@pigi/spec Documentation

Plasma Group

Jun 27, 2019

00: Introduction

Plasma Group is working on a generalized plasma construction that supports significantly more functionality than alternative plasma chains. Generalized plasma chains are complex objects of study. This website documents the entirety of the Plasma Group design, from the inner workings of the client to the specifics of the plasma smart contract.

CHAPTER 1

Introduction

Hello and welcome to Plasma Group's Generalized Plasma Specification! We've been working toward this spec for a long time now and we're excited to be able to share it with the public. We're going to kick the spec off with a high-level overview of our construction and then go into the structure of the spec itself.

1.1 TL;DR

Plasma is basically a way to build blockchains on top of blockchains. Plasma chains are sort of like sidechains, except they're a little less flexible and a lot more secure. We won't go into the details here.

Up until now, people have only been able to build very limited plasma chains that accomplish a few specific goals (like make payments or exchange assets). None of these chains support the sort of "smart contracts" that make Ethereum so useful. In order to build a "plasma application", you'd have to build an entire blockchain from scratch. That's obviously way too much work.

So we set out to design a general purpose chain. It took a lot of work, but we finally arrived at a design we're happy with (and that's what we've specified here!). Now, with this new design, developers can build apps and run them on top of a general purpose plasma chain instead of having to build an entire plasma chain from scratch. It's sort of like the jump of going from Bitcoin to Ethereum.

1.2 Required Background Knowledge

You're going to see a lot of terms and ideas from previous plasma research when you're reading through this specification. Although we've provided references to these terms or ideas wherever possible, we generally try to avoid rehashing what's already been explained well before. As a result, you should be familiar with the general theory behind plasma and with various specific plasma constructions (like Plasma MVP and Plasma Cash) before diving into this spec.

You don't need an extremely deep knowledge of plasma, but you should be comfortable with the idea of deposits, exits, and exit games. If you're not familiar with these concepts yet, we recommend reading through the original Plasma MVP and Plasma Cash posts, as well as the content on LearnPlasma.

Sometimes we have a little trouble remembering that readers don't have quite as much context as we do. If you feel familiar with these concepts but are still confused by some aspect of the spec, please let us know! We've set up a system that makes it easy for anyone to leave review comments in just a few seconds.

1.3 Specification Goals

This specification should be detailed, concise, and contained. Readers should have a very clear idea of the system as a whole without relying on guesswork about how a specific mechanism functions. Similarly, developers should be able to create a compliant implementation simply by reading through the specification.

If at any time you feel we haven't quite achieved that goal, please let us know! We're always open to constructive criticism. If you find something confusing, it's likely that others do too.

1.4 Specification Structure

We've laid out this specification in a very deliberate way that attempts to mirror the layout of the Lightning BOLT specifications.

Our specification is composed of eight primary sections:

- #00: Introduction
- #01: Core Design Components
- #02: Contract Specifications
- #03: Client Specification
- #04: Operator Specification
- #05: Client Architecture
- #06: Operator Architecture
- #07: Predicate Specifications

The first five sections (00 - 04) form the specification of the generalized plasma system itself. These sections explain, in detail, how the system works and what components someone would have to include in a compliant implementation.

The last three sections (05 - 07) describe the architecture for Plasma Group's implementation of the specification. A developer would **NOT** have to use the same architecture in their own implementation, but it may be useful in order to better understand how certain components are supposed to function. Someone who's primarily interested in understanding the system at a high-level could skip these sections.

1.5 Housekeeping

1.5.1 Source

This specification is a living document. You can find the source for this document on the Plasma Group monorepo.

We need your help! At the end of the day, this specification is meant to help others build better plasma chains. If you find anything confusing, please create an issue on GitHub and let us know how we can help. You're also more than welcome to create a pull request if you feel like you can fix an issue yourself. We're usually pretty responsive to issues and PRs.

1.5.2 Versioning

Most pages within this specification will continue to evolve as we develop our implementation. We've decided to use semantic versioning so that it's easy to see and understand the difference between versions of this document.

Our semantic versioning strategy is pretty simple, each version follows the format `major.minor.patch`. Changes to this specification that would make an implementation incompatible with implementations of previous versions are split into different major versions. Changes that simply add functionality but don't break backwards compatibility bump the minor version. Finally, fixes that don't add or remove functionality (layout edits, grammatical fixes, typo fixes) bump the patch version.

CHAPTER 2

Glossary

2.1 Miscellaneous

2.1.1 Merkle Interval Tree

2.1.2 Explicit and Implicit Bounds

2.1.3 Commitment

2.1.4 JSON RPC

2.2 Plasma

2.2.1 Plasma

2.2.2 Plasma MVP

2.2.3 Plasma Cash

2.2.4 Client

2.2.5 Operator

2.2.6 Deposit

2.2.7 Exit

2.2.8 Checkpoint

2.2.9 Challenges

Deprecated State Challenge

Invalid History Challenge

2.3 State System

2.3.1 State Object

2.3.2 Predicate

2.3.3 Predicate Plugin

2.3.4 Transaction

2.3.5 State Update

2.3.6 Range

2.3.7 History Proof

Deposit Proof Elements

Exclusion Proof Elements

State Update Proof Elements

2.3.8 Asset Tree

2.3.9 State Tree

CHAPTER 3

Contributors

- Jinglan Wang
- Karl Floersch
- Ben Jones
- Kelvin Fichter
- Mikerah Quintyne-Collins

CHAPTER 4

Generalized State System

The concept of state and state transitions is relatively universal. However, Layer 2 systems are unusual because they often require that state transitions be efficiently representable on the Layer 1 platform. As a result, existing models of state transitions on plasma are usually quite restrictive.

In order to develop a plasma system that could allow for more general state transitions, we found it important to develop a new system for representing state and state transitions. Our system was particularly designed to make state transitions easily executable on Ethereum.

This page describes our state model and the various terms we use in the rest of this specification. It's important to understand this model in detail before continuing.

4.1 State Objects

4.1.1 State Object Model

The core building block of our model is the "state object". Each state object represents a particular piece of state within the system and is composed of:

1. A globally unique identifier.
2. The address of a predicate contract.
3. Some arbitrary data.

The TypeScript interface for the state object is:

```typescript
interface StateObject {
  id: string
  predicate: string
  data: string
}
```

State objects are unique objects (hence the `id`) that may be **created**, **destroyed**, or **mutated**. The conditions under which a state object may undergo one of these changes, as well as the effects of such a change, are defined in a *predicate*. Each state object specifies a prediciate identifier (`predicate`) that points to the specific predicate that controls ("locks") the state object.

Requirements

- **State objects:**
  - **MUST** have a globally unique identifier.
  - **MUST** specify the 20-byte address of a predicate contract.
  - **MUST specify some additional arbitrary stored data.**
    - **MAY** specify the empty string in place of arbitrary stored data.
    - **SHOULD** specify the zero address to represent a "burned" state.

Rationale

We wanted our generalized plasma construction to be agnostic to the underlying design (Plasma MVP, Plasma Cash, etc.). As a result, it became important to find a very general-purpose way of representing state.

The idea of a "state object", an object that can be created, destroyed, or modified, seemed simple and intuitive. Perhaps more importantly, the "state object" model encapsulates both UTXOs and accounts. Ephemeral UTXO-like systems can be represented by objects which can only be created or destroyed, but not modified. Account-like systems can be represented by objects which can be modified.

The state object therefore allows us to simultaneously represent systems like Plasma MVP *and* Plasma Cash with a single unified notation. We believe this unification will lead to increased collaboration and decreased duplication of work.

4.1.2 Encoding and Decoding

State objects **MUST** be ABI encoded and decoded according to the following structure:

```
[
    id: bytes,
    predicate: address,
    data: bytes
]
```

Rationale

Plasma Group has decided to write its contracts in Solidity instead of in Vyper. We therefore needed an encoding scheme that would be easy to decode within a Solidity contract. Other teams have invested significant resources into developing smart contract libraries for encoding schemes. However, Solidity provides native support for ABI encoding and decoding. In order to minimize effort spent on encoding libraries, we've decided to simply use the native ABI encoding mechanisms.

Test Vectors

Todo: Add test vectors for encoding and decoding.

4.2 Predicates

Predicates are functions that define the ways in which state objects can be mutated. We require that the `id` of a specific state object never change (since it would then be a different state object). However, the `predicate` and `data` can be changed arbitrarily according to the rules defined in the predicate.

4.2.1 Predicate Methods

Predicates can provide one or more methods which take a state object in one state and transform it into another state. For example, a simple "ownership" predicate may define a function that allows the current owner of the object (defined in `object.data`) to specify a new owner.

For simplicity, we require that predicate methods may only allow input and output types that correspond to the primitive types in Solidity.

Rationale

Effectively all blockchain systems provide a model for different "methods" that determine how a given object can be transformed. Bitcoin's UTXO model allows for multiple "spending conditions" under which a UTXO can be consumed. Ethereum's account model allows a contract to specify multiple explicit state-transforming functions. The "method" model generalizes this concept.

We require that methods only use the primitive types available in Solidity so that predicates can easily be executed by treating them as Solidity contracts. Defining new types not understood by Solidity would require the development of a completely new EVM language.

Requirements

- **Predicate methods:**
  - **MUST** be executable within a single transaction to an Ethereum smart contract.
  - **MUST** only use the primitive types in Solidity.

4.2.2 Method Identifiers

Methods within each predicate are given a unique identifier computed as the keccak256 hash of the UTF-8 encoded version of the method's signature.

For any given method:

```
function methodName(Param1Type param1, Param2Type param2) public returns (ReturnType)
```

We get a corresponding signature:

```
methodName(Param1Type, Param2Type, ...)
```

Example

We'll use the SimpleOwnership predicate as an example. State objects locked with the `SimpleOwnership` have an "owner" field stored in `object.data`. `SimpleOwnership` defines a method that allows the current "owner" of a state object to specify a new owner:

```
function send(address _newOwner) public
```

The signature of this method is:

```
send(address)
```

In TypeScript we can compute the method ID as:

```
import { keccak256 } from 'js-sha3'

const methodId = keccak256('send(address)')
```

Rationale

We decided on this scheme for computing method signatures for several reasons.

Other languages, like Solidity and Vyper, define the method ID as the first 4 bytes of the keccak256 hash. One benefit of the 4-byte scheme is that it reduces the total amount of data on-chain. Unfortunately, this requires checking for any hash collisions between function names. For simplicity, therefore, we decided to use the *full* 32-byte hash. The additional required 28 bytes do not seem like a significant enough waste of gas to justify more complex collision-detection logic for predicates.

We also chose this system because `keccak256` hashes are cheaply computable on Ethereum.

Requirements

- **Method ID:**
  - **MUST** be computed as the keccak256 hash of the method signature.

Test Vectors

Todo: Add test vectors for method identifiers.

4.2.3 Predicate API

Predicates **MUST** provide a **Predicate API** that allows a client to interact with the predicate. A Predicate API is composed of an array of **API elements**. Each API element describes a single function, including the function's inputs and outputs. The structure of the API element has been based off of the Ethereum contract ABI specification.

TypeScript interfaces for valid Predicate API objects are provided below. Compare to the Ethereum ABI JSON format to understand similarities and differences.

```
interface PredicateApiInput {
  name: string
  type: string
}

interface PredicateApiOutput {
  type: string
}
```

(continues on next page)

(continued from previous page)

```
interface PredicateApiItem {
  name: string
  inputs: PredicateApiInput[]
  outputs: PredicateApiOutput[]
  constant: boolean
}
```

Example

We're going to describe a valid Predicate API by looking at the SimpleOwnership predicate. `SimpleOwnership` allows one valid state transition whereby the current owner of a state object may sign off on a new owner:

```
function send(address _newOwner) public
```

Note that this is **not** a `constant` method because it will update the state of the predicate.

`SimpleOwnership` also provides a method which returns the current owner:

```
funtion getOwner() public view returns (address)
```

This function **is** a `constant` method because it only reads information and does not change the state of the object.

Putting these together, the API for this predicate is therefore:

```
[
    {
        name: "send",
        constant: false,
        inputs: [
            {
                name: "newOwner",
                type: "address"
            }
        ],
        outputs: []
    },
    {
        name: "getOwner",
        constant: true,
        inputs: [],
        outputs: [
            {
                type: "address"
            }
        ]
    }
]
```

Rationale

Todo: Add rationale for Predicate API.

Requirements

Todo: Add requirements for Predicate API.

4.3 Transactions

4.3.1 Transaction Model

Mutations to state objects are carried out by **transactions**. Transactions specify:

1. The ID of a state object to mutate.
2. The ID of a method to call in the state object's predicate.
3. Parameters to be passed to the object's predicate.

