`,
  - `RawValuesUpdated[oracle_key]` MUST be set to false
  - `ExchangeRate[oracle_key]` MUST be set to the middle value of the sorted list of unexpired values from `RawValues[oracle_key]`. If there are an even number, one MAY be arbitrarily picked.
  - `ValidUntil[oracle_key]` MUST be set to `MaxDelay` plus the minimum timestamp from the unexpired values in `RawValues[oracle_key]`.

12.3 Events

12.3.1 FeedValues

12.3.2 SetExchangeRate

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

Event Signature

`FeedValues(oracle_id, Vec<(oracle_key, value)>),`

Parameters

- `oracle_id`: the oracle account calling this function.
- `oracle_key`: the key indicating which value is being set
- `value`: the new value

Function

- *feed_values*

CHAPTER

THIRTEEN

ISSUE

13.1 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.

13.1.1 Step-by-step

The nominal control flow is as follows:

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. The issue request includes the amount of interBTC the user wants to issue, the selected Vault, and a small collateral reserve to prevent *Griefing*.
3. A user sends the equivalent amount of BTC to issue as interBTC to the Vault on the Bitcoin blockchain.
4. The user or 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.

User Failsafe

To accommodate for user error, the bridge allows the execution of issue requests even when the user sends an incorrect BTC amount. Specifically, we distinguish the following cases:

- The user sends less than the expected amount. The user has the option to execute the issue with this amount. However, it will lose part of its griefing collateral. If it sends e.g. 10% of the expected amount, it loses 90% of the griefing collateral. It will also receive 10% of the wrapped tokens. Because there is a cost associated with this choice, automatic execution of this issue request by Vaults is disallowed. The alternative for the user is to make another Bitcoin transfer, and to execute the issue with that transaction. In this case, however, it loses the Bitcoin sent in the first transaction.
- The user sends more than the expected amount.
  - If the Vault has sufficient collateral to issue wrapped tokens for the sent amount, the size of the issue request is automatically increased and more collateral of the Vault is reserved. The user receives the amount corresponding to the received amount of Bitcoin. The issue fee is deducted from the updated (increased) amount.

– If the Vault does not have sufficient collateral to issue the additional amount, only the amount that was originally requested is issued. A refund request is sent to the Vault to return the surplus Bitcoin (excluding a fee). Note, however, that there is no penalty for the Vault if it does not return the surplus Bitcoin since this is a user fault rather than a Vault fault.

13.1.2 Security

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

13.1.3 Vault Registry

The data access and state changes to the Vault registry are documented in Fig. 13.1 below.

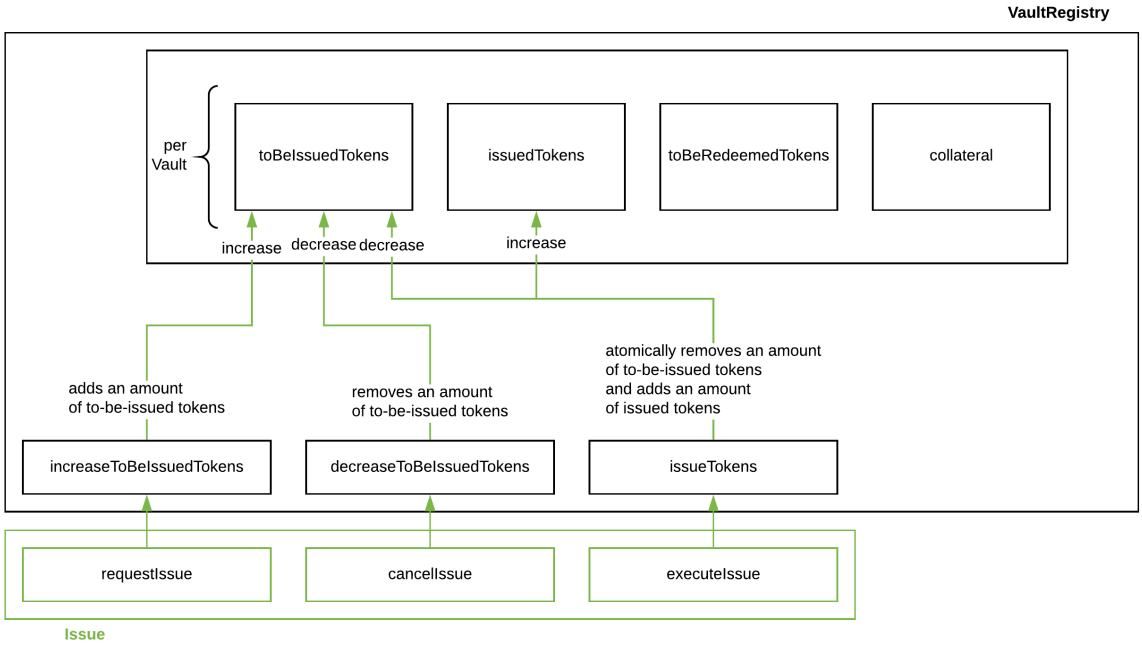


Fig. 13.1: The issue protocol interacts with three functions in the *Vault Registry* that handle updating the different token balances.

13.1.4 Fee Model

- 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.

13.2 Data Model

13.2.1 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.

IssueBtcDustValue

The minimum amount of BTC that is required for issue requests; lower values would risk the rejection of payment on Bitcoin.

13.2.2 Maps

IssueRequests

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

13.2.3 Structs

IssueRequest

Stores the status and information about a single issue request.

Parameter	Type	Description
`vault`	AccountId	The address of the Vault responsible for this issue request.
`opentime`	BlockNumber	The *ActiveBlockCount* when the issue request was created.
`period`	BlockNumber	Value of the *IssuePeriod* when the request was made.
`griefingCollateral`	DOT	Security deposit provided by a user.
`amount`	interBTC	Amount of interBTC to be issued.
`fee`	interBTC	Fee charged to the user for issuing.
`requester`	AccountId	User account receiving interBTC upon successful issuing.
`btcAddress`	BtcAddress	Vault's P2WPKH Bitcoin deposit address.
`btcPublicKey`	BtcPublicKey	Vault's Bitcoin public key used to generate the deposit address.
`btcHeight`	u32	The highest recorded height of the relay at time of opening.
`status`	Enum	Status of the request: Pending, Completed or Cancelled.

13.3 Functions

13.3.1 requestIssue

A user opens an issue request to create a specific amount of interBTC. When calling this function, a user provides their parachain account identifier, the to be issued amount of interBTC, and the Vault to use in this process (account identifier). Further, they provide some (small) amount of DOT collateral (`griefingCollateral`) to prevent griefing.

Specification

Function Signature

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

Parameters

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

Events

- *RequestIssue*

Preconditions

- The function call MUST be signed by `requester`.
- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The *BTC-Relay* MUST be initialized.
- The Vault MUST be registered and active.
- The Vault MUST NOT be banned.
- The `amount` MUST be greater than or equal to *IssueBtcDustValue*.
- The `griefingCollateral` MUST exceed or equal the value of request `amount` at the current exchange-rate, multiplied by *IssueGriefingCollateral*.
- The `griefingCollateral` MUST be equal or less than the requester's free balance in the *Griefing Collateral Currency*.
- The *tryIncreaseToBeIssuedTokens* function MUST return a new BTC deposit address for the Vault ensuring that the Vault's free collateral is above the *SecureCollateralThreshold* for the requested `amount` and that a unique BTC address is used for depositing BTC.
- A new unique `issuedId` MUST be generated via the *generateSecureId* function.

Postconditions

- The Vault's `toBeIssuedTokens` MUST increase by `amount`.
- The requester's free balance in the *Griefing Collateral Currency* MUST decrease by `griefingCollateral`.
- The requester's locked balance in the *Griefing Collateral Currency* MUST increase by `griefingCollateral`.
- A new BTC deposit address for the Vault MUST be generated by the *tryIncreaseToBeIssuedTokens*.
- The new issue request MUST be created as follows:
  - `issue.vault`: MUST be the `vault`.
  - `issue.opentime`: MUST be the *ActiveBlockCount* of the current block of this transaction.
  - `issue.period`: MUST be the current *IssuePeriod*.
  - `issue.griefingCollateral`: MUST be the `griefingCollateral` amount passed to the function.
  - `issue.amount`: MUST be `amount` minus `issue.fee`.
  - `issue.fee`: MUST equal `amount` multiplied by *IssueFee*.
  - `issue.requester`: MUST be the `requester`
  - `issue.btcAddress`: MUST be the BTC address returned from the *tryIncreaseToBeIssuedTokens*

  - `issue.btcPublicKey`: MUST be the BTC public key returned from the *tryIncreaseToBeIssuedTokens*
  - `issue.btcHeight`: MUST be the current Bitcoin height as stored in the BTC-Relay.
  - `issue.status`: MUST be `Pending`.
- The new issue request MUST be inserted into *IssueRequests* using the generated `issueId` as the key.

13.3.2 executeIssue

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

Specification

Function Signature

`executeIssue(executorId, issueId, rawMerkleProof, rawTx)`

Parameters

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

Events

- *ExecuteIssue*
- If the amount transferred IS not equal to the `issue.amount + issue.fee`, the *IssueAmountChange* MUST be emitted

Preconditions

- The function call MUST be signed by `executor`.
- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The issue request for `issueId` MUST exist in *IssueRequests*.
- The issue request for `issueId` MUST NOT have expired.
- The `rawTx` MUST be valid and contain a payment to the Vault.
- The `rawMerkleProof` MUST be valid and prove inclusion to the main chain.
- If the amount transferred is less than `issue.amount + issue.fee`, then the `executor` MUST be the account that made the issue request.

Postconditions

- If the amount transferred IS less than the `issue.amount + issue.fee`:
  - The Vault's `toBeIssuedTokens` MUST decrease by the deficit (`issue.amount - amountTransferred`).
  - The Vault's free balance in the *Griefing Collateral Currency* MUST increase by the `griefingCollateral * (1 - amountTransferred / (issue.amount + issue.fee))`.
  - The requester's free balance in the *Griefing Collateral Currency* MUST increase by the `griefingCollateral * amountTransferred / (issue.amount + issue.fee)`.
  - The `issue.fee` MUST be updated to the amount transferred multiplied by the *IssueFee*.
  - The `issue.amount` MUST be set to the amount transferred minus the updated `issue.fee`.

- If the amount transferred IS NOT less than the expected amount:
  - The requester's free balance in the *Griefing Collateral Currency* MUST increase by the `griefingCollateral`.
  - If the amount transferred IS greater than the expected amount:
    - If the Vault IS NOT liquidated and has sufficient collateral:
      - The Vault's `toBeIssuedTokens` MUST increase by the surplus (`amountTransferred - issue.amount`).
      - The `issue.fee` MUST be updated to the amount transferred multiplied by the *IssueFee*.
      - The `issue.amount` MUST be set to the amount transferred minus the updated `issue.fee`.
    - If the Vault IS NOT liquidated and does not have sufficient collateral:
      - There MUST exist a *Refund* request which references `issueId`.
- The requester's locked balance in the *Griefing Collateral Currency* MUST decrease by `issue.griefingCollateral`.
- The `issue.status` MUST be set to `Completed`.
- The Vault's `toBeIssuedTokens` MUST decrease by `issue.amount + issue.fee`.
- The Vault's `issuedTokens` MUST increase by `issue.amount + issue.fee`.
- The user MUST receive `issue.amount` interBTC in its free balance.
- Function *distributeReward* MUST complete successfully - parameterized by `issue.fee`.

13.3.3 cancelIssue

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

Specification

Function Signature

`cancelIssue(requester, issueId)`

Parameters

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

Events

- *CancelIssue*

Preconditions

- The function call MUST be signed by `requester`.
- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- The issue request for `issueId` MUST exist in *IssueRequests*.
- The issue request MUST have expired.

Postconditions

- If the vault IS liquidated:
  - The requester's free balance oinf the *Griefing Collateral Currency* MUST increase by the `griefingCollateral`.
- If the Vault IS NOT liquidated:

  - The vault's free balance in the *Griefing Collateral Currency* MUST increase by the `griefingCollateral`.
- The requester's locked balance in the *Griefing Collateral Currency* MUST decrease by the `griefingCollateral`.
- The vault's `toBeIssuedTokens` MUST decrease by `issue.amount + issue.fee`.
- The issue status MUST be set to `Cancelled`.

13.4 Events

13.4.1 RequestIssue

Emit an event if a user successfully open a issue request.

Event Signature

`RequestIssue(issueId, requester, amount, fee, griefingCollateral, vault, btcAddress, btcPublicKey)`

Parameters

- `issueId`: A unique hash identifying the issue request.
- `requester`: The user's account identifier.
- `amount`: The amount of interBTC requested.
- `fee`: The amount of interBTC to mint as fees.
- `griefingCollateral`: The security deposit provided by the user.
- `vault`: The address of the Vault involved in this issue request.
- `btcAddress`: The Bitcoin address of the Vault.
- `btcPublicKey`: The Bitcoin public key of the Vault.

Functions

- *requestIssue*

13.4.2 IssueAmountChange

Emit an event if the issue amount changed for any reason.

Event Signature

`IssueAmountChange(issueId, amount, fee, griefingCollateral)`

Parameters

- `issueId`: A unique hash identifying the issue request.
- `amount`: The amount of interBTC requested.
- `fee`: The amount of interBTC to mint as fees.
- `griefingCollateral`: Confiscated griefing collateral.

Functions

- *executeIssue*

13.4.3 ExecuteIssue

Event Signature

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

Parameters

- `issueId`: A unique hash identifying the issue request.
- `requester`: The user's account identifier.
- `amount`: The amount of interBTC issued to the user.
- `vault`: The address of the Vault involved in this issue request.
- `fee`: The amount of interBTC minted as fees.

Functions

- *executeIssue*

13.4.4 CancelIssue

Event Signature

`CancelIssue(issueId, requester, griefingCollateral)`

Parameters

- `issueId`: the unique hash of the issue request.
- `requester`: The sender of the cancel transaction.
- `griefingCollateral`: The released griefing collateral.

Functions

- *cancelIssue*

13.5 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

FOURTEEN

REFUND

14.1 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.

14.1.1 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 se**

    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 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).

14.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

14.2 Data Model

14.2.1 Scalars

RefundBtcDustValue

The minimum amount of BTC that is required for refund requests; lower values would risk the rejection of payment on Bitcoin.

14.2.2 Maps

RefundRequests

Overpaid issue payments create refund requests to return BTC. This mapping provides access from a unique hash `RefundId` to a `Refund` struct. <`RefundId, Refund`>.

14.2.3 Structs

Refund

Stores the status and information about a single refund request.

Parameter	Type	Description
`vault`	AccountId	The account of the Vault responsible for this request.
`amountWrapped`	interBTC	Amount of interBTC to be refunded.
`fee`	interBTC	Fee charged to the user for refunding.
`amountBtc`	interBTC	Total amount that was overpaid.
`issuer`	AccountId	Account that overpaid on issue.
`btcAddress`	BtcAddress	User's Bitcoin address.
`issueId`	H256	The id of the issue request.
`completed`	bool	True if the refund was processed successfully.

14.3 External Functions

14.3.1 executeRefund

This function finalizes a refund, also referred to as a user failsafe. It is typically called by the vault client that performed the refund.

Specification

Function Signature

`executeRefund(caller, refundId, merkleProof, rawTx)`

Parameters

- `caller`: address of the user finalizing the refund. Typically the vault client that performed the refund.
- `refundId`: the unique hash created during the internal `requestRefund` function.
- `rawMerkleProof`: raw Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: raw Bitcoin transaction of the refund payment, including the transaction inputs and outputs.

Events

- *ExecuteRefund*

Preconditions

- The function call MUST be signed by *someone*, i.e., not necessarily the Vault that performed the refund.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- A *pending* `RefundRequest` MUST exist with an id equal to `refundId`.
- `refundRequest.completed` MUST be `false`.
- The `rawTx` MUST decode to a valid transaction that transfers the amount specified in the `RefundRequest` struct. It MUST be a transaction to the correct address, and provide the expected OP_RETURN, based on the `RefundRequest`.
- The `rawMerkleProof` MUST be valid and prove inclusion to the main chain.
- The `vault.status` MUST be `active`.
- The refunding vault MUST have enough collateral to mint an amount equal to the refund fee.

Postconditions

- The `vault.issuedTokens` MUST increase by `fee`.
- The vault's free balance in wrapped currency MUST increase by `fee`.
- `refundRequest.completed` MUST be `true`.

14.4 Internal Functions

14.4.1 requestRefund

Used to request a refund if too much BTC was sent to a Vault by mistake.

Specification

Function Signature

`requestRefund(amount, vault, issuer, btcAddress, issueId)`

Parameters

- `amount`: the amount that the user has overpaid.
- `vault`: id of the vault the issue was made to.
- `issuer`: id of the user that made the issue request.
- `btcAddress`: the btc address that should receive the refund.

- `issueId`: corresponding issue request which was overpaid.

Events

- *RequestRefund*

Preconditions

- The function call MUST only be called by *executeIssue*.
- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- The `amount - fee` MUST be greater than or equal to *RefundBtcDustValue*.
- A new unique `refundId` MUST be generated via the *generateSecureId* function.

Postconditions

- The new refund request MUST be created as follows:
  - `refund.vault`: MUST be the `vault`.
  - `refund.amountWrapped`: MUST be the `amount - fee`
  - `refund.fee`: MUST equal `amount` multiplied by *RefundFee*.
  - `refund.amountBtc`: MUST be the `amount`.
  - `refund.issuer`: MUST be the `issuer`.
  - `refund.btcAddress`: MUST be the `btcAddress`.
  - `refund.issueId`: MUST be the `issueId`.
  - `refund.completed`: MUST be false.
- The new refund request MUST be inserted into *RefundRequests* using the generated `refundId` as the key.

