PolkaBTC Documentation: BTCRelay

Release v1.0

Interlay

Jan 12, 2020

INTRODUCTION

CHAPTER

ONE

BTC-RELAY AT A GLANCE

1.1 Overview

BTC-Relay is the key component of the BTC Parachain on Polkadot. It's 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.

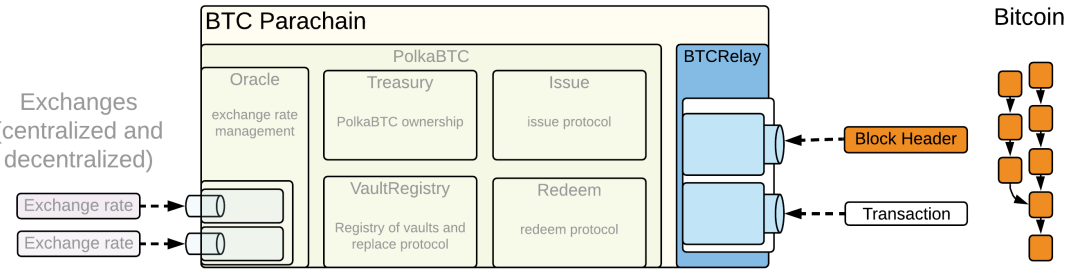

Fig. 1: BTC-Relay (highlighted in blue) is a key component of the BTC Parachain: it is necessary to verify and keep track of the state of Bitcoin.

1.2 How to Use this Document

This document provides a specification for BTC-Relay in the form of a Polkadot Parachain acting as a Bridge to Bitcoin, to be implemented on Substrate.

Before implementing or using BTC-Relay on Polkadot, make yourself familiar with this specification and read up on any content you are unfamiliar with by following the provided links (e.g. to academic papers and the Bitcoin developer reference).

1.2.1 Recommended Background Reading

We also recommend readers, unfamiliar with the problem of cross-chain communication, to study the following papers - in addition to acquiring a base understanding for the operation and security model of distributed ledgers.

- **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). [PDF]

- **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) [PDF]
- **Proof-of-Work Sidechains**. *Workshop on Trusted Smart Contracts, Financial Cryptography* Kiayias, A., & Zindros, D. (2018) [PDF]
- **Enabling Blockchain Innovations with Pegged Sidechains**. *Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra A., Timon J., & Wuille, P*. (2019) [PDF]

CHAPTER

TWO

BTC-RELAY ARCHITECTURE

BTC-Relay is a component / module of the BTC Parachain. It's main functionality is verification and storage of Bitcoin block headers, as well as verification of Bitcoin transaction inclusion proofs. Below, we provide an overview of it's components, as well as relevant actors - offering references to the full specification contained in the rest of this document.

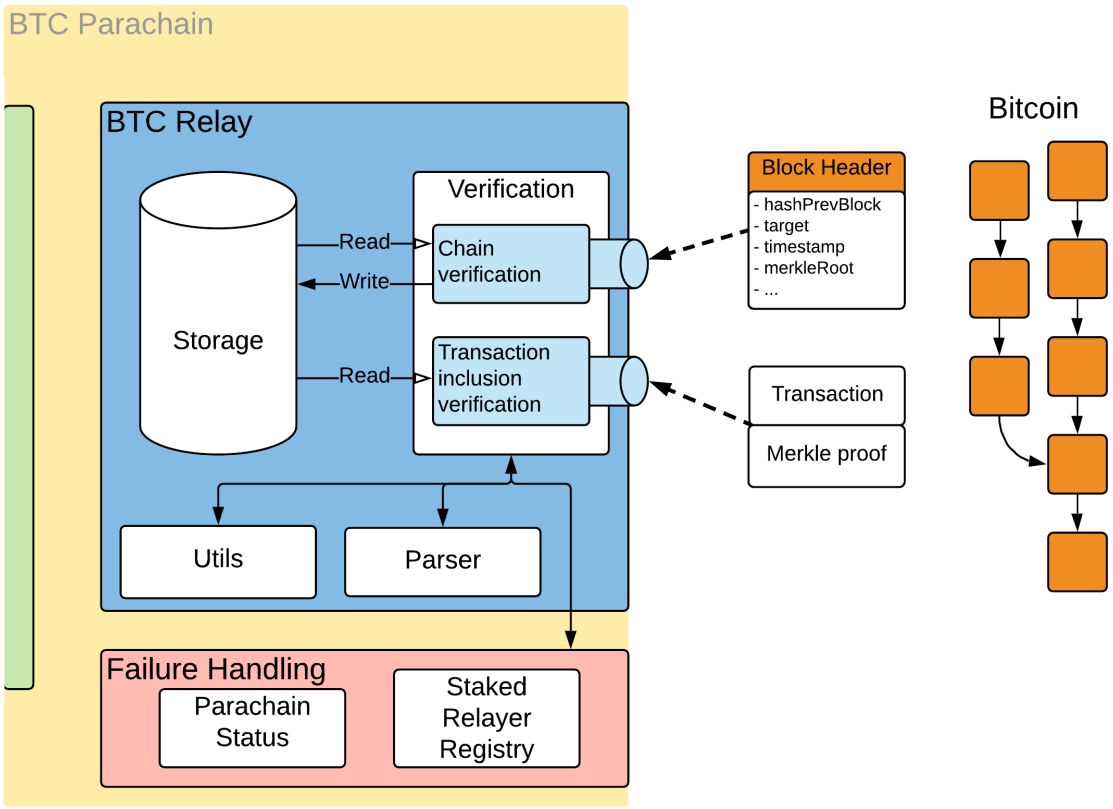

Fig. 1: Overview of the BTC-Relay architecture. Bitcoin block headers are submitted to the Verification Component, which interacts with the Utils, Parser and Failure Handling components, as well as the Parachain Storage.

2.1 Actors

BTC-Relay differentiates between the following actors:

- **Users** - BTC Parachain users which interact with the BTC-Relay directly or with other modules which make calls to BTC-Relay.

- **Staked Relayers** - staked relayers lock up collateral in the BTC Parachain and are responsible for running Bitcoin full nodes and verify that blocks submitted to BTC-Relay are valid (and that transactional data for these blocks is available). Staked relayers can halt BTC-Relay in case a failure is detected. See failure-handling for more details.

Note: While any user can submit block headers to BTC-Relay, this role can be assigned to staked relayers, given these participants already run Bitcoin full nodes and check validity of stored blocks.

- **(Parachain) Governance Mechanism** - the correct operation of the BTC Parachain, and hence of BTC-Relay, is overseen by the Parachain's governance mechanism (e.g. committee of consensus participants and other key actors of the BTC Parachain).

Note: At the time of writing, the exact composition of the governance mechanism / commitee is not fully defined in Polkadot. This specification assumes the existence and correct operation of this mechanism - although modifications may be necessary to accomodate complexities and additional functional requirements, once the governance mechanism is fully specified.

2.2 Components

2.2.1 Storage

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

2.2.2 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).

2.2.3 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.

2.2.4 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.