A TypeScript interface for a transaction:

```
interface Transaction {
  objectId: string
  methodId: string
  parameters: string
}
```

A Solidity struct:

```
struct Transaction {
    bytes objectId;
    bytes32 methodId;
    bytes parameters;
}
```

`methodId` corresponds to the identifier *computed* from the *Predicate API* of the referenced object's predicate contract.

Rationale

Todo: Add rationale for transaction model.

Requirements

Todo: Add requirements for transaction model.

4.3.2 Encoding and Decoding

Similarly, transactions **MUST** be ABI encoded and decoded in the form:

```
[ objectId: bytes,  methodId: bytes, parameters: bytes ]``
```

Rationale

Todo: Add rationale for transaction encoding and decoding.

Requirements

Todo: Add requirements for transaction encoding and decoding.

Test Vectors

Todo: Add test vectors for transaction encoding and decoding.

4.3.3 Transaction Hash

Test Vectors

Todo: Add test vectors for the transaction hash.

4.4 State Updates

Todo: Explain state updates at a high level.

4.4.1 State Update Model

Todo: Specify the model for a state update.

4.4.2 Encoding and Decoding

Todo: Specify how to encode and decode state updates.

Rationale

Todo: Add rationale for state update model.

Requirements

Todo: Add requirements for state update model.

4.4.3 State Update Hash

Todo: Explain how to compute state update hash.

Test Vectors

Todo: Add test vectors for computing state update hash.

CHAPTER 5

State Object Ranges

5.1 Background

Our generalized state system introduces the idea of *state objects*, which can be used to control the ownership of assets deposited into the plasma chain. Plasma Cash describes a version of the generalized state system in which each state object represents an indivisible asset of fixed value. This system significantly reduces the total amount of data each user of the plasma client must store.

Arbitrary value payments become difficult to make within Plasma Cash. This is primarily because state objects in Plasma cash may not be split apart or combined. A user may only make payments of a given amount if they are in possession of a set of state objects such that the value of these state objects is *exactly* the payment amount. Compare this, for example, to the simplicity of making payments in Bitcoin-like systems where a UTXO can be arbitrarily broken apart into outputs with different values.

The following diagram shows the basic idea behind transactions in Plasma Cash:

Each of the eight shown transactions can only operate on one of the eight state objects. As a result, they have to be treated as separate objects that a user might have to keep track of.

One method of simplifying payments within such a system is to require that each state object to have an identical small value. Payments of any amount can then be made by sending many state objects simultaneously. In order to support simultaneous transfer of state object, we introduce mechanisms that allow users to reference **ranges** of state objects within transactions. This was the primary advancement of Plasma Cashflow.

Here's a diagram of what this looks like in practice:

As you can see, we can manipulate all eight of the state objects with a single transaction. This means we can imagine the state objects as a single larger object, which reduces total storage requirements.

5.2 Transactions

The transaction format described in our generalized state system specifies that a transaction **MUST** provide a reference to the *state object*, or set of state objects, from which it spends. Conveniently, this allows us to create transactions over **ranges** of state objects.

The specification for a transaction over a range is very simple. Remember that each `objectId` in our system is a unique 32 byte identifier. We therefore require that the `objectIds` field be a 64 byte value. The first 32 bytes of this value represents the start of the transacted range and the last 32 bytes represents the end of the transacted range. Transactions over ranges are, in effect, transactions on each individual state object where `object.id` falls within the specified range.

5.2.1 Rationale

As described in the *background* section above, we need to be able to efficiently transact many state objects simultaneously. By allowing transactions to refer to a set of state objects with a range, it's no longer necessary to submit a transaction for each individual state object.

5.2.2 Requirements

- **The `objectIds` field of every transaction:**
  - **MUST** be a 64 byte value.
  - **MUST** begin with a 32 byte value that represents that start of the transacted range.
  - **MUST** end with a 32 byte value that represents the end of the transacted range.

CHAPTER 6

Merkle Interval Tree

6.1 Background

The Plasma Cashflow construction requires transactions that can efficiently reference ranges of state objects simultaneously. In order to preserve the properties of Plasma Cash, we require a Merkle tree structure that will not allow for the existence of two transactions that reference the same state object.

We provide a construction for such a tree, called a Merkle Interval Tree. This tree effectively commits to values that refer to corresponding ranges given by a `start` and an `end`. The tree provides the property that for any range, there can exist at most **one** leaf node that references the range and has a valid Merkle proof.

6.2 Tree Structure

The Merkle Interval Tree is a binary tree with special structures for leaf nodes and internal nodes.

6.2.1 Leaf Node

The leaf nodes in a Merkle Interval Tree represent a **range** and a value for that range. We describe leaf nodes as a tuple of `(start, end, data)`.

In TypeScript:

```
interface MerkleIndexTreeLeafNode {
  start: number
  end: number
  data: string
}
```

6.2.2 Internal Node

Internal nodes are a tuple of `(index, hash)`.

In TypeScript:

```
interface MerkleIndexTreeInternalNode {
  index: number
  hash: string
}
```

6.3 Tree Generation

A Merkle Index tree is generated from a list of leaf nodes. Merkle Interval Tree generation also requires the use of a hash function. Any hash function is suitable, but the security properties of the hash function will impact the properties of the tree.

An algorithm for generating the tree is described below. All lists used are zero-indexed.

The algorithm takes two inputs, a list of leaf nodes and a hash function.

1. If the list of leaf nodes is empty, return an empty array.
2. Assert that the range described by `start` and `end` of each leaf node does not intersect with the range described by any other leaf node. If any intersecting ranges exist, throw an error.
3. Sort the list of leaf nodes by their `start` value.
4. Store the list of leaf nodes as the first layer of the tree.
5. Generate a corresponding sorted list of internal nodes from the leaf nodes by creating an internal for each leaf node such that:
   a) The `index` of the node is equal to the `start` of the leaf node.
   b) The `hash` of the node is equal to the hash of the concatenation of `start`, `end`, and `data` of the leaf node, in that order.
6. Recursively generate the rest of the tree as follows:
   a) Store the list of internal nodes at the current height of the tree.
   b) If the list of internal nodes has only one element, return.
   c) Pair each node in the list of internal nodes such that any node where `node_index % 2 = 0` is paired with the node at `node_index + 1`. If the node at `node_index + 1` does not exist, pair the node with a new node such that `pair.index = node.index` and `pair.hash = 0`.
   d) Generate a list of parent nodes. For each pair of nodes, create a corresponding parent node such that:
      1. `parent.index = left_child.index` and `parent.hash` is the hash of the concatenation of `left_child.index`, `left_child.hash`, `right_child.index`, `right_child.hash`, in that order.
   e) Repeat this process for the generated list of parent nodes.
7. Return the generated tree.

6.3.1 Pseudocode

A pseudocode version of the above algorithm is given below:

```
def generate_tree(leaf_nodes, hash_function):
    tree = []

    # Empty tree
    if len(leaf_nodes) == 0:
        return tree

    # Leaves intersect
    for leaf in leaf_nodes:
        for other in leaf_nodes:
            if (intersects(leaf, other)):
                raise Exception()

    # Sort the leaves by start value
    leaf_nodes.sort()

    children = []
    for leaf in leaf_nodes:
        children.append({
            'index': leaf.start,
            'hash': hash_function(leaf.start + leaf.end + leaf.data)
        })

def generate_internal_nodes(children, tree):
    if len(children) == 1:
        return tree

    parents = []
    for x in range(0, len(children)):
        if x % 2 == 0:
            left_child = chilren[x]

            # Create an imaginary node if out of bounds
            if x + 1 == len(children):
                right_child = {
                    'index': left_child.index,
                    'hash': 0
                }
            else:
                right_child = children[x + 1]

            parents.append({
                'index': left_child.index,
                'hash': hash_function(left_child.index + left_child.hash + right_
↪child.index + right_child.hash)
            })

    tree.append(parents)
    return generate_internal_nodes(parents, tree)
```

6.4 Merkle Proofs

Our tree generation process allows us to create an efficient **proof** that for a given leaf node and a given Merkle Interval Tree root node such that:

1. The leaf node was contained in the tree that generated the root.
2. The range described by the leaf node intersects with no other ranges described by any other leaf node in the tree.

6.4.1 Proof Generation

Proofs can be generated after the full Merkle tree has been generated as per the algorithm *described above*. Proofs consist of a list of internal nodes within the Merkle tree.

The proof for a given leaf node is computed as follows:

1. If the leaf node is not in the tree, throw an error.
2. Find the internal node that corresponds to the leaf node in the bottom-most level of the tree.
3. Recursively:
   a) If the internal node is the root node, return.
   b) Find the sibling of the node. If no sibling exists, set the sibling to the empty node such that `sibling.index = node.index` and `sibling.hash = 0`.
   c) Insert the sibling into the proof.
   d) Repeat this process with the parent of the node.
4. Return the proof.

Pseudocode

```
def generate_proof(tree, leaf_node):
    leaves = tree[0]
    if leaf_node not in leaves:
        raise Exception()

    leaf_index = leaves.index(leaf_node)
    return find_siblings(tree, 1, leaf_index, [])

def find_siblings(tree, height, child_index, proof):
    if height == len(tree):
        return proof

    proof.append(get_sibling(child_index))
    parent_index = get_parent_index(child_index)
    return find_siblings(tree, height + 1, parent_index, proof)
```

6.4.2 Proof Verification

Verification of Merkle Interval Tree proofs is relatively straightforward. Given a leaf node, the index of that leaf node within the Merkle tree, a proof consisting of a list internal nodes, and the root of the tree:

1. Compute the internal node that corresponds to the leaf node such that `node.index = leaf.start` and `node.hash` is the hash of the concatenation of `leaf.start`, `leaf.end`, and `leaf.data`, in that order.
2. For each element of the proof:
   a) Use the index of the leaf node to determine whether the element is a left or right sibling of the current internal node.
   b) Compute the parent of the two siblings.
   c) Set the current internal node to be the parent.
3. Check if the current internal node is equal to the root node.

Pseudocode

```
def check_proof(leaf_node, leaf_index, proof, root_node, hash_function):
    current_node = {
        'index': leaf_node.start,
        'hash': hash_function(leaf_node.start + leaf_node.end + leaf_node.data)
    }

    for x in range(0, len(proof)):
        sibling = proof[x]
        if is_left_sibling(leaf_index, x):
            current_node = compute_parent(sibling, current_node)
        else:
            current_node = compute_parent(current_node, sibling)

    return current_node == root_node
```

6.5 Tree Diagram

A diagram of the Merkle Interval Tree is provided below. We've highlighted the nodes that one would need to provide to prove inclusion of a given state update.

CHAPTER 7

Double Layer Merkle Tree

7.1 Background

The amount of data a user needs to store in a Plasma Cash chain is linearly proportional to the number of state objects that user "owns" within the chain. Plasma Cashflow, however, relies on the concept of ranges of state objects in order to add a sense of fungibility to Plasma Cash. Users of a Plasma Cashflow chain instead store an amount of data linearly proportional to the number of **ranges** they own, not the number of individual state objects within those ranges.

The fewer ranges a user needs to track, the less data they need to store. It's for this reason that we need some sort of mechanism for defragmenting ranges. A detailed explanation of the defragmentation process is described outside of this page.

The basic idea behind defragmentation is that contiguous ranges can be "merged" into a single larger range. Users can execute atomic swaps with others users in order to trade disconnected ranges for contiguous ones. This process is relatively straightforward when the assets underlying the ranges are identical – trading 1 ETH for 1 ETH carries no risk. However, the process is significantly more complex if the underlying assets are different.

We can ensure that the underlying asset will always be the same by reserving some space for the asset. One way to accomplish this is to modify the ID of a state object with a unique prefix for each asset. For example, the range of state object IDs for ETH might be prefixed with a zero (`(0)(0 - 2^256)`) while the range of IDs for WETH might be prefixed with a one (`(1)(0 - 2^256)`). This scheme makes sure that the ranges next to yours will always have the same underlying asset.

Another way to achieve the same result is simply to have a different Plasma Cashflow chain for each asset. We do so by creating a unique deposit contract for each asset. The prefix for a range becomes the address of the deposit contract that corresponds to the asset.

However, this means that we're simultaneously running a lot of plasma chains. We don't want to have to submit a block root for each individual chain. We get around this by creating a **double-layered** Merkle Interval Tree. This page explains how the double-layered tree is computed.

7.2 State Tree

Todo: Explain the purpose and structure of the state tree.

7.3 Address Tree

Todo: Explain the purpose and structure of the address tree.

7.4 Tree Diagram

We've provided a diagram of the double-layer Merkle Interval Tree below. The diagram explains the relationship between the state updates that form the tree and the tree itself.

CHAPTER 8

JSON RPC

8.1 Description

We require that the clients via JSON RPC. A list of available RPC methods are specified separately.