14.5 Events

14.5.1 RequestRefund

Event Signature

`RequestRefund(refundId, issuer, amount, vault, btcAddress, issueId, fee)`

Parameters

- `refundId`: A unique hash created via *generateSecureId*.
- `issuer`: The user's account identifier.
- `amount`: The amount of interBTC overpaid.
- `vault`: The address of the Vault involved in this refund request.
- `issueId`: The unique hash created during *requestIssue*.
- `fee`: The amount of interBTC to mint as fees.

14.5.2 ExecuteRefund

Event Signature

`ExecuteRefund(refundId, issuer, vault, amount, fee)`

Parameters

- `refundId`: The unique hash created during via :ref:`requestRefund`.
- `issuer`: The user's account identifier.
- `vault`: The address of the Vault involved in this refund request.
- `amount`: The amount of interBTC refunded.
- `fee`: The amount of interBTC to mint as fees.

CHAPTER

FIFTEEN

REDEEM

15.1 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 they have 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.

15.1.1 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.
    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.

15.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

15.1.3 Vault Registry

The data access and state changes to the vault registry are documented in Fig. 15.1 below.

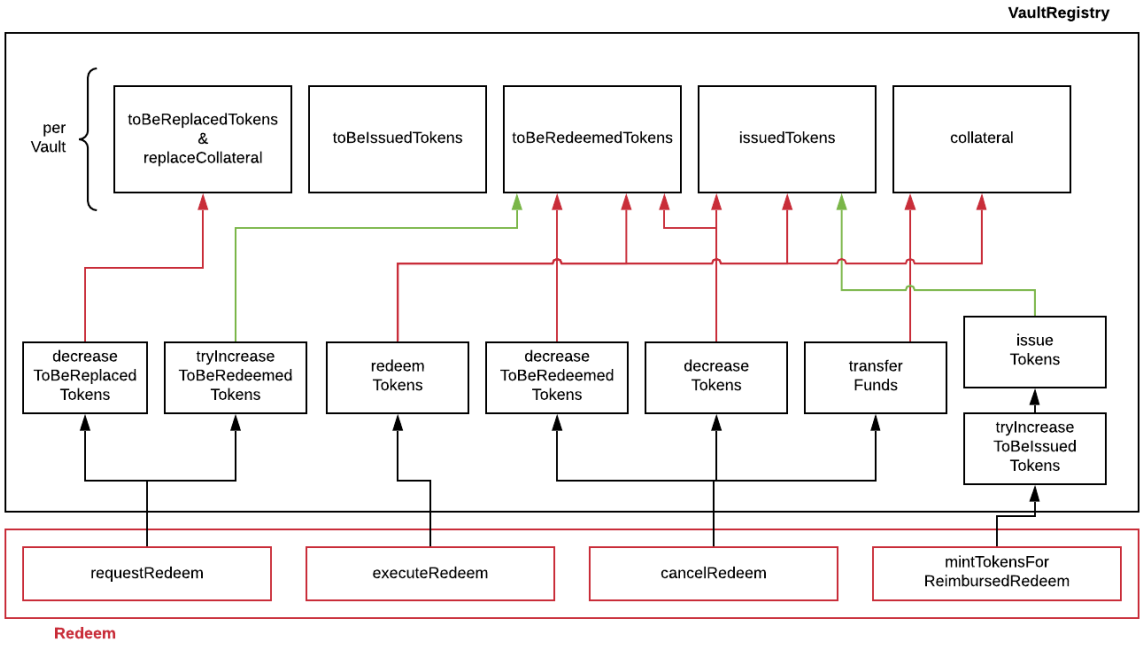


Fig. 15.1: The redeem module interacts through three different functions with the vault registry. The green arrow indicate an increase, the red arrows a decrease.

15.1.4 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` (if non-zero).
- If the vault self-redeems (the redeemer is the vault ID) no fee is paid.

15.2 Data Model

15.2.1 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.

15.2.2 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, RedeemRequest>`.

15.2.3 Structs

RedeemRequest

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.
`premium`	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

15.3 Functions

15.3.1 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, amountWrapped, btcAddress, vault)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountWrapped`: 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*

Preconditions

Let `burnedTokens` be `amountWrapped` minus the result of the multiplication of *RedeemFee* and `amountWrapped`. 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 `amountWrapped` free tokens.
- `burnedTokens` minus the inclusion fee MUST be above or equal to 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 `amountWrapped` minus the result of the multiplication of *RedeemFee* and `amountWrapped`. Then:

- The vault's `toBeRedeemedTokens` MUST increase by `burnedTokens`.
- `amountWrapped` 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` MUST be the requested `vault`
  - `redeem.opentime` MUST be the current *ActiveBlockCount*
  - `redeem.fee` MUST be *RedeemFee* multiplied by `amountWrapped` if `redeemer != vault`, otherwise this should be zero.
  - `redeem.transferFeeBtc` MUST be the inclusion fee, which is the multiplication of *RedeemTransactionSize* and the fee rate estimate reported by the oracle,
  - `redeem.amountBtc` MUST be `amountWrapped - redeem.fee - redeem.transferFeeBtc`,
  - `redeem.period` MUST be the current value of the *RedeemPeriod*,
  - `redeem.redeemer` MUST be the `redeemer` argument,
  - `redeem.btcAddress` MUST be the `btcAddress` argument,
  - `redeem.btcHeight` MUST be the current height of the btc relay,
  - `redeem.status` MUST be `Pending`,
  - If the vault's collateralization rate is above the *PremiumRedeemThreshold*, then `redeem.premium` MUST be `0`,
  - If the vault's collateralization rate is below the *PremiumRedeemThreshold*, then `redeem.premium` MUST be *PremiumRedeemFee* multiplied by the worth of `redeem.amountBtc`,

15.3.2 liquidationRedeem

A user executes a liquidation redeem that exchanges interBTC for collateral from the *LiquidationVault*. This function takes a `currencyId` argument that specifies which currency to the user wishes to receive. Since each currency uses a separate liquidation vault, the amount of collateral received depends only on the amount of tokens and collateral in that specific liquidation vault. If the user wants to obtain multiple currencies, they have to call this function multiple times, possibly through off-chain aggregation via batching. Since the 1:1 backing is being recovered in this function, interBTC is burned without releasing any BTC.

Specification

Function Signature

`liquidationRedeem(redeemer, amountWrapped, currencyId)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountWrapped`: the amount of interBTC to destroy.
- `currencyId`: the currency id of the funds to be received.

Events

- *LiquidationRedeem*

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 `amountWrapped` free tokens.

Postconditions

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

15.3.3 executeRedeem

A vault calls this function after receiving an `RequestRedeem` event with their 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(redeemId, rawMerkleProof, rawTx)`

Parameters

- `redeemId`: the unique hash created during the `requestRedeem` function.
- `rawMerkleProof`: Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- *ExecuteRedeem*

Preconditions

- The function call MUST be signed by *someone*, i.e. not necessarily the *vault*.
- 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 `rawTx` MUST decode to a valid transaction that transfers exactly 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 `rawMerkleProof` 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.amountBtc + redeemRequest.transferFeeBtc` of the tokens in the redeemer's account MUST be burned.
- The user's *lockedTokens* MUST decrease by *redeemRequest.amountBtc* + *redeemRequest.transferFeeBtc*.
- The vault's *toBeRedeemedTokens* MUST decrease by *redeemRequest.amountBtc* + *redeemRequest.transferFeeBtc*.
- The vault's *issuedTokens* MUST decrease by *redeemRequest.amountBtc* + *redeemRequest.transferFeeBtc*.
- `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`.

15.3.4 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 they receive 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.

The punishment fee paid to the user stays constant (i.e., the user always receives the punishment fee of e.g. 10%).

Specification

Function Signature

`cancelRedeem(redeemer, redeemId, reimburse)`

Parameters

- `redeemer`: account cancelling this redeem request.
- `redeemId`: the unique hash of the redeem request.
- `reimburse`: if true, user is reimbursed in collateral (slashed from the vault), else interBTC is returned (to retry with another vault).

Events

- *CancelRedeem*

Preconditions

- The function call MUST be signed by `redeemer`.
- 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 `redeemer` MUST equal `redeemRequest.redeemer`.
- The request MUST be expired.

Postconditions

Let `amountIncludingParachainFee` be equal to the worth in collateral of `redeem.amountBtc + redeem.transferFeeBtc`. Let `confiscatedCollateral` be equal to `vault.backingCollateral * (amountIncludingParachainFee / vault.toBeRedeemedTokens)`. Then:

- If the vault is liquidated:
  - If `reimburse` is true, an amount of `confiscatedCollateral` MUST be transferred from the vault to the redeemer.
  - If `reimburse` is false, an amount of `confiscatedCollateral` MUST be transferred from the vault to the liquidation vault.
- If the vault is *not* liquidated, the following 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 MUST be tranferred from the vault's backing collateral to the redeemer: *PunishmentFee*. 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 non-zero.
  - 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.amountBtc + redeem.fee + redeem.transferFeeBtc`.
  - The vault's `toBeRedeemedTokens` MUST decrease by `amountIncludingParachainFee`.
- The vault MUST be banned.

15.3.5 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

- `vault`: the vault that was unable to back the tokens.
- `redeemId`: the unique hash of the redeem request.

Events

- *MintTokensForReimbursedRedeem*

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 equal `redeemRequest.vault`.
- The vault MUST have sufficient collateral to remain above the *SecureCollateralThreshold* after issuing `redeem.amountBtc + redeem.transferFeeBtc` tokens.
- The function call MUST be signed by `redeem.vault`, i.e. this function can only be called by the vault.

Postconditions

- *tryIncreaseToBeIssuedTokens* and *issueTokens* MUST be called, both with the vault and `redeem.amountBtc + redeem.transferFeeBtc` as arguments.
- `redeem.amountBtc + redeem.transferFeeBtc` tokens MUST be minted to the vault.
- The `redeem.status` MUST be set to `Reimbursed(true)`.

15.4 Events

15.4.1 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, amountWrapped, feeWrapped, premium, vaultId, userBtcAddress, transferFeeBtc)`

Parameters

- `redeemID`: the unique identifier of this redeem request.
- `redeemer`: address of the user triggering the redeem.
- `amountWrapped`: the amount to be received by the user.
- `feeWrapped`: the fee to be paid to the reward 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*

15.4.2 LiquidationRedeem

Emit an event when a user does a liquidation redeem.

Event Signature

`LiquidationRedeem(redeemer, amountWrapped)`

Parameters

- `redeemer`: address of the user triggering the redeem.
- `amountWrapped`: the amount of interBTC to burned.

Functions

- ref:*liquidationRedeem*

15.4.3 ExecuteRedeem

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

Event Signature

`ExecuteRedeem(redeemId, redeemer, amountWrapped, feeWrapped, vault, transferFeeBtc)`

Parameters

- `redeemId`: the unique hash created during the `requestRedeem` function.
- `redeemer`: address of the user triggering the redeem.
- `amountWrapped`: the amount of interBTC to destroy and BTC to receive.
- `feeWrapped`: the amount of interBTC taken for fees.
- `vault`: the vault responsible for executing this redeem request.
- `transferFeeBtc`: the budget for the bitcoin inclusion fees, paid for by the user.

Functions

- ref:*executeRedeem*

15.4.4 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, vault, amountSlashed, status)`

Parameters

- `redeemId`: the unique hash of the redeem request.
- `redeemer`: The redeemer starting the redeem process.

- `vault`: the vault who failed to execute the redeem.
- `amountSlashed`: the amount that was slashed from the vault.
- `status`: the status of the redeem request.

Functions

- ref:*cancelRedeem*

15.4.5 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*

15.5 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

SIXTEEN

REPLACE

16.1 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. The DOT collateral of the *oldVault*, corresponding to the amount of replaced BTC, is then unlocked. The *oldVault* must provide griefing collateral for spam protection which is paid to *newVault* on failure.

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.

16.1.1 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 replace 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: *oldVault* can withdraw the request by calling the *withdrawReplace* function with a specified 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.
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*.

16.1.2 Security

- Unique identification of Bitcoin payments: *OP_RETURN*

16.1.3 Vault Registry

The data access and state changes to the *Vault Registry* are documented in Fig. 16.1 below.

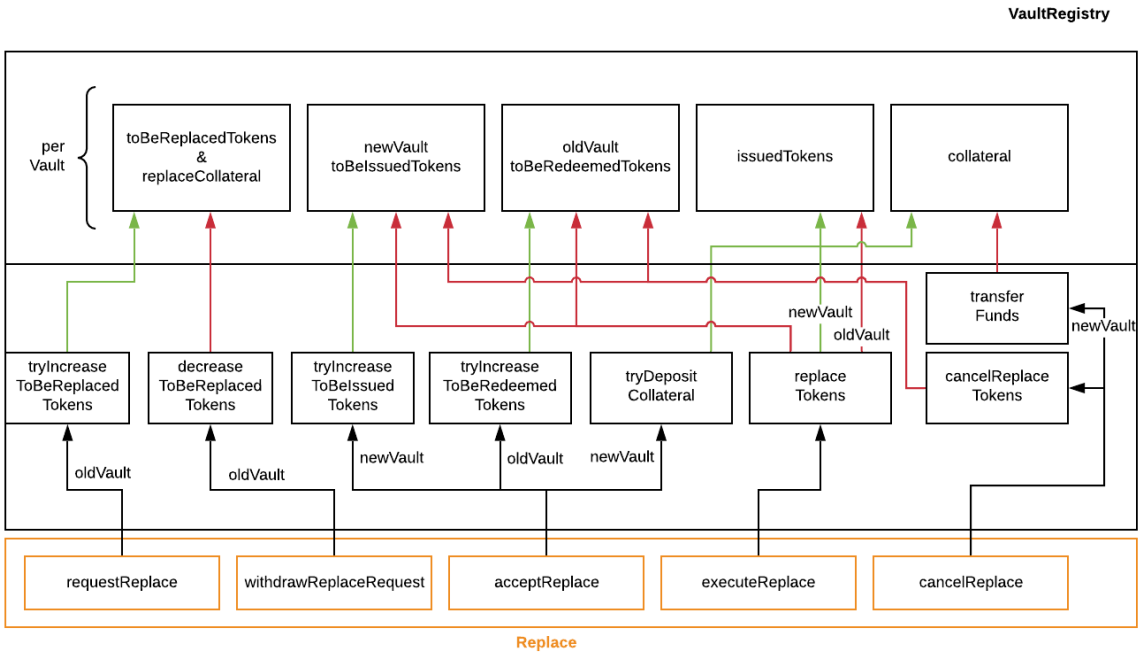


Fig. 16.1: The replace module interacts with functions in the Vault-Registry to handle updating token balances of vaults. The green lines indicate an increase, the red lines a decrease.

16.1.4 Fee Model

- If a replace request is cancelled, the griefing collateral is transferred to the *newVault*.
- If a replace request is executed, the griefing collateral is transferred to the *oldVault*.

16.2 Data Model

16.2.1 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.

16.2.2 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>`.

16.2.3 Structs

Replace

Stores the status and information about a single replace request.

Parameter	Type	Description
`oldVault`	AccountId	Account of the *oldVault* that is to be replaced.
`newVault`	AccountId	Account of the *newVault*, which accepts the replace request.
`amount`	interBTC	Amount of BTC / interBTC to be replaced.
`griefingCollateral`	DOT	Griefing protection collateral locked by *oldVault*.
`collateral`	DOT	Collateral locked by the new Vault.
`acceptTime`	BlockNumber	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.
`period`	BlockNumber	Value of *ReplacePeriod* when the redeem request was made, in case that value changes while this replace is pending.
`btcAddress`	BtcAddress	Vault's Bitcoin payment address.
`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	Status of the request: Pending, Completed or Cancelled

16.3 Functions

16.3.1 requestReplace

The *oldVault* (to-be-replaced) submits a request to be (partially) replaced. If it requests more than it can fulfill (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*

Preconditions

- The function call MUST be signed by *oldVault*.
- The BTC Parachain status in the *Security* component MUST be set to `RUNNING:0`.
- The *oldVault* MUST be registered.
- The *oldVault* MUST NOT be banned.
- The *oldVault* MUST NOT be nominated (if *Vault Nomination* is enabled).
- If the `btcAmount` is greater than the Vault's `replacableTokens = issuedTokens - toBeRedeemTokens - toBeReplaceTokens`, set the `btcAmount` to the `replaceableTokens` amount.
- The *oldVault* MUST provide sufficient `griefingCollateral` such that the ratio of all of its `toBeReplacedTokens` and `replaceCollateral` is above *ReplaceGriefingCollateral*.

- The *oldVault* MUST request sufficient `btcAmount` to be replaced such that its total is above `ReplaceBtcDustValue`.

Postconditions

- The *oldVault*'s `toBeReplacedTokens` MUST be increased by `tokenIncrease = min(btcAmount, vault.toBeIssuedTokens - vault.toBeRedeemedTokens)`.
- An amount of `griefingCollateral * (tokenIncrease / btcAmount)` MUST be locked in the *Griefing Collateral Currency* by the *oldVault* in this transaction.
- The *oldVault*'s `replaceCollateral` MUST be increased by the amount of collateral locked in this transaction.

16.3.2 withdrawReplace

The *oldVault* decreases its `toBeReplacedTokens`.

Specification

Function Signature

`withdrawReplace(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 `toBeReplacedTokens` to withdraw.

Events

- *WithdrawReplace*

Preconditions

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

Postconditions

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

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

Parameters

- `oldVault`: Account identifier of the *oldVault* who requested replacement (as tracked in `Vaults` in *Vault Registry*).
- `newVault`: Account identifier of the *newVault* 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.
- `btcAddress`: The *newVault*'s Bitcoin payment address for transaction verification.

Events

- *AcceptReplace*

Preconditions

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

Postconditions

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

- The *oldVault*'s `replaceCollateral` MUST be decreased by `consumedGriefingCollateral`.
- The *oldVault*'s `toBeReplacedTokens` MUST be decreased by `redeemableTokens`.
- The *oldVault*'s `toBeRedeemedTokens` MUST be increased by `redeemableTokens`.
- The *newVault*'s `toBeIssuedTokens` MUST be increased by `redeemableTokens`.
- The *newVault* locks additional collateral; its `backingCollateral` MUST be increased by `collateral * (redeemableTokens / oldVault.toBeReplacedTokens)`.
- A unique *replaceId* must be generated from *generateSecureId*.