2.2.5 Failure Handling

BTC-Relay interacts with the Failure Handling component of the BTC Parachain: since failures of BTC-Relay impact the entire operation of the BTC Parachain, the Failure Handling components tracks and reacts to changes in BTC-Relay's operation.

Specifically, Staked Relayers and the Governance Mechanism can restrict or entirely halt the operation of BTC-Relay, which in turn affects the available functionality of the entire BTC Parachain. See Failure Handling section of the PolkaBTC specification document for a full specification of failure handling on the BTC Parachain, and the interactions with BTC-Relay. On a high level, BTC-Relay can enter four possible states:

- `RUNNING`: correct operation, all functions are available.
- `PARTIAL`: transaction verification is disabled for blocks above a specified block height. This state is triggered by a `NO_DATA` failure by Staked Relayers (missing transaction data for submitted block headers), or manually by the Governance Mechanism.
- `HALTED`: transaction verification is entirely disabled. This state is triggered by a `INVALID` failure by Staked Relayers (invalid transaction was detected in a submitted block header) or manually by the Governance Mechanism.
- `SHUTDOWN`: submission of block headers (both main chain and forks) and transactions verification are disabled. This state can be triggered manually by the Governance Mechanism if a major failure is detected or a soft / hard fork has ocurred in Bitcoin (and hence BTC-Relay needs updating).

A full state machine, explaining how BTC-Relay transitions between different states, if provided in the Failure Handling section of the PolkaBTC specification.

Todo: Links to the PolkaBTC specification will be included once the documents are merged.

CHAPTER

THREE

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.

3.1 Block Headers

The 80 bytes block header hash encodes the following information:

Bytes	Parameter	Type	Description
4	`version`	u32	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	`time`	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.

3.2 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.

Bytes	Parameter	Type	Description
4	`version`	i32	Transaction version number.
var	`tx_in count`	uint	Number of transaction inputs.
var	`tx_in`	txIn	Transaction inputs.
var	`tx_out count`	uint	The number of transaction outputs.
var	`tx_out`	txOut	Transaction outputs.
4	`lock_time`	u32	A Unix timestamp OR block number.

3.3 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`).

3.4 Outputs

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

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

CHAPTER

FOUR

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.

4.1 Types

4.1.1 BTCBlockHeader

An 80 bytes long Bitcoin blockchain header.

Substrate

```
pub type BTCBlockHeader = [u8; 80];
```

4.2 Constants

4.2.1 DIFFICULTY_ADJUSTMENT_INTERVAL

The interval in number of blocks at which Bitcoin adjusts its difficulty. Defaults to `2016`.

Substrate

```
const DIFFICULTY_ADJUSTMENT_INTERVAL: u16 = 2016;
```

4.2.2 TARGET_TIMESPAN

Expected duration of the different adjustment interval in seconds. Defaults to `1209600` seconds or two weeks.

Substrate

```
const TARGET_TIMESPAN: U256 = 1209600;
```

4.2.3 UNROUNDED_MAX_TARGET

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

Substrate

```
const UNROUNDED_MAX_TARGET: U256 =␣
↪26959946667150639794667015087019630673637144422540572481103610249215;
```

4.3 Scalars

4.3.1 BestBlock

Byte 32 block hash identifying the current blockchain tip, i.e., the most significant block in `MainChain`.

Substrate

```
BestBlock: T::H256;
```

4.3.2 BestBlockHeight

Integer block height of `BestBlock` in `MainChain`.

Substrate

```
BestBlockHeight: U256;
```

4.4 Maps

4.4.1 BlockHeaders

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

Substrate

```
BlockHeaders: map T::H256 => BlockHeader<T::H256>;
```

4.4.2 MainChain

Mapping of `<blockHeight,blockHash>` (`<u256, byte32>`). Tracks the current Bitcoin main chain (refers to stored block headers in `BlockHeaders`).

Substrate

```
MainChain: map U256 => T::H256;
```

4.4.3 Forks

Mapping of `<forkId,Fork>` (`<u256, Fork>`), tracking ongoing forks in BTC-Relay.

Substrate

```
Forks: map U256 => Fork<Vec<T::H256>>;
```

4.5 Structs

4.5.1 BlockHeader

Representation of a Bitcoin block header.

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
`blockHeight`	u256	Height of this block in the Bitcoin main chain.
`merkleRoot`	byte32	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.
`version`	u32	[Optional] Version of the submitted block.
`hashPrevBlock`	byte32	[Optional] Block hash of the predecessor of this block.
`nonce`	u32	[Optional] Nonce used to solve the PoW of this block.

Substrate

```
#[derive(Encode, Decode, Default, Clone, PartialEq)]
#[cfg_attr(feature = "std", derive(Debug))]
pub struct BlockHeader<H256, DateTime> {
      blockHeight: U256,
      merkleRoot: H256,
      target: U256,
      timestamp: DateTime,
      // Optional fields
      version: U32,
      hashPrevBlock: H256,
      nonce: U32
}
```

4.5.2 Fork

Representation of an ongoing Bitcoin fork, tracked in BTC-Relay.

Warning: Forks tracked in BTC-Relay and observed in Bitcoin must not necessarily be the same. See *Relay Poisoning* for more details.

Parameter	Types	Description
`startHeight`	u256	Main chain block height of the block at which this fork starts (*forkpoint*).
`length`	u256	Length of the fork (in blocks).
`forkBlockHashes`	byte32[]	List of block hashes, which references Bitcoin block headers stored in `BlockHeaders`, contained in this fork (in insertion order).

Substrate

```
#[derive(Encode, Decode, Default, Clone, PartialEq)]
#[cfg_attr(feature = "std", derive(Debug))]
pub struct Fork<> {
      startHeight: U256,
      length: U256,
      forkBlockHahes: Vec<H256>
}
```

CHAPTER

FIVE

FUNCTIONS: STORAGE AND VERIFICATION

5.1 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 will be rejected**!

5.1.1 Specification

Function Signature

`initialize(blockHeaderBytes, blockHeight)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header.
- `blockHeight`: integer Bitcoin block height of the submitted block header

Returns

- `True`: if initialization is executed correctly (and for the first time only)
- `False` (or throws exception): otherwise.

Events

- `Initialized(blockHeight, blockHash)`: 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"`: raise exception if this function is called after BTC-Relay has already been initialized.

Substrate

```
fn initialize(origin, blockHeaderBytes: T::BTCBlockHeader, blockHeight: U256) ->␣
↪Result {...}
```

5.1.2 Preconditions

- This is the first time this function is called, i.e., when BTC-Relay is being deployed.

Note: Calls to `initialize` will likely be restricted through the governance mechanism of the BTC Parachain. This is to be defined.

5.1.3 Function sequence

The `initialize` function takes as input an 80 byte raw Bitcoin block header and the corresponding Bitcoin block height, and follows the sequence below:

1. Check if `initialize` is called for the first time. This can be done by checking if `BestBlock == None`. Raise `ERR_ALREADY_INITIALIZED` if BTC-Relay has already been initialized.
2. Parse `blockHeaderBytes`, extracting the `merkleRoot` using *extractMerkleRoot*, compute the Bitcoin block hash (`hashCurrentBlock`) of the block header (use *sha256d*), and store the block header data in `BlockHeaders`.
3. Store `hashCurrentBlock` in `MainChain` using the given `blockHeight` as key.
4. Set `BestBlock = hashCurrentBlock` and `BestBlockHeight = blockHeight`.
5. Return `True`.