We chose JSON RPC because the protocol is relatively simple and because it interacts well with TypeScript. Ethereum also uses JSON RPC for its client communication. The structure of our RPC methods are based heavily on the structure of Ethereum's RPC methods to reduce cognitive overhead for client developers.

8.2 Data Structures

8.2.1 JsonRpcRequest

```
interface JsonRpcRequest {
  jsonrpc: '2.0'
  method: string
  params?: any
  id: string | number
}
```

Description

Represents a JSON RPC request.

Fields

1. `jsonrpc` - `string`: **MUST** be exactly '2.0'.

2. `method` - `string`: Name of the method to call.
3. `params?` - `any`: Formatted request parameter object.
4. `id` - `string | number`: Request identifier.

8.2.2 JsonRpcSuccessResponse

```
interface JsonRpcSuccessResponse {
  jsonrpc: '2.0'
  result: any
  id: string | number
}
```

Description

Represents a successful JSON RPC response.

Fields

1. `jsonrpc` - `string`: **MUST** be exactly ‘2.0’.
2. `result` - `any`: Formatted response to the request.
3. `id` - `string | number`: Same identifier as the one given during the request.

8.2.3 JsonRpcError

```
interface JsonRpcError {
  code: number
  message: string
  data: any
}
```

Description

Represents information about a JSON RPC error.

Fields

1. `code` - `number`: Error code ID.
2. `message` - `string`: Additional short error message.
3. `data` - `any`: Additional error information.

8.2.4 JsonRpcErrorResponse

```
interface JsonRpcErrorResponse {
  jsonrpc: '2.0'
  error: JsonRpcError
}
```

Description

Represents a JSON RPC error response to request.

Fields

1. `jsonrpc` - `string`: **MUST** be exactly ‘2.0’.
2. `error` - `JsonRpcError`: RPC error object.

8.2.5 JsonRpcResponse

```
type JsonRpcResponse = JsonRpcSuccessResponse | JsonRpcErrorResponse
```

Description

Either a success response or an error response.

CHAPTER 9

Contracts Overview

9.1 Introduction

The PG contract architecture consists of three modular components which serve as the basis for creating plasma networks. These are:

Commitment contracts - where aggregators submit blocks, and where deposit contracts verify inclusion of State Updates. Deposit contracts - where users deposit money. Enforces the basic plasma exit game to guarantee safety. Predicate contracts - communicate with the deposit contract to extend the basic plasma exit game for particular applications (plapps).

9.1.1 Commitment Contract

The commitment contract defines a single public address aggregatorwhich can submit new blocks, and records the blocks in storage for later use. It also provides two functions, verifyAssetStateRootInclusion and verifyStateUpdateInclusion, which may be used by both the predicate and deposit contracts to authenticate block contents.

9.1.2 Deposit Contract

The deposit contract contains the base plasma logic for keeping funds secure. It allows users to deposit funds into it, and only allows withdrawals after an "exit game" has been passed. It provides a standard interface which predicate contracts may access to enforce their particular exit game details.

9.1.3 Predicate Contracts

Predicate contracts implement the specific logic of an exit game, and by extent define the transition logic for that type of state object. Most importantly, they dictate the rules of deprecation – the conditions by which an exit on the given state can be cancelled. For example, the ownership predicate allows an exit to be deprecated if the exiting owner has made a signature sending to another party.

9.2 Deposit Contract

The deposit contract custodies users' funds and enforces the plasma cash exit game on them. The exit game we have chosen is a modification of the exit games first proposed by Lucidity (https://ethresear.ch/t/luciditys-plasma-cash-easy-and-more-efficient/4029/8) and Dan Robinson (https://ethresear.ch/t/a-simpler-exit-game-for-plasma-cash/4917) to work on ranges of coins (cashflow).

9.2.1 Coin Fork Choice Rule

The simplest way to understand the rules the exit game enforces are as follows: for a given coin, the exitable state is that of its earliest state update which cannot be deprecated.

Thus, there are two conditions which a particular state update on a particular must satisfy to be exited. First, it must have no undeprecated state updates in its history. Second, it must itself not be deprecated. We have split these conditions into two distinct games.

9.2.2 Exits

Exits are the game for the second condition above: they represent a claim that some subrange of a state update is not deprecated. At any time, the predicate contract for that state update is allowed to make a call, deprecateExit, which cancels it. Only the predicate for that state update is authenticated to make the call. An exit points at a particular checkpoint.

9.2.3 Checkpoints

Checkpoints are the game for the first condition above: they represent a claim that all prior state updates must be deprecated for the range being exited. Finalization of a checkpoint means that the contract will strictly enforce this, disallowing any exits on a coin which come from an earlier plasma block in which a checkpoint on that coin is finalized.

9.2.4 Challenges

Creating a checkpoint on a state object requires an inclusion proof for a state update over that object. As such, if a state update is included by the aggregator without a previous state being deprecable, this is a malicious action which forces the user to exit. If the aggregator begins a checkpoint on the malicious state update, it may be blocked by a challenge. A challenge points at the earlier exit and a later intersecting checkpoint. That checkpoint is considered invalid unless the earlier exitobject can be deprecated, removing the challenge.

CHAPTER 10

Deposit Contract

10.1 Description

Deposit contracts are the Ethereum smart contracts into which assets are deposited–custodying the money as it is transacted on plasma and playing out the exit games to resolve the rightful owners of previously deposited assets. As such, it contains the bulk of the logic for the plasma exit games. The things it does not cover are 1) block commitments, and 2), state deprecation, which are handled by calls to the commitment contract and predicate contracts specifically.

10.2 API

10.2.1 Structs

Range

```
struct Range {
    uint256 start;
    uint256 end;
}
```

Description

Represents a range of state objects.

Fields

1. `start` - `uint256`: Start of the range of objects.

2. `end` - `uint256`: End of the range of objects.

StateObject

```
struct StateObject {
    string id;
    address predicate;
    bytes data;
}
```

Description

Represents a state object. Contains the address of the predicate contract and input data to that contract which control the conditions under which the object may be mutated.

Fields

1. `id` - `string`: A globally unique identifier for this state object.
2. `predicateAddress` - `address`: Address of the predicate contract that dictates how the object can be mutated.
3. `data` - `bytes`: Arbitrary state data for the object.

StateUpdate

```
struct StateUpdate {
    Range range;
    StateObject stateObject;
    address plasmaContract;
    uint256 plasmaBlockNumber;
}
```

Description

Represents a state update, which contains the contextual information for how a particular range of state objects was mutated.

Fields

1. `range` - `Range`: Range of state objects that were mutated.
2. `stateObject` - `StateObject`: Resulting state object created by the mutation of the input objects.
3. `plasmaContract` - `address`: Address of the plasma contract in which the update was included.
4. `plasmaBlockNumber` - `uint256`: Plasma block number in which the update occurred.

Checkpoint

```
struct Checkpoint {
    StateUpdate stateUpdate;
    Range subRange;
}
```

Description

Represents a *checkpoint* of a particular state update on which a “checkpoint game” is being or has been played out. Checkpoints which have successfully passed the checkpoint game are considered “finalized”, meaning the plasma contract should ignore all state updates on that range with an older plasma block number.

Fields

1. `stateUpdate` - `StateUpdate`: State update being checkpointed.
2. `subRange` - `Range`: Sub-range of the state update being checkpointed. We include this field because the update may be partially spent.

CheckpointStatus

```
struct CheckpointStatus {
    uint256 challengeableUntil;
    uint256 outstandingChallenges;
}
```

Description

Status of a particular checkpoint attempt.

Fields

1. `challengeableUntil` - `uint256`: Ethereum block number until which the checkpoint can still be challenged.
2. `outstandingChallenges` - `uint256`: Number of outstanding challenges.

Challenge

```
struct Challenge {
    Checkpoint challengedCheckpoint;
    Checkpoint challengingCheckpoint;
}
```

Description

Describes a challenge against a checkpoint. A challenge is a claim that the `challengingCheckpoint` has no valid transactions, meaning that the state update in the `challengedCheckpoint` could never have been reached and thus is invalid.

Fields

1. `challengedCheckpoint` - `Checkpoint`: Checkpoint being challenged.
2. `challengingCheckpoint` - `Checkpoint`: Checkpoint being used to challenge.

10.2.2 Public Variables

COMMITMENT_ADDRESS

```
address constant COMMITMENT_ADDRESS;
```

Description

Address of the commitment contract where block headers for the plasma chain are being published.

Requirements

Deposit contracts **MUST** specify the address of a commitment contract where plasma chain block headers are being published.

Rationale

Deposit contracts handle deposits and exits from a specific plasma chain. Commitment contracts hold the plasma block headers for that plasma chain and therefore make it possible to verify inclusion proofs.

TOKEN_ADDRESS

```
address constant TOKEN_ADDRESS;
```

Description

Address of the ERC-20 token which this deposit contract custodies.

Requirements

- **The deposit contract:**
  - **MUST** only support deposits of a single ERC-20 token.
- **`TOKEN_ADDRESS`:**
  - **MUST** be the address of an ERC-20 token.

Rationale

Each asset type needs to be allocated its own large contiguous "sub-range" within the larger Plasma Cashflow chain. Without these sub-ranges, defragmentation becomes effectively impossible. Although it's possible to achieve this result within a single deposit contract, it's easier to simply require that each asset have its own deposit contract and to allocate a large sub-range to every deposit contract.

CHALLENGE_PERIOD

```
uint256 constant CHALLENGE_PERIOD;
```

Description

Number of Ethereum blocks for which a checkpoint may be challenged.

EXIT_PERIOD

```
uint256 constant EXIT_PERIOD;
```

Description

Number of Ethereum blocks before an exit can be finalized.

---

totalDeposited

```
uint256 public totalDeposited;
```

Description

Total amount deposited into this contract.

---

checkpoints

```
mapping (bytes32 => CheckpointStatus) public checkpoints;
```

Description

Mapping from the ID of a checkpoint to the checkpoint's status.

---

depositedRanges

```
mapping (uint256 => Range) public depositedRanges;
```

Description

Stores the list of ranges that have not been exited as a mapping from the `start` of a range to the full range. Prevents multiple exits from the same range of objects.

---

exitRedeemableAfter

```
mapping (bytes32 => uint256) public exitRedeemableAfter;
```

Description

Mapping from the ID of an exit to the Ethereum block after which the exit can be finalized.

---

challenges

```
mapping (bytes32 => bool) public challenges;
```

Description

Mapping from the ID of a challenge to whether or not the challenge is currently active.

10.2.3 Events

CheckpointStarted

```
event CheckpointStarted(
    Checkpoint checkpoint,
    uint256 challengeableUntil
);
```

Description

Emitted whenever a user attempts to checkpoint a state update.

Fields

1. `checkpoint` - `bytes32`: ID of the checkpoint that was started.
2. `challengeableUntil` - `uint256`: Ethereum block in which the checkpoint was started.

CheckpointChallenged

```
event CheckpointChallenged(
    Challenge challenge
);
```

Description

Emitted whenever an invalid history challenge has been started on a checkpoint.

Fields

1. `challenge` - `Challenge`: The details of the challenge .

CheckpointFinalized

```
event CheckpointFinalized(
    bytes32 checkpoint
);
```

Description

Emitted whenever a checkpoint is finalized.

Fields

1. `checkpoint` - `bytes32`: ID of the checkpoint that was finalized.

ExitStarted

```
event ExitStarted(
    bytes32 exit,
    uint256 redeemableAfter
);
```

Description

Emitted whenever an exit is started.

Fields

1. `exit` - `bytes32`: ID of the exit that was started.
2. `redeembleAfter` - `uint256`: Ethereum block in which the exit will be redeemable.

ExitFinalized

```
event ExitFinalized(
    Checkpoint exit
);
```

Description

Emitted whenever an exit is finalized.

Fields

1. `exit` - `Checkpoint`: The checkpoint that had its exit finalized.

10.2.4 Methods

deposit

```
function deposit(
    address _depositer,
    uint256 _amount,
    StateObject _initialState
) public
```

Description

Allows a user to submit a deposit to the contract. Only allows users to submit deposits for the asset represented by this contract.

Parameters

1. `_depositer` - `address`: the account which has approved the ERC20 deposit.
2. `_amount` - `uint256`: Amount of the asset to deposit.
3. `_initialState` - `StateObject`: Initial state to put the deposited assets into. Can be any valid state object.

Requirements

- **MUST** keep track of the total deposited assets, `totalDeposited`.
- **MUST** transfer the deposited `amount` from the `depositer` to the deposit contract's address.
- **MUST** create a state update with a state object equal to the provided `initialState`.
- **MUST** compute the range of the created state update as `totalDeposited` to `totalDeposited + amount`.
- **MUST** update the total amount deposited after the deposit is handled.
- **MUST** insert the created state update into the `checkpoints` mapping with `challengeableUntil` being the current block number - 1.
- **MUST** emit a `CheckpointFinalized` event for the inserted checkpoint.