- A new `ReplaceRequest` MUST be added to the replace request mapping at the *replaceId* key.
  - `oldVault`: MUST be the `oldVault`.
  - `newVault`: MUST be the `newVault`.
  - `amount`: MUST be `redeemableTokens`.
  - `griefingCollateral`: MUST be `consumedGriefingCollateral`
  - `collateral`: MUST be `collateral`.
  - `accept_time`: MUST be the current active block number.
  - `period`: MUST be the current `ReplacePeriod`.
  - `btcAddress`: MUST be the `btcAddress` argument.
  - `btcHeight`: UST be the current height of the btc-relay.
  - `status`: MUST be `pending`.

16.3.4 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(replaceId, rawMerkleProof, rawTx)`

Parameters

- `replaceId`: The identifier of the replace request in `ReplaceRequests`.
- `rawMerkleProof`: Raw Merkle tree path (concatenated LE SHA256 hashes).
- `rawTx`: Raw Bitcoin transaction including the transaction inputs and outputs.

Events

- *ExecuteReplace*

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- Both *oldVault* and *newVault* (as specified in the request) MUST be registered in the *Vault Registry*.
- A pending `ReplaceRequest` MUST exist with id `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 `replaceId`.
- The `rawMerkleProof` 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

- The *replaceTokens* function in the *Vault Registry* MUST have been called, providing the `oldVault`, `newVault`, `replaceRequest.amount`, and `replaceRequest.collateral` as arguments.
- The griefing collateral as specifiied in the `ReplaceRequest` MUST be released back to *oldVault*'s free balance in the *Griefing Collateral Currency*.

- The `replaceRequest.status` MUST be set to `Completed`.

16.3.5 cancelReplace

If a replace request is not executed on time, the replace can be cancelled by the *newVault*. Since the *newVault* provided additional collateral in vain, it can claim the *oldVault*'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*

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be set to `SHUTDOWN:2`.
- Both *oldVault* and *newVault* (as specified in the request) MUST be registered in the *Vault Registry*.
- A pending `ReplaceRequest` MUST exist with id `replaceId`.
- The `newVault` MUST be equal to the *newVault* specified in the `ReplaceRequest`.
- The request MUST have expired.

Postconditions

- The *cancelReplaceTokens* function in the *Vault Registry* MUST have been called, providing the `oldVault`, `newVault`, `replaceRequest.amount`, and `replaceRequest.amount` as arguments.
- If *newVault* IS NOT liquidated:
  - If unlocking `replaceRequest.collateral` does not put the collateralization rate of the *newVault* below `SecureCollateralThreshold`, the collateral MUST be unlocked and its `backingCollateral` MUST decrease by the same amount.
- The griefing collateral MUST BE slashed from the *oldVault* to the *newVault*'s free balance.
- The `replaceRequest.status` MUST be set to `Cancelled`.

16.4 Events

16.4.1 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*

16.4.2 WithdrawReplace

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

Event Signature

`WithdrawReplace(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:*withdrawReplace*

16.4.3 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*

16.4.4 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*

16.4.5 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*

16.5 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

SEVENTEEN

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. **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.

17.1 Overview

17.1.1 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.

17.1.2 Oracle Offline

The *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`

17.2 Data Model

17.2.1 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.

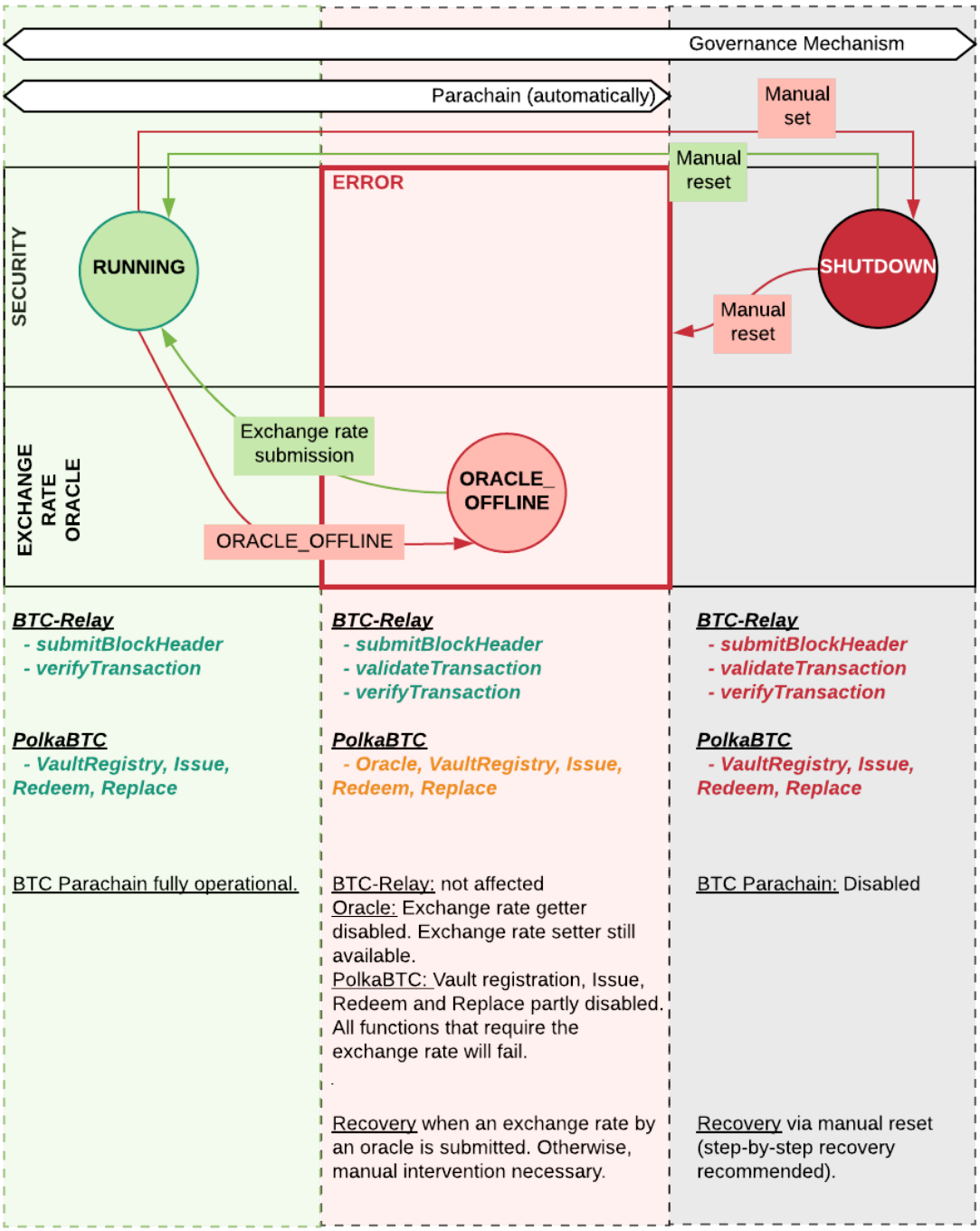


Fig. 17.1: (Informal) State machine showing the operational and failure modes and how to recover from or flag failures.

ErrorCode

Enum specifying error codes tracked in `Errors`.

- `NONE: 0`
- `ORACLE_OFFLINE: 1`

17.3 Data Storage

17.3.1 Scalars

ParachainStatus

Stores the status code (*StatusCode*) which defines the current state of the BTC Parachain.

Errors

Stores the set of error codes (*ErrorCode*), indicating the reason for the error.

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 when 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. This field MUST be set to the current block height on initialization.

17.4 Functions

17.4.1 generateSecureId

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

Specification

Function Signature

`generateSecureId(account)`

Parameters

- `account`: 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*).

Preconditions

- A parent block MUST exist (cannot be called on the parachain genesis block).

Postconditions

- Nonce MUST be incremented by one.
- MUST return the 256-bit hash of the `account`, ``nonce`, and `parent_hash` (the hash of the previous block of this transaction).

17.4.2 hasExpired

Checks if the `period` has expired since the `opentime`. This calculation is based on the *ActiveBlockCount*.

Specification

Function Signature

`hasExpired(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.

Preconditions

- The *ActiveBlockCount* MUST be greater than 0.

Postconditions

- MUST return `True` if `opentime + period < ActiveBlockCount`, `False` otherwise.

17.4.3 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.

17.4.4 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.

17.4.5 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.

17.5 Events

17.5.1 RecoverFromErrors

Event Signature

`RecoverFromErrors(StatusCode, ErrorCode[])`

Parameters

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

CHAPTER

EIGHTEEN

RELAY

The *Relay* module is responsible for handling theft reporting and block submission to the *BTC-Relay*.

18.1 Overview

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

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

In the second case, the module should check the accusation (using a Merkle proof), and liquidate the vault if valid. It is assumed that there is at least one honest relayer.

The Governance Mechanism 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*.

18.2 Data Storage

18.2.1 Maps

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.

18.3 Functions

18.3.1 report_vault_theft

A 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

`report_vault_theft(vault, raw_merkle_proof, raw_tx)`

Parameters

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

The `txId` is obtained as the `sha256d()` of the `raw_tx`.

Events

- *ReportVaultTheft*

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- A vault with id `vaultId` MUST be registered.
- The txId MUST NOT be in `TheftReports` mapping.
- The `verifyTransactionInclusion` function in the *BTC-Relay* component must return true for the derived `txId`.

Postconditions

- The vault MUST be liquidated.
- The vault's status MUST be set to `CommittedTheft`.
- All token accounts (`issuedTokens`, `toBeIssuedTokens`, etc.) MUST be added to the existing system's `LiquidationVault`.
- `TheftReports` MUST contain the reported txId.

18.3.2 report_vault_double_payment

A relayer reports a double payment from a vault, this can destabalize the system if the vault holds less BTC than is reported by the *Vault Registry*.

Like in *report_vault_theft*, if the vault actually misbehaved it is automatically liquidated.

Specification

Function Signature

`report_vault_double_payment(vault, raw_merkle_proof1, raw_tx1, raw_merkle_proof2, raw_tx2)`

Parameters

- `vaultId`: the account of the accused Vault.
- `raw_merkle_proof1`: The first raw Merkle tree path.
- `raw_tx1`: The first raw Bitcoin transaction.
- `raw_merkle_proof2`: The second raw Merkle tree path.
- `raw_tx2`: The second raw Bitcoin transaction.

Events

- *ReportVaultTheft*

Preconditions

- The BTC Parachain status in the *Security* component MUST NOT be `SHUTDOWN:2`.
- A vault with id `vaultId` MUST be registered.
- `raw_merkle_proof1` MUST NOT equal `raw_merkle_proof2`.
- `raw_tx1` MUST NOT equal `raw_tx2`.
- The `verifyTransactionInclusion` function in the *BTC-Relay* component must return true for the derived `txId`.
- Both transactions MUST NOT be in `TheftReports` mapping.

Postconditions

- The vault MUST be liquidated if both transactions contain the same `OP_RETURN` value.
- The vault's status MUST be set to `CommittedTheft`.
- All token accounts (`issuedTokens`, `toBeIssuedTokens`, etc.) MUST be added to the existing system's `LiquidationVault`.
- `TheftReports` MUST contain the reported transactions.

18.4 Events

18.4.1 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

- *report_vault_theft*
- *report_vault_double_payment*

CHAPTER

NINETEEN

TREASURY

19.1 Overview

Conceptually, the treasury serves as both the central storage for all interBTC and the interface though which to manage interBTC amount. It is implemented through the *Currency* pallet.

There are three main operations on interBTC to interact with the user or the *Issue* and *Redeem* components.

19.1.1 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 (via *executeIssue*) and the user has provided a valid proof that the required amount of BTC was sent to the correct vault. The issue module calls the *mint_to* function to create interBTC.
- **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_on* function that locks the requested amount of tokens for this user. Second, when a redeem request is completed (via *executeRedeem*) 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_from* function to destroy the previously locked interBTC.

CHAPTER

TWENTY

VAULT REGISTRY

20.1 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. Moreover, vaults use the registry to register public key for the *On-Chain Key Derivation Scheme* and register addresses for the *OP_RETURN* scheme.

20.1.1 Multi-Collateral

The parachain supports the usage of different currencies for usage as collateral. Which currencies are allowed is determined by governance - they have to explicitly white-list currencies to be able to be used as collateral. They also have to set the various safety thresholds for each currency.

Vaults in the system are identified by a VaultId, which is essentially a (AccountId, CollateralCurrency, WrappedCurrency) tuple. Note the distinction between the AccountId and the VaultId. A vault operator can run multiple vaults using a the same AccountId but different collateral currencies (and thus VaultIds). Each vault is isolated from all others. This means that if vault operator has two running vaults using the same AccountId but different CollateralCurrencies, then if one of the vaults were to get liquidated, the other vaults remains untouched. The vault client manages all VaultIds associated with a given AccountId. Vault operators will be able to register new VaultIds through the UI, and the vault client will automatically start to manage these.

When a user requests an issue, it selects a single vault to issue with (this choice may be made automatically by the UI). However, since the wrapped token is fully fungible, it may be redeemed with any vault, even if that vault is using a different collateral currency. When redeeming, the user again selects a single vault to redeem with. If a vault fails to execute a redeem request, the user is able to either get back its wrapped token, or to get reimbursed in the vault's collateral currency. If the user prefers the latter, the choice of vault becomes relevant because it determines which currency is received in case of failure.

The WrappedCurrency part of the VaultId is currently always required to take the same value - in the future support for different wrapped currencies may be added.

Moreover, the system implements a ceiling for the maximum amount of collateral than can be locked in the system per collateral and wrapped token pair. Governance is able to update the collateral ceilings.

Note: Please note that multi-collateral is a recent addition to the code, and the spec has not been fully updated .

20.2 Data Model

20.2.1 Scalars

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)

LiquidationVaultAccountId

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 `LiquidationVaultAccountId` and claims are later handled via the `LiquidationVault`.

20.2.2 Maps

LiquidationVault

Mapping from `CurrencyId` to the account identifier of an artificial vault (see *SystemVault*) maintained by the VaultRegistry to handle interBTC balances and collateral of liquidated Vaults that use the given currency. 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 collateral are transferred to the `LiquidationVault` as a result of automated liquidations and *report_vault_theft*.

MinimumCollateralVault

Mapping from `CurrencyId` to the minimum collateral a vault needs to provide to register.

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.

SecureCollateralThreshold

Mapping from `CurrencyId` to to the over-collateralization rate for collateral locked by Vaults, necessary for issuing tokens.

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:

$\max(\text{interBTC}) = \text{collateral} * \text{ExchangeRate} / \text{SecureCollateralThreshold}$.

- The Secure Collateral Threshold MUST be greater than the Liquidation Threshold.
- The Secure Collateral Threshold MUST be greater than the Premium Redeem Threshold.

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

Mapping from `CurrencyId` to the the collateral rate of Vaults, at which users receive a premium, allocated from the Vault's collateral, when performing a *Redeem* with this Vault.

- The Premium Redeem Threshold MUST be greater than the Liquidation Threshold.

LiquidationThreshold

Mapping from `CurrencyId` to the lower bound for the collateral rate in issued tokens. If a Vault's collateral rate drops below this, automatic liquidation is triggered.

- The Liquidation Threshold MUST be greater than 100% for any collateral asset.

SystemCollateralCeiling

Mapping from a collateral `CurrencyId` to a wrapped `CurrencyId`. Determines the maximum amount of collateral that Vaults are able to lock for backing a wrapped asset.

Vaults

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

20.2.3 Structs

Vault

Stores the information of a Vault.

Parameter	Type	Description
`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.
`status`	VaultStatus	Current status of the vault (Active, Liquidated, CommittedTheft)
`bannedUntil`	BlockNumber	Block height until which this vault is banned from being used for Issue, Redeem (except during automatic liquidation) and Replace .
`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.
`toBeReplacedTokens`	interBTC	Number of interBTC tokens requested for replacement.
`replaceCollateral`	DOT	Griefing collateral to be used for accepted replace requests.
`liquidatedCollateral`	DOT	Any collateral that is locked for remaining to_be_redeemed on liquidation.
`currencyId`	CurrencyId	The currency the vault uses for collateral

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.

SystemVault

A system vault that keeps track of tokens of liquidated vaults.

Parameter	Type	Description
`toBeIssuedtokens`	interBTC	Number of tokens pending issue
`issuedTokens`	interBTC	Number of issued tokens
`toBeRedeemedTokens`	interBTC	Number of tokens pending redeem
`currencyId`	CurrencyId	the currency used for collateral

20.3 External Functions

20.3.1 register_vault

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

`register_vault(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*.
- `currencyId`: the currency that the vault will use as collateral.

Events

- *RegisterVault*

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 vault's free balance in the given currency MUST decrease by `collateral`.
- The vault's reserved balance MUST in the given currency increase by `collateral`.
- The new vault MUST be created as follows:
  - `vault.wallet`: MUST be empty.
  - `vault.status`: MUST be set to `active=true`.
  - `vault.bannedUntil`: MUST be empty.
  - `vault.toBeIssuedTokens`: MUST be zero.
  - `vault.issuedTokens`: MUST be zero.
  - `vault.toBeRedeemedTokens`: MUST be zero.
  - `vault.toBeReplacedTokens`: MUST be zero.
  - `vault.replaceCollateral`: MUST be zero.
  - `vault.liquidatedCollateral`: MUST be zero.
  - `vault.currencyId`: MUST be the supplied `currencyId`
- The new vault MUST be inserted into *Vaults* using their account identifier as key.

20.3.2 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*

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` MUST be added to the vault's wallet.

20.3.3 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*

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 MUST be set to `publicKey`.

20.3.4 deposit_collateral

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

Specification

Function Signature

`deposit_collateral(vaultId, collateral)`

Parameters

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

Events

- *DepositCollateral*

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 in the currency determined by `vault.currencyId` to lock.