Warning: Attention: the Bitcoin block header submitted to `initialize` must be in the Bitcoin main chain - this must be checked outside of the BTC Parachain **before** making this function call! A wrong initialization will cause the entire BTC Parachain to fail, since verification requires that all submitted blocks **must** (indirectly) point to the initialized block (i.e., have it as ancestor, just like the actual Bitcoin genesis block).

5.2 storeMainChainBlockHeader

Method to submit block headers to the BTC-Relay, which extend the Bitcoin main chain (as tracked in `MainChain` in BTC-Relay). This function calls *verifyBlockHeader* proving the 80 bytes Bitcoin block header as input, and, if the latter returns `True`, extracts from the block header and stores (i) the hash, height and Merkle tree root of the given block header in `BlockHeaders` and (ii) the hash and block height in `MainChain`.

5.2.1 Specification

Function Signature

`storeMainChainBlockHeader(blockHeaderBytes)`

Parameters

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

Returns

- `True`: if `verifyBlockHeader` returns `True` and the extraction and storage of the block header data was executed correctly.
- `False` (or throws exception): otherwise.

Events

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

Errors

- `ERR_SHUTDOWN = "BTC Parachain has shut down"`: the BTC Parachain has been shutdown by a manual intervention of the governance mechanism.
- `ERR_NOT_MAIN_CHAIN = "Main chain submission indicated, but submitted block is on a fork"`: raise exception if the block header submission indicates that it is extending the current longest chain, but is actually on a (new) fork.

Substrate

```
fn storeMainChainBlockHeader(origin, blockHeaderBytes: T::BTCBlockHeader) ->␣
↪Result {...}
```

5.2.2 Preconditions

- The failure handling state must not be set to `SHUTDOWN: 3`.
- The to-be-submitted Bitcoin block header must extend `MainChain` as *tracked by the BTC-Relay*.

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 Analysis*).

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`).

5.2.3 Function sequence

The `storeMainChainBlockHeader` function takes as input the 80 byte raw Bitcoin block header and follows the sequence below:

1. Check if the failure handling state is set to `SHUTDOWN`. If true, raise `ERR_SHUTDOWN` and return.
2. Check that the submitted block header is extending the `MainChain` of BTC-Relay. That is, `hashPrevBlock` (extract using *extractHashPrevBlock*) must be equal to `BestBlock`. Raise `ERR_NOT_MAIN_CHAIN` error if this check fails.
3. Call *verifyBlockHeader* passing `blockHeaderBytes)` as function parameter. If this call **does not return** `True` (i.e., fails or returns `False`), then abort and return `False`.
4. Extract the `merkleRoot` (*extractMerkleRoot*), `timestamp` (*extractTimestamp*) and `target` (*extractNBits* amd *nBitsToTarget*) from `blockHeaderBytes`, and compute the block hash using *sha256d* (passing `blockHeaderBytes` as parameter)..
5. Store the `height`, `merkleRoot`, `timestamp` and `target` as a new entry in the `BlockHeaders` map, using `hashCurrentBlock` as key.
    - `hashCurrentBlock` is the double SHA256 hash over the 80 bytes block header and can be calculated via *sha256d*.
    - `merkleRoot` is the root of the transaction Merkle tree of the block header. Use *extractMerkleRoot* to extract from block header.

- `height` is the blockchain height of the submitted block header. Compute by incrementing the height of the block header referenced by `hashPrevBlock` (retrieve from `BlockHeaders` using `hashPrevBlock` as key).

6. Store `hashCurrentBlock` as a new entry in `MainChain`, using `blockHeight` as key.
7. Emit a `StoreMainChainBlockHeader` event using `height` and `hashCurrentBlock` as input (`StoreMainChainHeader(height, hashCurrentBlock)`).
8. Return `True`.

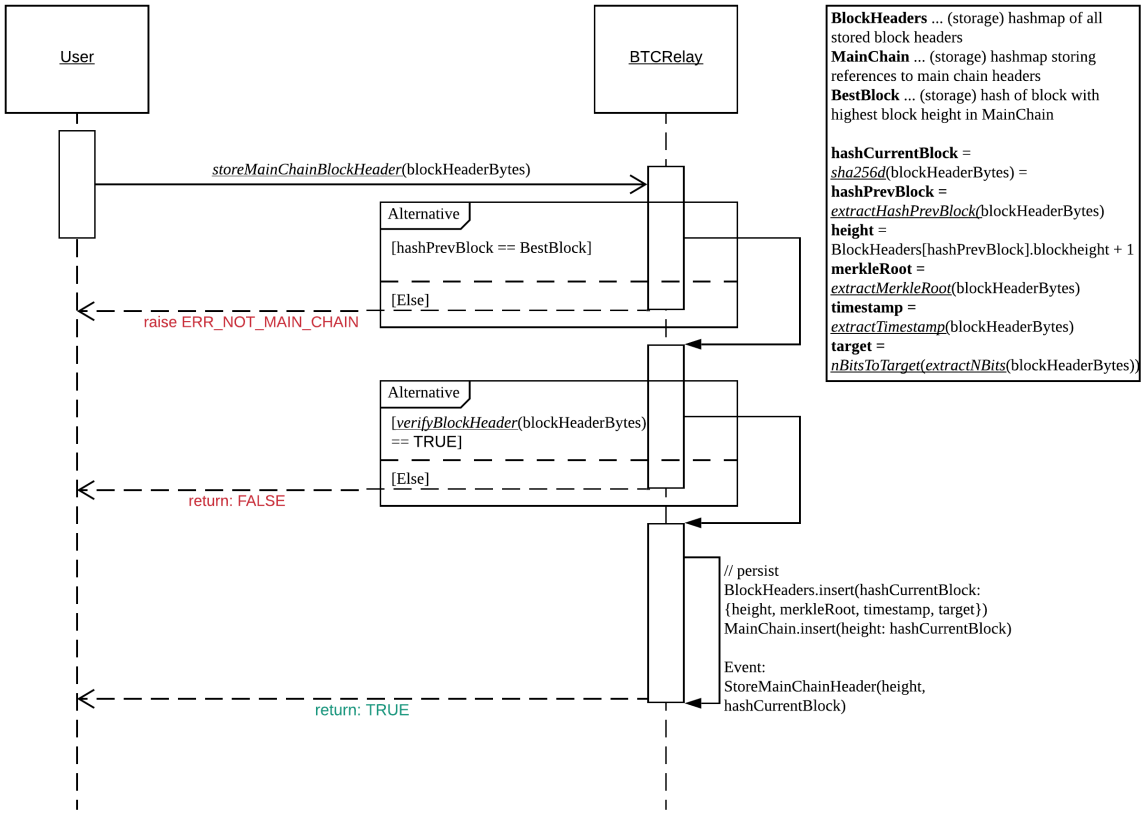

Fig. 1: Sequence diagram showing the function sequence of *storeMainChainBlockHeader*.

