plasma-core Documentation

Release latest

Mar 08, 2019

Developer Documentation

Hello and welcome to the documentation of Plasma Group's `plasma-core`! A quick note: `plasma-core` is **not** our `Node.js` client, `plasma-client`. Documentation for `plasma-client` is available at the plasma-client docs.

CHAPTER 1

What is plasma-core?

`plasma-core` makes up the core of the Plasma Group node ecosystem. It contains almost all of the functionality that a full plasma node needs. `plasma-core` handles things like watching Ethereum, keeping the local state up to date, and talking to the operator. A full list of services that `plasma-core` provides is documented in our **architecture_** page.

`plasma-core` is **not** a full plasma node! This means that you'll need to **'extend plasma-core'_** to expose the full set of necessary functionality. However, we've already done this for you! We've built out two different full plasma nodes using `plasma-core` as a backend. Our most user-friendly node is `plasma-extension`, a full plasma node inside a Chrome extension!

Note: `plasma-extension` is still under construction.

CHAPTER 2

Contributing

Welcome! A huge thank you for your interest in contributing to Plasma Group. Plasma Group is an open source initiative developing a simple and well designed plasma implementation. If you're looking to contribute to `plasma-core`, you're in the right place! It's contributors like you that make open source projects work, we really couldn't do it without you.

We don't just need people who can contribute code. We need people who can run this code for themselves and break it. We need people who can report bugs, request new features, and leave helpful comments. **We need you!**

We're always available to answer your questions and to help you become a contributor! You can reach out to any of the members of Plasma Group on GitHub, or send us an email at contributing@plasma.group.

Here at Plasma Group we're trying to foster an inclusive, welcoming, and accessible open source ecosystem. The best open source projects are those that make contributing an easy and rewarding experience. We're trying to follow those best practices by maintaining a series of resources for contributors to Plasma Group repositories.

If you're a new contributor to `plasma-core`, please read through the following information. These resources will help you get started and will help you better understand what we're building.

2.1 Contributing Guide and Code of Conduct

Plasma Group follows a Contributing Guide and Code of Conduct adapted slightly from the Contributor Covenant. **All contributors are expected to read through this guide.** We're here to cultivate a welcoming and inclusive contributing environment. Every new contributor needs to do their part to uphold our community standards.

2.2 Getting Started as a Contributor

2.2.1 Design and Architecture

Before you start contributing, please read through our Architecture document. This will give you a high-level understanding of what `plasma-core` is and what `plasma-core` isn't.

2.2.2 Requirements and Setup

Node.js

`plasma-core` is a Node.js application. You'll need to install `Node.js` (and it's corresponding package manager, `npm`) for your system before continuing.

`plasma-core` has been tested on the following versions of Node:

- 10.14.2

If you're having trouble getting a component of `plasma-core` running, please try running one of the above versions.

Packages

`plasma-core` makes use of several `npm` packages.

Install all required packages with:

2.2.3 Running Tests

`plasma-core` makes use of a combination of Mocha (a testing framework) and Chai (an assertion library) for testing.

Run all tests with:

Contributors: remember to run tests before submitting a pull request! Code with passing tests makes life easier for everyone and means your contribution can get pulled into this project faster.

CHAPTER 3

Architecture

This document describes, in detail, the architecture of `plasma-core`. If you're a new contributor to `plasma-core`, welcome! Hopefully this document will help you better understand how `plasma-core` works under the hood.

3.1 plasma-core 101

`plasma-core` is composed of a set of *services* that, when woven together, form (almost) a complete node! Each of these services performs a very specific role. A more in-depth explanation of each individual service is available in the *Service API Reference* section of our documentation.

3.2 Architecture Diagram

This diagram shows the basic architecture of `plasma-core`:

3.3 External services

`plasma-core` talks to three external services: **Ethereum**, the plasma chain **Operator**, and user **applications**. These three services are *outside* of the scope of `plasma-core`. Instead, `plasma-core` provides interfaces through which it can talk to and hear from these external services.

3.3.1 ContractService

ContractService handles interactions with the plasma chain contract, like submitting deposits or starting withdrawals. Non-contract specific Ethereum interactions are handled by ETHService.

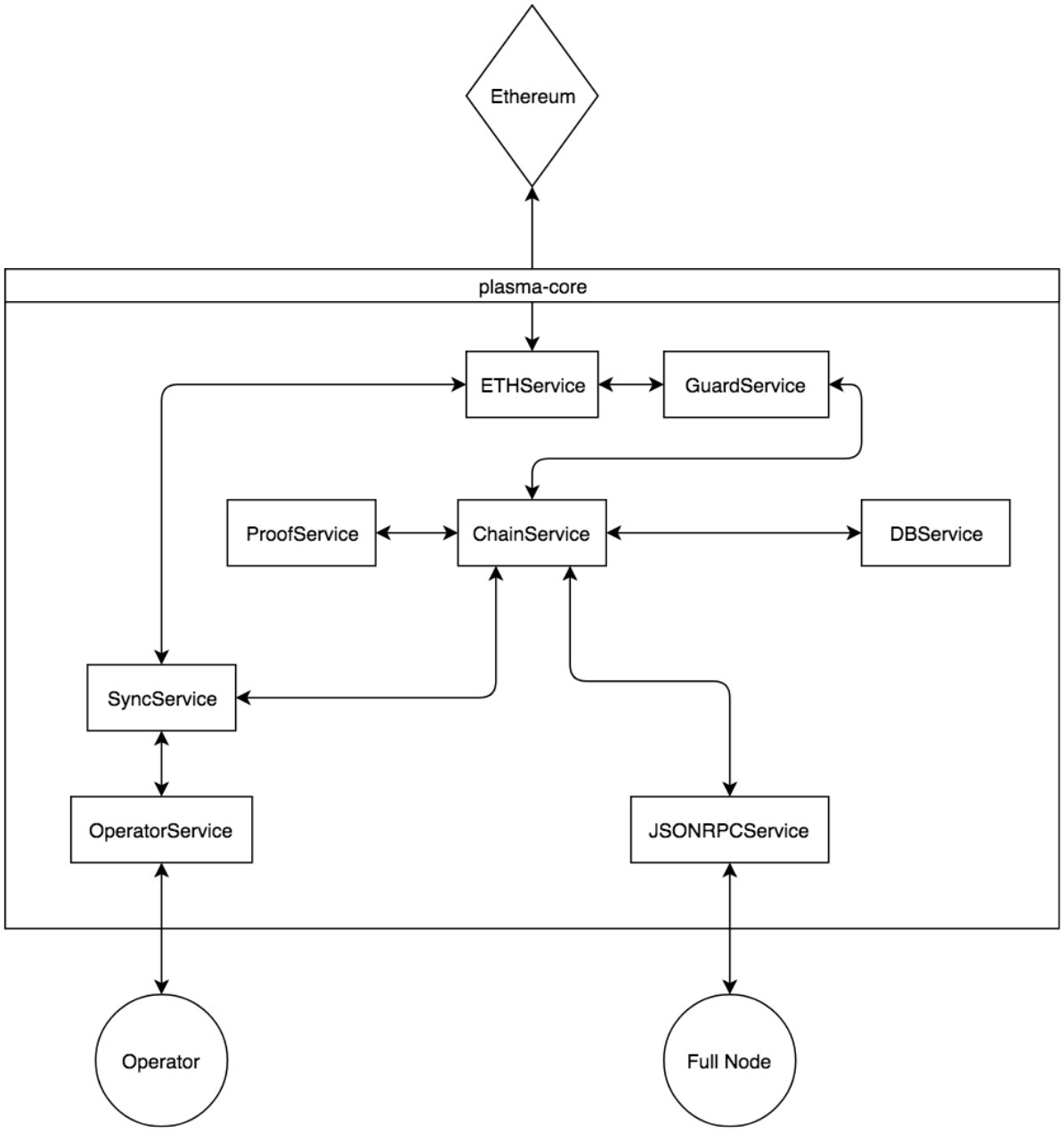


Fig. 1: PG Plasma Architecture Diagram

3.3.2 ETHService

ETHService exposes the functionality necessary to read information from Ethereum. `ETHService` is used for non-contract specific interactions.

3.3.3 OperatorService

As its name suggests, OperatorService handles all communication with the plasma chain operator. This includes sending and receiving plasma chain transactions.

3.3.4 JSONRPCService

The JSONRPCService acts as a handler for commands sent by user **applications**. By default, applications must interact *directly* with `JSONRPCService`. `plasma-core` can be extended to expose additional interfaces to `JSONRPCService`, such as an HTTP API.

3.4 Internal services

The remaining services of `plasma-core` manage things internally.

3.4.1 SyncService

Possibly the most important internal service, SyncService ensures that your node always has the latest transactions. `SyncService` watches Ethereum for any new plasma chain blocks and automatically pulls any necessary information from the Operator. `SyncService` makes sure your balances are always up-to-date and that you can always send transactions when you need to!

3.4.2 ChainService

ChainService is another extremely important internal service. `ChainService` manages `plasma-core`'s internal blockchain. This includes storing any necessary transaction and block information. `ChainService` also handles returning information about the stored local state with convenient wrapper functions.

3.4.3 GuardService

GuardService takes on the important role of keeping your funds safe at all times. The `GuardService` keeps a constant eye on Ethereum and blocks others from trying to move funds without your permission. `GuardService` queries Ethereum through the `ETHService` and pulls other relevant user data from `ChainService`.

3.4.4 DBService

DBService simply provides a database that `ChainService` uses to store user data. Currently, we support two database backends, LevelDB and an in-memory ephemeral database (`EphemDB`). Most services talk to `ChainService` to retrieve data from `DBService` instead of talking to `DBService` directly.