Postconditions

- Function *depositStake* MUST complete successfully - parameterized by `vaultId` and `collateral`.
- The vault MUST lock an amount of `collateral` of its collateral, using the currency set in `vault.currencyId`.

20.3.5 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*

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`.
- After the withdrawal, the vault's ratio of nominated collateral to own collateral must remain above the value returned by *getMaxNominationRatio*.

Postconditions

- Function *withdrawStake* MUST complete successfully - parameterized by `vaultId` and `withdrawAmount`.
- The vault's free balance in the currency configured by `vault.currencyID` MUST increase by `withdrawAmount`.
- The vault's locked balance in the currency configured by `vault.currencyID` MUST decrease by `withdrawAmount`.

20.4 Internal Functions

20.4.1 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 *get_price*).

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*

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` MUST be increased by an amount of `tokens`.

20.4.2 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*

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` MUST be decreased by an amount of `tokens`.
- If the vault *is* liquidated, the liquidation vault's `toBeIssuedTokens` MUST be decreased by an amount of `tokens`.

20.4.3 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*

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` MUST be decreased by `tokens`, while its `issuedTokens` MUST be increased by `tokens`.
- If the vault is *not* liquidated, function *depositStake* MUST complete successfully - parameterized by `vaultId` and `tokens`.
- If the vault *is* liquidated, the `toBeIssuedTokens` of the liquidation vault MUST be decreased by `tokens`, while its `issuedTokens` MUST be increased by `tokens`.

20.4.4 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*

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` MUST be increased by `tokens`.

20.4.5 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*

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` MUST be decreased by `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` of the liquidation vault MUST be decreased by `tokens`.

20.4.6 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*

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` MUST be decreased by `tokens`.
- If the vault *is* liquidated, the `toBeRedeemedTokens` and `issuedTokens` of the liquidation vault MUST be decreased by `tokens`.

20.4.7 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*
- *RedeemTokensPremium*
- *RedeemTokensLiquidatedVault*

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` (as calculated via *computeStakeAtIndex*) 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:
  - If `premium > 0`, then `premium` MUST be transferred from the vault's collateral to the redeemer's free balance.
  - Function *withdrawStake* MUST complete successfully - parameterized by `vaultId` and `tokens`.
- If the vault *IS* liquidated:
  - The amount `toBeReleased` is calculated as `(vault.liquidatedCollateral * tokens) / vault.toBeRedeemedTokens`.
  - The vault's `liquidatedCollateral` MUST decrease by `toBeReleased`.
  - Function *depositStake* MUST complete successfully - parameterized by `vaultId`, `vaultId`, and `toBeReleased`.
- The vault's `toBeRedeemedTokens` MUST decrease by `tokens`.

- The vault's `issuedTokens` MUST decrease by `tokens`.

20.4.8 redeemTokensLiquidation

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

Specification

Function Signature

`redeemTokensLiquidation(redeemerId, tokens, currencyId)`

Parameters

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

Events

- *RedeemTokensLiquidation*

Preconditions

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

Postconditions

- The used liquidation vault MUST be the one with the given `currencyId`.
- The liquidation vault's `issuedTokens` MUST decrease by `tokens`.
- The redeemer MUST have received an amount of collateral equal to `(tokens / liquidationVault.issuedTokens) * liquidationVault.backingCollateral`.

20.4.9 increaseToBeReplacedTokens

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

Specification

Function Signature

`increaseToBeReplacedTokens(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 tuple of the new total `toBeReplacedTokens` and `replaceCollateral`.

Events

- *IncreaseToBeReplacedTokens*

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` MUST be increased by `tokens`.
- The vault's `replaceCollateral` MUST be increased by `collateral`.

20.4.10 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 tuple of the new total `toBeReplacedTokens` and `replaceCollateral`.

Events

- *DecreaseToBeReplacedTokens*

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

20.4.11 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*

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 the `oldVault` *IS* liquidated:
  - The amount `toBeReleased` MUST be calculated as `(oldVault.liquidatedCollateral * tokens) / oldVault.toBeRedeemedTokens`.
  - The `oldVault`'s `liquidatedCollateral` MUST decrease by `toBeReleased`.
  - Function *depositStake* MUST complete successfully - parameterized by `oldVault`, `oldVault` and `toBeReleased`.
- The `oldVault`'s `toBeRedeemed` MUST decrease by `tokens`.
- The `oldVault`'s `issuedTokens` MUST decrease by `tokens`.
- The `newVault`'s `toBeIssuedTokens` MUST decrease by `tokens`.
- The `newVault`'s `issuedTokens` MUST increase by `tokens`.

20.4.12 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.

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

20.4.13 liquidateVault

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

Specification

Function Signature

`liquidateVault(vault, reporter)`

Parameters

- `vault`: Account identifier of the vault to be liquidated.
- `reporter`: [Optional] Account that initiated the liquidation (e.g. theft report).

Events

- *LiquidateVault*

Preconditions

Postconditions

- `usedCollateral` MUST be calculated as `exchangeRate * (issuedTokens + toBeIssuedTokens)) * secureCollateralThreshold`.
- `usedCollateral` MUST be set to `backingCollateral` if `backingCollateral < usedCollateral`.
- `usedTokens` MUST be calculated as `issuedTokens + toBeIssuedTokens`.
- `toBeLiquidated` MUST be calculated as `(usedCollateral * (usedTokens - toBeRedeemedTokens)) / usedTokens`.
- `remainingCollateral` MUST be calculated as `max(0, usedCollateral - toBeLiquidated)`.
- Function *withdrawStake* MUST complete successfully - parameterized by `vault` and `issuedTokens`.
- Function *withdrawStake* MUST complete successfully - parameterized by `vault` and `remainingCollateral`.
- `liquidatedCollateral` MUST be increased by `remainingCollateral`.
- `toWithdraw` MUST be calculated as `toBeLiquidated - backingCollateral` OR `toBeLiquidated` if `backingCollateral > toBeLiquidated`.
- `toSlash` MUST be calculated as the remainder of the previous calculation.
- Function *withdrawStake* MUST complete successfully - parameterized by `vault` and `toWithdraw`.
- Function *slashStake* MUST complete successfully - parameterized by `vault` and `toSlash`.
- The liquidation vault MUST be updated as follows:
  - `liquidationVault.issuedTokens` MUST increase by `vault.issuedTokens`
  - `liquidationVault.toBeIssuedTokens` MUST increase by `vault.toBeIssuedTokens`
  - `liquidationVault.toBeRedeemedTokens` MUST increase by `vault.toBeRedeemedTokens`
- The vault MUST be updated as follows:
  - `vault.issuedTokens` MUST be set to zero
  - `vault.toBeIssuedTokens` MUST be set to zero
- If *reporter* IS specified, *min(TheftFee(liquidatedAmountinBTC), TheftFeeMax)* MUST be transferred from the liquidated vault to the `reporter`.

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

20.4.14 getMaxNominationRatio

Returns the nomination ratio, denoting the maximum amount of collateral that can be nominated in a given currency.

- `MaxNominationRatio = (SecureCollateralThreshold / PremiumRedeemThreshold) - 1)`

Example

- `SecureCollateralThreshold = 1.5 (150%)`
- `PremiumRedeemThreshold = 1.2 (120%)`
- `MaxNominationRatio = (1.5 / 1.2) - 1 = 0.25 (25%)`

In this example, a Vault with 10 DOT locked as collateral can only receive 2.5 DOT through nomination.

20.5 Events

20.5.1 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`: the amount of the to-be-locked collateral.

Functions

- *register_vault*

20.5.2 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

- *deposit_collateral*

20.5.3 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*

20.5.4 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*

20.5.5 UpdatePublicKey

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

Event Signature

`UpdatePublicKey(vault, publicKey)`

Parameters

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

Functions

- *updatePublicKey*

20.5.6 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*

20.5.7 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*

20.5.8 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*

20.5.9 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*

20.5.10 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 redeemed.

Functions

- *decreaseToBeRedeemedTokens*

20.5.11 IncreaseToBeReplacedTokens

Emit an event when the `toBeReplacedTokens` is increased.

Event Signature

`IncreaseToBeReplacedTokens(vault, tokens)`

Parameters

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

Functions

- *increaseToBeReplacedTokens*

20.5.12 DecreaseToBeReplacedTokens

Emit an event when the `toBeReplacedTokens` is decreased.

Event Signature

`DecreaseToBeReplacedTokens(vault, tokens)`

Parameters

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

Functions

- *decreaseToBeReplacedTokens*

20.5.13 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*

20.5.14 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*

20.5.15 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*

20.5.16 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*

20.5.17 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*

20.5.18 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*

20.5.19 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*

20.6 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

TWENTYONE

VAULT NOMINATION

21.1 Overview

Nomination is a feature aimed at increasing *interBTC* issuance capacity by allowing Nominators to back a particular Vault. Nominators lock their free collateral so that trusted Vaults can issue *interBTC* backed by the nominated collateral. Nominators are rewarded a fraction of the fees generated by the Vault, while the remaining fees are given to the Vault. Vaults are assumed to be trusted by their Nominators not to steal Bitcoin backed by nominated collateral.

21.1.1 Step-by-step

1. Vaults may opt in to Nomination, expanding the total possible issuance amount.
2. The maximum amount that can be nominated is bounded by the Vault's locked collateral.
3. Nominators select one or more Vaults and lock their collateral balance on the BTC Parachain. The used currency for each nomination is determined by the vault's collateral currency. If a nominator is nominating different vaults, a different currency can be used for each vault, i.e., a nominator is not limited to a single currency.
4. Nominators can go offline and their nominated collateral will generate rewards passively.
5. Vaults and Nominators can withdraw their collateral at any point subject to the `SecureCollateralThreshold`.
6. Upon liquidation, Nominators are returned some collateral after remaining requests have been executed.

21.2 Protocol

21.2.1 Assumptions

Security Assumptions

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.

Liveness Assumptions

1. Nominators are mostly-offline agents, who are slow to respond to system changes.
2. Vaults are always-online agents, who can promptly react to system updates.

21.2.2 Vault Nomination Protocol

1. Vaults must choose to opt in to the Nomination protocol.
2. Nominators select a Vault to which they can delegate collateral. They will earn a fraction of any rewards generated by the Vault.
3. Vault replacement is disallowed with nominated collateral. Otherwise, Security Assumptions 1 and 2 would be violated.
4. The nominated collateral:
    1. Is in the vault's collateral currency,
    2. Cannot be withdrawn by the Vault,
    3. Is locked on the parachain,
    4. Is capped at a fraction of the Vault's deposited collateral (*Max Nomination Ratio*) to bound the risk for both Vaults and Nominators.
5. Liquidation slashing is handled as follows:
    1. **Proportional Slashing** In case the collateral managed by the Vault falls below the liquidation threshold, the Vault and its Nominators are slashed proportionally to their collateral.
    2. **Vault First Slashing** In case the Vault steals Bitcoin deposited at its address, its collateral is used to cover as much of the slashed amount as possible. If the Vault's collateral was not enough to cover the entire amount, the Nominators are slashed proportionally for the remaining amount.
6. Vaults may opt out of the Nomination protocol which force refunds Nominators if there is enough collateral over the `SecureCollateralThreshold`.

Max Nomination Ratio

This ratio prevents the Vault from withdrawing its entire collateral and only exposing Nominators to economic risk, or stealing without liquidation consequences. This means that a Vault can only withdraw collateral as long as the fraction of nominated collateral does not exceed the threshold cap. Capping Nominator collateral also prevents Vaults being "outnumbered" by Nominators and their relative fee earnings being marginalized. The calculation is defined in *Vault Registry*, in the `getMaxNominationRatio` function.

21.2.3 Security Considerations

The Vault Nomination protocol changes the economic incentive for Vaults to misbehave, i.e., violate the security of the XCLAIM protocol by stealing BTC.

Economic Security without Vault Nomination

Informally and not considering reputation or rewards, a rational Vault should not steal BTC if the economic value of the collateral is above the value of the BTC held in custody. The *effective collateralization* rate at which Vaults should steal BTC is below 100%. More formally, we can express this as:

$$C > b$$

Where C is the value of the locked collateral (e.g., DOT) and b is the value of the backing asset (e.g., BTC). Note, that we will add an extension to this model such that we account for the *expected* value from the perspective of the Vault for both assets. However, for this simple model, the above should suffice.

As an aside, Vaults are liquidated before reaching 100% collateralization as defined by the *LiquidationThreshold*.

Economic Security with Vault Nomination

Introducing Vault Nomination changes the effective collateralization rate at which Vaults have an economic incentive to steal BTC.

In both, the Vault First and Proportional Slashing, the effective collateralization rate at which Vaults should steal can be calculated by considering that if the value of *only* the Vault's collateral is below 100% of the locked BTC, a Vault has an incentive to steal BTC. We can then calculate the effective collateralization, under the assumption that a Vault is fully nominated, by taking the 100% collateralization provided by the Vault and adding the *Max Nomination Ratio*:

$$100\% + (100\% * \text{maxNominationRatio})$$

Note: If we take DOT as an example and use a secure collateral ratio of 150% and a premium redeem threshold of 135%, Vaults have an incentive to steal BTC if their collateralization falls below 125%.

Above the effective collateralization rate to steal BTC, the incentives to violate the security of the system (i.e., being under-collateralized *or* steal BTC), are different depending on the slashing strategy.

Proportional Slashing

For the under-collateralization failure, both Nominators and Vaults need to be active to (1) add more collateral to prevent such a failure, (2) reduce the amount of backed tokens, i.e., the number of backed interBTC, or (3) a combination of 1 and 2. In this strategy each the Vault and its Nominators are punished proportionally to their collateral holdings. We visualize this with the example below:

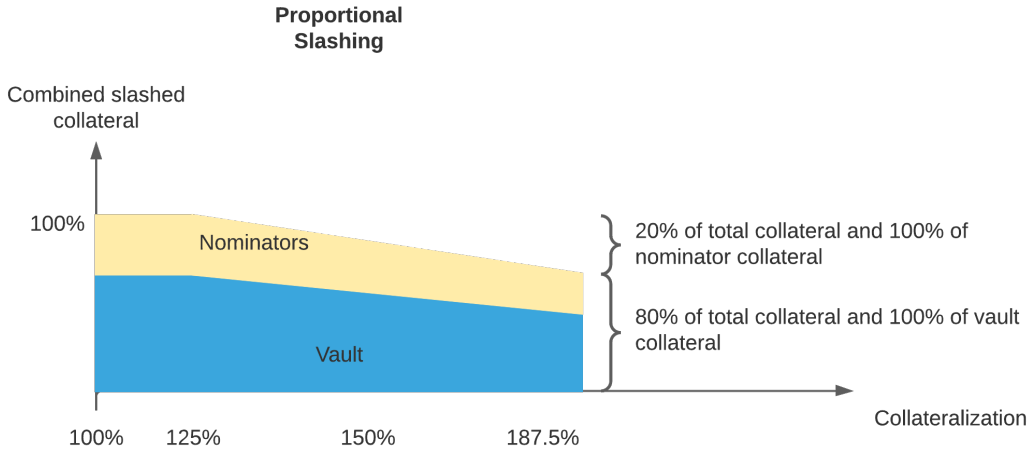


Fig. 21.1: The slashed collateral (in %) in Proportional Slashing of a Vault and its Nominators.

Note: Assume the similar DOT example from above. Effective threshold when Vault has an incentive to steal Bitcoin: 100% + (100% * 25%) = 125% collateralization. In case of a liquidation, the Vault is slashed all collateral and the Nominators are slashed all collateral since we slash up to the secure collateral threshold.

Note: It is not recommended to use this strategy in case of Vault theft. If the Vault steals Bitcoin at collateralization of 187.5% (i.e., 150% + (150% * 25%)), the Vault's and Nominators' collateral are slashed proportionally such that 150%/187.5% = 80% of the collateral is slashed from both the Vault and its Nominators. Normally, the vault should not be motivated to steal but it might be the case if e.g., the DOT/BTC exchange rate drops, the exchange

rate update is not yet reflected on chain, nominators are offline and cannot react, and the new exchange rate would bring the combined collateralization below 125% (such that Vault's future collateral is below 100%)).

Vault First Slashing

Nominators cannot control if Vaults decide to steal BTC. While Nominators trust Vaults (see Security Assumption 1 and 2), the protocol still tries to minimize this case by slashing Vaults first in case of theft. Therefore, in case of theft all of the Vaults available collateral are slashed before its Nominators. At the lower bound of $100\% + (100\% * \text{maxNominationRatio})$, both Proportional Slashing and Vault First Slashing slash the same amount of collateral from a Vault and its Nominators. However, at higher collateralization rates, Vaults are comparatively more slashed. See the figure below for an illustration using the threshold examples as above:

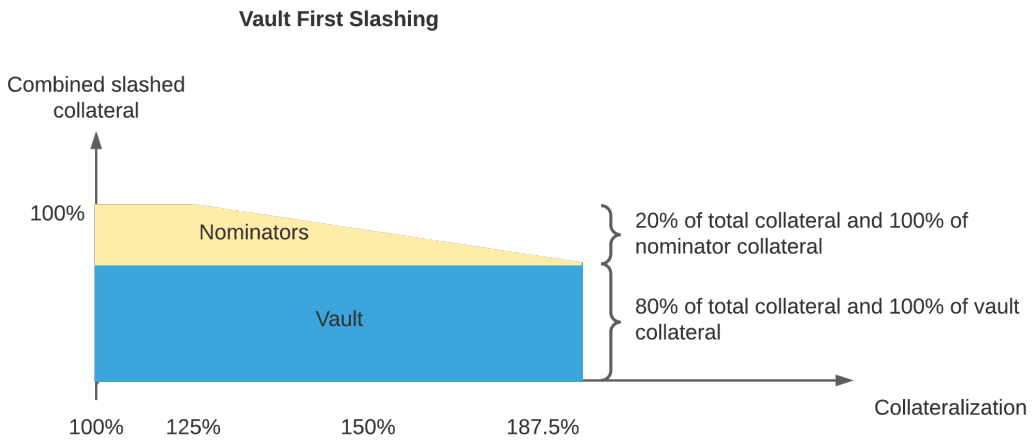


Fig. 21.2: The slashed collateral (in %) in Vault First Slashing of a Vault and its Nominators.