5.3 storeForkBlockHeader

Method to submit block headers to the BTC-Relay, which extend an existing (as tracked in `Forks` in BTC-Relay) of create a new *fork*. This function calls *verifyBlockHeader* passing the 80 bytes Bitcoin block header as parameter, and, if the latter returns `True`, extracts from the block header and stores (i) the hash, height and Merkle tree root of the given block header in `BlockHeaders` and (ii) the hash of the block header as well as the starting block height of the fork and the current length (1 if a new fork) in `Forks`.

5.3.1 Specification

Function Signature

`storeForkHeader(blockHeaderBytes, forkId)`

Parameters

- `blockHeaderBytes`: 80 byte raw Bitcoin block header.
- `forkId`: integer tracked fork identifier. Set to `0` if a new fork is being created (default).

Returns

- `True`: if the block header passes all checks and creates a new or extends an existing fork of the currently known longest chain
- `False` (or raises exception): otherwise.

Events

- `StoreForkHeader(forkId, blockHeight, blockHash)`: if the submitted block header is on a fork, emit an event with the fork's id (`forkId`), block height (`blockHeight`) and the (PoW) block hash (`blockHash`).
- `ChainReorg(newChainTip, startHeight, forkId)`: 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 (`newChainTip`), the start block height of the fork (`startHeight`) and the fork identifier (`forkId`).

Errors

- `ERR_SHUTDOWN = "BTC Parachain has shut down"`: the BTC Parachain has been shutdown by a manual intervention of the governance mechanism.
- `ERR_INVALID_FORK_ID = "Incorrect fork identifier"`: raise an exception when a non-existent is passed.
- `ERR_FORK_PREV_BLOCK = "Previous block hash does not match last block in fork submission"`: raise exception if the block header does not reference the highest block in the fork specified by `forkId` (via `prevBlockHash`).
- `ERR_NOT_FORK = "Indicated fork submission, but block is in main chain"`: raise exception if the submitted block header is actually extending the current longest chain tracked by BTC-Relay (`MainChain`) instead of a fork.

Substrate

```
fn storeForkBlockHeader(origin, blockHeaderBytes: T::BTCBlockHeader, forkId: U256)␣
↪-> Result {...}
```

5.3.2 Preconditions

- The failure handling state must not be set to `SHUTDOWN: 3`.
- The submitted block header must either create a new fork or extend an existing fork (in `Forks`) as tracked by BTC-Relay.
- If the submission extends an existing fork, the `forkId` must be set to the correct identifier as tracked in `Forks`.
- If the submission creates a new fork, the `forkId` must be set to `0`.

5.3.3 Function Sequence

The `storeForkBlockHeader` function takes as input the 80 byte raw Bitcoin block header and a `forkId` and follows the following sequence:

1. Check if the failure handling state is set to `SHUTDOWN`. If true, raise `ERR_SHUTDOWN` and return.
2. Call *verifyBlockHeader* passing `blockHeaderBytes` as parameter. If this call **does not return** `True` (i.e., fails or returns `False`), then abort and return `False`.
3. Check if `forkId == 0`.

a. If `forkId == 0`, generate a new `forkId` and create a new entry in `Forks`, setting the `height` of the block header as the `startHeight` of the fork.

b. Otherwise:

b.1) Check if a fork is tracked in `Forks` under the specified `forkId`. If no fork can be found, raise an `ERR_INVALID_FORK_ID` exception and abort.

b.2) Check that the `hashPrevBlock` of the submitted block header indeed references the last block submitted to the fork, specified by `forkId`. Raise `ERR_FORK_PREV_BLOCK` exception and abort if this check fails.

4. Extract the `merkleRoot` (*extractMerkleRoot*), `timestamp` (*extractTimestamp*) and `target` (*extractNBits* amd *nBitsToTarget*) from `blockHeaderBytes`, and compute the block hash using *sha256d* (passing `blockHeaderBytes` as parameter).
5. Store the `height`, `merkleRoot`, `timestamp` and `target` as a new entry in the `blockHeaders` map, using `hashCurrentBlock` as key (compute using *sha256d*).
6. Update `Fork[forkId]` entry, incrementing the fork `length` and inserting `hashCurrentBlock` into the list of block hashes contained in that fork (`forkBlockHashes`).
7. Emit a `StoreForkBlockHeader` event using `height` and `hashCurrentBlock` as input (`StoreMainChainHeader(height, hashCurrentBlock)`).
8. Check if the fork at `forkId` has become longer than the current `MainChain`. This is the case if the block height `height` of the submitted block header exceeds the `BestBlockHeight`.
   a. If `height > BestBlockHeight` call `chainReorg(forkId)` and return the value returned form this call.
9. Return `True`.

5.4 verifyBlockHeader

The `verifyBlockHeader` function parses and verifies Bitcoin block headers.

Warning: This function must called and return `True` **before** a Bitcoin block header is stored in the BTC-Relay (i.e., must be called by the *storeMainChainBlockHeader* and *storeForkBlockHeader* functions).

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

Other operations, such as verification of transaction inclusion, can only be executed once a block header has been verified and consequently stored in the BTC-Relay.

5.4.1 Specification

Function Signature

`verifyBlockHeader(blockHeaderBytes)`

Parameters

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

Returns

- `True`: if the block header passes all checks.
- `False` (or throws exception): otherwise.