Rationale

Depositing is the mechanism which locks an asset into the plasma escrow agreement, allowing it to be transacted off-chain. The `initialState` defines its spending conditions, in the same way that a `StateUpdate` does once further transactions are made. Because deposits are verified on-chain transactions, they can be treated as checkpoints which are unchallengeable.

---

startCheckpoint

```
function startCheckpoint(
    Checkpoint _checkpoint,
    bytes _inclusionProof,
    uint256 _depositedRangeId
) public
```

Description

Starts a checkpoint for a given state update.

Parameters

1. `_checkpoint` - `Checkpoint`: Checkpoint to be initiated.
2. `_inclusionProof` - `bytes`: Proof that the state update was included in the block specified within the update.
3. `_depositedRangeId` - `uint256`: The key in the `depositedRanges` mapping which includes the `subRange` as a subrange.

Requirements

- **MUST** verify the that `checkpoint.stateUpdate` was included with `inclusionProof`.
- **MUST** verify that `subRange` is actually a sub-range of `stateUpdate.range`.
- **MUST** verify that the `subRange` is still exitable with the `depositedRangeId` .
- **MUST** verify that an indentical checkpoint has not already been started.
- **MUST** add the new pending checkpoint to `checkpoints` with `challengeableUntil` equalling the current ethereum `block.number + CHALLENGE_PERIOD` .
- **MUST** emit a `CheckpointStarted` event.

Rationale

Checkpoints are assertions that a certain state update occured/was included, and that it has no intersecting unspent state updates in its history. Because the operator may publish an invalid block, it must undergo a challenge period in which the parties who care about the unspent state update in the history exit it, and use it to challenge the checkpoint.

---

deleteExitOutdated

```
function deleteExitOutdated(
    Checkpoint _olderExitt,
    Checkpoint _newerCheckpoint
) public
```

Description

Deletes an exit by showing that there exists a newer finalized checkpoint. Immediately cancels the exit.

Parameters

1. `_olderExitt` - `Checkpoint`: **‘The exit‘_** to delete.
2. `_newerCheckpoint` - `Checkpoint`: The checkpoint used to challenge.

Requirements

- **MUST** ensure the checkpoint ranges intersect.
- **MUST** ensure that the plasma blocknumber of the `_olderExitt` is less than that of `_newerCheckpoint`.
- **MUST** ensure that the `newerCheckpoint` has no challenges.
- **MUST** ensure that the `newerCheckpoint` is no longer challengeable.
- **MUST** delete the entries in `exitRedeemableAfter`.

Rationale

If a checkpoint game has finalized, the safety property should be that nothing is valid in that range’s previous blocks–”the history has been erased.” However, since there still might be some `StateUpdates` included in the blocks prior, invalid checkpoints can be initiated. This method allows the rightful owner to demonstrate that the initiated `olderCheckpoint` is invalid and must be deleted.

---

challengeCheckpoint

```
function challengeCheckpoint(
    Challenge _challenge
) public
```

Description

Starts a challenge for a checkpoint by pointing to an exit that occurred in an earlier plasma block. Does **not** immediately cancel the checkpoint. Challenge can be blocked if the exit is cancelled.

Parameters

1. `_challenge` - `Challenge`: Challenge to submit.

Requirements

- **MUST** ensure that the checkpoint being used to challenge exists.
- **MUST** ensure that the challenge ranges intersect.
- **MUST** ensure that the checkpoint being used to challenge has an older `plasmaBlockNumber`.
- **MUST** ensure that an identical challenge is not already underway.

- **MUST** ensure that the current ethereum block is not greater than the `challengeableUntil` block for the checkpoint being challenged.
- **MUST** increment the `outstandingChallenges` for the challenged checkpoint.
- **MUST** set the `challenges` mapping for the `challengeId` to true.

Rationale

If the operator includes an invalid `StateUpdate` (i.e. there is not a deprecation for the last valid `StateUpdate` on an intersecting range), they may checkpoint it and attempt a malicious exit. To prevent this, the valid owner must checkpoint their unspent state, exit it, and create a challenge on the invalid checkpoint.

---

removeChallenge

```
function removeChallenge(
    Challenge _challenge
) public
```

Description

Decrements the number of outstanding challenges on a checkpoint by showing that one of its challenges has been blocked.

Parameters

1. `_challenge` - `Challenge`: **‘The challenge‘_** that was blocked.

Requirements

- **MUST** check that the challenge was not already removed.
- **MUST** check that the challenging exit has since been removed.
- **MUST** remove the challenge if above conditions are met.
- **MUST** decrement the challenged checkpoint’s `outstandingChallenges` if the above conditions are met.

Rationale

Anyone can exit a prior state which was since spent and use it to challenge despite it being deprecated. To remove this invalid challenge, the challenged checkpointer may demonstrate the exit is deprecated, deleting it, and then call this method to remove the challenge.

---

startExit

```
function startExit(Checkpoint _checkpoint) public
```

Description

Allows the predicate contract to start an exit from a checkpoint. Checkpoint may be pending or finalized.

Parameters

1. `_checkpoint` - `Checkpoint`: The checkpoint from which to exit.

Requirements

- **MUST** ensure the checkpoint exists.
- **MUST** ensure that the `msg.sender` is the `_checkpoint.stateUpdate.predicateAddress` to authenticate the exit's initiation.
- **MUST** ensure an exit on the checkpoint is not already underway.
- **MUST** set the exit's `redeemableAfter` status to the current Ethereum `block.number + LOCKUP_PERIOD`.
- **MUST** emit an `exitStarted` event.

Rationale

For a user to redeem state from the plasma chain onto the main chain, they must checkpoint it and respond to all challenges on the checkpoint, and await a `LOCKUP_PERIOD` to demonstrate that the checkpointed subrange has not been deprecated. This is the method which starts the latter process on a given checkpoint.

---

deprecateExit

```
function deprecateExit(
    Checkpoint _checkpoint
) public
```

Description

Allows the predicate address to cancel an exit which it determines is deprecated.

Parameters

1. `_checkpoint` - `Checkpoint`: The checkpoint referenced by the exit.

Requirements

- **MUST** ensure the `msg.sender` is the `_checkpoint.stateUpdate.predicateAddress` to ensure the deprecation is authenticated.
- **MUST** delete the exit from `exitRedeemableAfter` at the `checkpointId` .

Rationale

If a transaction exists spending from a checkpoint, the checkpoint may still be valid, but an exit on it is not. This method allows the predicate to remove the exit if it has determined it to be outdated.

---

finalizeExit

```
function finalizeExit(Checkpoint _exit, uint256 _depositedRangeId) public
```

Description

Finalizes an exit that has passed its exit period and has not been successfully challenged.

Parameters

1. `_exit` - `Checkpoint`: The checkpoint on which the exit is not finalizable.
2. `_depositedRangeId` - `uint256`: the entry in `depositedRanges` demonstrating the range is not yet exited.

Requirements

- **MUST** ensure that the exit finalization is authenticated from the predicate by `msg.sender == _exit.stateUpdate.state.predicateAddress`.
- **MUST** ensure that the checkpoint is finalized (current Ethereum block exceeds `checkpoint.challengeableUntil`).
- **MUST** ensure that the checkpoint's `outstandingChallenges` is 0.
- **MUST** ensure that the exit is finalized (current Ethereum block exceeds `redeemablAfter`).
- **MUST** ensure that the checkpoint is on a subrange of the currently exitable ranges via `depositedRangeId`.
- **MUST** make an ERC20 transfer of the `end - start` amount to the predicate address.
- **MUST** delete the exit.
- **MUST** remove the exited range by updating the `depositedRanges` mapping.
- **MUST** delete the checkpoint.
- **MUST** emit an `exitFinalized` event.

Rationale

Exit finalization is the step which actually allows the assets locked in plasma to be used on the main chain again. Finalization requires that the exit and checkpoint games have completed successfully.

CHAPTER 11

Commitment Contract

11.1 Description

Each plasma chain **MUST** have at least one **commitment contract**. Commitment contracts hold the block headers for the plasma chain. Whenever the operator creates a new plasma block, they **MUST** publish this block to the commitment contract.

11.2 API

11.2.1 Events

BlockSubmitted

```
event BlockSubmitted(
    uint256 _number,
    bytes _header
);
```

Description

Emitted whenever a new block root has been published.

Fields

1. `_number` - `uint256`: Block number that was published.
2. `_header` - `bytes`: Header for that block.

Rationale

Users need to know whenever a new block has been published so that they can stay in sync with the operator.

11.2.2 Public Variables

currentBlock

```
uint256 public currentBlock;
```

Description

Block number of the most recently published plasma block.

Rationale

Users need to know the current plasma block for various operations. Contract also needs to keep track of this so it knows what block is being published when `submitBlock` is called.

---

blocks

```
mapping (uint256 => bytes) public blocks;
```

Description

Mapping from block number to block header.

Rationale

It's often important to be able to pull a specific block header given a block number. This is necessary, for example, when verifying **'inclusion proofs'_**.

Other implementations often represent this mapping as `uint256 -> bytes32` under the assumption that the block header will always be a `bytes32` Merkle tree root. We instead represent the mapping as `uint256 -> bytes` for more flexibility in the structure of the block root.

11.2.3 Methods

submitBlock

```
function submitBlock(bytes _header) public
```

Description

Allows a user to submit a block with the given header.

Parameters

1. `_header` - `bytes`: Block header to publish.

Rationale

It's obviously necessary to expose some functionality that allows a user to submit a block header. However, the rationale around authentication logic is more interesting here.

Authentication in our original construction was handled by checking that msg.sender was the operator. This works well in a single-operator construction, but it doesn't work if we wanted some more complex system. In order to solve this problem, we initinally wanted to add a `witness: bytes` parameter to the method which could then be used to authenticate the submitted header. Fortunately, we stumbled on an even better solution.

Conveniently, if a contract calls another contract, then msg.sender within that second contract will be the address of the first contract. We can therefore outsource verification of a given block to some external contract and simply check that `msg.sender` is that contract.

Requirements

- **SHOULD** authenticate the block header in some manner.
- **MUST** increment `currentBlock` by one.
- **MUST** store the block header in `blocks` at `currentBlock`.
- **MUST** emit a `BlockSubmitted` event.

CHAPTER 12

Predicate Contracts

12.1 Description

Predicate contracts define the rules for particular state objects' exit game. The most fundamental thing they define is the deprecation logic, which informs the plasma contract that an exit on some state is invalid because it is outdated. Usually, this logic revolves around proving to the predicate that some transaction has invalidated a previous exit. Because the predicate contract is a stateful main-chain contract, more advanced predicates can also define custom exit logic which must be evaluated before any state transitions are approved by the predicate. Thus, predicates can be used as fully customized extensions to the base plasma cash exit game.

12.2 Deprecation

The foremost thing a predicate accomplishes is to enable the deprecation of its state, so that a new state included by the operator becomes valid. This enables state transitions to occur. Because the deposit contract only allows an exit's `predicateAddress` to call `deprecateExit`, the predicate must enable some function which leads to a subcall on `depositContract.deprecateExit`.

12.3 Exit Initiation

Just like the `msg.sender` check by the deposit contract for deprecations, only the predicate is allowed to begin an exit for a given checkpoint. Thus, the predicate must provide some method which leads to a subcall on `depositContract.beginExit`. This enables the predicate to prevent unwanted exits, e.g. by only allowing the current owner to start an exit.

12.4 Exit Finalization

If the predicate contract has custom/stateful exit logic, it may not know who to send money to even after the standard plasma exit period has elapsed. Thus, we require that the `msg.sender == predicateAddress` for a given exit to be finalized and funds to be released. This requires a call by the predicate to the `depositContract.finalizeExit`.

These are the only real requirements for a predicate contract to be compatible with the deposit contract spec. However, it lacks a useful abstraction usually made by blockchains and plasma implementations: the notion of transactions and state transitions. In the next section, we define a standard predicate base which is used throughout the rest of the spec, that treats deprecations, exits, etc. in terms of transactions and state transitions.

CHAPTER 13

Transaction-based Predicate Standard

13.1 Description

Most developers are more familiar with a transaction-based model of cryptocurrencies, where state transitions as the result of transactions form the basis for computation. Our client and operator implementations also work this way–it's really a great model! The transaction-based predicate standard helps us to do this cleanly, by providing a wrapper which interprets transactions in the context of deprecation and disputes.

In essence, the way to enable this is: if a transaction from one state has been authenticated (e.g. signed by its single owner or signed by the multiple participants of a multisig), it must be deprecable using that transaction.