Note: Assume the similar DOT example from above. Effective threshold when Vault has an incentive to steal Bitcoin: 100% + (100% * 25%) = 125% collateralization. In case of theft, the vault is slashed all collateral, the nominators are slashed all collateral since we slash up to the secure collateral threshold. However, if the Vault steals Bitcoin at collateralization of 187.5% (i.e., 150% + (150% * 25%)), all of the vault's collateral are slashed and none of the nominators collateral is slashed. Normally, the vault should not be motivated to steal but it might be the case if e.g., you modify my example from the comment above (exchange rate drops, not yet reflected on chain, nominators are offline and cannot react, new exchange rate would bring combined collaterealization below 125% (such that vault's future collateral is below 100%)). In this case, the vault should steal BTC but in this case, we would only slash the vault for this.

Risk Summary

1. **Increased Exchange Rate Risk on Collateral Withdrawal**: 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.
2. **Vaults Have an Increased Incentive to Commit Theft**: The effective collateralization rate at which Vault's should steal Bitcoin increases from 100% to :math::*100% + (100% * text{maxNominationRatio})*.
3. **Different Slashing Strategies Reduce the Impact of Theft for Nominators**: By applying Vault First Slashing, the impact of the slashed collateral for Nominators is reduced if the collateralization is $> 100\% + (100\% * \text{maxNominationRatio})$.

21.3 Data Model

21.3.1 Scalars

NominationEnabled

Flag indicating whether this feature is enabled.

- If set to `True`, Vaults MAY opt-in to be nominated.
- If set to `False`, Vaults MUST NOT be able to opt-in to nomination. Already nominated Vaults MUST keep being nominated as Vaults may have issued *interBTC* with nominated collateral when this feature was enabled.

21.3.2 Maps

Vaults

Set of Vault accounts that have enabled nomination.

21.3.3 Structs

21.4 Functions

21.4.1 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.

Preconditions

- The calling account MUST be root (system level origin).

Postconditions

- The `NominationEnabled` scalar MUST be set to the value of the `enabled` parameter.

21.4.2 optInToNomination

Allow the Vault to receive nominated collateral.

Specification

Function Signature

`optInToNomination(vaultId)`

Parameters

- `vaultId`: the id of the Vault to enable nomination for.

Events

- *NominationOptIn*

Preconditions

- The BTC Parachain status in the *Security* component MUST be `RUNNING:0`.
- *NominationEnabled* MUST be true.
- A Vault with id `vaultId` MUST be registered.
- The Vault MUST NOT be opted in.

Postconditions

- The Vault MUST be allowed to receive nominated collateral.

21.4.3 optOutOfNomination

Disallow the Vault from receiving nominated collateral and force refund Nominators.

Specification

Function Signature

`optOutOfNomination(vaultId)`

Parameters

- `vaultId`: the id of the Vault to deregister from the nomination feature.

Events

- *NominationOptOut*

Preconditions

- The BTC Parachain status in the *Security* component MUST be `RUNNING:0`.
- A Vault with id `vaultId` MUST be registered.
- A Vault with id `vaultId` MUST exist in the `Vaults` mapping.

Postconditions

- The Vault MUST be removed from the `Vaults` mapping.
- The Vault MUST remain above the secure collateralization threshold.
- `getTotalNominatedCollateral(vaultId)` must return zero.
- For all nominators, `getNominatorCollateral(vaultId, userId)` must return zero.
- Staking pallet `nonce` must be incremented by one.
- The return value of calling *computeRewardAtIndex* parameterized with `(nonce - 1, INTERBTC, vaultId, userId)` must be equal to the user's nomination just before the vault opted out.

21.4.4 depositCollateral

Nominate collateral to a selected Vault.

Specification

Function Signature

`depositCollateral(vaultId, nominatorId, amount)`

Parameters

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

Events

- *DepositCollateral*

Preconditions

- The BTC Parachain status in the *Security* component MUST be `RUNNING:0`.
- *NominationEnabled* MUST be true.
- A Vault with id `vaultId` MUST be registered.
- A Vault with id `vaultId` MUST exist in the `Vaults` mapping.
- The nominator's free balance in the vault's used currency MUST be at least *amount*.
- The Vault MUST remain below the max nomination ratio.

Postconditions

- The Vault's backing collateral MUST increase by the amount nominated.
- The Nominator's balance in the vault's `currencyId` MUST decrease by the amount nominated.

21.4.5 withdrawCollateral

Withdraw collateral from a nominated Vault.

Specification

Function Signature

`withdrawCollateral(vaultId, nominatorId, amount)`

Parameters

- `vaultId`: the id of the previously nominated Vault.
- `nominatorId`: the id of the user who nominated collateral.
- `amount`: the amount of collateral to withdraw.

Events

- *WithdrawCollateral*

Preconditions

- The BTC Parachain status in the *Security* component MUST be `RUNNING:0`.
- *NominationEnabled* MUST be true.

- A Vault with id `vaultId` MUST be registered.
- A Vault with id `vaultId` MUST exist in the `Vaults` mapping.
- The Vault MUST remain above the secure collateralization threshold.
- Nominator MUST have a nomination with the given vault (including slashes) of at least `amount`.

Postconditions

- The Vault's collateral MUST decrease by `amount`.
- The Nominator's balance in the vault's `currencyId` MUST increase by `amount`.

21.5 Events

21.5.1 NominationOptIn

Event Signature

`NominationOptIn(vaultId)`

Parameters

- `vaultId`: the id of the Vault who opted in.

Functions

- *optInToNomination*

21.5.2 NominationOptOut

Event Signature

`NominationOptOut(vaultId)`

Parameters

- `vaultId`: the id of the Vault who opted out.

Functions

- *optOutOfNomination*

21.5.3 DepositCollateral

Event Signature

`DepositCollateral(vaultId, nominatorId, amount)`

Parameters

- `vaultId`: the id of the Vault who receives the nomination.
- `nominatorId`: the id of the nominator who is depositing collateral.
- `amount`: the amount of nominated collateral.

Functions

- *depositCollateral*

21.5.4 WithdrawCollateral

Event Signature

`WithdrawCollateral(vaultId, nominatorId, amount)`

Parameters

- `vaultId`: the id of the previously nominated Vault.
- `nominatorId`: the id of the nominator who is withdrawing collateral.
- `amount`: the amount of nominated collateral.

Functions

- *withdrawCollateral*

CHAPTER

TWENTYTWO

REWARD

22.1 Overview

This pallet provides a way distribute rewards to any number of accounts, proportionally to their stake. It does so using the Scalable Reward Distribution algorithm. It does not directly transfer any rewards - rather, the stakeholders have to actively withdraw their accumulated rewards, which they can do at any time. Stakeholders can also change their stake at any time, without impacting the rewards gained in the past.

22.2 Invariants

- For each `currencyId`,
  - `TotalStake[currencyId]` MUST be equal to the sum of `Stake[currencyId, accountId]` over all accounts.
  - `TotalReward[currencyId]` MUST be equal to the sum of `Stake[currencyId, accountId] * RewardPerToken[currencyId] - RewardTally[currencyId, accountId]` over all accounts.
  - For each `accountId`,
    - `RewardTally[currencyId, accountId]` MUST be smaller than or equal to `Stake[currencyId, accountId] * RewardPerToken[currencyId]`
    - `Stake[currencyId, accountId]` MUST NOT be negative
    - `RewardTally[currencyId, accountId]` MUST NOT be negative

22.3 Data Model

22.3.1 Maps

TotalStake

The total stake deposited to the reward with the given currency.

TotalRewards

The total unclaimed rewards in the given currency distributed to this reward pool. This value is currently only used for testing purposes.

RewardPerToken

The amount of reward the stakeholders get for the given currency per unit of stake.

Stake

The stake in the given currency for the given account.

RewardTally

The amount of rewards in the given currency a given account has already withdrawn, plus a compensation that is added on stake changes.

22.4 Functions

22.4.1 getTotalRewards

This function gets the total amount of rewards distributed in the pool with the given currencyId.

Specification

Function Signature

`getTotalRewards(currencyId)`

Parameters

- `currencyId`: Determines of which currency the amount is returned.

Postconditions

- The function MUST return the total amount of rewards that have been distributed in the given currency.

22.4.2 depositStake

Adds a stake for the given account and currency in the reward pool.

Specification

Function Signature

`depositStake(currencyId, accountId, amount)`

Parameters

- `currencyId`: The currency for which to add the stake
- `accountId`: The account for which to add the stake
- `amount`: The amount by which the stake is to increase

Events

- *DepositStake*

Preconditions

Postconditions

- `Stake[currencyId, accountId]` MUST increase by `amount`
- `TotalStake[currencyId]` MUST increase by `amount`
- `RewardTally[currencyId, accountId]` MUST increase by `RewardPerToken[currencyId] * amount`. This ensures the amount of rewards the given accountId can withdraw remains unchanged.

22.4.3 distributeReward

Distributes rewards to the stakeholders.

Specification

Function Signature

`distributeReward(currencyId, reward)`

Parameters

- `currencyId`: The currency being distributed
- `reward`: The amount being distributed

Events

- *DistributeReward*

Preconditions

- `TotalStake[currencyId]` MUST NOT be zero.

Postconditions

- `RewardPerToken[currencyId]` MUST increase by `reward / TotalStake[currencyId]`
- `TotalRewards[currencyId]` MUST increase by `reward`

22.4.4 computeReward

Computes the amount a given account can withdraw in the given currency.

Specification

Function Signature

`computeReward(currencyId, accountId)`

Parameters

- `currencyId`: The currency for which the rewards are being calculated
- `accountId`: Account for which the rewards are being calculated.

Postconditions

- The function MUST return `Stake[currencyId, accountId] * RewardPerToken[currencyId] - RewardTally[currencyId, accountId]`.

22.4.5 withdrawStake

Decreases a stake for the given account and currency in the reward pool.

Specification

Function Signature

`withdrawStake(currencyId, accountId, amount)`

Parameters

- `currencyId`: The currency for which to decrease the stake
- `accountId`: The account for which to decrease the stake
- `amount`: The amount by which the stake is to decrease

Events

- *WithdrawStake*

Preconditions

- `amount` MUST NOT be greater than `Stake[currencyId, accountId]`

Postconditions

- `Stake[currencyId, accountId]` MUST decrease by `amount`
- `TotalStake[currencyId]` MUST decrease by `amount`
- `RewardTally[currencyId, accountId]` MUST decrease by `RewardPerToken[currencyId] * amount`. This ensures the amount of rewards the given accountId can withdraw remains unchanged.

22.4.6 withdrawReward

Withdraw all available rewards of a given account and currency

Specification

Function Signature

`withdrawReward(currencyId, reward)`

Parameters

- `currencyId`: The currency being withdrawn
- `accountId`: The account for which to withdraw the rewards

Events

- *WithdrawReward*

Preconditions

- `TotalStake[currencyId]` MUST NOT be zero.

Postconditions

Let `reward` be the result *computeReward* when it is called with `currencyId` and `accountId` as arguments. Then:

- `TotalRewards[currencyId]` MUST decrease by `reward`
- `RewardPerToken[currencyId]` MUST be set to `RewardPerToken[currencyId] * Stake[currencyId, accountId]`

22.5 Events

22.5.1 DepositStake

Event Signature

`DepositStake(currencyId, accountId, amount)`

Parameters

- `currencyId`: the currency for which the stake has been changed
- `accountId`: the account for which the stake has been changed
- `amount`: the increase in stake

Functions

- *depositStake*

22.5.2 WithdrawStake

Event Signature

`WithdrawStake(currencyId, accountId, amount)`

Parameters

- `currencyId`: the currency for which the stake has been changed
- `accountId`: the account for which the stake has been changed
- `amount`: the decrease in stake

Functions

- *withdrawStake*

22.5.3 DistributeReward

Event Signature

`DistributeReward(currencyId, accountId, amount)`

Parameters

- `currencyId`: the currency for which the reward has been withdrawn
- `amount`: the distributed amount

Functions

- *distributeReward*

22.5.4 WithdrawReward

Event Signature

`WithdrawReward(currencyId, accountId, amount)`

Parameters

- `currencyId`: the currency for which the reward has been withdrawn
- `accountId`: the account for which the reward has been withdrawn
- `amount`: the withdrawn amount

Functions

- *withdrawReward*

CHAPTER

TWENTYTHREE

STAKING

23.1 Overview

This pallet is very similar to the *Reward* pallet - it is also based on the Scalable Reward Distribution algorithm. The reward pallet keeps track of how much rewards vaults have earned. However, when nomination is enabled, there needs to be a way to relay parts of the vault's rewards to its nominators. Furthermore, the nominator's collaterals can be consumed, e.g., when a redeem is cancelled. This pallet is responsible for both tracking the rewards, and the current amount of contributed collaterals of vaults and nominators.

The idea is to have one reward pool per vault, where both the vault and all of its nominators have a stake equal to their contributed collateral. However, when collateral is consumed, either in *cancelRedeem* or *liquidateVault*, the collateral of each of these stakeholders should decrease proportionally to their stake. To be able to achieve this without iteration, in addition to tracking `RewardPerToken`, a similar value `SlashPerToken` is introduced. Similarly, in addition to `RewardTally`, we now also maintain a `SlashTally` is for each stakeholder. When calculating a reward for a stakeholder, a compensated stake is calculated, based on `Stake`, `SlashPerToken` and `SlashTally`.

When a vault opts out of nomination, all nominators should receive their collateral back. This is achieved by distributing all funds from the vault's shared collateral as rewards. However, a vault is free to opt back into nominator after having opted out. It is possible for the vault to do this before all nominators have withdrawn their reward. To ensure that the bookkeeping remains intact for the nominators to get their rewards at a later point, all variables are additionally indexed by a nonce, which increases every time a vault opts out of nomination. Effectively, this create a new pool for every nominated period.

Note: Most of the functions in this pallet that have a `_at_index` also have a version without this suffix that does not take a `nonce` argument, and instead uses the value stored in *Nonce*. For brevity, these functions without the suffix are omitted in this specification.

23.2 Data Model

23.2.1 Maps

TotalStake

Maps `(currencyId, nonce, vaultId)` to the total stake deposited by the given vault and its nominators, with the given nonce and currencyId.

TotalCurrentStake

Maps `(currencyId, nonce, vaultId)` to the total stake deposited by the given vault and its nominators, with the given nonce and currencyId, excluding stake that has been slashed.

TotalRewards

Maps `(currencyId, nonce, vaultId)` to the total rewards distributed to the vault and its nominators. This value is currently only used for testing purposes.

RewardPerToken

Maps `(currencyId, nonce, vaultId)` to the amount of reward the vault and its nominators get per unit of stake.

RewardTally

Maps `(currencyId, nonce, vaultId, nominatorId)` to the reward tally the given nominator has for the given vault's reward pool, in the given nonce and currency. The tally influences how much the nominator can withdraw.

Stake

Maps `(currencyId, nonce, vaultId, nominatorId)` to the stake the given nominator has in the given vault's reward pool, in the given nonce and currency. Initially, the stake is equal to its contributed collateral. However, after a slashing has occurred, the nominator's collateral must be compensated, using *computeStakeAtIndex*.

SlashPerToken

Akin to *RewardPerToken*: maps `(currencyId, nonce, vaultId)` to the amount the vault and its nominators got slashed for per unit of stake. It is used for computing the effective stake (or equivalently, its collateral) in *computeStakeAtIndex*.

SlashTally

Akin to *RewardTally*: maps `(currencyId, nonce, vaultId, nominatorId)` to the slash tally the given nominator has for the given vault's reward pool, in the given nonce and currency. It is used for computing the effective stake (or equivalently, its collateral) in *computeStakeAtIndex*.

Nonce

Maps `(currencyId, vaultId)` current value of the nonce the given vault uses in the given currency. The nonce is increased every time *forceRefund* is called, i.e., when a vault opts out of nomination. Since nominators get their collateral back as a withdrawable reward, the bookkeeping must remain intact when the vault once again opts into nomination. By incrementing this nonce, effectively a new reward pool is created for the new session. All externally callable functions use the nonce stored in this map, except for the reward withdrawal function *withdrawRewardAtIndex*.

23.3 Functions

23.3.1 depositStake

Adds a stake for the given account and currency in the reward pool.

Specification

Function Signature

`depositStake(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency for which to add the stake
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator
- `amount`: The amount by which the stake is to increase

Events

- *DepositStake*

Postconditions

- `Stake[currencyId, nonce, vaultId, nominatorId]` MUST increase by `amount`
- `TotalStake[currencyId, nonce, vaultId]` MUST increase by `amount`
- `TotalCurrentStake[currencyId, nonce, vaultId]` MUST increase by `amount`
- `RewardTally[currencyId, nonce, vaultId, nominatorId]` MUST increase by `RewardPerToken[currencyId, nonce, vaultId] * amount`.
- `SlashTally[currencyId, nonce, vaultId, nominatorId]` MUST increase by `SlashPerToken[currencyId, nonce, vaultId] * amount`.

23.3.2 withdrawStake

Withdraws the given amount stake for the given nominator or vault. This function also modifies the nominator's `SlashTally` and `Stake`, such that the `Stake` is once again equal to its collateral.