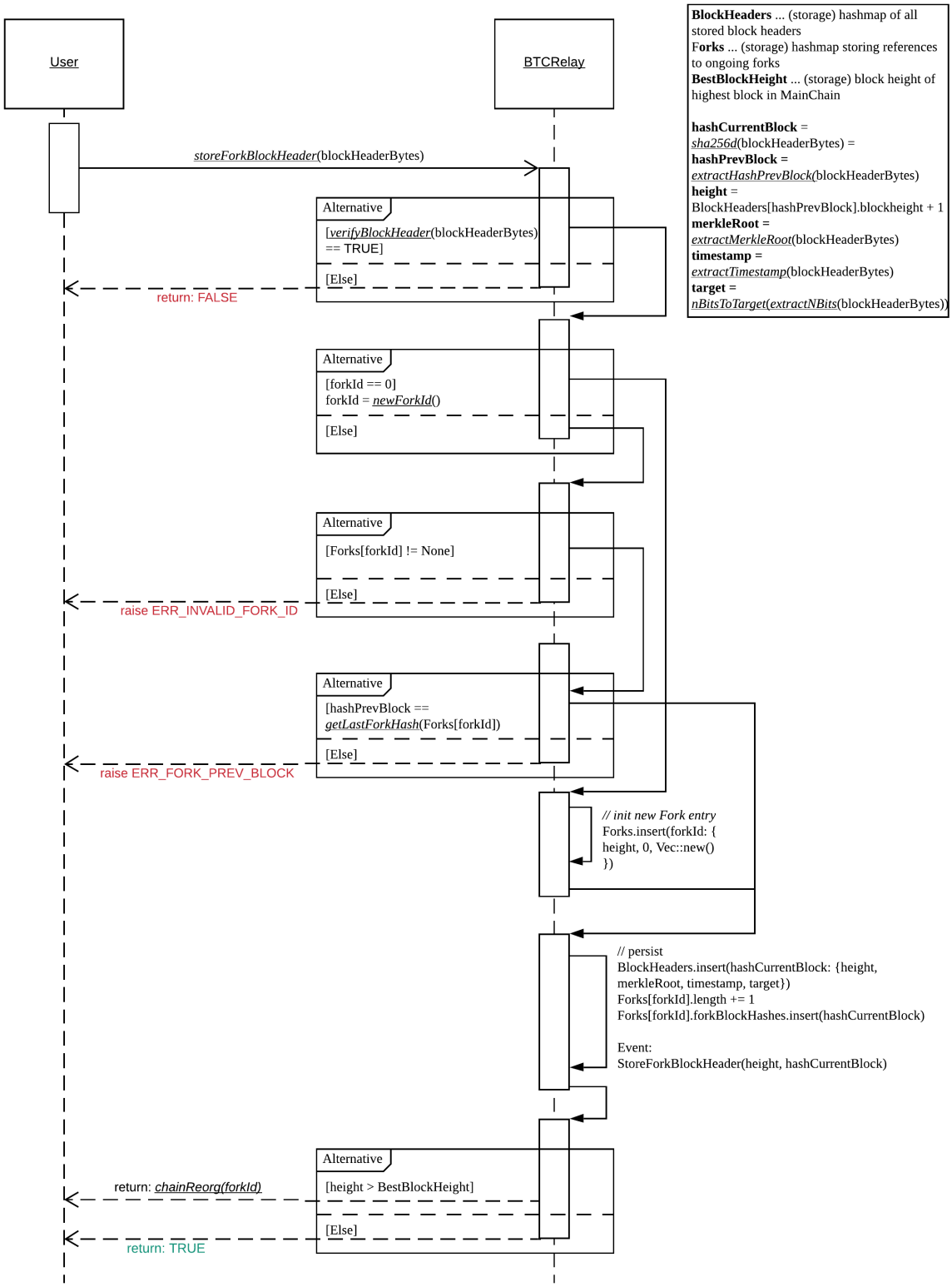

Fig. 2: Sequence diagram showing the function sequence of *storeForkBlockHeader*.

Errors

- `ERR_INVALID_HEADER_SIZE = "Invalid block header size"`: raise exception if the submitted block header is not exactly 80 bytes long.
- `ERR_DUPLICATE_BLOCK = "Block already stored"`: raise exception if the submitted block header is already stored in BTC-Relay (duplicate PoW `blockHash`).
- `ERR_PREV_BLOCK = "Previous block hash not found"`: raise exception if the submitted block does not reference an already stored block header as predecessor (via `prevBlockHash`).
- `ERR_LOW_DIFF = "PoW hash does not meet difficulty target of header"`: raise exception 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"`: raise exception if the `target` specified in the block header is incorrect for its block height (difficulty re-target not executed).

Substrate

```
fn verifyBlockHeader(origin, blockHeaderBytes: T::BTCBlockHeader) -> Result {...}
```

5.4.2 Function Sequence

The `verifyBlockHeader` function takes as input the 80 byte raw Bitcoin block header and follows the sequence below:

1. Check that the `blockHeaderBytes` is 80 bytes long. Raise `ERR_INVALID_HEADER_SIZE` exception and abort otherwise.
2. Check that the block header is not yet stored in BTC-Relay (`blockHash` is unique in `blockHeaders`). Raise `ERR_DUPLICATE_BLOCK` exception and abort otherwise.
3. Check that the previous block referenced by the submitted block header (`hashPrevBlock`, extract using *extractHashPrevBlock*) exists in `BlockHeaders`. Raise `ERR_PREV_BLOCK` exception and abort otherwise.
4. Check that the Proof-of-Work hash (`blockHash`) is below the `target` specified in the block header. Raise `ERR_LOW_DIFF` exception and abort otherwise.
5. Check that the `target` specified in the block header (extract using *extractNBits* and *nBitsToTarget*) is correct by calling *checkCorrectTarget* passing `hashPrevBlock`, `height` and `target` as parameters (as per Bitcoin's difficulty adjustment mechanism, see here). If this call returns `False`, raise `ERR_DIFF_TARGET_HEADER` exception and abort.
6. Return `True`

5.5 verifyTransaction

The `verifyTransaction` 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 `MainChain`), 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 `MainChain`).