CHAPTER 4

Extending plasma-core

Note: This article is for developers who want to build their own full plasma nodes using `plasma-core`.

As explained in **'What is plasma-core?'_** `plasma-core` is just the *core* of our plasma node ecosystem. It's not really a *full* plasma client. In order to build out a full client that exposes all of the functionality a user will expect, you'll need to *extend* `plasma-core`.

To this end, `plasma-core` is designed to be maximally extensible. It tries to make as few decisions on your behalf as possible. This means you can make a lot of different clients out of `plasma-core`! So far we've tested out creating a full *Node.js* node and a full node inside of a Chrome extension!

4.1 What's missing

`plasma-core` is missing a few key features that a user might expect. Here's a list of what's missing so that you know what to expect if you want to build a full plasma node.

4.1.1 Front-facing Services

`plasma-core` doesn't come with any sort of front-facing services (i.e. an HTTP server that handles JSON-RPC requests). If you want users to be able to interact with your node software, you'll need to implement a service like this. We left this to node developers because different types of nodes might handle this in completely different ways. For example, the node that we're building inside of a Chrome extension talks to apps via Chrome's native message passing interface instead of over HTTP. Front-facing services need to wrap and pipe calls into JSONRPCService.

4.1.2 User interface

Because `plasma-core` doesn't provide any front-facing services, it also doesn't provide any sort of user interface. As a node developer, you'll probably want to create some sort of simple interface that allows users to interact with the node. This might take the form of a CLI that sends requests to an HTTP server or a local website that connects to the node.

4.1.3 Wallet Management

Private key storage and transaction signing is not handled by `plasma-core`. `plasma-core` only provides a mock wallet for testing that should **not** be used in production. You will therefore have to implement your own WalletService.

However, key management is hard and you probably shouldn't be building your own wallets. We therefore recommend deferring this functionality to a user's Ethereum node. This can be as easy as forwarding the necessary API calls (as described on the WalletService documentation page) to the Ethereum node. `plasma-extension` uses this method by forwarding all wallet-related activity to MetaMask.

CHAPTER 5

Reference

This page provides a series of miscellaneous reference articles that can be helpful when installing Plasma Group components.

5.1 Running a Terminal

Before you keep going, it's probably good to become familiar with using the terminal on your computer. Here are some resources for getting started:

- Windows: Command Prompt: What It Is and How to Use It
- MacOS: Introduction to the Mac OS X Command Line
- Linux: How to Start Using the Linux Terminal

5.2 Installing Git

`git` is an open source version control system. You don't really need to know how it works, but you *will* need it in order to install most Plasma Group components.

5.2.1 Windows

Atlassian has a good tutorial on installing `git` on Windows. It's basically just installing an `.exe` and running a setup wizard.

5.2.2 MacOS

Installing `git` on a Mac is pretty easy. You basically just need to type `git` into your terminal. If you have `git` installed, you'll see a bunch of output. Otherwise, you'll get a pop-up asking you to install some command-line tools (including `git`).

5.2.3 Linux

Installing `git` on Linux is also pretty easy. However, the exact install process depends on your distribution. Here's a guide for installing `git` on some popular distributions.

5.3 Installing Node.js

Most of the Plasma Group apps are built in JavaScript and make use of a tool called Node.js. In order to run our tools, you'll need to make sure that you've got `Node.js` installed.

Here's a list of ways to install `Node.js` on different operating systems:

5.3.1 Windows

If you're on a windows computer, you can download the latest Long-term Support (LTS) version of `Node.js` here. You'll just need to install the `.msi` file that `Node.js` provides and restart your computer.

5.3.2 MacOS

You have some options if you want to install `Node.js` on a Mac. The simplest way is to download the `.pkg` file from the `Node.js` downloads page. Once you've installed the `.pkg` file, run this command on your terminal to make sure everything is working properly:

```
node -v
```

If everything is working, you should see a version number pop up that looks something like this:

```
v10.15.1
```

Homebrew

Note: If you've already installed `Node.js` with the above steps, you can skip this section!

You can also install `Node.js` using Homebrew. First, make sure Homebrew is up to date:

```
brew update
```

Now just install `Node.js`:

```
brew install node
```

5.3.3 Linux

There are different ways to install `Node.js` depending on your Linux distribution. Here's an article that goes through installing `Node.js` on different distributions.

CHAPTER 6

Plasma Chain Operator

When learning about plasma you'll eventually run into the idea of the plasma chain 'operator'. For the most part, the operator is exactly what it sounds like — a single entity that's responsible for aggregating transactions into blocks and then publishing those blocks to some other "main" blockchain (like Ethereum). Blockchains are only usable if new blocks are being created, so, by producing these new blocks, the operator is quite literally responsible for keeping the whole plasma chain running.

This might be a little confusing at first. Initial reactions are usually something along the lines of, "What? Plasma chains have operators? Aren't blockchains supposed to be decentralized?" Well, it turns out that the answer is, as you might've guessed, "Kinda."

6.1 Back to Basics

Let's go back to basics and talk a little bit about why most blockchains use decentralized block production mechanisms in the first place. Blockchains are big logs of things that have happened, broken into concrete and ordered time-steps we call blocks. Sometimes these "things that have happened" are simple - "A sent X amount of money to B" - sometimes they're more complex - "Kitty A mated with Kitty B and created Kitty C". No matter what these events are, we usually want to make sure that we have a few key properties:

1. No one should be able to re-write history.
2. No one should be blocked from making transactions.

Let's imagine we have a blockchain run by a single person. For the sake of argument, assume that you *must* use the blockchain run by that person for some reason. Well, unfortunately it's quite easy for that person to break the first property. If that person says "transaction X happened" and then later says "transaction X never happened", there's not much you can really do. *You* know that the transaction happened, but the blockchain itself doesn't.

It's also really easy for that person to break the second property. If they don't want you to send transactions, they can just refuse to add any transactions that come from you. Had a bunch of money on that blockchain? Too bad, you're not getting it back.

6.2 Plasma Magic

The problems with blockchains run by a single person are why blockchains usually have fancy mechanisms that ensure that it's extremely expensive to rewrite history. It's also why blockchains usually have lots of different people who can create blocks – no single person can stop someone from making transactions. So why can we have a single person running plasma chain? It's because we cheat (sort of).

In plasma world, we get the first property by taking advantage of the "main blockchain" that we were talking about earlier. Plasma chain operators need to publish a block "commitment" (sort of like a very compressed version of the block) to the main blockchain for every block they produce. A smart contract on the main blockchain ensures that the operator can never publish the same block twice. As long as the main blockchain has the first property, so does the plasma chain! There's no way for the operator to re-write history unless they can re-write history on the main chain.

The second property (blocking users from transacting) is where things get interesting. Unfortunately, it's still possible for the operator to censor transactions from anyone they want. *However*, this is where the magic of plasma comes in. Plasma chains are designed in a way that **no matter what, a user can always withdraw their money** from the plasma chain back to the main chain. Even if the operator is actively trying to steal money from you, you'll still be able to get it back. Being censored obviously isn't great, but it's not as bad when you can always take your money somewhere else.

6.3 Decentralizing the Operator

The one thing that wasn't really mentioned here is the fact that the "operator" can actually consist of multiple people making decisions about what blocks to publish. This could be as simple as a having a few designed people who take turns making blocks, or as complex as a Proof-of-Stake system that selects block producers randomly. Either way, it's complicated than just having a single person run everything, but it's probably the way to go for projects that want to get rid of censorship. At the same time, it tends to be unimportant research-wise whether the operator is a single person or many people. As a result, you'll often see people just assuming the operator is a single person for simplicity.

CHAPTER 7

Coin ID Assignment

This explains the process by which assets deposited into our plasma chain are assigned unique identifiers.

You usually won't have to think about these things when you're interacting with a PG plasma chains. Most of this is handled automagically by `plasma-utils` or `plasma-js-lib`. The exact byte-per-byte binary representations of all data structures for each structure can be found in our schemas.

7.1 Coin IDs

The base unit of any asset on our chain is the "coin". Just like coins in Plasma Cash, these coins are non-fungible. Each coin is given a unique 16 byte identifier, `Coin ID`. Coin IDs are assigned to assets in deposit-order on a per-asset (ERC 20/ETH) basis.

Note that all assets in the chain share the same ID-space, even if they're different ERC20s or ETH. This means that transactions across all asset classes (which we refer to as the `tokenType` or `token`) share the same tree. This decision was made primarily as an optimization to make the transaction tree as small as possible. However, it's important because it introduces some added complexity to coin IDs.

7.1.1 Token Type

The first 4 bytes of a coin's ID refer to the `tokenType` of a coin. `tokenType` is assigned to a given token depending on when that token was listed in the smart contract. We give ETH a `tokenType` of `00000000`.

7.1.2 Token ID

The next 12 bytes of the ID represents the actual ID of that specific token. Note that two different coins can have the same "Token ID" as long as they have a different `tokenType`.

7.2 Denominations

The base denomination of each asset is automatically drawn from the asset's smart contract. If it's an ERC20, we look at the `decimals` variable. If no `decimals` is available, we set the denomination to "1". The base denomination of ETH is `wei`.

7.3 Examples

7.3.1 ETH Deposit

Let's say a user deposits `2 wei` into the plasma chain contract. That user will be given spending rights for two coins, `0000000000000000000000000000000000` and `0000000000000000000000000000000001`. The total coins received per deposit is precisely `(amount of token deposited)/(minimum token denomination)`.

7.3.2 ERC20 Deposit

Let's say that the `tokenType` for `DAI` is `00000001` and the base denomination is ``0.1 DAI``. If the first user to deposit `DAi` deposits `0.5 DAI`, they'll recieve the coins `0x0000000100000000000000000000000000` up to and including coin `0x0000000100000000000000000000000004`.