Specification

Function Signature

`withdrawStake(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency for which to add the stake
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator
- `amount`: The amount by which the stake is to decrease

Events

- *WithdrawStake*

Preconditions

- Let `nonce` be `Nonce[currencyId, vaultId]`, and
- Let `stake` be `Stake[nonce, currencyId, vaultId, nominatorId]`, and
- Let `slashPerToken` be `SlashPerToken[currencyId, nonce, vaultId]`, and
- Let `slashTally` be `slashTally[nonce, currencyId, vaultId, nominatorId]`, and
- Let `toSlash` be `stake * slashPerToken - slashTally`

Then:

- `stake - toSlash` MUST be greater than or equal to `amount`

Postconditions

- Let `nonce` be `Nonce[currencyId, vaultId]`, and
- Let `stake` be `Stake[nonce, currencyId, vaultId, nominatorId]`, and
- Let `slashPerToken` be `SlashPerToken[currencyId, nonce, vaultId]`, and
- Let `slashTally` be `slashTally[nonce, currencyId, vaultId, nominatorId]`, and
- Let `toSlash` be `stake * slashPerToken - slashTally`

Then:

- `Stake[currencyId, nonce, vaultId, nominatorId]` MUST decrease by `toSlash + amount`
- `TotalStake[currencyId, nonce, vaultId]` MUST decrease by `toSlash + amount`
- `TotalCurrentStake[currencyId, nonce, vaultId]` MUST decrease by `amount`
- `SlashTally[nonce, currencyId, vaultId, nominatorId]` MUST be set to `(stake - toSlash - amount) * slashPerToken`
- `RewardTally[nonce, currencyId, vaultId, nominatorId]` MUST decrease by `rewardPerToken * amount`

23.3.3 slashStake

Slashes a vault's stake in the given currency in the reward pool. Conceptually, this decreases the stakes, and thus the collaterals, of all of the vault's stakeholders. Indeed, *computeStakeAtIndex* will reflect the stake changes on the stakeholder.

Specification

Function Signature

`slashStake(currencyId, vaultId, amount)`

Parameters

- `currencyId`: The currency for which to add the stake
- `vaultId`: Account of the vault
- `amount`: The amount by which the stake is to decrease

Preconditions

- `TotalStake[currencyId, Nonce[currencyId, vaultId], vaultId]` MUST NOT be zero

Postconditions

Let `nonce` be `Nonce[currencyId, vaultId]`, and `initialTotalStake` be `TotalCurrentStake[currencyId, nonce, vaultId]`. Then:

- `SlashPerToken[currencyId, nonce, vaultId]` MUST increase by `amount / TotalStake[currencyId, nonce, vaultId]`

- `TotalCurrentStake[currencyId, nonce, vaultId]` MUST decrease by `amount`
- if `initialTotalStake - amount` is NOT zero, `RewardPerToken[currencyId, nonce, vaultId]` MUST increase by `RewardPerToken[currencyId, nonce, vaultId] * amount / (initialTotalStake - amount)`

23.3.4 computeStakeAtIndex

Computes a vault's stakeholder's effective stake. This is also the amount collateral that belongs to the stakeholder.

Specification

Function Signature

`computeStakeAtIndex(nonce, currencyId, vaultId, amount)`

Parameters

- `nonce`: The nonce to compute the stake at
- `currencyId`: The currency for which to compute the stake
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator

Postconditions

Let `stake` be `Stake[nonce, currencyId, vaultId, nominatorId]`, and Let `slashPerToken` be `SlashPerToken[currencyId, nonce, vaultId]`, and Let `slashTally` be `slashTally[nonce, currencyId, vaultId, nominatorId]`, then

- The function MUST return `stake - stake * slash_per_token + slash_tally`.

23.3.5 distributeReward

Distributes rewards to the vault's stakeholders.

Specification

Function Signature

`distributeReward(currencyId, reward)`

Parameters

- `currencyId`: The currency being distributed
- `vaultId`: the vault for which distribute rewards
- `reward`: The amount being distributed

Events

- *DistributeReward*

Postconditions

Let `nonce` be `Nonce[currencyId, vaultId]`, and Let `initialTotalCurrentStake` be `TotalCurrentStake[currencyId, nonce, vaultId]`, then:

- If `initialTotalCurrentStake` is zero, or if `reward` is zero, then:
  - The function MUST return zero.
- Otherwise (if `initialTotalCurrentStake` and `reward` are not zero), then:

- `RewardPerToken[currencyId, nonce, vaultId)]` MUST increase by `reward / initialTotalCurrentStake`
- `TotalRewards[currencyId, nonce, vaultId]` MUST increase by `reward`
- The function MUST return `reward`.

23.3.6 computeRewardAtIndex

Calculates the amount of rewards the vault's stakeholder can withdraw.

Specification

Function Signature

`computeRewardAtIndex(nonce, currencyId, vaultId, amount)`

Parameters

- `nonce`: The nonce to compute the stake at
- `currencyId`: The currency for which to compute the stake
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator

Postconditions

Let `stake` be the result of `computeStakeAtIndex(nonce, currencyId, vaultId, nominatorId)`, then: Let `rewardPerToken` be `RewardPerToken[currencyId, nonce, vaultId]`, and Let `rewardTally` be `rewardTally[nonce, currencyId, vaultId, nominatorId]`, then

- The function MUST return `max(0, stake * rewardPerToken - reward_tally)`

23.3.7 withdrawRewardAtIndex

Withdraws the rewards the given vault's stakeholder has accumulated.

Specification

Function Signature

`withdrawRewardAtIndex(currencyId, vaultId, amount)`

Parameters

- `nonce`: The nonce to compute the stake at
- `currencyId`: The currency for which to compute the stake
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator

Events

- *WithdrawReward*

Preconditions

- `computeRewardAtIndex(nonce, currencyId, vaultId, nominatorId)` MUST NOT return an error

Postconditions

Let `reward` be the result of `computeRewardAtIndex(nonce, currencyId, vaultId, nominatorId)`, then: Let `stake` be `Stake(nonce, currencyId, vaultId, nominatorId)`, then: Let `rewardPerToken` be `RewardPerToken[currencyId, nonce, vaultId]`, and

- `TotalRewards[currency_id, nonce, vault_id]` MUST decrease by `reward`
- `RewardTally[currencyId, nonce, vaultId, nominatorId]` MUST be set to `stake * rewardPerToken`
- The function MUST return `reward`

23.3.8 forceRefund

This is called when the vault opts out of nomination. All collateral is distributed among the stakeholders, after which the vault withdraws his part immediately.

Specification

Function Signature

`forceRefund(currencyId, vaultId)`

Parameters

- `currencyId`: The currency for which to compute the stake
- `vaultId`: Account of the vault

Events

- *ForceRefund*
- *IncreaseNonce*

Preconditions

Let `nonce` be `Nonce[currencyId, vaultId]`, then:

- `distributeReward(currencyId, vaultId, TotalCurrentStake[currencyId, nonce, vaultId])` MUST NOT return an error
- `withdrawRewardAtIndex(nonce, currencyId, vaultId, vaultId)` MUST NOT return an error
- `depositStake(currencyId, vaultId, vaultId, reward)` MUST NOT return an error
- `Nonce[currencyId, vaultId]` MUST be increased by 1

Postconditions

Let `nonce` be `Nonce[currencyId, vaultId]`, then:

- `distributeReward(currencyId, vaultId, TotalCurrentStake[currencyId, nonce, vaultId])` MUST have been called
- `withdrawRewardAtIndex(nonce, currencyId, vaultId, vaultId)` MUST have been called
- `Nonce[currencyId, vaultId]` MUST be increased by 1
- `depositStake(currencyId, vaultId, vaultId, reward)` MUST have been called AFTER having increased the nonce

23.3.9 DepositStake

Event Signature

`DepositStake(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator
- `amount`: The amount by which the stake is to increase

Functions

- *depositStake*

23.3.10 WithdrawStake

Event Signature

`WithdrawStake(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator
- `amount`: The amount by which the stake is to increase

Functions

- *withdrawStake*

23.3.11 DistributeReward

Event Signature

`DistributeReward(currencyId, vaultId, amount)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault
- `amount`: The amount by which the stake is to increase

Functions

- *distributeReward*

23.3.12 WithdrawReward

Event Signature

`WithdrawReward(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault
- `nominatorId`: Account of the nominator
- `amount`: The amount by which the stake is to increase

Functions

- *withdrawRewardAtIndex*

23.3.13 ForceRefund

Event Signature

`ForceRefund(currencyId, vaultId)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault

Functions

- *forceRefund*

23.3.14 IncreaseNonce

Event Signature

`IncreaseNonce(currencyId, vaultId, nominatorId, amount)`

Parameters

- `currencyId`: The currency of the reward pool
- `vaultId`: Account of the vault
- `amount`: The amount by which the stake is to increase

Functions

- *forceRefund*

CHAPTER

TWENTYFOUR

ESCROW

24.1 Overview

The Escrow module allows users to lockup tokens in exchange for a non-fungible voting asset. The total "power" of this asset decays linearly as the lock approaches expiry - calculated based on the block height. Historic points for the linear function are recorded each time a user's balance is adjusted which allows us to re-construct voting power at a particular point in time.

This architecture was adopted from Curve, see: Vote-Escrowed CRV (veCRV).

Note: This specification is still a Work-in-Progress (WIP), some information may be outdated or incomplete.

24.1.1 Step-by-step

1. A user may lock any amount of defined governance currency (KINT on Kintsugi, INTR on Interlay) up to a maximum lock period.
2. Both the amount and the unlock time may be increased to improve voting power.
3. The user may unlock their fungible asset after the lock has expired.

24.2 Data Model

24.2.1 Constants

Span

The locktime is rounded to weeks to limit checkpoint iteration.

MaxPeriod

The maximum period for lockup.

24.2.2 Scalars

Epoch

The current global epoch for `PointHistory`.

24.2.3 Maps

Locked

Stores the `amount` and `end` block for an account's lock.

PointHistory

Stores the global `bias`, `slope` and `height` at a particular point in history.

UserPointHistory

Stores the `bias`, `slope` and `height` for an account at a particular point in history.

UserPointEpoch

Stores the current epoch for an account.

SlopeChanges

Stores scheduled changes of slopes for ending locks.

24.2.4 Structs

LockedBalance

The `amount` and `end` height for a locked balance.

Parameter	Type	Description
`amount`	Balance	The amount deposited to receive vote-escrowed tokens.
`end`	BlockNumber	The end height after which the balance can be unlocked.

Point

The `bias`, `slope` and `height` for our linear function.

Parameter	Type	Description
`bias`	Balance	The bias for the linear function.
`slope`	Balance	The slope for the linear function.
`height`	BlockNumber	The current block height when this point was stored.

24.3 External Functions

24.3.1 create_lock

Create a lock on the account's balance to expire in the future.

Specification

Function Signature

`create_lock(who, amount, unlock_height)`

Parameters

- `who`: The user's address.
- `amount`: The amount to be locked.
- `unlock_height`: The height to lock until.

Events

- *Deposit*

Preconditions

- The function call MUST be signed by `who`.
- The `amount` MUST be non-zero.
- The account's `old_locked.amount` MUST be non-zero.
- The `unlock_height` MUST be greater than `now`.
- The `unlock_height` MUST NOT be greater than `now + MaxPeriod`.

Postconditions

- The account's `LockedBalance` MUST be set as follows:
  - `new_locked.amount`: MUST be the `amount`.
  - `new_locked.end`: MUST be the `unlock_height`.
- The `UserPointEpoch` MUST increase by one.
- A new `Point` MUST be recorded at this epoch:
  - `slope = amount / max_period`
  - `bias = slope * (unlock_height - now)`
  - `height = now`
- Function *withdrawStake* MUST complete successfully using the account's total stake.
- Function *depositStake* MUST complete successfully using the current balance (*balance_at*).

24.3.2 increase_amount

Deposit additional tokens for a pre-existing lock to improve voting power.

Specification

Function Signature

`increase_amount(who, amount)`

Parameters

- `who`: The user's address.
- `amount`: The amount to be locked.

Events

- *Deposit*

Preconditions

- The function call MUST be signed by `who`.
- The `amount` MUST be non-zero.
- The account's `old_locked.amount` MUST be non-zero.
- The account's `old_locked.end` MUST be greater than `now`.

Postconditions

- The account's `LockedBalance` MUST be set as follows:
  - `new_locked.amount`: MUST be `old_locked.amount + amount`.
  - `new_locked.end`: MUST be the `old_locked.end`.
- The `UserPointEpoch` MUST increase by one.
- A new `Point` MUST be recorded at this epoch:
  - `slope = new_locked.amount / max_period`
  - `bias = slope * (new_locked.end - now)`
  - `height = now`

24.3.3 extend_unlock_height

Push back the expiry on a pre-existing lock to retain voting power.

Specification

Function Signature

`extend_unlock_height(who, unlock_height)`

Parameters

- `who`: The user's address.
- `unlock_height`: The new expiry deadline.

Events

- *Deposit*

Preconditions

- The function call MUST be signed by `who`.
- The `amount` MUST be non-zero.
- The account's `old_locked.amount` MUST be non-zero.
- The account's `old_locked.end` MUST be greater than `now`.
- The `unlock_height` MUST be greater than `old_locked.end`.
- The `unlock_height` MUST NOT be greater than `now + MaxPeriod`.

Postconditions

- The account's `LockedBalance` MUST be set as follows:
  - `new_locked.amount`: MUST be `old_locked.amount`.
  - `new_locked.end`: MUST be the `unlock_height`.
- The `UserPointEpoch` MUST increase by one.
- A new `Point` MUST be recorded at this epoch:
  - `slope = new_locked.amount / max_period`
  - `bias = slope * (new_locked.end - now)`
  - `height = now`

24.3.4 withdraw

Remove the lock on an account to allow access to the account's funds.

Specification

Function Signature

`withdraw(who)`

Parameters

- `who`: The user's address.

Events

- *Withdraw*

Preconditions

- The function call MUST be signed by `who`.
- The account's `old_locked.amount` MUST be non-zero.
- The current height (`now`) MUST be greater than or equal to `old_locked.end`.

Postconditions

- The account's `LockedBalance` MUST be removed.
- Function *withdrawStake* MUST complete successfully using the account's total stake.

24.4 Internal Functions

24.4.1 balance_at

Using the `Point`, we can calculate the current voting power (`balance`) as follows:

```
balance = point.bias - (point.slope * (height - point.height))
```

Specification

Function Signature

`balance_at(who, height)`

Parameters

- `who`: The user's address.
- `height`: The future height.

Preconditions

- The `height` MUST be `>= point.height`.

24.5 Events

24.5.1 Deposit

Emit an event if a user successfully deposited tokens or increased the lock time.

Event Signature

`Deposit(who, amount, unlock_height)`

Parameters

- `who`: The user's account identifier.
- `amount`: The amount locked.
- `unlock_height`: The height to unlock after.

Functions

- *create_lock*

24.5.2 Withdraw

Emit an event if a user withdrew previously locked tokens.

Event Signature

`Withdraw(who, amount)`

Parameters

- `who`: The user's account identifier.
- `amount`: The amount unlocked.

Functions

- *withdraw*

CHAPTER

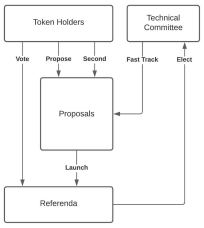
TWENTYFIVE

GOVERNANCE

25.1 Overview

On-chain governance is useful for controlling system parameters, authorizing trusted oracles and upgrading the core protocols. The architecture adopted by interBTC is modelled on Polkadot with some significant changes:

- **Optimistic Governance**
  - No **Council**, only public proposals from community
  - Community can elect a **Technical Committee** to fast-track proposals
  - Referenda are Super-Majority Against (Negative Turnout Bias) by default
- **Stake-To-Vote**
  - Adopted from Curve's governance model
  - Users lock the native governance token
  - Longer lockups give more voting power



An important distinction is the `negative turnout bias` (Super-Majority Against) voting threshold. This is best summarized by the Polkadot docs:

A heavy super-majority of nay votes is required to reject at low turnouts, but as turnout increases towards 100%, it becomes a simple majority-carries as below.

$$\frac{\text{against}}{\sqrt{\text{electorate}}} < \frac{\text{approve}}{\sqrt{\text{turnout}}}$$

25.2 Terminology

- **Proposals** are community-supported motions to perform system-level actions.
- **Referenda** are accepted proposals undergoing voting.

25.3 Processes

25.3.1 Proposals

1. Account submits public proposal with deposit (> `MinimumDeposit`)
2. Account "seconds" proposal with additional deposit
3. New referenda are started every `LaunchPeriod`
4. Community can vote on referenda for the `VotingPeriod`
5. Votes are tallied after `VotingPeriod` expires
6. System update executed after `EnactmentPeriod`

25.3.2 Technical Committee

1. Community creates proposal as above
2. TC may fast track before `LaunchPeriod`
3. The new referendum is started immediately
4. Community can vote on referenda for the `FastTrackVotingPeriod`

25.4 Parameters

EnactmentPeriod

The period to wait before any approved change is enforced.

LaunchPeriod

The interval after which to start a new referenda from the queue.

VotingPeriod

The period to allow new votes for a referenda.

MinimumDeposit

The minimum deposit required for a proposal.

FastTrackOrigin

Used to fast-track a proposal before the `LaunchPeriod`.

FastTrackVotingPeriod

The period to allow new votes for a fast-tracked referendum.

CancellationOrigin

Used to cancel a proposal before it is launched.

MaxProposals

The maximum number of public proposals allowed in the queue.

MaxMembers

The maximum number of possible members in the TC.

CHAPTER

TWENTYSIX

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**.

26.1 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 *LiquidationThreshold*.

In both cases, the Vault's entire BTC holdings are liquidated and its collateral is slashed - up to the *SecureCollateralThreshold* of the liquidated BTC value.

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

26.2 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 amount to be redeemed:

- **Punishment Fee**: 10% of the failed redeem value.

26.3 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

26.3.1 Issue

- **`requestIssue`**
  - disallowed
- **`executeIssue`**
  - Overpayment protection is disabled; if a user transfers too many BTC, the user loses it.
- **`cancelIssue`**
  - User's griefing collateral is released back to the user, rather than slashed to the Vault.

26.3.2 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.

26.3.3 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

26.3.4 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

TWENTYSEVEN

XCLAIM SECURITY ANALYSIS

27.1 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*.

27.1.1 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.

27.1.2 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).

27.2 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 expect Vaults 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.

27.3 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.

27.3.1 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.

27.3.2 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'.

27.4 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.

27.5 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.

27.5.1 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.

27.5.2 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.

27.6 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.