5.5.1 Specification

Function Signature

`verifyTransaction(txId, txBlockHeight, txIndex, merkleProof, confirmations)`

Parameters

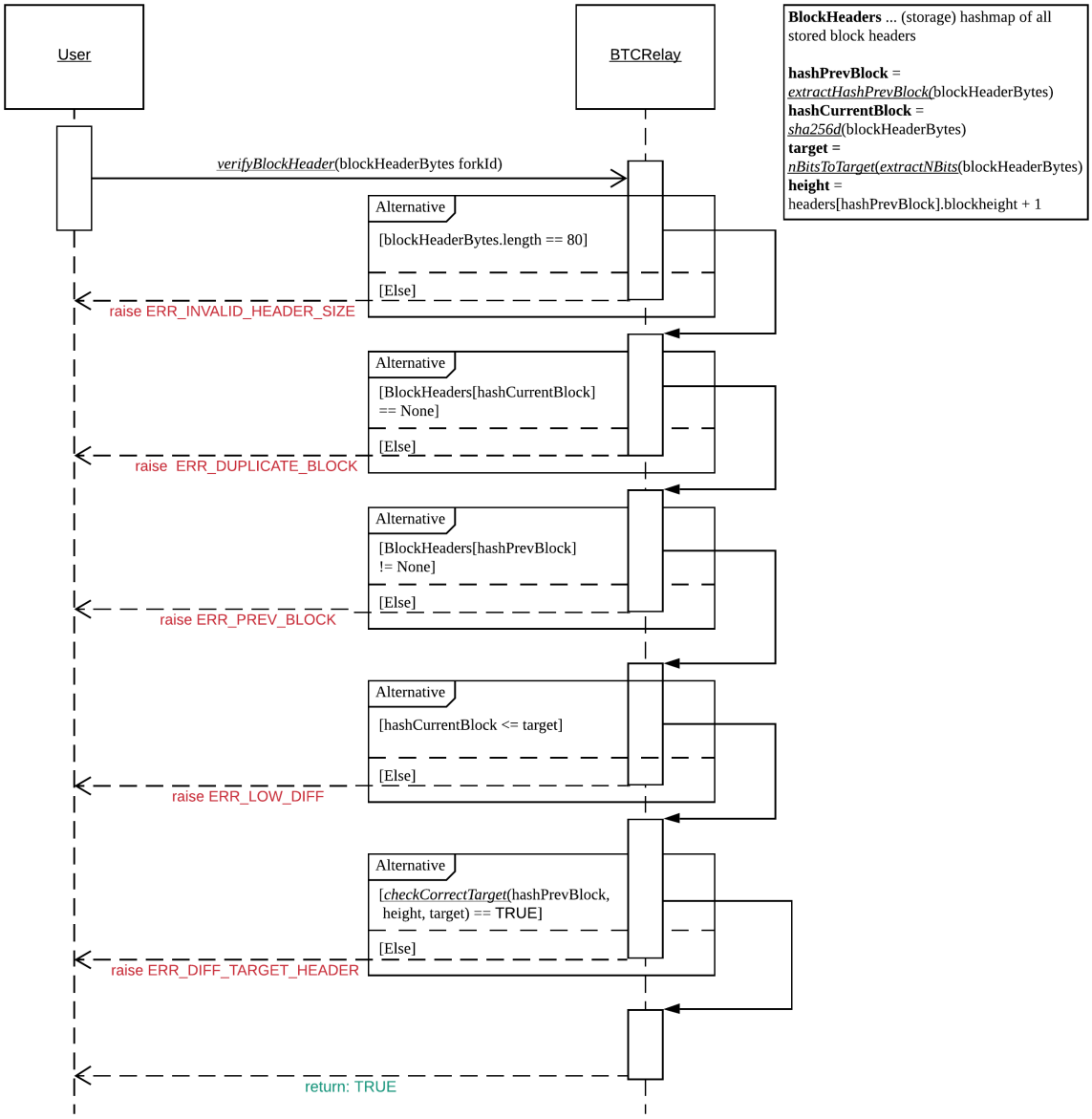

Fig. 3: Sequence diagram showing the function sequence of *verifyBlockHeader*.

- `txId`: 32 byte hash identifier of the transaction.
- `txBlockHeight`: integer block height at which transaction is supposedly included.
- `txIndex`: integer index of transaction in the block's tx Merkle tree.
- `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.

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.
- `False` (or throws exception): 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_PARTIAL = "BTC Parachain partially deactivated"`: the BTC Parachain has been partially deactivated since a specific block height.
- `ERR_HALTED = "BTC Parachain is halted"`: the BTC Parachain has been halted.
- `ERR_SHUTDOWN = "BTC Parachain has shut down"`: the BTC Parachain has been shutdown by a manual intervention of the governance mechanism.
- `ERR_INVALID_TXID = "Invalid transaction identifier"`: raise exception if the transaction identifier (`txId`) is malformed.
- `ERR_CONFIRMATIONS = "Transaction has less confirmations than requested"`: raise exception if the block in which the transaction specified by `txId` was included has less confirmations than requested.
- `ERR_MERKLE_PROOF = "Invalid Merkle Proof structure"`: raise exception if the Merkle proof is malformed.

Substrate

```
fn verifyTransaction(origin, txId: T::Hash, txBlockHeight: U256, txIndex: u64,␣
↪merkleProof: String, confirmations: U256) -> Result {...}
```

5.5.2 Preconditions

- If the failure handling status is set to `PARTIAL: 1`, transaction verification is disabled for the latest blocks.
- The failure handling status must not be set to `HALTED: 2`. If `HALTED`, all transaction verification is disabled.
- The failure handling status must not be set to `SHUTDOWN: 3`. If `SHUTDOWN`, all transaction verification is disabled.

5.5.3 Function Sequence

The `verifyTransaction` function follows the function sequence below:

1. Check if the failure handling state is set to `HALTED` or `SHUTDOWN`. If true, raise `ERR_HALTED` or `ERR_SHUTDOWN` and return.
2. Check if the failure handling state is set to `PARTIAL`. If true, check if the `txBlockHeight` is equal to or greater than the first `NO_DATA` block. If false, raise `ERR_PARTIAL` and return.
3. Check that `txId` is 32 bytes long. Raise `ERR_INVALID_FORK_ID` error if this check fails.
4. Check that the current `BestBlockHeight` exceeds `txBlockHeight` by the specified number of `confirmations`. Raise `ERR_CONFIRMATIONS` if this check fails.
5. Extract the block header from `BlockHeaders` using the `blockHash` tracked in `MainChain` at the passed `txBlockHeight`.
6. 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. Raise `ERR_MERKLE_PROOF` error if one of these checks fails.
7. 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 `False`.

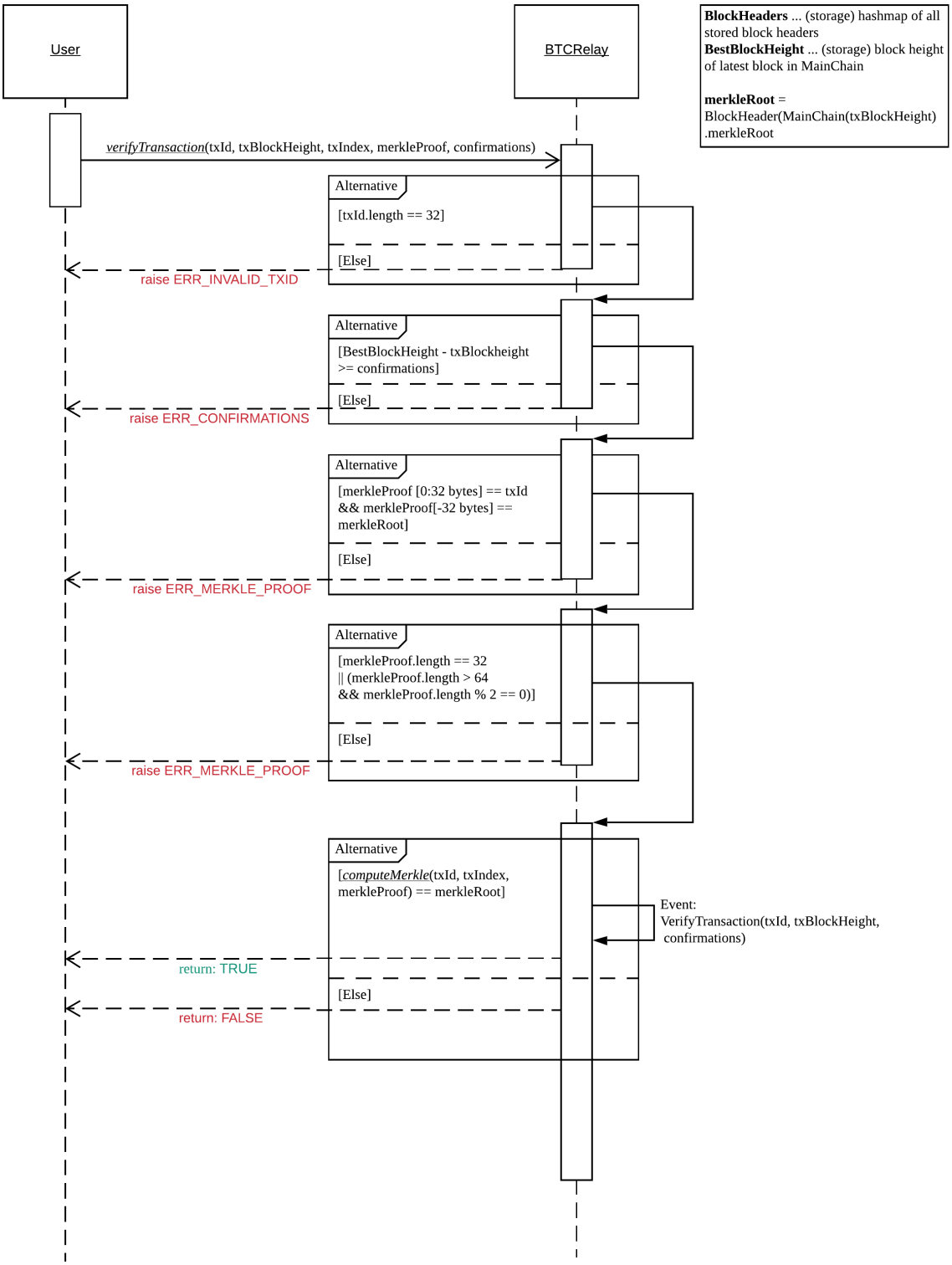

Fig. 4: The steps to verify a transaction in the *verifyTransaction* function.