13.2 Transaction Execution

13.2.1 verifyTransaction

```
function verifyTransaction(
    StateUpdate _preState,
    Transaction _transaction,
    bytes _witness,
    StateUpdate _postState
) public
```

Description

The main thing that must be defined for a state transition model is this `verifyTransaction` function which accepts a `preState` state update, and verifies against a `transaction` and `witness` that a given `postState` is correct.

Parameters

1. `_preState` - `StateUpdate`: the state update which the transaction is being applied on.
2. `_transaction` - `Transaction`: The transaction being applied. Follows the standard format as outlined in the transaction generation page in Section #03.
3. `_witness` - `bytes`: Additional witness data which authenticates the transaction validity, e.g. a signature. Defined on a per-predicate basis.
4. `_postState` - `StateUpdate`: the output of the transaction to be verified.

Requirements

- Predicates **MUST** define a custom `_witness` struct for their particular type of state.
- Predicates **MUST** disallow state transitions which pass verification without some interested party's consent, e.g. the owner's signature

13.3 Deprecation

13.3.1 proveDeprecation

```
function proveExitDeprecation(
    Checkpoint _deprecatedExit,
    Transaction _transaction,
    bytes _witness,
    StateUpdate _postState
) public
```

Description

If a state transition away from a given state update exists, then it is not valid to exit that state–it should be deprecated! This function allows a user to demonstrate this to the predicate so that it may cancel an exit

Parameters

1. `_deprecatedExit` - `Checkpoint`: the deprecated checkpoint being exited.
2. `_transaction` - `Transaction`: The transaction which deprecates the exit. Follows the standard format as outlined in the transaction generation page in Section #03.
3. `_witness` - `bytes`: Additional witness data which authenticates the transaction validity, e.g. a signature. Defined on a per-predicate basis.
4. `_postState` - `StateUpdate`: the output of the transaction to be verified.

Requirements

- **MUST** check that the transaction is valid with a call to `verifyTransaction(_deprecatedExit.stateUpdate, _transaction, _witness, _postState`.
- **MUST** check that the `_postState.range` intersects the `_deprecatedExit.subrange`

- **MUST** call `deprecateExit(_deprecatedExit)` on the `_deprecatedExit.stateUpdate.state.predicateAddress`.

CHAPTER 14

Limbo Exit Predicate Standard

14.1 Explanation

An important attack on plasma systems is for the operator to start withholding blocks after they have been sent a transaction. For instance, if Alice signs off agreeing to send to Bob, but the operator does not reveal the corresponding Bob ownership `StateUpdate`, then there is insufficient information to exit because Alice's exit can be deprecated by her signature, but Bob doesn't have an inclusion proof to exit.

One solution is *confirmation signatures*, in which Alice does not create a signature until she sees inclusion of the Bob ownership state. However, this has bad UX properties–for one, Alice has to wait until the next block before she can sign, which means she cannot simply send a signature and go offline. Further, it requires an out-of-protocol authentication process: how does the operator know to include the Bob ownership before Alice has signed?

Limbo exits are a better way to solve this problem. They allow Alice to sign a transaction before the block is submitted, go offline, and have Bob receive the coin at a later time, without reducing safety or extending the exit period.

Intuitively, limbo exits work by allowing Alice to exit her ownership state *on Bob's behalf*. A limbo-compatible ownership predicate allows Alice to exit her ownership state "to Bob." Once she does this, the `finalizeExit` call will send the money to Bob instead of Alice. So, if Alice agrees to a limbo exit, it cannot be deprecated by her spend to Bob.

There are two ways to deprecate a limbo exit. One is to show a conflicting spend other than the one the limbo exit claims to be valid: for example, if Alice limbo exits to Bob, this can be deprecated by providing an alternate spend, e.g. from Alice to Carol. The other way is to show that the limbo "target" is itself deprecated: for example, if Alice sends to Bob, and Bob sends to Carol, a limbo exit from Alice to Bob may be cancelled by showing Bob's transaction to Carol.

Note that this all means limbo exits have a stronger notion of state than the deposit contract requires–instead of just dealing with deprecation, they must have a notion of transactions and state transitions. Luckily, this is just generally a great idea–Plasma Group will support transaction and limbo functionality across all predicates we create. This page will document the standard interface which predicates supporting limbo functionality must follow, so that it doesn't have to be reecreated for different predicates.

14.2 API

14.2.1 Structs

LimboStatus

```
struct LimboStatus {
    bytes32 targetId;
    bool wasReturned;
}
```

Description

Represents the status of a limbo exit.

Fields

1. `targetId` - `bytes32`: Hash of the target `StateUpdate` being limbo exited "to".
2. `wasReturned` - `bool`: Whether the state being limbo exited to cooperatively agreed to return the limbo exit back to the limbo source.

14.2.2 Public Variables

limboTargets

```
mapping (bytes32 => LimboStatus) public limboExits;
```

Description

This mapping maps the limbo exit IDs to their current status.

Rationale

This is the storage on which limbo exit logic is based. If Alice is limbo exiting to Bob, we record the Bob ownership in `targetId` field, and if Bob cooperatively decides to return the outcome of the exit to Alice, we store that in `wasReturned`.

14.2.3 Events

CheckpointStarted

```
event LimboTargeted(
    Checkpoint limboExitSource,
    StateUpdate limboExitTarget
);
```

Description

Emitted whenever a deposit contract exit is targeted as a

Fields

1. `limboExitSource` - `Checkpoint`: The exit being converted into a limbo exit.
2. `limboExitTarget` - `StateUpdpate`: The "target" of a limbo exit. A transaction resulting in the `limboExitTarget` cannot be used to deprecate `limboExitSource` once this event has been emitted.

---

limboExitReturned

```
event limboExitReturned(
    Checkpoint limboExitSource,
);
```

Description

Emitted whenever a deposit contract exit is targeted as a

Fields

1. `limboExitSource` - `Checkpoint`: The exit returned by the limbo target to the source state.

14.2.4 Methods

targetLimboExit

targetLimboExit(originCheckpoint, transaction, target)

```
function targetLimboExit(
    Checkpoint _sourceExit,
    Transaction _transaction,
    bytes _witness
    StateUpdate _limboTarget
) public
```

Description

Allows a user to convert a normal exit into a limbo exit by "targeting" a transaction which they made but cannot prove inclusion of.

Parameters

1. `_sourceExit` - `Checkpoint`: the exit being converted into a limbo exit
2. `_transaction` - `Transaction`: The transaction from the `_sourceExit.stateUpdate`.
3. `_witness` - `bytes`: the witness which proves the transaction validity.
4. `_limboTarget` - `StateUpdate`: the output of the transaction being verified.

Requirements

- **MUST** ensure that the transaction is valid by calling `verifyTransaction(_sourceExit.stateUpdate, _transaction, _witness, _limboTarget`.
- **MUST** ensure the `_limboTarget.range` is a subrange of the `_sourceExit.subRange`.
- **MUST** ensure the `_sourceExit` has not already been made a limbo exit.
- **MUST** call `onTargetedForLimboExit(_sourceExit, _target)` on the `_target` predicate.
- **MUST** set the `limboTargets` mapping with a key of the ID of `_sourceExit` and value of `hash(_limboTarget)`
- **MUST** emit a `LimboTargeted` event.

Justification

This is the base function which converts a regular exit into a limbo exit. Both the source and target predicates must support the limbo interface outlined here for it to work. For example, if Alice limbo exits to a Bob and Carol multisig, she exits her ownership, then limbo targets the mutisig with her transaction. This is because the ownership `targetLimboExit` method subcalls the `onTargetedForLimboExit` of the mutisig.

Functions

onTargetedForLimboExit

```
function onTargetedForLimboExit(
    Checkpoint _sourceExit,
    StateUpdate _limboTarget
) public
```

Description

Hook allowing for the target predicate to initiate any custom logic needed for stateful limbo exits.

Parameters

1. `_sourceExit` - `Checkpoint`: the exit being converted into a limbo exit
2. `_limboTarget` - `StateUpdate`: the output of the transaction being verified.

Requirements

N/A

Justification

This method will simply return true for basic predicates like ownership or multisigs, but allows for more complex stateful exit subgames to be initiated if they need to happen during limbo exits.

proveDeprecation

```
function proveExitDeprecation(
    Checkpoint _deprecatedExit,
    Transaction _transaction,
    bytes _witness,
    StateUpdate _postState
) public
```

Description

This function serves the same purpose as regular state transition predicates, on the condition that the `_deprecatedExit` is not a limbo exit.

Parameters

1. `_deprecatedExit` - `Checkpoint`: the deprecated checkpoint being exited.
2. `_transaction` - `Transaction`: The transaction which deprecates the exit. Follows the standard format as outlined in the transaction generation page in Secion #03.
3. `_witness` - `bytes`: Additional witness data which authenticates the transaction validity, e.g. a signature. Defined on a per-predicate basis.
4. `_postState` - `StateUpdate`: the output of the transaction to be verified.

Requirements

- **MUST** ensure that the `_deprecatedExit` is not a limbo exit by checking the `limboTargets` mapping.
- **MUST** check that the transaction is valid with a call to `verifyTransaction(_deprecatedExit.stateUpdate, _transaction, _witness, _postState` on the source predicate (i.e. this predicate itself).
- **MUST** check that the `_postState.range` intersects the `_deprecatedExit.subrange`
- **MUST** call `deprecateExit(_deprecatedExit)` on the `_deprecatedExit.stateUpdate.plasmaContractAddress`.

Justification

If the exit is not a limbo exit, deprecation may occur normally, by proving an intersecting transaction spending the exit.

proveTargetDeprecation

```
function proveTargetDeprecation(
    Checkpoint _limboSource,
    StateUpdate _limboTarget
    Transaction _transaction,
    bytes _witness,
    StateUpdate _postState
) public
```

Description

This function allows a limbo exit to be cancelled if the `target` has been spent.

Parameters

1. `_limboSource` - `Checkpoint`: the limbo exit whose target state update is deprecable.
2. `_limboTarget` - `StateUpdate` the target of the limbo exit which is deprecable.
3. `_transaction` - `Transaction`: The transaction which spends from the `_limboTarget`
4. `_witness` - `bytes`: Additional witness data which authenticates the transaction validity.
5. `_postState` - `StateUpdate`: the output of the transaction on the target.

Requirements

- **MUST** ensure that the `_limboSource` is indeed a limbo exit with the `hash(_limboTarget)` in its `limboTargets` value.
- **MUST** check that the transaction is valid with a call to the **target** predicate's `verifyTransaction(_deprecatedExit.stateUpdate, _transaction, _witness, _postState`.
- **MUST** check that the `_postState.range` intersects the `_limboTarget.range`.
- **MUST** call `deprecateExit(_limboSource)` on the `_limboSource.stateUpdate.plasmaContractAddress`.
- **MUST** clear the limbo exit from the `limboTargets` mapping.

Justification

An example usage of this would be: if Alice->Bob->Carol, and Alice limbo exits with Bob ownership as the target, this function will be used to cancel the exit by showing Bob->Carol.

proveSourceDoubleSpend

```
function proveSourceDoubleSpend(
    Checkpoint _limboSource,
    StateUpdate _limboTarget
    Transaction _conflictingTransaction,
    bytes _conflictingWitness,
    StateUpdate _conflictingPostState
) public
```

Description

This function allows a limbo exit which has an alternate transaction spending from the source to be deprecated.

Parameters

1. `_limboSource` - `Checkpoint`: the limbo exit which has a double spend which conflicting its target.
2. `_limboTarget` - `StateUpdate` the target of the limbo exit which has a conflicting spend
3. `_conflictingTransaction` - `Transaction`: The transaction which spends from the `_limboTarget`
4. `_conflictingWitness` - `bytes`: Additional witness data which authenticates the transaction validity.
5. `_conflictingPostState` - `StateUpdate`: the output of the transaction on the source which has a different `state` than the target.

Requirements

- **MUST** ensure that the `_limboSource` is indeed a limbo exit with the `hash(_limboTarget)` in its `limboTargets` value.
- **MUST** check that the transaction is valid with a call to the **source** predicate's `verifyTransaction(_deprecatedExit.stateUpdate, _transaction, _witness, _postState`.
- **MUST** check that the `_postState.range` intersects the `_limboTarget.range`.
- **MUST** check that the `_postState.state` is not equal to the `_conflictingPostState.state`.
- **MUST** call `deprecateExit(_limboSource)` on the `_limboSource.stateUpdate.plasmaContractAddress`.
- **MUST** clear the limbo exit from the `limboTargets` mapping.