27.7 Concurrency

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

27.7.1 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, they can only fulfil one and would lose his collateral with the other.

27.7.2 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.

27.7.3 Solution

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

CHAPTER

TWENTYEIGHT

BTC-RELAY SECURITY ANALYSIS

This section provides an overview of security considerations related to BTC-Relay. We refer the reader to this paper (Section 7) for more details.

28.1 Security Parameter *k*

Blockchains using Nakamoto consensus as underlying agreement protocol (i.e., leveraging PoW for random leader election in a dynamically changing set of consensus participants) exhibit so called *stabilizing consensus*. Specifically, finality of transactions included in the blockchain converges with a *security parameter k*, measured in confirmations (i.e., blocks mined on top of a block containing the observed transaction). That is, the probability of a transaction being reverted in a blockchain reorganization decreases exponentially in k. We refer the reader to this paper for more details on Nakamoto consensus.

In Bitcoin, this security parameter is often set to $k = 6$, i.e., transactions are considered "final" after 6 blocks have been mined on top. However, there is *no mathematical reasoning* behind this, nor is there a proof that 6 confirmations are sufficient.

In fact, research has shown that when estimating the necessary confirmations before accepting a transaction, the *transaction value* itself must also be considered: the higher the value, the more confirmations are necessary to maintain the same level of security. However, recent analysis suggests that it is insufficient to consider the value of a single transaction - instead, to estimate the necessary k one must study the value of the entire block. The existence of bribing attacks, which can even be executed cross-chain, makes the situation worse: in theory, it is impossible to estimate k reliably, as there can always be a large transaction that is being attacked by a reorg in an older block.

What does this mean for BTC-Relay?

BTC-Relay does not specify a recommended value for k. This task lies with the applications which interact with the relay. BTC-Relay itself only *mirrors* the state of Bitcoin to Polkadot, including all forks and failures which may occur.

28.2 Liveness Failures

The correct operation of BTC-Relay relies on receiving a steady stream of Bitcoin block headers as input. A high delay between block generation in Bitcoin and submission to BTC-Relay yields the system susceptible to attacks: an adversary can attempt to *poison* the relay by submitting a fork, even if the fork was not submitted to Bitcoin itself (see *Relay Poisoning* below).

While by design, any user can submit Bitcoin block headers to BTC-Relay, it is recommended to introduce an explicit set of participants for this task. These can be *Staked Relayers*, which already run Bitcoin full nodes for validation purposes, or *Vaults* which are used for the creation of Bitcoin-backed assets in the interBTC component.

28.3 Safety Failures

28.3.1 51% Attack on Bitcoin

One of the major questions that arises in cross-chain communication is: what to do if one of the interlinked chains fails?

In the case of BTC-Relay, a major chain reorganization in Bitcoin would be accepted, if the new chain exceeds the tracked `Chains` in BTC-Relay. If the length of the fork exceeds the security parameter k relied upon by applications using BTC-Relay, this can have sever impacts, beyond that of users loosing BTC.

However, as BTC-Relay acts only as mirror of the Bitcoin blockchain, the only possible mitigation of a 51% attack on Bitcoin **halting BTC-Relay** via manual intervention of *Staked Relayers* or the *Governance Mechanism*. See **Failure Handling** for more details on BTC-Relay failure modes and recovery procedures.

A major challenge thereby is to ensure the potential financial loss of *Staked Relayers* and/or participants of the *Governance Mechanism* exceeds the potential gains from colluding with an adversary on Bitcoin.

28.3.2 Relay Poisoning

BTC-Relay poisoning is a more subtle way of interfering with correct operation of the system: an adversary submits a Bitcoin fork to BTC-Relay, but does not broadcast it to the actual Bitcoin network. If Liveness of BTC-Relay is breached, e.g. *Staked Relayers* are unavailable, BTC-Relay can be tricked into accepting an alternate `Chains` than actually maintained in Bitcoin.

However, as long as a single honest participant is online and capable of submitting Bitcoin block headers from the Bitcoin main chain to BTC-Relay within k blocks, poisoning attacks can be mitigated.

28.3.3 Replay Attacks

Since BTC-Relay does not store Bitcoin transactions, nor can it be aware of all possible applications using `verifyTransactionInclusion`, duplicate submission of transaction inclusion proofs **cannot be easily detected** by BTC-Relay.

As such, it lies in the responsibility of each application interacting with BTC-Relay to introduce necessary replay protection mechanisms (e.g. nonces stored in OP_RETURN outputs of verified transactions) and to check the latter using the *Functions: Parser* component of BTC-Relay.

28.4 Hard and Soft forks

Permanent chain splits or *hard forks* occur where consensus rules are "loosened" or new conflicting rules are introduced. As a result, multiple instances of the same blockchain are created, e.g. as in the case of Bitcoin and Bitcoin Cash.

BTC-Relay by default will follow the old consensus rules, and must be updated accordingly if it is to follow the new version of the system.

Thereby, is it for the *Governance Mechanism* to determine (i) whether an update will be executed and (ii) if two parallel blockchains result from the hard fork, whether an additional new instance of BTC-Relay is to be deployed (and how).

Note: to differentiate between the two resulting chains after a hard fork, replay protection is necessary for secure operation. While typically accounted for by the developers of the verified blockchain, the absence of replay protection can lead to undesirable behavior. Specifically, payments made on one fork may be accepted as valid on the other as well - and propagated to BTC-Relay. To this end, *if a fork lacks replay protection*, **halting of the relay** may be necessary until the matter is resolved.

CHAPTER

TWENTYNINE

PERFORMANCE ANALYSIS

Contrary to permissionless blockchains, such as Ethereum, Polkadot's Parachains can easily implement the cryptographic primitives of the verified blockchains, instead of relying on pre-compiled smart contracts or manual and costly implementation of primitives. In the case of Bitcoin, the BTC Parachain can provide native support for the SHA256 and RIPEMD-160 hash functions, as well as for ECDSA using the secp256k1 curve.

Consequently, storage resembles the main cost factor of BTC-Relay on Polkadot.

29.1 Estimation of Storage Costs

BTC-Relay only stores Bitcoin block headers. Transactions are not stored directly in the relay – this responsibility lies with other components or applications interacting with BTC-Relay.

The size of the necessary storage allocation hence grows linear with the length of the Bitcoin blockchain (tracked in BTC-Relay) – specifically, the block headers stored in `BlockHeaders` which are referenced in `Chains` or in an entry of `Forks`.

Recall, for each block header, BTC-Relay merely stores:

- the 32 byte `blockHash`
- 4 byte `blockHeight` (twice for better referencing, so 8 bytes in total)
- the 32 byte `merkleRoot`
- the 4 byte `timestamp` (u32, wrapped in DateTime)
- and the 32 byte `target` (u256 integer)

That is, in total 108 bytes per submitted Bitcoin block header (fork or main chain block).

For example, if we were to sync BTC-Relay from the genesis block all the way to block height **612450**, the storage requirements would amount to around **66 MB** – an arguably negligible number. At the current rate and under this configuration, we would reach 100 MB in about 10 years.

Note: Fork submissions take up additional storage space, depending om the length of the tracked fork. Compared to the (already negligible) size of the main chain block headers, this overhead is negligible. Furthermore, fork entries are deleted when a chain reorganization occurs, while old entries (with sufficient confirmations) can be subject to pruning.

29.2 BTC-Relay Optimizations

29.2.1 Pruning

Optionally, to further reduce storage requirements (e.g., in case more data is to be stored per block in the future), *pruning* of `Chains` and `BlockHeaders` can be introduced. While the storage overhead for Bitcoin itself may be acceptable, Polkadot is expected to connect to numerous blockchains and tracking the entire blockchain history for each could unnecessarily bloat Parachains (even more so, if Parachains are non-exclusive to specific blockchains).

With pruning activated, `Chains` would be implemented as a FIFO queue, where sufficiently old block headers are removed from `BlockHeaders` (and the references from `Chains` and `Forks` accordingly). The pruning depth can be set to e.g. 10 000 blocks. There is no need to store more block headers, as verification of transactions contained in older blocks can still be performed by requiring users to *re-spend*. More detailed analysis of the spending behavior in Bitcoin, i.e., UTXOs of which age are spent most frequently and at which "depth" the spending behavior declines, can be considered to optimize the cost reduction.

Warning: If pruning is implemented for `BlockHeaders` and `Chains` as performance optimization, it is important to make sure there are no `Forks` entries left which reference pruned blocks.

29.2.2 Batch Submissions

Currently, BTC-Relay supports submissions of a single Bitcoin block header per transaction.

To reduce network load on the Parachain, multiple block header submissions can be batched into a single transaction. Note: the improvement in terms of data broadcast to the Parachain depends on the fixed costs per Parachain transaction (if Parachain transactions are considered a negligible cost, batching may be unnecessary).

The potential improvement can especially be useful for blockchains with higher block generation rates than Bitcoin's 1 block / 10 minutes, as in the case of Ethereum.

29.2.3 Outlook on Sub-Linear Verification in Bitcoin

Recently, so called "sub-linear" light clients were proposed for Bitcoin, which use random sampling of blocks to deter malicious actors from tricking light clients into accepting an invalid chain.

We refer the reader to the Superblock NiPoPoW and the FlyClient papers for more details.

As of this writing, both techniques require soft fork modifications to Bitcoin, if to be deployed in a secure and useful manner. The design of BTC-Relay as specified in this document (split into storage, verification, parser, etc. components) thereby allows for introduction of additional verification methods, without major modifications to the architecture.

CHAPTER

THIRTY

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 to lock their DOT in the system and operate Vault clients. This problem is amplified as the BTC Parachain does not exist in isolation and Vaults can choose to participate in other protocols (e.g., staking, stablecoin issuance) as well. In the following, we outline the constraints we see, a viable incentive model, and pointers to further research questions we plan to solve by getting feedback from potential Vaults as well as quantitative modeling.

30.1 Currencies

The BTC-Parachain features four asset types:

- *BTC* - the backing-asset (locked on Bitcoin)
- *interBTC* - the issued cryptocurrency-backed asset (on Polkadot)
- *DOT* - the currency used to pay for transaction fees
- *COL* - the currencies used as collateral (e.g., *DOT*, *KSM*, ...)

30.2 Actors: Roles, Risks, and Economics

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

We can classify four groups of users, or actors, in the interBTC bridge.

Below, we provide an overview of the protocol role, the risks, and the economics of each of the four actors. Specifically, we list the following:

- **Protocol role** The intended interactions of the actor with the bridge.
- **Risks** An informal overview of the risks of using the bridge.
- **Economics** An informal overview of the following economic factors:
  - **Income**: revenue achieved by using the bridge. We differentiate between *primary* income that is achieved when the bridge works as intended and *secondary* income that is available in failure cases (e.g., misbehavior of Vaults or Users).

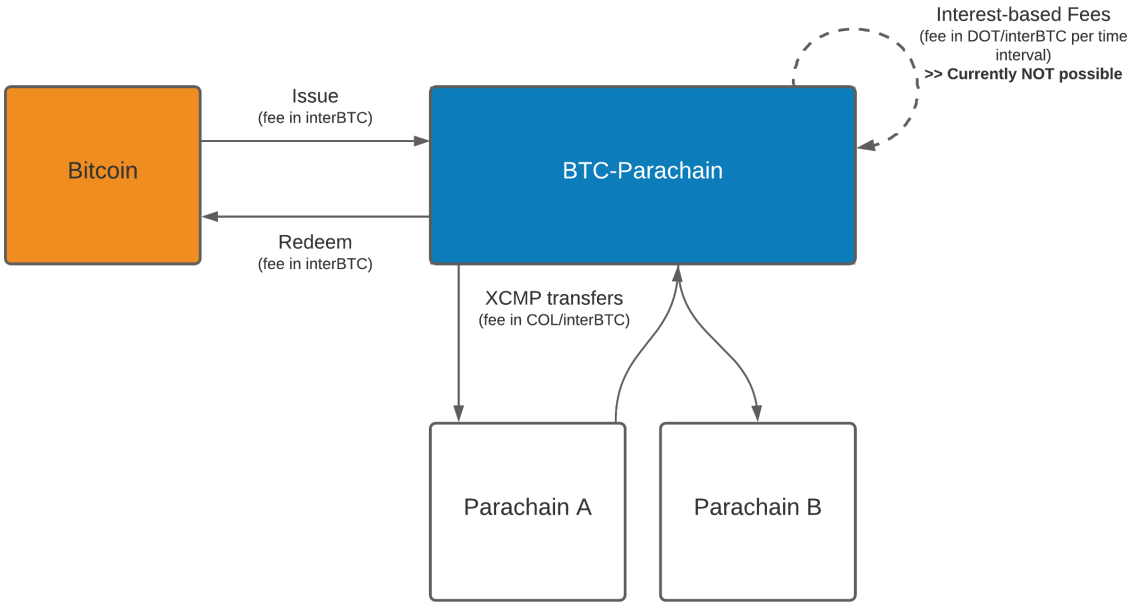


Fig. 30.1: High-level overview of fee accrual in the BTC-Parachain (external sources only).

- **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)

30.2.1 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.
- **Risks** A user gives up custody over their BTC to a Vault. The Vault is over-collateralized in *COL*, (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 steal the BTC. Users will be reimbursed with *COL* in that case - not the currency they initially started out with.
- **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.
  - **Income**
    - Primary: Use of interBTC in external applications (outside the bridge)
    - Secondary: Slashed collateral of Vaults on failed redeems paid in *COL*, see *cancelRedeem*
    - Secondary: Slashed collateral of Vaults on premium redeems paid in *COL*, see *requestRedeem*
    - Secondary: Arbitrage interBTC for *COL*, see *liquidationRedeem*
  - **Internal Cost**
    - Issue and redeem fees paid in *interBTC*, see *requestIssue* and *requestRedeem*
    - Parachain transaction fees on every transaction with the system paid in *DOT*
    - Optional: Additional BTC fees on refund paid in *BTC*, see *executeRefund*

  - **External Costs**
    - *None*
  - **Opportunity Cost**
    - Locking BTC with a Vault that could be used in another protocol

30.2.2 Vaults

- **Protocol role** Vaults lock up 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.
- **Risks** A Vault backs a set of interBTC with collateral. If the exchange rate of the *COL/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.
- **Economics** Vaults hold *COL* and thus have exposure to the *COL* price against *BTC*. Vaults inherently make a bet that *COL* will either stay constant or increase in value against BTC – otherwise they would simply exchange *COL* against their preferred asset(s). This is a simplified view of the underlying problem. We assume Vaults 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.
  - **Income**
    - Primary: Issue and redeem fees paid in *interBTC*, see *requestIssue* and *requestRedeem*
    - Secondary: Slashed collateral of Users on failed issues paid in *DOT*, see *cancelIssue*
    - Secondary: Slashed collateral of Vaults on failed replace paid in *COL*, see *cancelReplace*
    - Secondary: Additional BTC of Users on refund paid in *BTC*, see *executeRefund*
  - **Internal Cost**
    - Parachain transaction fees on every transaction with the system paid in *DOT*
    - Optional: Slashed collateral on failed redeems paid in *COL*, see *cancelRedeem*
    - Optional: Slashed collateral on theft paid in *COL*, see *report_vault_theft*
    - Optional: Slashed collateral on liquidation paid in *COL*, see *liquidateVault*
  - **External Costs**
    - Vault client operation/maintenance costs
    - Bitcoin full node operation/maintenance costs
  - **Opportunity Cost**
    - Locking *COL* that could be used in another protocol

30.2.3 Relayers

- **Protocol role** 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 misbehaving Vaults who have allegedly stolen BTC (move BTC outside of BTC Parachain constraints).
- **Risks** Relayers have no financial stake in the system. Their highest risk is that they do not get sufficient rewards for submitting transactions (i.e., reporting Vault theft or submitting BTC block headers).
- **Economics** 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.

- **Income**
  - Primary: *None*
  - Secondary: Slashed collateral on theft paid in *COL*, see *report_vault_theft*
- **Internal Cost**
  - Parachain transaction fees on every transaction with the system paid in *DOT*
- **External Costs**
  - Bitcoin full node operation/maintenance costs
  - Parachain node operation/maintenance costs
- **Opportunity Cost**
  - *None*

Note: Operating a Vault requires access to a Bitcoin wallet. Currently, the best solution to access a Bitcoin wallet programmatically is by using the inbuilt wallet of the Bitcoin core full node. Hence, the Vault client is already running a Bitcoin full node. Therefore, the Relayer and the Vault roles are bundled together in the implementation of the Vault/Relayer clients.

30.2.4 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
- **Risks** Collators have no financial stake in the system. Hence running a collator has no inherent risk.
- **Economics** Collators have to run a full node for the parachain incurring external costs. In return, they can receive fees.
  - **Income**
    - Primary: Parachain transaction fees on every transaction with the system paid in *DOT*
  - **Internal Cost**
    - *None*
  - **External Costs**
    - Parachain full node operation/maintenance costs
  - **Opportunity Cost**
    - *None*

30.3 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., 150%) more collateral than minted interBTC to ensure security against exchange rate fluctuations (see *SecureCollateralThreshold*). Dynamically modifying the secure collateral threshold 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 1.5 BTC worth of collateral (e.g. in DOT), resulting in a 250% 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. As such, it is 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).

30.4 External Economic Risks

A range of external factors also have to be considered in the incentives for the actors.