CHAPTER 8

Transactions Over Ranges

Our plasma design introduces an new construction that allows users to transact over *ranges* of coins, rather than simply transacting single coins at a time. This page explains how we achieve that.

8.1 Transfers

A transaction consists of a specified block number and an array of Transfer objects. These Transfer objects describe the actual details of the transaction, including which ranges are being sent and to whom.

From the schema in `plasma-utils`:

```
const TransferSchema = new Schema({
 sender: {
   type: Address,
   required: true
 },
 recipient: {
   type: Address,
   required: true
 },
 token: {
   type: Number,
   length: 4,
   required: true
 },
 start: {
   type: Number,
   length: 12,
   required: true
 },
 end: {
   type: Number,
   length: 12,
```

(continues on next page)

(continued from previous page)

```
    required: true
  }
```

(Note that `length` is in bytes)

We can see that each Transfer in a Transaction specifies a `tokenType`, `start`, `end`, `sender`, and `recipient`.

8.2 Typed and Untyped Bounds

One thing to note above is that the `start` and `end` values are *12 bytes* and *not 16 bytes* like the Coin ID. This is because these values are "untyped" - they don't take the *tokenType* into account. We can calculate the "typed" values by concatenating the `token` field to either `start` or `end`. This design choice was made for API simplicity.

8.3 Atomic Transactions

The Transaction schema contains an *array* of Transfer objects. This means that a transaction can describe several transfers at the same time. Multiple transfers in the same transaction are all atomically executed *if any only if* the *entire transaction* is included and valid. This will form the basis for both decentralized exchange and defragmentation in later releases.

8.4 Serialization

`plasma-utils` implements a custom serialization library for the above data structures. Both the JSON-RPC API and the smart contract use byte arrays as encoded by the serializer.

Our encoding scheme is quite simple. The encoded version of a piece of data structure is the concatenation of each its values. Because each value has a fixed number of bytes defined by a schema, we can decode by slicing off the appropriate number of bytes. The choice to use a custom scheme instead of an existing one (like RLP encoding) was made to reduce smart contract complexity.

For encodings which involve variable-sized arrays, like Transaction objects which contain 1 or more Transfer objects, we prepend a single byte that represents the number of array elements.

CHAPTER 9

Proof Structure and Verification

9.1 What are History Proofs?

Unlike traditional blockchain nodes, our full plasma nodes don't store every single transaction. Instead, they only ever need to store information relevant to assets they own. This means that when you're receiving an asset, you don't have all of the information necessary to know that the person sending that asset is, in fact, the true owner. The `sender` needs to explicitly prove to the `recipient` that the `sender` actually owns the asset being sent.

Currently, the `sender` needs to give the `recipient` a list of every single transaction that has ever moved the asset around. The `sender` also needs to provide proof that they're not omitting any transactions. Once the `recipient` checks this proof, they can see the whole chain of custody for the asset and be convinced that the `sender` is the current owner. Generally, we call this proof a "history proof" because it tells the `recipient` about the history of an asset.

This page describes the methodology `plasma-core` follows to verify these history proofs.

9.2 Proof Structure

History proofs consist of a set of Deposits a long list of relevant Transactions with corresponding TransactionProofs. Here we'll discuss all of the various components of the proof.

9.2.1 Deposits

Deposits form the beginning of each history proof. An asset's history always starts from the point at which it was created. When we're talking about a range, we might need to provide more than one deposit.

Let's look at an example. Imagine that a sender is trying to create a proof for the range `(0, 100)`. The range `(0, 25)` was created in deposit #1, and the range `(25, 100)` was created in deposit #2. The sender **must** provide these two deposits as part of the proof.

9.2.2 Transaction Proofs

A TransactionProof contains all the necessary information to check the validity of a given Transaction. Namely, it is simply an array of TransferProof objects (described below). A given TransactionProof is valid if and only if all its TransferProofs are valid.

9.2.3 Transfer Proofs

A TransferProof contains all the necessary information required to check that a specific Transfer inside of a Transaction is valid. This includes:

- The Merkle tree branch that shows the transaction was included in a block.
- The position of the Merkle tree in which the transaction was included.
- The "parsed sum" for that transaction - a special value necessary to verify the Merkle proof.
- The transaction signature from the sender of the transfer.

Here's the schema taken right from `plasma-utils`:

```
const TransferProofSchema = new Schema({
  parsedSum: {
    type: Number,
    length: 16
  },
  leafIndex: {
    type: Number,
    length: 16
  },
  signature: {
    type: SignatureSchema
  },
  inclusionProof: {
    type: [Bytes],
    length: 48
  }
})
```

Note that the `inclusionProof` is a variable-length array whose size depends on the depth of the tree.

9.3 Proof Verification

The process of verifying a proof for an incoming transaction involves applying each proof element to the current "verified" state, starting with the deposits. If any proof element doesn't result in a valid state transition, we simply ignore that element and go onto the next. At the very end, we check that each of the transfers in the incoming transaction is part of the verified state.

9.3.1 Snapshot Objects

We keep track of the current owner of a range using an object called a Snapshot. Quite simply, a Snapshot represents the verified owner of a range at a block:

```
{
  start: number,
  end: number,
  block: number,
  owner: address
}
```

9.3.2 Checking for Exits

Before doing anything else, the verifier **must** check that the ranges being received have no pending or finalized exits. If any part of the received ranges have pending or finalized exits, the transaction should be rejected.

9.3.3 Applying Deposits

Every received range has to come from a corresponding deposit. A deposit record consists of its `token`, `start`, `end`, `depositer`, and `blockNumber`.

For each deposit record, the verifier **must** double-check with Ethereum to verify that the claimed deposit did indeed occur. The verifier must then add a verified Snapshot for each valid deposit, where `snapshot.owner = deposit.depositer`.

9.3.4 Applying Transactions

Next, the verifier must apply all given TransactionProofs and update the set of verified Snapshots accordingly. For each Transaction and corresponding TransactionProof, the verifier **must** first perform the following validation steps:

1. Check that the transaction encoding is well-formed.
2. For each Transfer in the Transaction:

1. Check that the Transfer has a corresponding **Signature_** created by `transfer.sender`.
2. Check that the Transfer was included in the plasma block using the `inclusionProof`, `leafIndex`, and `parsedSum`.
3. Calculate the `implicitStart` and `implicitEnd` of the Transfer, and verify that `implicitStart <= transfer.start < transfer.end <= implicitEnd`.

If any of the above checks fail, the transaction **must** be ignored and the verifier should continue onto the next transaction.

If all of the checks are successful, the verifier **must** apply each Transfer to the verified state:

1. For each Transfer in the Transaction, do the following:
    1. Break the Transfer into *implicit* components (`[implicitStart, typedStart]`, `[typedEnd, implicitEnd]`) and *explicit* components (`[typedStart, typedEnd]`).
    2. For each component:
        1. Find all verified Snapshots that overlap with the component.
        2. For each Snapshot that overlaps:
            2. Remove the Snapshot from the verified state.
            3. Split the Snapshot into overlapping and non-overlapping components.
            4. Re-insert any non-overlapping components into the verified state.

5. If `snapshot.block === transaction.blockNumber - 1` and `snapshot.owner === component.sender || component.implicit`:
   1. Increment `snapshot.block`.
   2. Set `snapshot.owner = transfer.sender`.
6. Insert the overlapping snapshot back into the verified state.

9.3.5 Verifying Transactions

Once all Deposits and Transactions have been applied to the verified state, the verifier can check the validity of the incoming transaction. The verifier **must** check that for each Transfer in the Transaction, there exists some Snapshot in the verified state such that:

1. `snapshot.owner === transfer.recipient`.
2. `snapshot.start <= transfer.typedStart`.
3. `snapshot.end >= transfer.typedEnd`.

If this condition is true for each Transfer in the Transaction, the proof can be accepted.

CHAPTER 10

Smart Contract and Exit Games

The proof for a chain of custody isn't useful unless it can also be passed to the main chain to keep funds secure. The mechanism which accepts proofs on-chain is the core of plasma's security model. We usually call this mechanism an "exit game".

When a user wants to move their money from plasma chain back to Ethereum, they start an "exit". However, the user doesn't get their funds immediately. Instead, each exit has to stand a "dispute period".

During the dispute period, users may submit "challenges" which claim the money being withdrawn isn't rightfully owned by the person who started the exit. If the person who started the exit *does* in fact own the money, they are always able to calculate and present a "challenge response" which closes the challenge. If there are no outstanding disputes at the end of the dispute period, the user can finalize their exit and receive the money.

The goal of the exit game is to keep user assets secure, even in the case of a malicious operator. Particularly, there are three main attacks which we must mitigate:

- **Data withholding:** the operator may publish a root hash to the contract, but not tell anybody what the contents of the block are.
- **Invalid transactions:** the operator may include a transaction in a block whose `sender` was not the previous `recipient` in the chain of custody.
- **Censorship:** the operator may refuse to publish any transactions from a specific user.

In all of these cases, the challenge/response protocol of the exit game ensures that these behaviors do not allow theft. Importantly, we also ensure that each challenge can be closed in at most one response.

10.1 Keeping Track of Deposits and Exits

10.1.1 Deposits Mapping

Each time a new set of coins is deposited, the contract updates a mapping that contains `deposit` structs.

From the contract:

```
struct deposit:
  untypedStart: uint256
  depositer: address
  precedingPlasmaBlockNumber: uint256
```

Note that this struct contains neither the `untypedEnd` or `tokenType` for the deposit. That's because the contract uses those values as the keys in a mapping of mappings. For example, to access the depositer of a given deposit, we can query `deposits` directly:

```
someDepositer: address = self.deposits[tokenType][untypedEnd].depositer
```

This choice was made to save gas and to simplify the smart contract. Namely, it means that we don't need to store any sort of deposit ID in order to reference a deposit.