Justification

An example usage of this would be: if Alice->Bob->Carol, and Alice limbo exits with Mallory ownership as the target, this function will be used to cancel the exit by showing Alice->Bob–a double spend.

returnLimboExit

```
function returnLimboExit(
    Checkpoint _limboSource,
    StateUpdate _limboTarget
    bytes _witness
) public
```

Description

This function allows the source state of a limbo exit to agree to give the money back to the source state of the limbo exit.

Parameters

1. `_limboSource` - `Checkpoint`: the limbo exit which is being returned
2. `_limboTarget` - `StateUpdate` the target of the limbo exit which has a conflicting spend
3. `_witness` - `bytes`: Arbitrary witness data used by the target predicate to authenticate the return. Not necessarily the same as a transaction witness.

Requirements

- **MUST** ensure that the `_limboSource` is indeed a limbo exit with the `hash(_limboTarget)` in its status.
- **MUST** ensure that the target state is allowing the return by calling the target predicate's `canReturnLimboExit(_limboSource, _limboTarget, _witness)`
- **MUST** set `wasReturned` to true for the limbo exit's status.
- **MUST** emit a `limboExitReturned` event.

Justification

If Alice sends a transaction to Bob and then observes block withholding, she must limbo exit with Bob as the target. However, because of the `proveSourceDoubleSpend` method, Bob cannot guarantee until the exit period has passed that Alice will not sign a conflicting message and deprecate the exit. Thus, we want Bob to be able to return the exit to Alice, perhaps conditionally on some payment which he can validate without waaaiting a full exit period

canReturnLimboExit

```
function canReturnLimboExit(
    Checkpoint _limboSource,
    StateUpdate _limboTarget
    bytes _witness
) public returns (bool)
```

Description

This function allows the target state of a limbo exit to authenticate a guaranteed return to the source as described above.

Parameters

1. `_limboSource` - `Checkpoint`: the limbo exit which requesting to be returned.
2. `_limboTarget` - `StateUpdate` the target of the limbo exit which has a conflicting spend
3. `_witness` - `bytes`: Arbitrary witness data used by the target predicate to authenticate the return. Not necessarily the same as a transaction witness.

Requirements

- **MUST** handle some sort of authentication which provides the target state a guarantee that it will not be returnable without permission.

Justification

See justification for `returnLimboExit` above, which explains this function.

finalizeExit

```
function finalizeExit(
    Checkpoint _exit
) public
```

Description

Finalizes an exit which is either not a limbo exit or has been returned

Parameters

1. `_exit` - `Checkpoint`: the exit being finalized.

Requirements

- **MUST check that the exit is either:**
  - **not** a limbo exit, or
  - is a limbo exit which `wasReturned`.
- **MUST** call `finalizeExit` on the deposit contract.
- **MUST** handle the resulting ERC20 transfer of the exit amount in some way.

Justification

If an exit is not a limbo exit or was returned by the target state, we execute a normal exit procedure for the exited `Checkpoint`.

finalizeLimboExit

```
function finalizeExit(
    Checkpoint _exit,
    StateUpdte _target
) public
```

Description

Finalizes a successful, unreturned limbo exit.

Parameters

1. `_exit` - `Checkpoint`: the limbo exit being finalized.
2. `_target` - `StateUpdate`: the target of the limbo exit.

Requirements

- **MUST** ensure that the `_limboSource` is indeed a limbo exit with the `hash(_limboTarget)` in its status.
- **MUST** ensure that the limbo exit was **not** returned.
- **MUST** call `finalizeExit` on the deposit contract.
- **MUST** transfer ALL ERC20 funds from the deposit contract to the target predicate.
- **MUST** call `onFinalizeTargetedExit` on the target predicate.

Justification

If an exit is a limbo exit which was not returned, the source predicate finalizes the exit, sends it to the target, and allows the target to handle the receipt.

onFinalizeTargetedExit

```
function onFinalizeTargetedExit(
    Checkpoint _exit,
    StateUpdate _target
) public
```

Description

Logic for the target of a limbo exit to handle the exit's finalization.

Parameters

1. `_exit` - `Checkpoint`: the limbo exit being finalized.
2. `_target` - `StateUpdate`: the target of the limbo exit.

Requirements

- **MUST** handle the money received from the exit as it pertains to this target.

Justification

The target of a limbo exit must be able to handle the money arbitrarily, this logic is called by the source predicate on the target to perform that logic.

CHAPTER 15

Introduction

Todo: Write introduction for client.

CHAPTER 16

Deposit Generation

Users will usually first interact with a plasma chain by depositing their assets into a plasma deposit contract. Deposits usually consist of a single Ethereum transaction that locks some assets into the depoist contract. This page describes how a client can make a deposit and start using a plasma chain.

Plasma chain transactions transform the state of a range of state objects. The state of each range at a moment in time is described by a state object. Each state object specifies the address of a predicate contract and some additional arbitrary data which are used in tandem to manage ownership of an asset.

Users submit deposit transactions to a plasma deposit contract. Each deposit contract exposes a method `deposit`:

```
function deposit(uint256 _amount, StateObject _state) public payable
```

Where `StateObject` is the following struct:

```
struct StateObject {
    address predicate;
    bytes data;
}
```

`deposit` requires that users specify the `_amount` the asset being deposited and an **initial state object**, `_state`, that controls ownership of the asset. For example, users might use the SimpleOwnership predicate to control their asset.

CHAPTER 17

Event Handling

17.1 Checkpoint Events

Todo: Explain how to handle checkpoint events.

17.2 Exit Game Events

Todo: Explain how to handle exit game events.

CHAPTER 18

Transaction Generation

Once a user has submitted a deposit, they're ready to start making transactions. It's important that transactions are relatively standardized so that transaction generation is as simple as possible. This page describes a standard transaction format and generation process that clients **MUST** adhere to.

18.1 Transaction Format

From the perspective of each predicate, a transaction just consists of an arbitrary string of bytes. Each predicate could parse these bytes in a unique way and therefore define its own transaction format. However, clients should be able to correctly generate a transaction for any given predicate. As a result, we've developed a standard transaction format that simplifies the transaction generation process.

The interface for a `Transaction` object looks like this:

```
interface Transaction {
  plasmaContract: string
  start: number
  end: number
  methodId: string
  parameters: string
}
```

Where the components of this interface are:

1. `plasmaContract` - `string`: The address of the specific plasma deposit contract which identifies the asset being transferred. This is somewhat equivalent to Ethereum's chain ID transaction parameter.
2. `start` - `number`: Start of the range being transacted.
3. `end` - `number`: End of the range being transacted.
4. `methodId` - `string`: A unique method identifier that tells a given predicate what type of state transition a user is trying to execute. This is necessary because a predicate may define multiple ways in which a state object can be mutated. `methodId` **should** be computed as the keccak256 hash of the method's signature, as given by the Predicate API.

5. `parameters` - `string`: Input parameters to be sent to the predicate along with `method` to compute the state transiton. Must be ABI encoded according to the Predicate API. This is similar to the transaction input value encoding in Ethereum.

18.1.1 Transaction Encoding and Decoding

Plasma transactions **must** be ABI encoded or decoded according to the following schema:

```
{
    plasmaContract: address,
    start: uint256,
    end: uint256,
    methodId: bytes32,
    parameters: bytes
}
```

18.2 Sending Transactions

The client **SHOULD** verify the history of the range being transacted before sending the transaction to the operator. Doing so will confirm that no **'invalid transactions'_** have been maliciously inserted into the blockchain by the operator between the block in which the user received a state update and the latest block. Otherwise the client may have to start **'limbo exit'_**, which is more costly than a standard exit.

Transactions can be submitted to a node via the sendTransaction RPC method. If the node that receives this request is not the operator, then it will forward the transaction to the operator on the requester's behalf.

18.3 Example: SimpleOwnership Predicate

We're going to look at the whole process for generating a valid transaction to interact with some state objects locked by the SimpleOwnership predicate. This example will explain how a client can use the Predicate API to generate a valid state-changing transaction. In this case, we'll generate a transaction that changes the ownership of the objects. We'll then look at the process of encoding the transaction and sending it to the operator.

First, let's pick some arbitary values for `plasmaContract`, `start`, and `end`. Users will know these values in advance, so we don't really need to explain the process of getting them in the first place. Let's say that the `plasmaContract` of the `SimpleOwnership` predicate is `0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c` and we want to send the range `(0, 100)`.

Now we just need to figure out our values for `methodId` and `parameters`. We're going to use the Predicate API for SimpleOwnership in order to generate these values. Users can get this API from a variety of places, but it's likely that most wallet software will come with a hard-coded API. Once we have the API, we know that `send` looks like this:

```
{
    name: "send",
    constant: false,
    inputs: [
        {
            name: "newOwner",
            type: "address"
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    outputs: []
}
```

This is already enough information to generate `methodId` and `parameters`. As we previously described, `methodId` is generated by taking the keccak256 hash of the method's signature. In this case:

```
const methodId = keccak256('send(bytes)')
```

Now let's generate `parameters`. Our only parameter to `send` is `newOwner`. We're going to send to a random address, `0xd98165d91efb90ecef0ddf089ce06a06f6251372`. We need to ABI encode this address:

```
const newOwner = '0xd98165d91efb90ecef0ddf089ce06a06f6251372'
const parameters = abi.encode(['address'], newOwner)
```

This is all we need to generate the transaction:

```
const transaction = abi.encode([
  'address',
  'uint256',
  'uint256',
  'bytes32',
  'bytes'
], [
  plasmaContract,
  start,
  end,
  methodId,
  parameters
])
```

Finally, we need to generate a valid *witness* for this transaction. `SimpleOwnership` requires a signature from the previous owner over the whole encoded transaction (except, of course, the signature itself) as a witness:

```
const key = '0x...'
const witness = sign(transaction, key)
```

We now have everything we need to send this transaction off to the operator!

CHAPTER 19

History Proofs

19.1 Introduction

In every plasma block, a range of state objects can either be deposited, transacted, or not transacted. Whenever clients want to verify a transaction on a specific range, they need to verify the entire "history" of what happened to the range between the block in which it was first deposited and the block in which the transaction occurred.

For example, let's imagine that a range `(0, 100)` was deposited in block 1, not transacted in block 2, and then transacted in block 3. The history proof for a transaction in block 4 would contain the deposit, a proof that the range wasn't transacted in block 2, and a proof that the range was transacted in block 3.

This page describes the basic components of the history proof. Details about generating a history proof and verifying a history proof are described later.

19.2 Proof Elements

History proofs consist of a series of "proof elements", which correspond to some action that took place within a specific plasma block. A proof element can be a **Deposit Proof Element**, **Exclusion Proof Element**, or a **State Update Proof Element**. Each element type conveys different information and needs to be handled differently.

19.2.1 Deposit Proof Elements

Deposit Proof Elements consist of a deposit ID. Deposit IDs correspond to a deposit on Ethereum. Deposits on Ethereum contain a state update, which a client can query and insert into their local state.

19.2.2 Exclusion Proof Elements

Exclusion Proof Elements consist of a single state update and an inclusion proof for that state update. They prove that a specific range was **not** transacted during a given block, but they **do not** prove that the given state update is valid.

These proof elements take advantage of the fact that items of our Merkle Interval Tree have both an explicit range and an implicit range. A valid inclusion proof for an item in the tree also proves that there aren't any valid items that intersect with that element's implicit range. A user can check the inclusion proof for the state update in the Exclusion Proof Element and be sure that its implicit range wasn't transacted in the given block.

19.2.3 State Update Proof Elements

State Update Proof Elements prove that a given state update was correctly created from the previous state update. Despite the name, State Update Proof Elements actually include **transactions** and not state updates. The provided transactions are used to compute the newer state update. These elements also include a **block number** in which the computed state update was included and an **inclusion proof** that the update was actually included.

CHAPTER 20

History Generation

Clients need to be able to correctly generate history proofs. This page describes the process of querying and generating these proofs.

20.1 History Query

Clients can query a history proof by creating a `HistoryQuery`:

```
interface HistoryQuery {
  plasmaContract: string
  start: number
  end: number
  startBlock?: number
  endBlock?: number
}
```

Where:

1. `plasmaContract` - `string`: Address of the specific plasma contract to query. Clients may watch several plasma contracts simultaneously, so this parameter is required for the client to return the correct history.
2. `start` - `number`: Start of the range of state objects to query.
3. `end` - `number`: End of the range of state objects to query.
4. `startBlock` - `number`: Block to start querying history from. If not provided, will default to 0.
5. `endBlock` - `number`: Block to query history to. If not provided, will default to the latest block known by the client.

Once created, these queries can be sent to a node via the history query RPC method.

20.2 History Calculation

When a node receives a history query, they **MUST** follow the following process to generate the correct history proof.

20.2.1 Range Intersection

Generation of a history for a given range begins by performing an intersection of the range with the historical state for the provided block range. Intersection **MUST** be performed on a start-inclusive, end-exclusive basis. Basically, for each block in the block range, the client needs to find all historical state updates with an implicit range or an explicit range that intersects with the queried range.