CHAPTER

SIX

FUNCTIONS: UTILS

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

6.1 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`.

Substrate

```
fn sha256d(data: String) -> T::Hash {...}
```

6.1.1 Function Sequence

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

6.2 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`.

Substrate

```
fn concatSha256d(left: T::Hash, right: T::Hash) -> T::Hash {...}
```

6.2.1 Function Sequence

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

6.3 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.

Substrate

```
fn nBitsToTarget(nBits: u32) -> U256 {...}
```

6.3.1 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`.

6.4 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.

Substrate

```
fn checkCorrectTarget(hashPrevBlock: T::Hash, blockHeight: U256, target: U256) ->␣
↪bool {...}
```

6.4.1 Function Sequence

1. Retrieve the previous block header with the `hashPrevBlock` from the `BlockHeaders` storage and extract the `target` difficulty of the previous block.
2. Check if the `target` 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 target of the previous block and check that the target of the previous block is not `0`.
        i. If the target difficulties match, return `True`.
        ii. Otherwise, return `False`.
    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 `MainChain` 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 new target difficulty matches `target`, return `True`.
        ii. Otherwise, return `False`.

6.5 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.

Substrate

```
fn computeNewTarget(prevTime: T::DateTime, startTime: T::DateTime, prevTarget:␣
↪U256) -> U256 {...}
```

6.5.1 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`.

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

Substrate

```
fn computeMerkle(txId: T::Hash, txIndex: u64, merkleProof: String) -> Hash {...}
```

6.6.1 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_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`.

6.6.2 Example

Assume we have the following input:

- txId: `330dbbc15169c538583073fd0a7708d8de2d3dc155d75b361cbf5c24b73f3586`
- txIndex: `0`
- merkleProof: `86353fb7245cbf1c365bd755c13d2dded808770afd73305838c56951c1bb0d33b635f586cf`

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.

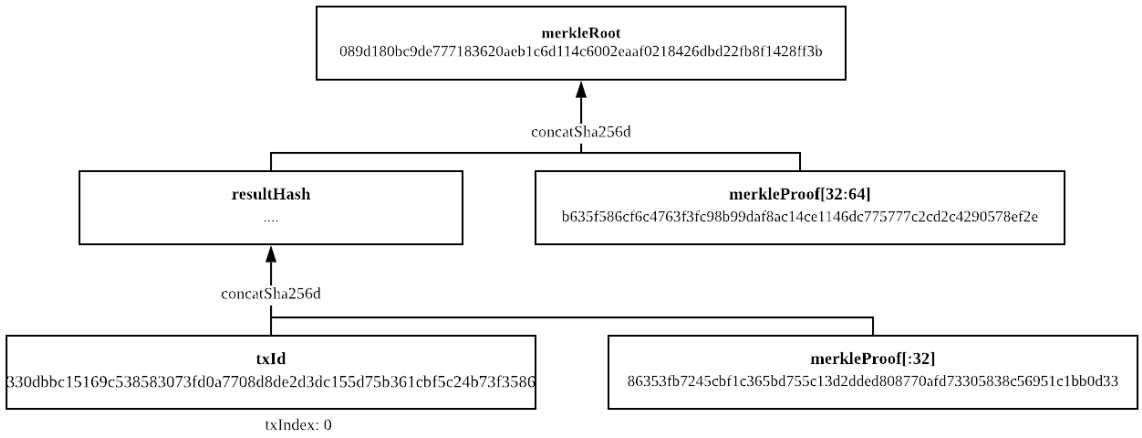

Fig. 1: An example of the `computeMerkle` function with a transaction from a block that contains two transactions in total.

6.7 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`.

Substrate

```
fn calculateDifficulty(target: U256) -> U256 {...}
```

6.7.1 Function Sequence

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

6.8 chainReorg

The `chainReorg` function is called from `storeForkBlockHeader` and handles blockchain reorganizations in BTC-Relay, i.e., when a fork overtakes the tracked main chain in terms of length (and accumulated PoW). As a result, the `MainChain` references the stored block headers (in `BlockHeaders`) are updated to point to the blocks contained in the overtaking fork.

6.8.1 Specification

Function Signature

`chainReorg(forkId)`

Parameters

- `forkId`: identifier of the fork as stored in `Forks`, which is to replace the `MainChain`.

Returns

- `True`: if the `MainChain` is updated to point to the block headers contained in the fork specified by `forkId`.
- `False` (or throws exception): otherwise.

Substrate

```
fn chainReorg(forkId: U256) -> bool {...}
```

6.8.2 Function Sequence

1. Retrieve fork data (`Fork`, see *Data Model*) via `Fork[forkId]`
2. Create new entry in `Forks`, (generate a new identifier `newForkId`), setting `Forks[newForkId].startHeight = Forks[forkId].startHeight` and `Forks[newForkId].length = Forks[forkId].length - 1`.
3. Replace the current `MainChain` references to `BlockHeaders` (i.e., the `blockHash` at each `blockHeight`) with the corresponding entry in `forkHashes` of the given fork. In this process, store the replaced `MainChain` entries to a new fork. In detail: starting at `Fork[forkId].startHeight`, loop over `Fork[forkId].forkHashes` (`forkHash`) and for each `forkHash` (loop counter `counter = 0` incremented each round):
   a. Copy the `blockHash` referenced in `mainChain` at the corresponding block height (`startHeight + counter`) to `Forks[newForkId].forkHashes`.
   b. Overwrite the `blockHash` in `MainChain` at the corresponding block height (`startHeight + counter`) with the given `forkHash`.
4. Update `BestBlock` and `BestBlockHeight` to point to updated highest block in `MainChain`.
5. Delete `Fork[forkId]`.

Note: The last block hash in `forkHashes` will be added to `MainChain` with a block height exceeding the current `BestBlockHeight`, since the fork that caused the reorganization is by definition 1 block longer than the `MainChain` tracked in BTC-Relay.