- **Exchange rate fluctuations**. Vaults have a risk of having their *COL* liquidated if the *COL/BTC* exchange rate drops below the *LiquidationThreshold*. In this case, the collateral is liquidated as described in *Vault Liquidations*. Liquidations describe that users can restore the *interBTC* to *BTC* peg by burning *interBTC* for *COL*. However, in a continuous drop of the exchange rate the value of *COL* will fall below the value of the burned *interBTC*. As such, the system relies on actors that execute fast arbitrage trades of *interBTC* for *COL*.
- **Counterparty risk for BTC in custody**. When a user locks BTC with the Vault, they implicitly sell 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).
- **interBTC Liquidity Shortage**. Related to the exchange rate fluctuations, arbitrageurs rely on their own *interBTC* or a place to buy *interBTC* for *COL* to execute an arbitrage trade. In a *interBTC* liquidity shortage, simply not enough *interBTC* might be available. In combination with a severe exchange rate drop (more than *LiquidationThreshold* - 100%), there will be no financial incentive to restore the *interBTC* to *BTC* peg.
- **BTC and COL Liquidity Shortage**. *interBTC* is a "stablecoin" in relation to *BTC*. Since owning *interBTC* gives a claim to redeem *BTC*, the price of *interBTC* to *BTC* should remain roughly the same. However, in case *interBTC* demand is much larger than either the *COL* and/or *BTC* supply, the price for *interBTC* might increase much faster than *BTC*. In practice, this should not be an issue since the collateral thresholds are computed based on the *BTC* to *COL* exchange rate rather than the *interBTC* rates.
- **Opportunity costs**: Each actor 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 and Keepers might pick other protocols to earn interest on their DOT holdings.

CHAPTER

THIRTYONE

FEE MODEL

The interBTC bridge uses conceptually three different and independent fee models:

1. **interBTC Fee Model**. The internal interBTC bridge fee model covers any payments made through the operation of the bridge, e.g., the issue, redeem, or replace processes. This process concerns Users, Vaults (and its Nominators), and Relayers.
2. **Griefing Fee Model**. These are *DOT* fees paid to the Vault on a failed issue or replace.
3. **Transaction Fee Model**. The transaction fees are essentially the *DOT* fees paid on every transaction to the Collators.

31.1 Payment Flows

We detail the payment flows for both models in the figure below:

31.2 interBTC Fee Model

31.2.1 Issue and Redeem Fee Distribution

The primary fees in interBTC are paid by users during *Issue* and *Redeem* as a relative fee on the issued or redeemed interBTC.

Vaults earn fees based on their currently backed interBTC (i.e., `vault.issuedTokens`). To reduce variance of payouts, the interBTC bridge implements a pooled fee model. This means that Vaults earn a share of each fee based on their share of issued interBTC in the bridge.

If the Vault does not back interBTC then it does not have a stake in the system and it will not receive any rewards, i.e., its stake is 0. Conversely, if the Vault has any issued interBTC, the Vault will earn rewards. Thus, only Vaults directly locking Bitcoin in the system will earn rewards from users.

Each time a user issues or redeems interBTC, they pay the following fees to a global fee pool:

- **Issue Fee**: A relative fee paid based on the requested interBTC paid in *interBTC*, for the current parameterization see *IssueFee*
- **Redeem Fee**: A relative fee paid based on the requested BTC paid in *interBTC*, for the current parameterization see *RedeemFee*

Note: Since redeem fees are backed by the Vault, they must use the Replace protocol to exit the system. To solve this issue, we allow self-redeems based on the Vault's account ID which sets the redeem fee to zero.

From this fee pool 100% is distributed among all active Vaults.

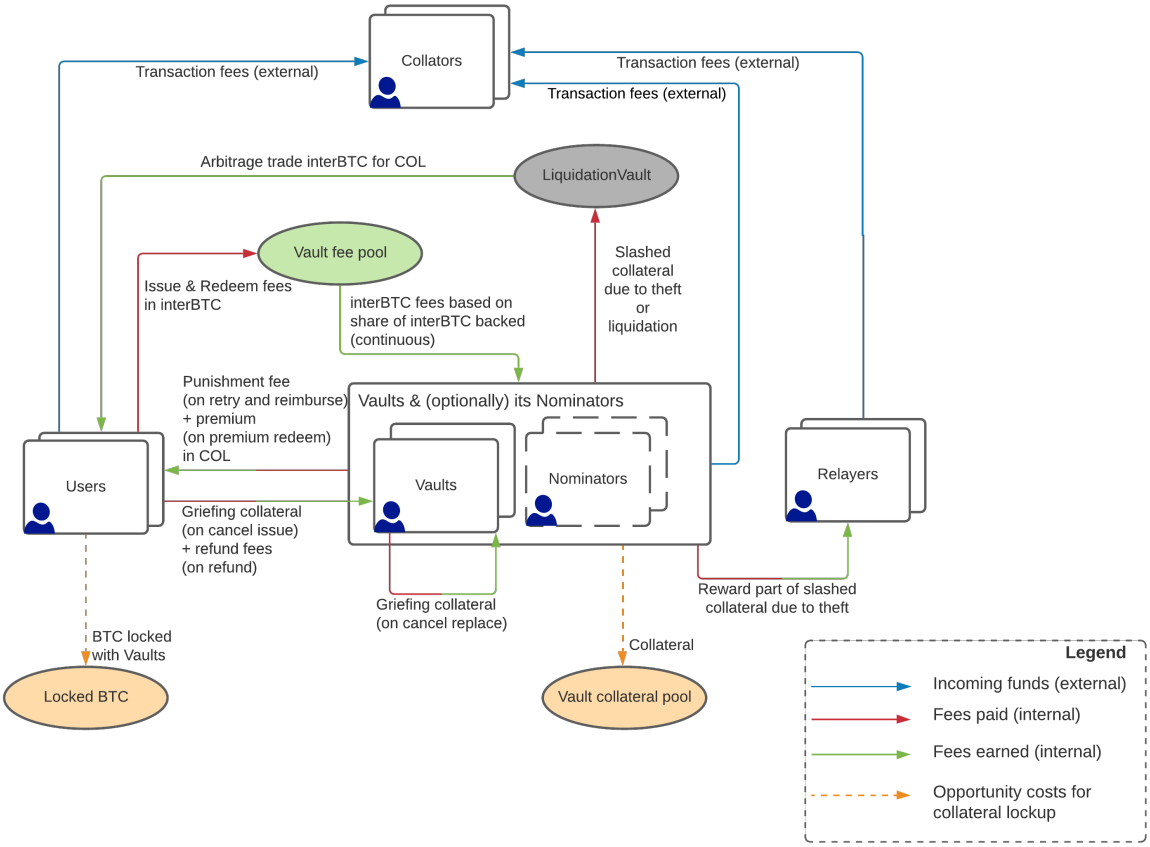


Fig. 31.1: Detailed overview of fee accrual in the interBTC bridge, showing interBTC Fee Model and Transaction Fee Model payment flows, as well as opportunity costs.

Each Vault is receiving a fair share of this fee pool by considering its stake in the system. The stake in the system is just the amount of BTC a vault is currently insuring with collateral. Calculating the rewards for a Vault is equivalent to this formula:

$$\text{rewards} = \text{stake}(\text{totalRewards}/\text{totalStake})$$

Eq. 1: Vault reward distribution.

Note: As an example, if we had 1 interBTC to distribute among all Vaults with total stake 200 and assuming the individual Vault has stake 100, the reward share could be calculated by: 100 * (1 interBTC / 200) = 0.5 interBTC

To be exact, the stake is expressed as the interBTC issued by a Vault. The issued interBTC are the interBTC currently being backed by the Vault. This shows how much a Vault's collateral is "occupied" by users:

$$\text{stake} = \text{interBTCIssued}$$

Eq. 2: Parameterized stake updates.

Stake Updates

Whenever a Vault is increasing or decreasing the number of issued interBTC it is backing, we MUST update their stake in the reward pool accordingly. These updates are achieved through the issue, redeem, and replace operations.

Fee Payouts

The Vault fee is paid each time an Issue or Redeem request is executed. Naively speaking, the bridge behaves as if on each issue and redeem, the bridge would loop through all Vaults to determine their share of stake, i.e., `vault.issuedTokens / totalSupply`, and distribute a percentage of the paid fees to the Vault.

Since a naive implementation would result in unbounded iteration, the fee payout is implemented in a different way. However, the outcome it is equivalent to the naive approach. The payouts are based on the pull-based Scalable Reward Distribution with Changing Stake Sizes. This scheme allows rewards to be drawn by each Vault (and Nominator) individually and at any time without the interBTC bridge having to loop over all Vaults each time rewards are paid out. Read the *Excursion: Scalable Reward Distribution* section if you would like to understand how the payout system works under the hood.

31.2.2 Griefing Fees

Griefing collateral is locked on *requestIssue* and *requestReplace* to prevent *Griefing*. If the requests are indeed cancelled, the griefing collateral is paid to the free balance of the Vault that locked collateral in vain. On successful execute, the griefing collateral is refunded to the party making the request.

- **Issue Griefing Collateral**: A relative collateral locked based on the requested interBTC paid in *DOT*, for the current parameterization see *IssueGriefingCollateral*
- **Replace Griefing Collateral**: A relative collateral locked based on the request interBTC paid in *DOT*, for the current parameterization see *ReplaceGriefingCollateral*

Griefing Collateral Currency

The currency that is used for griefing collateral used for issue and replace. This value is set to the currency of the transaction fees, i.e., *DOT*, regardless of the vault's configured backing collateral currency.

31.2.3 Premium Redeem Fee

When Vaults are below the *PremiumRedeemThreshold*, users are able to redeem with the Vault and receive an extra "bonus" slashed from the Vault's collateral. This mechanism is to ensure that (1) Vaults have a higher incentive to stay above the *PremiumRedeemThreshold* and (2) users have an additional incentive to redeem with Vaults that are close to the *LiquidationThreshold*.

- **Premium Redeem Fee**: A relative fee slashed from the Vault's collateral paid to the user in the vault's *COL* if a Vault is below the *PremiumRedeemThreshold*, for the current parameterization see *PremiumRedeemFee*

31.2.4 Punishment Fees

Punishment fees are slashed from the Vault's collateral on failed redeems. A user can choose to either retry with another Vault or reimburse the *interBTC* amount. In both cases, the a punishment fee is deducted from the Vault's collateral to ensure that Vault's are punished in both cases.

- **Punishment Fee**: A relative fee slashed from the Vault's collateral paid to the user in the vault's *COL* if a Vault failed to execute a redeem request, for the current parameterization see *PunishmentFee*

31.2.5 Theft Fee

Relayers receive a reward for reporting Vaults for committing theft (see *report_vault_theft* and *report_vault_double_payment*).

- **Theft Fee**: A relative fee slashed form the Vault's collateral paid to the Relayer in the vault's *COL* if a Vault commits theft, for the current parameterization see *TheftFee*

31.2.6 Arbitrage

Arbitrage trades are executed by anyone that exchanges *interBTC* for *COL* against the LiquidationVault. The LiquidationVault is essentially an AMM with two balances:

- *issuedTokens*: amount of *interBTC* that have been liquidated through safety failures, see *Vault Liquidations*
- *lockedCollateral*: amount of *COL* that have been confiscated through safety failures, see *Vault Liquidations*

Anyone can now burn *interBTC* for *COL* at the exchange rate of the `issuedTokens/lockedCollateral` from the LiquidationVault. As the *LiquidationThreshold* is strictly above the current exchange rate of the *BTC/COL* pair at the time of liquidation, this *should* represent an arbitrage opportunity: the value of burned *interBTC* should be lower than the value of received *COL*.

However, in practice, the arbitrage process might not work as intended. See *External Economic Risks* for a discussion of related problems. Note that there are no fees being collected to execute trades against the LiquidationVault.

31.2.7 Excursion: Scalable Reward Distribution

We recommend reading first the Scalable Reward Distribution paper and then the extension for changing rewards. Note that this scheme is "just" an efficient equivalent of the Vault distribution outlined above. Last, we extend this scheme to account for *Vault Nomination* and *Vault Liquidations*. The adopted scheme is described in the README of the implementation.

Notable changes to the Scalable Reward Distribution with Changing Rewards are:

- **Staking Pools** Fees are forwarded to a *Reward Pool* and then distributed to a *Staking Pool*. There is one Staking Pool for each Vault and all of its Nominators.
- **Slashing** On liquidation of Vaults, no more fees are forwarded to the Staking Pool of that Vault.

See the figure below for an indication how the Staking Pools are used.

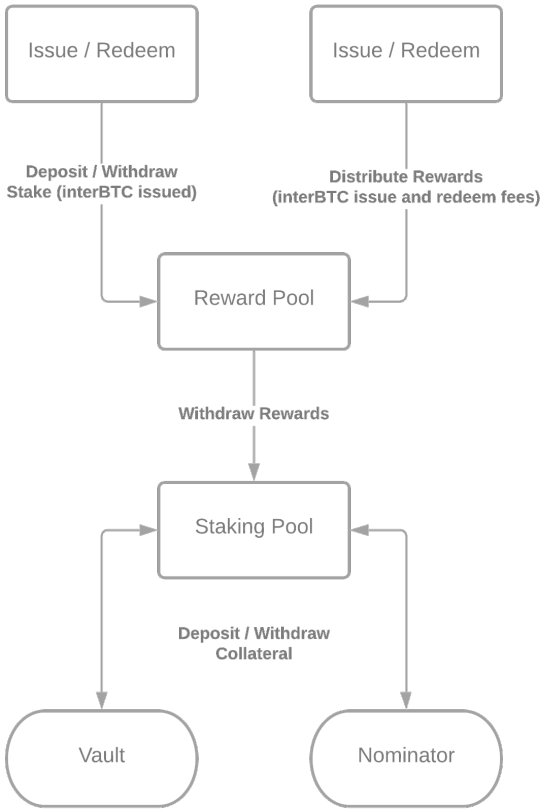


Fig. 31.2: Distribution of fees according to Staking Pools. Each Vault and all its Nominators are represented by a Staking Pool. This allows to distribute the applicable fees based on the global share of issued interBTC based on the stake of the Staking Pool as well as an individual distribution of fees between the Vault and its Nominators based on their share in the pool.

In the scalable reward distribution, a single source of truth is used to calculate rewards: the "stake". The "stake" can be any numeric representation. In interBTC, stake is defined as: *the current amount of issued interBTC*. A Vault's stake is adjusted based on the change in issued interBTC - for instance we increase the issued interBTC on successful issues and decrease this on executed redeems.

Note: For example, if a Vault executes issue requests amounting to 2,456,000 interSatoshi (smallest denomination) being added to the system, its stake would increase by 2,456,000. If the Vault then executes redeem requests, its rewards are reduced. So if the Vault redeems all 2,456,000 interSatoshi, its stake is 0 again. On a liquidation, this is again set to zero since the Vault no longer backs these tokens.

Now, each Vault's rewards are calculated according to the following formula (equivalent to Eq. 1):

$$\text{deposit}(\text{stakeDelta}) : \text{rewardTally} \mathrel{+}= \text{rewardPerToken} \cdot \text{stakeDelta}$$

$$\text{stake} \mathrel{+}= \text{stakeDelta}$$

$$\text{totalStake} \mathrel{+}= \text{stakeDelta}$$

$$\text{distributeReward}(\text{reward}) : \text{rewardPerToken} \mathrel{+}= \text{reward}/\text{totalStake}$$

$$\text{computeReward}() : \text{return stake} \cdot \text{rewardPerToken} - \text{rewardTally}$$

Eq. 3: Vault reward distribution using the SRD.

Definitions

- **stake:** the amount of interBTC issued by this Vault.
- **reward_tally**: the Vault's accumulated rewards (can be negative or positive).
- **stake_delta**: the stake impact based on issuing or redeeming interBTC.
- **total_stake**: the total amount of interBTC issued by all Vaults.
- **reward_per_token**: the current reward per current stake (the total_stake).
- **reward**: the rewards paid from issue and redeem requests.

The reward is influenced by the total of all stakes. So the share of rewards paid to a Vault is determined by how many other Vaults are in the system and their individual stake.

Example Without Nomination

Current stake

Note: stake is always non-zero.

- Vault Alice has a stake of 250
- Vault Bob has a stake of 30
- Vault Charlie has a stake of 100

The total stake is therefore `380`.

Reward claims

Let's assume there is a total of 1 interBTC in the reward pool based on the accumulated issue and redeem request. Then the `reward_per_token = 1 interBTC / 380`.

- Vault Alice has a claim of `250 * 1 interBTC/380 = 0.6578947368421052 interBTC`
- Vault Bob has a claim of `30 * 1 interBTC/380 = 0.07894736842105263 interBTC`
- Vault Charlie has a claim of `100 * 1 interBTC/380 = 0.2631578947368421 interBTC`

Example With Nomination

Current stake

Note: stake is always non-zero.

- Vault Alice and her Nominators have a stake of 250. Alice is fully nominated such that Alice is backing 200 and her Nominators are backing 50.
- Vault Bob has a stake of 30
- Vault Charlie has a stake of 100

The total stake is therefore `380`.

Reward claims

Let's assume there is a total of 1 interBTC in the reward pool based on the accumulated issue and redeem request. Then the `reward_per_token = 1 interBTC / 380`.

- Vault Alice has a claim of `200 * 1 interBTC/380 = 0.526315789 interBTC`
- Alice's Nominators have a claim of `50 * 1 interBTC/380 = 0.131578947 interBTC`
- Vault Bob has a claim of `30 * 1 interBTC/380 = 0.07894736842105263 interBTC`
- Vault Charlie has a claim of `100 * 1 interBTC/380 = 0.2631578947368421 interBTC`

31.3 Transaction Fee Model

The interBTC bridge chain adopts the Polkadot relay chain model with *DOT* as the native currency for paying transaction fees. In this model, collators receive 100% of the transaction fees paid by Users, Vaults, and Relayers. We refer to the official Polkadot documentation for full details.

CHAPTER

THIRTYTWO

LICENSE

CHAPTER

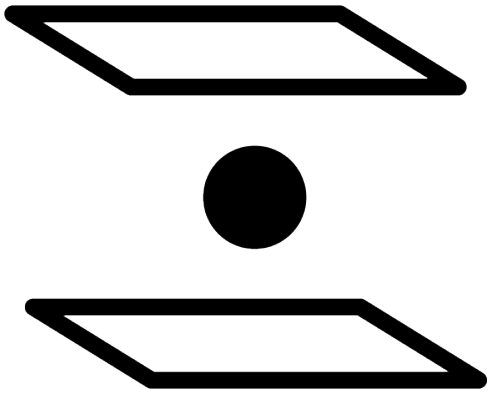
THIRTYTHREE

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