Intersection with the historical state will return three types of relevant elements: **Deposit Proof Elements**, **Exclusion Proof Elements**, and **State Update Proof Elements**. Each of these elements are necessary to generate a full history proof. However, each element must be handled differently during the proof generation process. A more detailed explanation of these elements can be found here.

Deposit Proof Elements and **State Update Proof Elements** will be found when the explicit range described by a state update directly intersects with queried range.

Exclusion Proof Elements are found when the implicit range described by a state update intersects with the queried range but the explicit range **does not**. Exclusion Proof Elements prove that a given range was **not** transacted during a specific block but **do not** prove that the given state update was actually valid.

20.2.2 Proof Generation

Deposit Proof Elements, and Exclusion Proof Elements, and State Update Proof Elements must each be handled individually when generating proofs. The process for handling each element is described below.

For each element returned by the range intersection, the client **MUST** generate a corresponding proof element according to the following rules. Proof elements **MUST** be returned in ascending block number order.

Deposit Proof Elements

Deposit Proof Elements consist of state updates that are created on the plasma chain when a deposit is submitted on Ethereum. Because we assume that clients have access to Ethereum, we **SHOULD NOT** include the full state update in the proof. Instead, clients **MUST** include the deposit ID logged on Ethereum when the deposit was submitted.

Exclusion Proof Elements

State updates are all associated with two ranges. A state update's *explicit range* is the range of state objects the update mutates. An update's *implicit range* is the range of objects that the update proves are *not* spent within a given block. This mechanism is a byproduct of our Merkle Interval Tree construction.

An Exclusion Proof Element consists of a **state update** and an **inclusion proof** . For each Exclusion Proof Element, the client **MUST** provide an inclusion proof that shows the element was included in a block. An Exclusion Proof Element does **not** prove that the given state update is valid. Therefore, the client **SHOULD NOT** provide any information that proves the validity of the state update beyond the inclusion proof.

State Update Proof Elements

State Update Proof Elements describe the transition of one or more state objects that fall within the specified range. State Update Proof Elements correspond to one or more transactions that generated the state update.

For each state update, the client **MUST** provide all transactions that produced the state update. However, as these transaction can be used to calculate the state update, the client **SHOULD NOT** provide the state update itself.

The client **MUST** also provide an inclusion proof for the state update that proves the update was included in the block specified in the state update

It's possible that the validity of a given transaction may also rely on the existence of some other plasma transaction. When this is the case, the verifier must first verify some additional proof elements before executing a given transaction. For each transaction that corresponds to a state update, the client **MUST** ask for a list of additional proof elements from the predicate plugin of the state objects from which the transaction spends. Any additional proof elements **MUST** be inserted *before* the transaction itself so that the client can verify the necessary state before verifying the transaction.

CHAPTER 21

History Verification

Once a client has received a history proof, they need to be able to correctly check the proof's validity. This page describes the process for applying each individual proof element and determining the overal validity of the proof.

21.1 Proof Structure

A HistoryProof consists of a series of "proof elements". Each of these elements is either a **Deposit Proof Element**, an **Exclusion Proof Element**, or a **State Update Proof Element**. Elements of each different element type must be handled differently as they convey different information.

21.2 Applying Proof Elements

Proof elements each correspond to some action that took place within a specific plasma block. Proof elements **MUST** be applied in ascending block order. Otherwise, a client may not have the necessary information to verify a specific state transition.

21.2.1 Deposit Proof Elements

Deposit Proof Elements consist of a **deposit ID**. Deposit IDs correspond to a deposit on Ethereum, which contains a state update. Whenever the client encounters a Deposit Proof Element, the client **MUST** download the deposit with the given ID by querying for the checkpoint with the same ID.

For efficiency, the client **SHOULD** check their local database for the deposit before querying Ethereum. It's possible that the client already downloaded while verifying another history proof.

If the deposit is not found, the client **MUST** either throw an error or skip the element.

Once the client has downloaded the corresponding deposit, the client **MUST** insert the deposit's state update into their local state. The client does **not** need to verify anything about this state update since it came directly from a deposit.

21.2.2 Exclusion Proof Elements

Elements of our Merkle Interval Tree have both an explicit range and an implicit range. A valid inclusion proof for an element of the tree also conveys the fact that no other elements intersect with the included element's implicit range.

Exclusion Proof Elements simply consist of a state update and an inclusion proof. These elements make no statements about the validity of the state update, but prove that there were no valid transactions on the update's implicit range.

The client **MUST** verify the inclusion proof for each Exclusion Proof Element. If the inclusion proof fails, the client **MUST** either throw an error or skip to the next proof element.

The client **MUST** then find all state updates that intersect with the implicit range of the proof element where `update.verifiedBlockNumber` is equal to `element.block - 1`. We're only interested in these state updates because the implicit proof only applies to the `element.block` but not any previous blocks.

Next, for each found state update, the client **MUST** split any elements where the implicit range only partially covers the intersected update. For example, if the implicit range is `(50, 100)` but the found update is over `(0, 100)`, the client will split the update into two elements that cover `(0, 50)` and `(50, 100)`. This process will leave the client with a set of state updates that are entirely covered by the implicit range and a set that no longer intersect at all.

Next, for each element that is **entirely** covered by the implicit range, the client **MUST** set `update.verifiedBlockNumber` to `element.block`. This process effectively finds any state updates covered by the implicit range and "bumps up" the block to which they're considered valid.

21.2.3 State Update Proof Elements

State Update Proof Elements prove that a given state update has transitioned to a newer one. Despite the name, State Update Proof Elements actually consist of transactions and not state updates. The provided transactions are used to compute the newer state update. These elements also include a block number in which the computed state update was included and an inclusion proof that the state update was actually included in the given block.

For simplicity, we require that all of the transactions within a single State Update Proof Element refer to the same range. If any of the transactions do not refer to the same range, the client **MUST** either throw an error or skip to the next element.

Next, the client **MUST** find all previously verified state updates that intersect with the range on the transations **and** where `update.verifiedBlockNumber` is equal to `element.block - 1`.

For each update found in the previous step, the client then **MUST** execute each transaction against the update. Each transaction execution will generate a resulting state update.

After executing the transactions against all updates, the client **MUST** verify that the resulting state updates are all equivalent. If any state update is not equivalent, the client **MUST** either throw an error or skip to the next proof element. The resulting update **MUST** have a block number equal to `element.block`.

The client then **MUST** verify the inclusion proof provided as part of the proof element. If the inclusion proof is invalid, the client **MUST** either throw an error or skip to the next proof element.

Finally, the client **MUST** insert the resulting state update into the local state. This process **MUST** split or overwrite any existing state updates over the same range. For example, if the local state contains an update that covers `(0, 100)` and the computed update covers `(50, 100)`, the client will modify the first update such that it only covers `(0, 50)`.

CHAPTER 22

Exit Guarding

22.1 Event Handling

22.1.1 Predicate Event Watching

Todo: Explain how predicate plugins can watch for exit events.

22.2 Predicate Guarding

22.2.1 Exit Validity Determination

Todo: Explain how predicate plugins determine whether an exit is valid or not.

22.2.2 Challenge Generation

Todo: Explain how predicate plugins generate a challenge.

22.2.3 Challenge Submission

Todo: Explain how predicate plugins actually send the challenge to Ethereum.

CHAPTER 23

State Queries

Clients need to be able to query the local state to build effective front-ends. For example, a wallet might be interested in querying all state objects that a specific user owns. We've designed a querying system to make this process as easy as possible.

23.1 Query Generation

23.1.1 Parsing Predicate API

The Predicate API provided by each predicate specifies a list of queries that can be made on state objects locked with that predicate. For example, the API of the SimpleOwnership predicate specifies the following function:

```
{
  "name": "getOwner",
  "constant": true,
  "inputs": [],
  "outputs": [
    {
      "name": "owner",
      "type": "address"
    }
  ]
}
```

Users first need to parse this API to figure out what methods are available for the predicate they're attempting to query. Once the user has determined the name and required parameters for the method they want to query, they can generate a StateQuery. The user can then send this query to a client via the state query RPC method.

23.1.2 Parsing Query Results

The result of a state query is a list of **'StateQueryResult'_** objects. If a filter was provided as part of the `StateQuery`, only `StateQueryResult` objects that passed the provided filter will be returned.

23.1.3 Example: SimpleOwnership

We'll now provide an example state query for the getOwner method of the SimpleOwnership predicate. The Predicate API for SimpleOwnership describes getOwner as:

```
{
  "name": "getOwner",
  "constant": true,
  "inputs": [],
  "outputs": [
    {
      "name": "owner",
      "type": "address"
    }
  ]
}
```

Let's assume that our plasmaContract is 0x1b33c35be86be9d214f54af218c443c2623d3d0a and our SimpleOwnership predicate is located at 0xf25746ac8621a7998e0992b9d88e260c117c145f.

To query all state updates where 0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c is the owner, we construct the following query:

```
const query: StateQuery = {
  plasmaContract: '0x1b33c35be86be9d214f54af218c443c2623d3d0a',
  predicateAddress: '0xf25746ac8621a7998e0992b9d88e260c117c145f',
  method: 'getOwner',
  params: [],
  filter: {
    $eq: [ '$0', '0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c' ]
  }
}
```

Our filter here specifies that the first output of the function call (which we know to be the owner in this case) should be equal to a given address. Results where this is not the case will not be returned.

Once we send this request via the state query RPC method, we'll receive a result that looks like this:

```
[
  {
    stateUpdate: {
      block: 123,
      start: 0,
      end: 100,
      predicate: '0xf25746ac8621a7998e0992b9d88e260c117c145f',
      data: '0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c'
    },
    result: ['0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c']
  },
  ...
]
```

We can then present this data in any way that we might want to.

23.2 Query Handling

23.2.1 Range Intersection

Clients will receive a StateQuery object when receiving a state query. Clients first **MUST** use the range provided by the `StateQuery` to find all state updates in the current head state that match the provided `predicateAddress`.

23.2.2 Passing Queries to Predicate Plugins

Once the client has found all relevant state updates, they **MUST** call the queryState method in the predicate plugin that corresponds to the provided `predicateAddress`. `queryState` takes the `method` and `parameters` from the `StateQuery` and returns an array of results.

23.2.3 Filtering Queries

Users filter their results by providing an Expression that performs some operation on the provided result. If a `filter` was given in the `StateQuery`, then the client **MUST** correctly remove results according to the filter. More information about filters is given in the page about Expressions.

Users can filter based on the outputs of the query method by inserting strings in the form of `\$[0-9]+` (starting at `$0`).

For example, a user could filter results where the first output is greater than `0` and the second result is less than `100` like this:

```
{
  $and: [
    { $gt: [ '$0', 0 ] },
    { $lt: [ '$1', 100 ] }
  ]
}
```

Any results that have not been removed by the filter can then be returned to the requesting client.

CHAPTER 24

Synchronization

24.1 Introduction

Clients need to be able to synchronize their local state with that of the operator. However, this process is somewhat non-trivial as clients may be simultaneously connected to multiple plasma chains.

The actual process for synchronizing a client can generally be determined by the client implementer. However, we're going to go through some recommendations that will ensure that the client is as feature-complete as possible.

24.2 Deposit and Commitment Contracts

Our plasma construction makes use of deposit contracts that store the assets that users transact on the plasma chain. These deposit contracts act like their own individual plasma chains. For example, the range `(0, 100)` will be valid on two different deposit contracts but will refer to different assets.

Furthermore, we're generally using a single deposit contract per asset type to simplify things for the client. It's therefore very likely that a client will be interested in transactions on several different deposit contracts. As a result, clients **SHOULD** be able to send and receive transactions on multiple deposit contracts simultanously.

We're also using the new concept of commitment contracts that store plasma block commitments instead of throwing all of this logic inside the deposit contract. Each deposit contract points to a specific commitment contract, and it's possible for multiple deposit contracts to point to the same contract. Therefore a client **SHOULD** also be able to watch for commitments to multiple commitment contracts and **SHOULD** maintain a mapping between commitment contracts and deposit contracts.

24.3 Receiving Transactions

Transactions are unique to a given deposit contract, but blocks are unique to a commitment contract. For each commitment contract the client is interested in, clients **SHOULD** watch for new blocks being published to Ethereum.

It's important to note that, unlike in previous plasma constructions, there's no easy way for an operator to tell that a given address will be interested in a specific transaction. Instead, clients **SHOULD**, upon seeing the publication of a new block, send a **'state query'_** to the operator for all state updates the client is interested in.

For example, imagine we have a predicate that allows anyone to mutate a state object as long as they have the pre-image to some hash. Without the pre-image, the operator has no way to know which user "owns" that state object. A client would have to specifically form a query for all state objects that use that predicate and lock the state object with a specific hash.