10.1.2 Exitable Ranges Mapping

The contract also needs to keep track of finalized exits in order prevent multiple exits on the same range. This is a little trickier because exits don't occur in order like deposits do, and it'd be too expensive to search through a giant list of exits.

Our contract implements a constant-sized solution, which instead stores a list of "exitable ranges". This list is updated as new exits occur.

From the smart contract:

```
struct exitableRange:
  untypedStart: uint256
  isSet: bool
```

Again, we use a double-nested mapping with keys `tokenType` and `untypedEnd` so that we may call `self.exitable[tokenType][untpyedEnd].untypedStart` to access the start of the range. Note that Vyper returns `0` for all unset mapping keys, so we need an `isSet` bool so that users may not "trick" the contract by passing an unset `exitableRange`.

The contract's `self.exitable` ranges are split and deleted based on successful calls to `finalizeExit` via a helper function called `removeFromExitable`. Note that exits on a previously exited range do not even need to be challenged; they'll never pass the `checkRangeExitable` function called in `finalizeExit`. You can find that code here.

10.2 Similarities to Plasma Cash

At heart, the exit games in our spec are very similar to the original Plasma Cash design. Exits are initiated with calls to:

```
beginExit(tokenType: uint256, blockNumber: uint256, untypedStart: uint256,␣
↪untypedEnd: uint256) -> uint256
```

All exit challenges specify a particular coin ID, and a Plasma Cash-style challenge game is carried out on that particular coin. Only a single coin needs to be proven invalid to cancel the entire exit.

Both exits and challenges are assigned a unique `exitID` and `challengeID`. These IDs are assigned in order based on an incrementing `challengeNonce` and `exitNonce`.

10.3 Block-specific Transactions

In the original Plasma Cash spec, the exiter is required to specify both the exited transaction and its previous "parent" transaction to prevent the "in-flight" attack. This attack occurs when the operator delays inclusion of a valid transaction and then inserts an invalid transaction before the valid one.

This poses a problem for our range-based schemes because a transaction may have multiple parents. For example, if Alice sends `(0, 50)` to Carol, and Bob sends `(50, 100)` to Carol, Carol can now send `(0, 100)` to Dave. If Dave wants to exit `(0, 100)`, he would need to specify both `(0, 50)` and `(50, 100)` as parents.

If a range has dozens or even hundreds of parents, it becomes basically impossible to publish all of these parents on chain. Instead, we opted for a simpler alternative in which each transaction specifies the block in which it should be included. If the transaction is included in a different block, it's no longer valid. This solves the in-flight attack because it becomes impossible for the operator to delay inclusion of the transaction.

This does, unfortunately, introduce one downside – if a transaction isn't included in the specified block (for whatever reason), it needs to be re-signed and re-submitted. Hopefully this won't happen too often in practice, but it's something to think about.

For those interested in a formal writeup and safety proof for this scheme, it's worth giving this great post a look.

10.4 Per-coin Transaction Validity

An unintuitive property of our exit games that's worth noting up front is that a certain transaction might be "valid" for some of the coins in its range, but not for others.

For example, imagine that Alice sends `(0, 100)` to Bob, who in turn sends `(50, 100)` to Carol. Carol doesn't need to verify that Alice was the rightful owner of the full `(0, 100)`. Carol only needs an assurance that Alice owned `(50, 100)` –the part of the custody chain which applies to Carol's range.

Though the transaction to Dave might in a sense be "invalid" if Alice didn't own `(0, 50)`, the smart contract doesn't care for the purposes of disputes on exits for the coins `(50, 100)`. As long as the received coins are valid, invalid transactions on any other coins don't matter.

This is a **very important requirement** to preserve the size of light client proofs. If Carol had to check the full `(0, 100)`, she might also have to check an overlapping parent of `(0, 10000)`, and then all of its parents, and so on. This "cascading" effect could massively increase the size of proofs if transactions were very interdependent.

Note that this property also applies to atomic multisends, in which multiple ranges are *swapped*. If Alice trades 1 ETH for Bob's 1 DAI, it is Alice's responsibility to check that Bob owns the 1 DAI before signing. However, after, if Bob then sends the 1 ETH to Carol, Carol need not verify that Bob owned the 1 DAI, only that Alice owned the 1 ETH she sent to Bob. Alice incurred the risk, so Carol doesn't have to.

From the standpoint of the smart contract, this property is a direct consequence of challenges always being submitted for a particular `coinID` within the exit.

10.5 Transaction Verification

Only funds that came from valid transactions can be withdrawn. We can check the validity of a transaction at the contract level via:

```
def checkTransactionProofAndGetTypedTransfer(
  transactionEncoding: bytes[277],
  transactionProofEncoding: bytes[1749],
```

(continues on next page)

(continued from previous page)

```
  transferIndex: int128
) -> (
  address, # transfer.to
  address, # transfer.from
  uint256, # transfer.start (typed)
  uint256, # transfer.end (typed)
  uint256  # transaction.blockNumber
)
```

An important feature here is the `transferIndex` argument. Remember that a transaction may contain multiple transfers and that the transaction must be included in the tree once for each transfer. However, since challenges refer to a specific `coinID`, only a single transfer will be relevant. As a result, challengers and responders have to give a `transferIndex` –a reference to the index of the relevant transfer.

Once we decode the `TransactionProof`, we can check the relevant `TransferProof`:

```
def checkTransferProofAndGetTypedBounds(
  leafHash: bytes32,
  blockNum: uint256,
  transferProof: bytes[1749]
) -> (uint256, uint256)
```

10.6 Challenges That Immediately Block Exits

Two kinds of challenges immediately cancel exits: those that show a specific coin is already spent, and those that show an exit comes before the deposit.

10.6.1 Spent-Coin Challenge

This challenge is used to demonstrate that coins being withdrawn have already been spent.

```
@public
def challengeSpentCoin(
  exitID: uint256,
  coinID: uint256,
  transferIndex: int128,
  transactionEncoding: bytes[277],
  transactionProofEncoding: bytes[1749],
)
```

It uses `checkTransactionProofAndGetTypedTransfer` and then checks the following:

1. The challenged coinID lies within the specified exit.
2. The challenged coinID lies within the `typedStart` and `typedEnd` of the `transferIndex``th element of ``transaction.transfers`.
3. The `plasmaBlockNumber` of the challenge is greater than that of the exit.
4. The `transfer.sender` is the exiter.

The introduction of atomic swaps does mean one thing: the spent coin challenge period must be strictly less than others. There's an edge case in which the operator withholds an atomic swap between two or more parties. Those parties must exit their coins from *before* the swap because they don't know if the swap was included. If the swap was

not included, then these exits will finalize successfully. However, if the swap *was* included, then operator can submit a Spent-Coin Challenge and block these exits.

If we allowed the operator to submit this challenge at the last minute, we'd be creating a race condition in which the parties have no time to use the newly revealed information to cancel other exits. Thus, the timeout is made shorter (1/2) than the regular challenge window, eliminating "last-minute response" attacks.

10.6.2 Before-Deposit Challenge

This challenge is used to demonstrate that an exit comes from a `plasmaBlockNumber` earlier than the coin's deposit.

```
@public
def challengeBeforeDeposit(
  exitID: uint256,
  coinID: uint256,
  depositUntypedEnd: uint256
)
```

The contract looks up `self.deposits[self.exits[exitID].tokenType][depositUntypedEnd].precedingPlasmaBlockNumber` and checks that it's is later than the exit's block number. If so, it cancels the exit immediately.

10.7 Optimistic Exits and Inclusion Challenges

Our contract allows an exit to occur without actually checking that the transaction referenced in the exit was included in the plasma chain. This is called an "optimistic exit," and allows us to reduce gas costs for users who are behaving honestly. However, this means that it's possible for someone start an exit from a transaction that never happened.

As a result, we expose a way for someone to challenge this type of exit:

```
@public
def challengeInclusion(exitID: uint256)
```

Then, the user who started the exit can respond by showing that the transaction or deposit from which they are exiting really did happen:

```
@public
def respondTransactionInclusion(
  challengeID: uint256,
  transferIndex: int128,
  transactionEncoding: bytes[277],
  transactionProofEncoding: bytes[1749],
)
...
@public
def respondDepositInclusion(
  challengeID: uint256,
  depositEnd: uint256
)
```

We need this special second case so that users can withdraw money even if the operator is censoring all transactions after their deposit.

Both responses cancel the challenge if: 1. The deposit or transaction was indeed at the exit's plasma block number. 2. The depositer or recipient is indeed the exiter. 3. The start and end of the exit were within the deposit or transfer's start and end

10.8 Invalid-History Challenge

The Invalid-History Challenge is the most complex challenge-response game in both vanilla Plasma Cash and this spec. This part of the protocol mitigates the attack in which the operator includes an forged "invalid" transaction whose sender is not the previous recipient.

Effectively, this challenge allows the rightful owner of a coin to request that the exiter provide a proof that the owner has spent their funds. The idea here is that if the rightful owner really is the rightful owner, then the exiter will not be able to provide such a transaction.

Both invalid history challenges and responses can be either deposits or transactions.

10.8.1 Challenging

There are two ways to challenge, depending on the current rightful owner:

```
@public
def challengeInvalidHistoryWithTransaction(
  exitID: uint256,
  coinID: uint256,
  transferIndex: int128,
  transactionEncoding: bytes[277],
  transactionProofEncoding: bytes[1749]
)
```

and

```
@public
def challengeInvalidHistoryWithDeposit(
  exitID: uint256,
  coinID: uint256,
  depositUntypedEnd: uint256
)
```

Both of these methods call an additional method, `challengeInvalidHistory`:

```
@private
def challengeInvalidHistory(
  exitID: uint256,
  coinID: uint256,
  claimant: address,
  typedStart: uint256,
  typedEnd: uint256,
  blockNumber: uint256
)
```

This method does the legwork of checking that the `coinID` is within the challenged exit, and that the `blockNumber` is earlier than the exit.