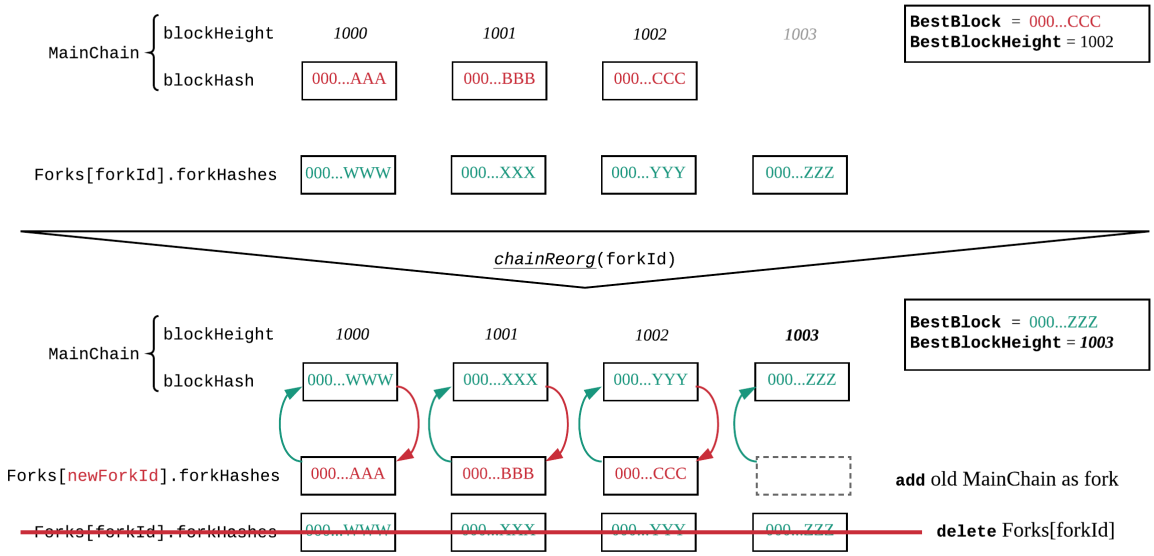

Fig. 2: Overview of a the BTC-Relay state before (above) and after (below) `chainReorg(forkId)`.

Warning: **Do not instantly delete** the block headers that were removed from the `MainChain` through the reorganization. If deletion is required, wait at least until sufficient confirmations have passed, as defined by the security parameter *k* (see *Security Analysis*).

6.9 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.

6.9.1 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`.

Substrate

```
fn getForkIdByBlockHash(blockHash: T::Hash) -> U256 {...}
```

6.9.2 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.

CHAPTER

SEVEN

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.

7.1 Block Header

7.1.1 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.

Substrate

```
fn extractHashPrevBlock(blockHeaderBytes: T::BTCBlockHeader) -> T::Hash {...}
```

Function Sequence

1. Return `blockHeaderBytes[4:32]` (`hashPrevBlock` starts an position 4 of the 80 byte block header).

7.1.2 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

Substrate

```
fn extractMerkleRoot(blockHeaderBytes: T::BTCBlockHeader) -> T::Hash {...}
```

Function Sequence

1. Return `blockHeaderBytes[36:32]` (`merkleRoot` starts at position 36 of the 80 byte block header).

7.1.3 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

Substrate

```
fn extractNBits(blockHeaderBytes: T::BTCBlockHeader) -> T::Bytes {...}
```

Function Sequence

1. Return `blockHeaderBytes[72:4]` (`nBits` starts at position 72 of the 80 byte block header).

7.1.4 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

Substrate

```
fn extractTimestamp(blockHeaderBytes: T::BTCBlockHeader) -> T::DateTime {...}
```

Function Sequence

1. Return `blockHeaderBytes[68:4]` (`timestamp` starts at position 68 of the 80 byte block header).

7.2 Transactions

Todo: The parser functions used for transaction processing (called by other modules) will be added on demand. See PolkaBTC specification for more details.

CHAPTER

EIGHT

EVENTS

8.1 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*

Substrate

```
Initialized(U256, Hash);
```

8.2 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

- *storeMainChainBlockHeader*

Substrate

```
StoreMainChainHeader(U256, Hash);
```

8.3 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

- *storeForkBlockHeader*

Substrate

```
StoreForkHeader(U256, U256, Hash);
```

8.4 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 start block height of the fork and the fork identifier.

Event Signature

`ChainReorg(newChainTip, startHeight, forkId)`

Parameters

- `newChainTip`: hash of the new highest block.
- `startHeight`: height of the new highest block.
- `forkId`: a unique id for the fork.

Functions

- *storeForkBlockHeader*

Substrate

```
ChainReorg(Hash, U256, U256);
```

8.5 VerifyTransaction

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

- *verifyTransaction*

Substrate

```
VerifyTransaction(Hash, U256, U256);
```

CHAPTER

NINE

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:** *storeMainChainBlockHeader*
- **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:** *storeForkBlockHeader*
- **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**: *storeForkBlockHeader*
- **Cause**: Raised if raise exception if the submitted block header is actually extending the current longest chain tracked by BTC-Relay (`MainChain`), instead of a fork.

`ERR_INVALID_FORK_ID`

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

`ERR_INVALID_HEADER_SIZE`

- **Message**: "Invalid block header size":
- **Function**: *verifyBlockHeader*
- **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_INVALID_TXID`

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

`ERR_CONFIRMATIONS`

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

`ERR_MERKLE_PROOF`

- **Message**: "Invalid Merkle Proof structure"
- **Function**: *verifyTransaction*
- **Cause**: Exception raised in `verifyTransaction` 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_PARTIAL`

- **Message**: "BTC Parachain partially deactivated"
- **Function**: *verifyTransaction*
- **Cause**: The BTC Parachain has been partially deactivated since a specific block height.

`ERR_HALTED`

- **Message**: "BTC Parachain is halted"
- **Function**: *verifyTransaction*
- **Cause**: The BTC Parachain has been halted.

`ERR_SHUTDOWN`

- **Message**: "BTC Parachain has shut down"
- **Function**: *verifyTransaction* | *storeForkBlockHeader* | *storeMainChainBlockHeader*
- **Cause**: The BTC Parachain has been shutdown by a manual intervention of the governance mechanism.

CHAPTER

TEN

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.

10.1 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.

10.2 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 PolkaBTC component.

10.3 Safety Failures

10.3.1 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 `MainChain` 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.

Todo: Add reference to Failure Handling spec, once deployed.

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.

10.3.2 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 `MainChain` 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.

10.3.3 Replay Attacks

Since BTC-Relay does not store Bitcoin transactions, nor can it be aware of all possible applications using `verifyTransaction`, 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.

10.4 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

ELEVEN

PERFORMANCE AND COSTS

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.

11.1 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 `MainChain` 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.

11.2 BTC-Relay Optimizations

11.2.1 Pruning

Optionally, to further reduce storage requirements (e.g., in case more data is to be stored per block in the future), *pruning* of `MainChain` 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, `MainChain` would be implemented as a FIFO queue, where sufficiently old block headers are removed from `BlockHeaders` (and the references from `MainChain` 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 `MainChain` as performance optimization, it is important to make sure there are no `Forks` entries left which reference pruned blocks.

11.2.2 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.

11.2.3 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

TWELVE

LICENSE