10.8.2 Responding

Of course it's also possible for someone to submit a fraudulent Invalid-History Challenge. Therefore we give exiters two ways to respond to this type of challenge.

The first is to respond with a transaction showing that the challenger did, in fact, spend their money:

```
@public
def respondInvalidHistoryTransaction(
  challengeID: uint256,
  transferIndex: int128,
  transactionEncoding: bytes[277],
  transactionProofEncoding: bytes[1749],
)
```

The smart contract then performs the following checks: 1. The `transferIndex``th ``Transfer` in the `transactionEncoding` covers the challenged `coinID`. 2. The `transferIndex``th ``transfer.sender` was indeed the claimant for that invalid history challenge. 3. The transaction's plasma block number lies between the invalid history challenge and the exit.

The second response is to show the challenge came *before* the coins were actually deposited-making the challenge invalid. This is similar to a `challengeBeforeDeposit`, but for the exit itself.

```
@public
def respondInvalidHistoryDeposit(
  challengeID: uint256,
  depositUntypedEnd: uint256
)
```

In this case, there is no check on the sender being the challenge recipient, since the challenge was invalid. So the contract just needs to check: 1. The deposit covers the challenged `coinID`. 2. The deposit's plasma block number lies between the challenge and the exit.

If all of these conditions are true, the exit is cancelled.

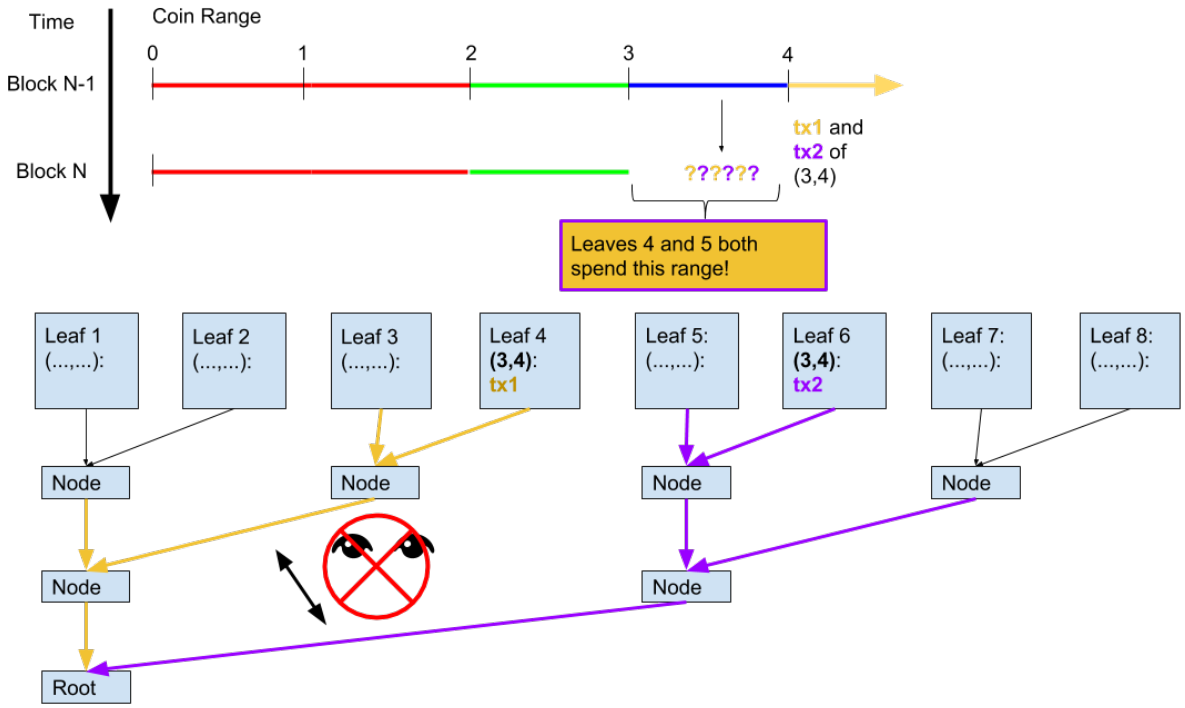
CHAPTER 11

Merkle Sum Tree Block Structure

One of the most important improvements Plasma Cash introduced was "light proofs." Previously, plasma constructions required that users download the entire plasma chain to ensure safety of their funds. With Plasma Cash, they only have to download the branches of a Merkle tree relevant to their own funds.

This was accomplished by introducing a new transaction validity condition: transactions of a particular coinID are only valid at the coinIDth leaf in the Merkle tree. Thus, it is sufficient to download just that branch to be confident no valid transaction exists for that coin. The problem with this scheme is that transactions are "stuck" at this denomination: if you want to transact multiple coins, you need multiple transactions, one at each leaf.

Unfortunately, if we put the range-based transactions into branches of a regular Merkle tree, light proofs would become insecure. This is because having one branch does not guarantee that others don't intersect:



Leaves 4 and 6 both describe transactions over the range (3,4). Having one branch DOES NOT guarantee that the other doesn't exist.

With a regular Merkle tree, the only way to guarantee no other branches intersect is to download them all and check. But that's no longer a light proof!

At the heart of our plasma implementation is a new block structure, and an accompanying new transaction validity condition, which allows us to get light proofs for range-based transactions. The block structure is called a Merkle sum tree, where next to each hash is a *sum* value.

The new validity condition uses the `sum` values for a particular branch to compute a a `start` and `end` range. This calculation is specially crafted so that it is *impossible for two branches' computed ranges to overlap.* A *transfer* is only valid if its own range is within that range, so this gets us back our light clients!

This section will specify the exact spec of the sum tree, what the range calculation actually is, and how we actually construct a tree which satisfies the range calculation. For a more detailed background and motivation on the research which led us to this spec, feel free check out *this* post.

We have written two implementations of the plasma Merkle sum tree: one done in a database for the operator, and another in-memory for testing in plasma-utils.

11.1 Sum Tree Node Specification

Each node in the Merkle sum tree is 48 bytes, as follows:

```
[32 byte hash][16 byte sum]
```

It's not a coincidence that the `sum`'s 16 bytes length is the same as a `coinID`! We have two helper properties, `.hash` and `.sum`, which pull out these two parts. For example, for some `node = 0x1b2e79791f28c27ed669f257397e1deb3e522cf1f27024c161b619d276a25315ffffffffffffffffffffffff` we have `node.hash == 0x1b2e79791f28c27ed669f257397e1deb3e522cf1f27024c161b619d276a25315` and `node.sum == 0xffffffffffffffffffffffffffffffff`.

11.2 Parent Calculation

In a regular Merkle tree, we construct a binary tree of hash nodes, up to a single root node. Specifying the sum tree format is a simple matter of defining the parent(left, right) calculation function which accepts the two siblings as arguments. For example, a regular Merkle sum tree has: .. code-block:: javascript

    parent = function (left, right) { return Sha3(left.concat(right)) }

Where `Sha3` is the hash function and `concat` appends the two values together. To create a merkle *sum* tree, the `parent` function must also concatenate the result of an addition operation on its children's own "su

```
parent = function (left, right) { return Sha3(left.concat(right)).concat(left.sum +␣
↪right.sum)  }
```

For example, we might have

```
parent(0xabc...0001, 0xdef...0002) === hash(0xabc...0001.
concat(0xdef...0002)).concat(0001 + 0002) === 0x123...0003
```

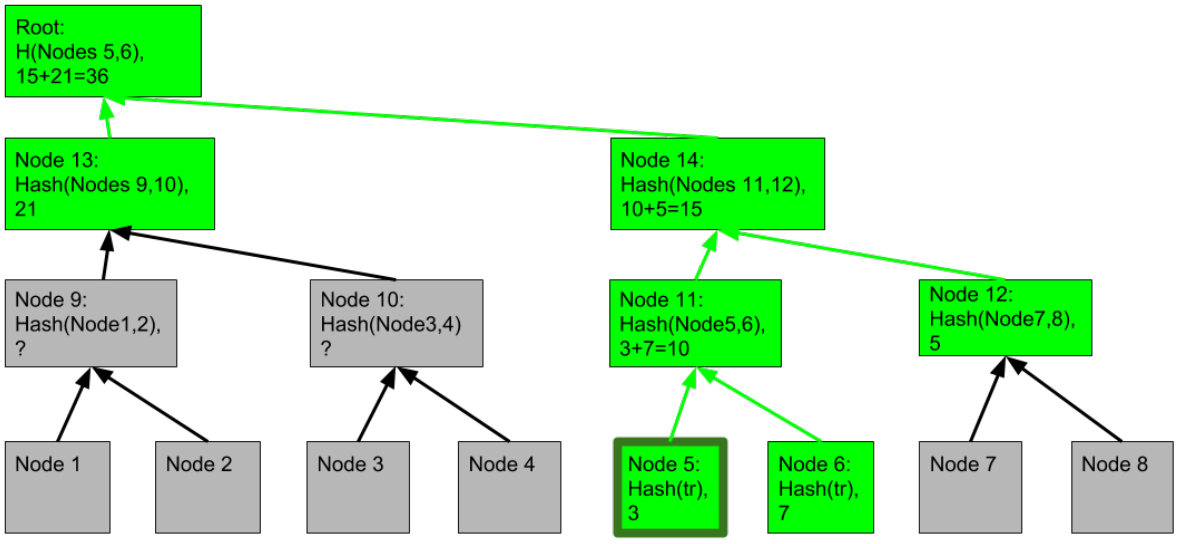
Note that the `parent.hash` is a commitment to each `sibling.sum` as well as the hashes: we hash the full 96 bytes of both.

11.3 Calculating a Branch's Range

The reason we use a merkle sum tree is because it allows us to calculate a specific range which a branch describes, and be 100% confident that no other valid branches exist which overlap that range.

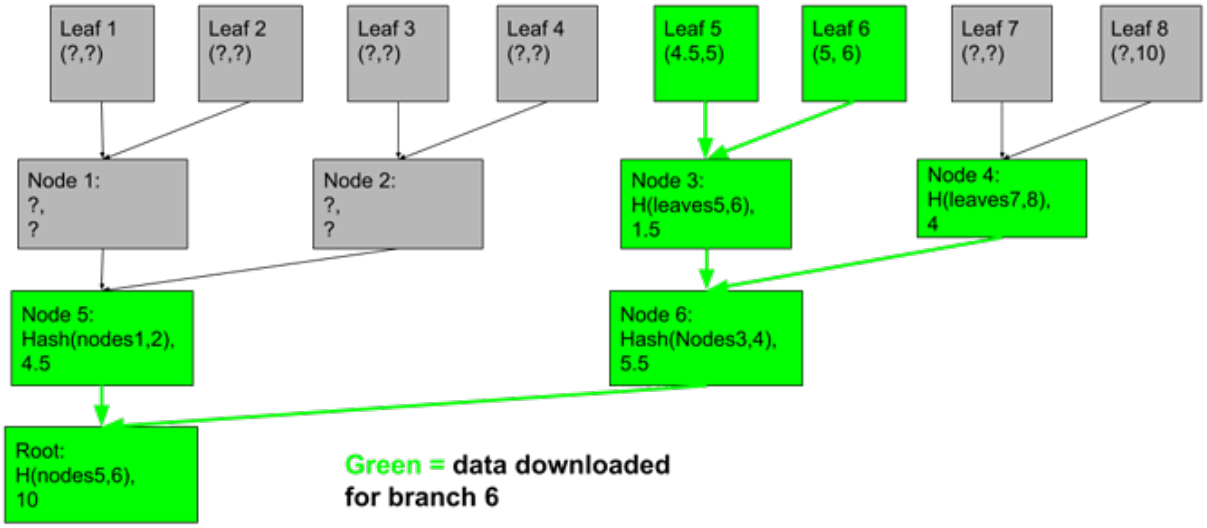
We calculate this range by adding up a `leftSum` and `rightSum` going up the branch. Initializing both to 0, at each parent verification, if the leaf lies somewhere under the `left` child, we take `rightSum += right.sum`, and if the leaf is under the `right`, we add `leftSum += left.sum`.

Then, the range the branch describes is `(leftSum, root.sum - rightSum)`. See the following example:

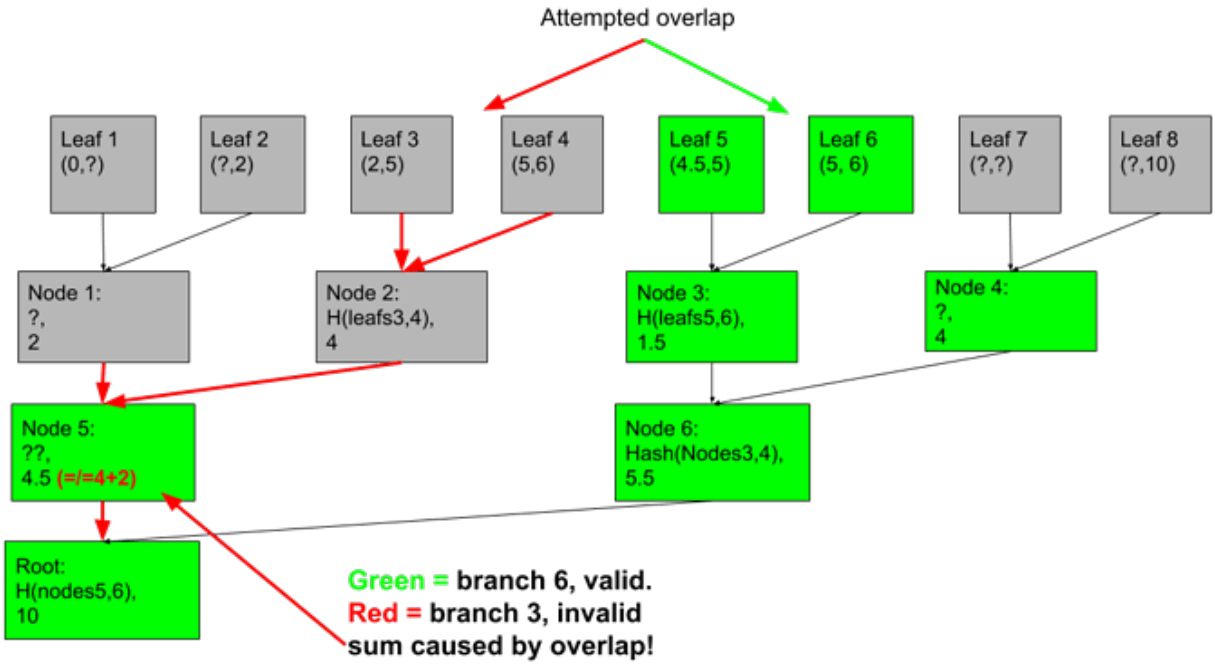


In this example, branch 6's valid range is `[21+3, 36-5) == [24, 31)`. Notice that `31-24=7`, which is the sum value for leaf 6! Similarly, branch 5's valid range is `[21, 36-(7+5)) == [21, 24)`. Notice that its end is the same as branch 6's start!

If you play around with it, you'll see that it's impossible to construct a Merkle sum tree with two different branches covering the same range. At some level of the tree, the sum would have to be broken! Go ahead, try to "trick" leaf 5 or 6 by making another branch that intersects the range (4.5,6). Filling in only the ‘‘?‘‘s in grey boxes:



You'll see it's always impossible at some level of the tree:



This is how we get light clients. We call the branch range bounds the `implicitStart` and `implicitEnd`, because they are calculated "implicitly" from the inclusion proof. We have a branch checker implemented in `plasma-utils` via `calculateRootAndBounds()` for testing and client-side proof checking:

let leftSum = new BigNum(0) let rightSum = new BigNum(0) for (let i = 0; i < inclusionProof.length; i++) {

let encodedSibling = inclusionProof[i] if (path[i] === '0') {

computedNode = PlasmaMerkleSumTree.parent(computedNode, sibling) rightSum = rightSum.add(sibling.sum)

} else { computedNode = PlasmaMerkleSumTree.parent(sibling, computedNode) leftSum = leftSum.add(sibling.sum)

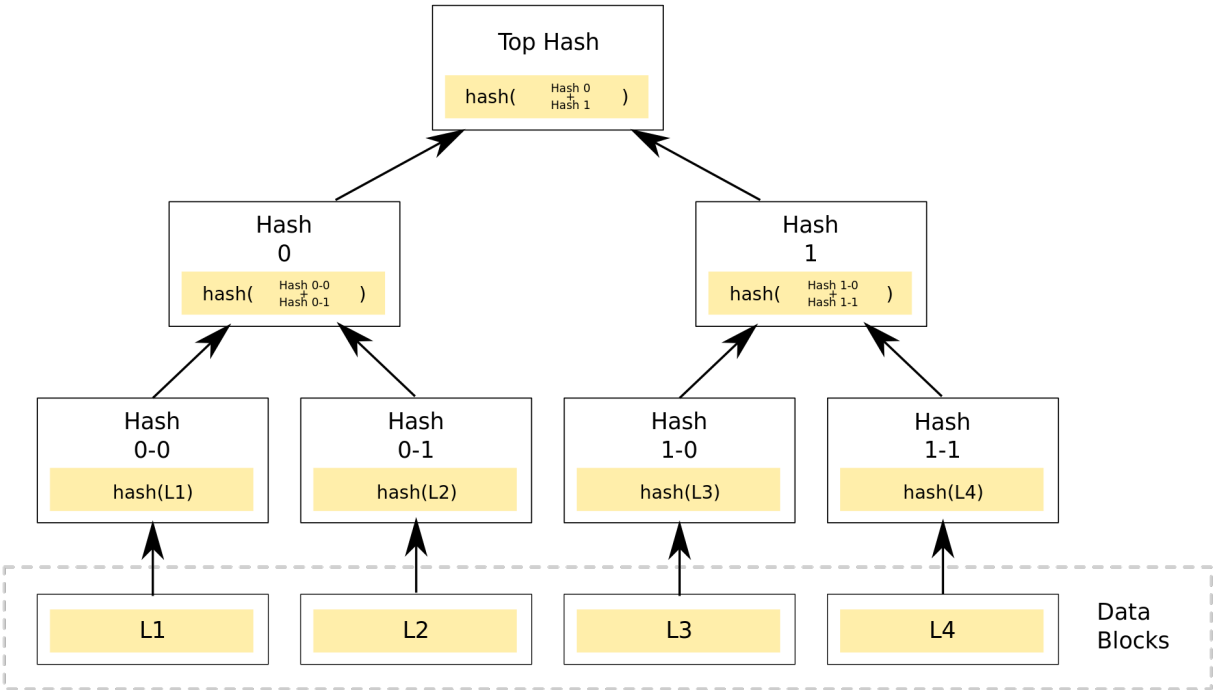
}

}

as well as in Vyper for the smart contract via `checkTransferProofAndGetTypedBounds` in `PlasmaChain.vy`

11.4 Parsing Transfers as Leaves

In a regular merkle tree, we construct the bottom layer of nodes by hashing the "leaves":

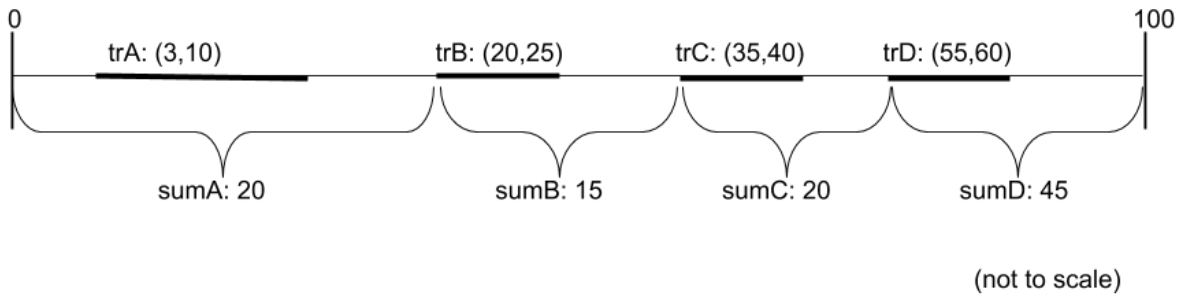


In our case, we want the leaves to be the transactions of ranges of coins. More specifically, we actually want 'Transfer's–signatures don't need to be included, they can be stored by the clients and submitted to the smart contract separately. (For more details on objects and serialization, see the serialization section.)

So–the hashing is straightforward–but what should the bottom nodes' *.sum* values be?

Given some `txA` with a single `transferA`, what should the sum value be? It turns out, _not_ just `transferA.end - transferA.start`. The reason for this is that it might screw up other branches' ranges if the transfers are not touching. We need to "pad" the sum values to account for this gap, or the root.sum will be too small.

Interestingly, this is a non-deterministic choice because you can pad either the node to the right or left of the gap. We've chosen the following "left-aligned" scheme for parsing leaves into blocks:



We call the bottommost `.sum` value the `parsedSum` for that branch, and the `TransferProof` schema includes a `.parsedSum` value which is used to reconstruct the bottom node.

11.5 Branch Validity and Implicit NoTx

Thus, the validity condition for a branch as checked by the smart contract is as follows: `implicitStart <= transfer.start < transfer.end <= implicitEnd`. Note that, in the original design of the sum tree in Plasma Cashflow, some leaves were filled with `NoTx` to represent that ranges were not transacted. With this format, any coins which are not transacted are simply those between `(implicitStart, transfer.start)` and `(transfer.end, implicitEnd)`. The smart contract guarantees that no coins in these ranges can be used in any challenge or response to an exit.

11.6 Atomic Multisends

Often (to support transaction fees and exchange) transactions require multiple transfers to occur or not, atomically, to be valid. The effect is that a valid transaction needs to be included once for each of its `.transfers``-each with a valid sum in relation to that particular ``transfer.typedStart` and `.typedEnd`. However, for each of these inclusions, it's still the hash of the full `UnsignedTransaction``-NOT the individual ``Transfer`- that is parsed to the bottom `.hash`.

CHAPTER 12

JSON-RPC Calls

12.1 pg_getBalance

```
pg_getBalance
```

Returns the balance of a specific account.

12.1.1 Parameters

1. `address` - `string`: Address of the account to query.

12.1.2 Returns

`Array`: A list of token balances in the form *(token, balance)*.

12.2 pg_getBlock

```
pg_getBlock
```

Pulls the hash of the block at a specific height.

12.2.1 Parameters

1. `block` - `number`: Number of the block to query.

12.2.2 Returns

`string`: The block hash.

12.3 pg_getTransaction

```
pg_getTransaction
```

Pulls information about a specific transaction.

12.3.1 Parameters

1. `hash` - `string`: The hash of the transaction.

12.3.2 Returns

`SignedTransaction`: The specified transaction.

12.4 pg_sendTransaction

```
pg_sendTransaction
```

Sends a transaction to the node to be processed.

12.4.1 Parameters

1. **`transaction` - `Object`:**
    - `from` - `string`: Address from which the transaction was sent.
    - `to` - `string`: Address to which the transaction was sent.
    - `token` - `string`: ID of the token to be sent.
    - `value` - `number`: Value of tokens to be sent.

12.4.2 Returns

`string`: The transaction receipt.

12.5 pg_sendRawTransaction

```
pg_sendRawTransaction
```

Sends an encoded SignedTransaction to the node to be processed.

12.5.1 Parameters

1. `transaction` - `string`: Encoded signed transaction.

12.5.2 Returns

`string`: The transaction receipt.

12.6 pg_getHeight

```
pg_getHeight
```

Returns the current plasma block height.

12.6.1 Returns

`number`: The current block height.

12.7 pg_getRecentTransactions

```
pg_getRecentTransactions
```

Returns the most recent transactions. Because there are a *lot* of transactions in each block, this method is paginated.

12.7.1 Parameters

1. `start` - `number`: Start of the range of recent transactions to return.
2. `end` - `number`: End of range of recent transactions to return.

12.7.2 Returns

`Array<SignedTransaction>`: A list of SignedTransaction objects.

12.8 pg_getAccounts

```
pg_getAccounts
```

Returns a list of all available accounts.

12.8.1 Returns

`Array<string>`: A list of account addresses.

12.9 pg_getTransactionsByAddress

```
pg_getTransactionsByAddress
```

Returns the latest transactions by an address. This method is paginated and requires a `start` and `end`. Limited to a total of **25** transactions at a time.

12.9.1 Parameters

1. `address - ``string`: The address to query.
2. `start` - `number`: Start of the range of recent transactions to return.
3. `end` - `number`: End of range of recent transactions to return.

12.9.2 Returns

`Array<SignedTransaction>`: A list of SignedTransaction objects.

CHAPTER 13

ContractProvider

`ContractProvider` is a wrapper that interacts with the plasma chain smart contract.

13.1 address

```
contract.address
```

Returns the contract's address.

13.1.1 Returns

`string`: Address of the connected contract.

13.2 hasAddress

```
contract.hasAddress
```

Whether or not the contract has an address.

13.2.1 Returns

`boolean`: `true` if the contract is ready to be used, `false` otherwise.

13.3 ready

```
contract.ready
```

Whether or not the contract is ready to be used.

13.3.1 Returns

`boolean`: `true` if hte contract is ready, `false` otherwise.

13.4 web3

```
contract.web3
```

Returns the web3 instance being used by the contract.

13.4.1 Returns

`Web3`: Contract's web3 instance.

13.5 plasmaChainName

```
contract.plasmaChainName
```

Name of the plasma chain this contract is connected to.

13.5.1 Returns

`string`: Plasma chain name.

13.6 checkAccountUnlocked

```
contract.checkAccountUnlocked(address)
```

Checks whether an account is unlocked and attempts to unlock it if not.

13.6.1 Parameters

1. `address` - `string`: Address of the account to check.

13.7 getBlock

```
contract.getBlock(block)
```

Queries the hash of a given block.

13.7.1 Parameters

1. `block` - `number`: Number of the block to query.

13.7.2 Returns

`Promise<string>`: Root hash of the block with that number.

13.8 getNextBlock

```
contract.getNextBlock()
```

Returns the number of the next block that will be submitted.

13.8.1 Returns

`Promise<number>`: Next block number.

13.9 getCurrentBlock

```
contract.getCurrentBlock()
```

Returns the number of the last block to be submitted.

13.9.1 Returns

`Promise<number>`: Last block number.

13.10 getOperator

```
contract.getOperator()
```

Returns the address of the operator.

13.10.1 Returns

`Promise<string>`: Plasma chain operator address.

13.11 getTokenAddress

```
contract.getTokenAddress(token)
```

Returns the address for a given token ID.

13.11.1 Parameters

1. `token` - `string`: A token ID.

13.11.2 Returns

`Promise<string>`: Address of the contract for that token.

13.12 listToken

```
contract.listToken(tokenAddress)
```

Lists a token with the given address so that it can be deposited.

13.12.1 Parameters

1. `tokenAddress` - `string`: Address of the token to list.

13.12.2 Returns

`EthereumTransaction`: The Ethereum transaction result.

13.13 getChallengePeriod

```
contract.getChallengePeriod()
```

Returns the current challenge period in number of blocks.

13.13.1 Returns

`Promise<number>`: Challenge period.

13.14 getTokenId

```
contract.getTokenId(tokenAddress)
```

Gets the token ID for a specific token.

13.14.1 Parameters

1. `tokenAddress` - `string`: Token contract address.

13.14.2 Returns

`Promise<string>`: ID of the token.

13.15 depositValid

```
contract.depositValid(deposit)
```

Checks whether a Deposit actually exists. Used when checking transaction proofs.

13.15.1 Parameters

1. `deposit` - `Deposit`: A Deposit to validate.

13.15.2 Returns

`boolean`: `true` if the deposit exists, `false` otherwise.

13.16 deposit

```
contract.deposit(address, token, amount)
```

Deposits some value of a token to the plasma smart contract.

13.16.1 Parameters

1. `address` - `string`: Address to deposit with. 1. `token` - `string`: Address of the token to deposit. 2. `amount` - `number`: Amount to deposit.

13.16.2 Returns

`EthereumTransaction`: An Ethereum transaction receipt.

13.17 startExit

```
contract.startExit(block, token, start, end, owner)
```

Starts an exit for a user. Exits can only be started on *transfers*, meaning you need to specify the block in which the transfer was received.

13.17.1 Parameters

1. `block` - `BigNum`: Block in which the transfer was received.
2. `token` - `BigNum`: Token to be exited.
3. `start` - `BigNum`: Starts of the range received in the transfer.
4. `end` - `BigNum`: End of the range received in the transfer.
5. `owner` - `string`: Address to withdraw from.

13.17.2 Returns

`EthereumTransaction`: Exit transaction receipt.

13.18 finalizeExit

```
contract.finalizeExit(exitId, exitableEnd, owner)
```

Finalizes an exit for a user.

13.18.1 Parameters

1. `exitId` - `string`: ID of the exit to finalize.
2. `exitableEnd` - `BigNum`: The "exitable end" for that exit.
3. `owner` - `string`: Address that owns the exit.

13.18.2 Returns

`EthereumTransaction`: Finalization transaction receipt.

13.19 submitBlock

```
contract.submitBlock(hash)
```

Submits a block with the given hash. Will only work if the operator's account is unlocked and available to the node.

13.19.1 Parameters

1. `hash` - `string`: Hash of the block to submit.

13.19.2 Returns

`EthereumTransaction`: Block submission transaction receipt.

CHAPTER 14

OperatorService

`OperatorService` handles all interaction with the operator. This includes things like sending transactions and pulling any pending transactions.

14.1 getNextBlock

```
operator.getNextBlock()
```

Returns the next block that will be submitted.

14.1.1 Returns

`Promise<number>`: Next block number.

14.2 getEthInfo

```
operator.getEthInfo()
```

Returns information about the smart contract.

14.2.1 Returns

`Promise<Object>`: Smart contract info.

14.3 getTransactions

```
operator.getTransactions(address, startBlock, endBlock)
```

Returns a list of transactions received by an address between two blocks.

14.3.1 Parameters

1. `address` - `string`: Address to query.
2. `startBlock` - `number`: Block to query from.
3. `endBlock` - `number`: Block to query to.

14.3.2 Returns

`Promise<Array>`: List of encoded transactions.

14.4 getTransaction

```
operator.getTransaction(encoded)
```

Returns a transaction proof for a given transaction.

14.4.1 Parameters

1. `encoded` - `string`: The encoded transaction.

14.4.2 Returns

`Promise<Object>`: Proof information for the transaction.

14.5 sendTransaction

```
operator.sendTransaction(transaction)
```

Sends a **SignedTransaction_** to the operator.

14.5.1 Parameters

1. `transaction` - `string`: The encoded **SignedTransaction_**.

14.5.2 Returns

`Promise<string>`: The transaction receipt.

14.6 submitBlock

```
operator.submitBlock()
```

Attempts to have the operator submit a new block. Won't work if the operator is properly configured, but used for testing.

CHAPTER 15

JSONRPCService

`JSONRPCService` handles incoming JSON-RPC method calls. A full list of methods is documented at the JSON-RPC Methods Specification. Note that `JSONRPCService` does **not** expose any form of external interface (such as an HTTP server). Full nodes should implement these services so that users can interact with the node. For more information about these external services, see our document on extending plasma-core.

15.1 getAllMethods

```
jsonrpc.getAllMethods()
```

Returns all available RPC methods.

15.1.1 Returns

`Object`: All subdispatcher methods as a single object in the form `{ name: methodref }`.

15.2 getMethod

```
jsonrpc.getMethod(name)
```

Returns a method by its name.

15.2.1 Parameters

1. `name` - `string`: Name of the method.

15.2.2 Returns

`Function`: A reference to the method with that name.

15.3 handle

```
jsonrpc.handle(method, params = [])
```

Calls a method with the given name and params.

15.3.1 Parameters

1. `method` - `string`: Name of the method to call.
2. `params` - `Array`: An array of parameters.

15.3.2 Returns

`Promise<any>`: The result of the method call.

15.4 handleRawRequest

```
jsonrpc.handleRawRequest(request)
```

Handles a raw JSON-RPC request.

15.4.1 Parameters

1. `request` - `Object`: A JSON-RPC request object.

15.4.2 Returns

`Promise<Object>`: A JSON-RPC response object.

CHAPTER 16

ProofService

`ProofService` handles checking the validity of transactions. In the Plasma Group plasma chain design, each client only receives transactions that are relevant to that client. The client then needs to check that the received transaction is actually valid. This is carried out via a transaction proof attached to each transaction. If you're interested in learning more about transaction proofs, check out our more detailed transaction proof specification document.

16.1 checkProof

```
proofService.checkProof(transaction, proof)
```

Checks the validity of a transaction using the given proof.

16.1.1 Parameters

1. `transaction` - `Object`: A Transaction object.

2. `deposits` - `Array<Deposit>`: An array of Deposits. 2. `proof` - `Object`: A Proof object.

16.1.2 Returns

`Promise<boolean>`: `true` if the transaction is valid, `false` otherwise.

CHAPTER 17

SyncService

`SyncService` makes sure that the client's wallet is always synchronized. This service automatically pulls any pending transactions and keeps transaction proofs up to date.

CHAPTER 18

ChainService

`ChainService` does most of the heavy lifting when it comes to receiving and sending transactions. This service handles processing new transactions and computing the latest state.

18.1 getBalances

```
chain.getBalances(address)
```

Returns a list of balances for a user.

18.1.1 Parameters

1. `address` - `string`: Address of the user to query.

18.1.2 Returns

`Promise<Object>`: An object where keys are tokens and values are the balances.

18.1.3 Example

```
const balances = await chain.getBalances(address)
console.log(balances)
> { '0': '1194501', '1': '919ff01' }
```

18.2 addDeposits

```
chain.addDeposits(deposits)
```

Applies a series of deposits to the state.

18.2.1 Parameters

1. `deposits` - `Array<Deposit>`: An array of Deposit objects to apply.

18.3 getExitsWithStatus

```
chain.getExitsWithStatus(address)
```

Returns any exits started by a specific user. Identifies exits that are finalized or ready to be finalized.

18.3.1 Parameters

1. `address` - `string`: Address of the user to query.

18.3.2 Returns

`Array<Exit>`: An array of Exits started by the user.

18.4 addExit

```
chain.addExit(exit)
```

Applies an exit to the local state. Internally, sends the exited range to the zero address.

18.4.1 Parameters

1. `exit` - `Exit`: An Exit to apply.

18.5 pickRanges

```
chain.pickRanges(address, token, amount)
```

Picks the best ranges to use for a transaction.

18.5.1 Parameters

1. `address` - `string`: Address sending the transaction.
2. `token` - `BigNum`: ID of the token being sent.
3. `amoun` - `BigNum`: Amount of the token being sent.

18.5.2 Returns

`Array<Range>`: Best ranges for the transaction.

18.6 pickTransfers

```
chain.pickTransfers(address, token, amount)
```

Picks the best Transfers to use for an exit. This is currently necessary because of a quirk in how we're processing exits.

18.6.1 Parameters

1. `address` - `string`: Address sending the transaction.
2. `token` - `BigNum`: ID of the token being sent.
3. `amoun` - `BigNum`: Amount of the token being sent.

18.6.2 Returns

`Array<Range>`: Best ranges for the transaction.

18.7 startExit

```
chain.startExit(address, token, amount)
```

Attempts to start an exit for a user. May submit more than one exit if neccessary to withdraw the entire amount.

18.7.1 Parameters

1. `address` - `string`: Account to withdraw from.
2. `token` - `BigNum`: ID of the token to exit.
3. `amount` - `BigNum`: Amount to exit.

18.7.2 Returns

`Array<String>`: An array of Ethereum transaction hashes.

18.8 finalizeExits

```
chain.finalizeExits(address)
```

Attempts to finalize all pending exits for an account.

18.8.1 Parameters

1. `address` - `string`: Address to finalize exits for.

18.8.2 Returns

`Array<String>`: An array of Etheruem transaction hashes.

18.9 sendTransaction

```
chain.sendTransaction(transaction)
```

Sends a transaction to the operator.

18.9.1 Parameters

1. `transaction` - `Transaction`: Transaction to be sent.

18.9.2 Returns

`string`: The transaction receipt.

18.10 loadState

```
chain.loadState()
```

Loads the current head state as a SnapshotManager.

18.10.1 Returns

`SnapshotManager`: The current head state.

18.11 saveState

```
chain.saveState(stateManager)
```

Saves the current head state from a SnapshotManager.

18.11.1 Parameters

1. `stateManager` - `SnapshotManager`: A SnapshotManager to save.

CHAPTER 19

GuardService

`GuardService` ensures that user funds always remain safe. This service watches Ethereum and blocks any invalid withdrawals of a user's assets. If you'd like to understand more about invalid withdrawals, we've provided a detailed article.

CHAPTER 20

DBService

`DBService` handles all interaction with the user's local database. Currently, **all** services talk to ChainService to interact with the database instead of talking with `DBService` directly.

20.1 Backends

`plasma-core` uses key-value store when storing information in the database. `plasma-core` provides serveral different backends for `DBService` depending on the user's preference. `DBService` supports:

1. `EphemDBProvider`, an in-memory database (mostly for testing).
2. `LevelDBProvider`, a wrapper for LevelDB.

20.2 get

```
db.get(key)
```

Returns the value stored at the given key.

20.2.1 Parameters

1. `key` - `String`: The key to query.

20.2.2 Returns

`any`: The value stored at that key.

20.3 set

```
db.set(key, value)
```

Stores a value at the given key.

20.3.1 Parameters

1. `key` - `String`: The key to set.
2. `value` - `any`: The value to store.

20.4 delete

```
db.delete(key)
```

Deletes the value at a given key.

20.4.1 Parameters

1. `key` - `String`: The key to delete.

20.5 exists

```
db.exists(key)
```

Checks if a given key is set.

20.5.1 Parameters

1. `key` - `String`: The key to check.

20.5.2 Returns

`boolean`: `true` if the key exists, `false` otherwise.