CHAPTER 25

Query Expressions

25.1 Description

We provide a system for filtering state queries via expressions similar to those used by mongoDB.

25.2 Data Structures

25.2.1 Expression

```
interface Expression {
  [operator: string]: Array<string | number | Expression>
}
```

Description

Represents an expression. All expressions consist of an operator and a list of input values. Some expressions take other operators as input values.

25.3 Operators

25.3.1 Boolean Operators

$and

```
{ $and: [ <expression1>, <expression2>, ... ] }
```

Description

Returns `true` if **all** arguments evaluate to `true`.

Parameters

1. `expressions` - `Expression[]`: Any number of `Expression` objects.

Returns

`boolean`: `true` if **all** arguments resolve to `true`, `false` otherwise.

$not

```
{ $not: [ <expression> ] }
```

Description

Returns the boolean opposite of the result of the argument expression.

Parameters

1. `expression`: `Expression`: A single `Expression` object.

Returns

`boolean`: `true` if the expression resolves to `false`, `false` otherwise.

$or

```
{ $or: [ <expression1>, <expression2>, ... ] }
```

Description

Returns `true` if **any** argument evaluates to `true`.

Parameters

1. `expressions` - `Expression[]`: Any number of `Expression` objects.

Returns

`boolean`: `true` if **any** argument resolves to `true`, `false` otherwise.

25.3.2 Comparison Operators

$eq

```
{ $eq: [ <argument1>, <argument2>, ... ] }
```

Description

Checks if **all** arguments are equal.

Parameters

1. `arguments` - `any[]`: List of input values.

Returns

`boolean`: `true` if **all** arguments are equal, `false` otherwise.

$gt

```
{ $gt: [ <argument1>, <argument2> ] }
```

Description

Checks if the first argument is greater than the second.

Parameters

1. `argument1` - `any`: First input value.
2. `argument2` - `any`: Second input value.

Returns

`boolean`: `true` if the first argument is greater than the second, `false` otherwise.

$gte

```
{ $gte: [ <argument1>, <argument2> ] }
```

Description

Checks if the first value is greater than or equal to the second.

Parameters

1. `argument1` - `any`: First input value.
2. `argument2` - `any`: Second input value.

Returns

`boolean`: `true` if the first value is greater than or equal to the second, `false` otherwise.

$lt

```
{ $lt: [ <argument1>, <argument2> ] }
```

Description

Checks if the first value is less than the second.

Parameters

1. `argument1` - `any`: First input value.
2. `argument2` - `any`: Second input value.

Returns

`boolean`: `true` if the first value is less than the second, `false` otherwise.

$lte

```
{ $lte: [ <argument1>, <argument2> ] }
```

Description

Checks if the first value is less than or equal to the second.

Parameters

1. `argument1` - `any`: First input value.
2. `argument2` - `any`: Second input value.

Returns

`boolean`: `true` if the first value is less than or equal to the second, `false` otherwise.

$ne

```
{ $ne: [ <argument1>, <argument2>, ... ] }
```

Description

Returns `true` if input values are not all equivalent.

Parameters

1. `arguments` - `any[]`: List of input values.

Returns

`boolean`: `true` if input values are not equivalent, `false` otherwise.

CHAPTER 26

RPC Methods

26.1 Description

We require that the clients interact via JSON RPC. Here we provide a list of available RPC methods that the client **MUST** implement.

26.2 Methods

26.2.1 pg_accounts

Description

Returns the list accounts in the client's wallet.

Returns

`string[]`: List of addresses owned by the client.

26.2.2 pg_blockNumber

Description

Returns the current plasma block number.

Returns

`number`: Current plasma block number.

26.2.3 pg_sign

Description

Signs a message with the private key of a specified account.

Parameters

1. `account` - `string`: Address of the account to sign with.
2. `message` - `string`: Message to sign.

Returns

`string`: Signature over the given message.

26.2.4 pg_sendRawTransaction

Description

Sends a transaction to the client. If the client is not the operator, the transaction will be forwarded to the operator.

Parameters

1. `transaction` - `string`: Properly encoded transaction to send to the operator.

Returns

`string`: Receipt for the transaction.

26.2.5 pg_sendQuery

Description

Sends a state query to the client.

Parameters

1. `query` - `StateQuery`: A StateQuery object.

Returns

`any[]`: Result of the state query. Returns one query result for each state object that intersected with the range specified in the `StateQuery`.

26.2.6 pg_getTransactionByHash

Description

Returns a full transaction from a transaction hash.

Parameters

1. `hash` - `string`: Hash of the transaction to query.

Returns

`Transaction`: The Transaction object with the given hash.

26.2.7 pg_getProof

Description

Returns a history proof for a given range.

Parameters

1. `query` - `HistoryQuery`: A HistoryQuery object.

Returns

`HistoryProof`: A HistoryProof composed of a list of proof elements that can be ingested.

26.2.8 pg_getInstalledPredicatePlugins

Description

Returns the list of predicates installed by the client.

Returns

`string[]`: Address of each predicate for which the client has an installed plugin.

26.2.9 pg_clientVersion

Description

Returns the name and version of the client.

Returns

`string`: Version and name of the client in the form `<name>/<version>/<os>`.

CHAPTER 27

RPC Error Messages

27.1 Wallet

27.1.1 Account Not Found

```
{
  code: -20001,
  message: "Account Not Found",
  data: <address>
}
```

Description

Error returned when an account has not been found for a given address.

Code

-20001

Data

1. `address` - `string`: Address of the queried account.

27.1.2 Invalid Password

```
{
  code: -20002,
  message: "Invalid Password",
  data: <address>
}
```

Description

Returned when the user attempts to unlock an account with an invalid password.

Code

-20002

Data

1. `address` - `string`: Address of the account the user attempted to unlock.

27.1.3 Account Locked

```
{
  code: -20003,
  message: "Account Locked",
  data: <address>
}
```

Description

Returned when a user attempts to sign some data with a locked account.

Code

-20003

Data

1. `address` - `string`: Address of the account the user attempted to sign with.

27.2 Transactions

27.2.1 Invalid Transaction Encoding

```
{
  code: -20004,
  message: "Invalid Transaction Encoding"
}
```

Description

Returned when a user attempts to submit an incorrectly encoded transaction.

Code

-20004

27.2.2 Invalid Transaction

```
{
  code: -20005,
  message: "Invalid Transaction"
}
```

Description

Returned when a user submits a transaction that executes incorrectly.

Code

-20005

27.3 State Queries

27.3.1 Invalid State Query

```
{
  code: -20006,
  message: "Invalid State Query"
}
```

Description

Returned when the user attempts to make an invalid state query.

Code

-20006

CHAPTER 28

Introduction

CHAPTER 29

Transaction Ingestion

29.1 Incoming Transaction Endpoint

Clients send transactions to the operator via the send transaction RPC method, which is then received by the RPC server. The RPC server then pipes the transaction to the operator's state manager.

29.2 Transaction Decoding

Before the operator can execute the transaction, it **MUST** first check that the transaction is correctly formatted by attempting to decode the transaction.

If decoding fails, the operator **MUST** return an Invalid Transaction Encoding error response.

29.3 Transaction Execution

Once the operator has determined that the transaction was correctly encoded, they can attempt to execute the transaction against the local state.

29.3.1 State Update Resolution

The operator **MUST** resolve the set of state updates that the transaction operates on. The operator uses the `start` and `end` values from the transaction and finds all state updates the range overlaps with.

Once the operator finds all overlapping state updates, they **MUST** assert that the entire range described by the transaction is covered by existing state updates. If this is not the case, the operator **MUST** return an Invalid Transaction error response.

29.3.2 State Transition Execution

For each state update resolved in the previous step, the operator then **MUST** call the executeStateTransition method of the plugin that corresponds to the predicate address specified in the state update. This function call will return a new resulting state update. If any of these calls throw an error, the operator **MUST** return an Invalid Transaction error response.

The operator **MUST** then validate that all of the resulting state updates are identical. If any state update is not identical, the operator **MUST** return an Invalid Transaction error response.

29.3.3 Transaction Queueing

Once the transaction has been verified, the operator can add the resulting state update to the queue of state updates to be published in the next block. If the queue already contains a state update on the range specified in the transaction, the operator **MUST** return a Duplicate Transaction error response.

29.4 Transaction Receipt

Finally, once the new state update has been added to the block queue, the operator **MUST** return a transaction receipt to the client.

CHAPTER 30

Block Generation

30.1 Transaction Ingestion

30.2 Block Merklization

30.3 Block Submission

CHAPTER 31

Operator RPC Methods

31.1 Description

Todo: Add description for Operator RPC methods.

31.2 Methods

31.2.1 pgop_sendTransaction

Description

Sends a transaction to be ingested by the operator.

Parameters

1. `transaction` - `Transaction`: The Transaction to be ingested.

Returns

`string`: The transaction receipt for the given transaction.

CHAPTER 32

Introduction

32.1 Components

Our client is broken up into several key components, each of which are specified separately. Here we'll discuss a high-level overview of the different components and what they do.

32.1.1 RpcServer

RpcServer is the exposes a JSON RPC server that users can interact with. It pipes calls to different RPC methods to the various other components that can fulfill these requests.

32.1.2 RpcClient

RpcClient handles interactions with other clients and with the operator.

32.1.3 StateDB

StateDB stores information about the current head state.

32.1.4 StateManager

StateManager executes transactions and applies them to the head state. `StateManager` also handles queries about the current state. `StateManager` is the only component with access to `StateDB`.

32.1.5 HistoryDB

HistoryDB stores historical information about the plasma chain. `HistoryDB` is primarily used to store history proof information necessary to assert the validity of a given transaction.

32.1.6 HistoryManager

HistoryManager handles queries about historical state and generates history proofs. `HistoryManager` is the only component with access to `HistoryDB`.

32.1.7 PluginManager

PluginManager handles access to the various predicate plugins loaded into the client. Other components are expected to go through `PluginManager` whenever they want to interact with a plugin.

32.1.8 EventWatcher

EventWatcher watches for various important events on Ethereum. Components can request that `EventWatcher` watch a specific event and will be notified whenever the event is fired. `EventWatcher` is designed to be robust as not to miss events or notify components of the same event multiple times.

32.1.9 ContractWrapper

ContractWrapper is a simple wrapper around the smart contracts the client needs to interact with. All components, except for `EventWatcher`, are expected to only interact with Ethereum through `ContractWrapper`.

32.2 Architecture Diagram

We've provided a diagram of the interactions between the various client components below.

CHAPTER 33

MerkleIntervalTree

33.1 Description

Our plasma implementation uses a special Merkle tree called a Merkle Interval Tree. This page describes the interface for an implementation of the tree.

33.2 API

33.2.1 Structs

MerkleIntervalTreeLeafNode

```
interface MerkleIntervalTreeLeafNode {
  start: number
  end: number
  data: string
}
```

Description

Represents a leaf node in the tree.

Fields

1. `start` - `number`: Start of the range this leaf node covers.
2. `end` - `number`: End of the range this leaf node covers.

3. `data` - `string`: Arbitrary data held by the leaf node.

MerkleIntervalTreeInternalNode

```
interface MerkleIntervalTreeInternalNode {
  index: number
  hash: string
}
```

Description

Represents an internal node in the interval tree.

Fields

1. `index` - `number`: Index value of the node's left child.
2. `hash` - `string`: Hash of the node's children.

33.2.2 Methods

getInclusionProof

```
function getInclusionProof(
  leaf: MerkleIntervalTreeLeafNode
): Promise<MerkleIntervalTreeInternalNode[]>
```

Description

Generates an inclusion proof for a given leaf node.

Parameters

1. `leaf` - `MerkleIntervalTreeLeafNode`: Leaf node to generate a proof for.

Returns

`MerkleIntervalTreeInternalNode[]`: List of internal nodes that form the inclusion proof.

---

checkInclusionProof

```
function checkInclusionProof(
  leaf: MerkleIntervalTreeLeafNode,
  leafIndex: number,
  root: MerkleIntervalTreeInternalNode,
  inclusionProof: MerkleIntervalTreeInternalNode[]
): boolean
```

Description

Checks an inclusion proof for a given leaf node.

Parameters

1. `leaf` - `MerkleIntervalTreeLeafNode`: Leaf node to check inclusion for.
2. `leafIndex` - `number`: Index of the leaf node in the list of leaf nodes.
3. `root` - `MerkleIntervalTreeInternalNode`: Root of the Merkle Interval Tree.
4. `inclusionProof` - `MerkleIntervalTreeInternalNode[]`: List of internal nodes that form the inclusion proof.

Returns

`boolean`: `true` if the proof is valid, `false` otherwise.

CHAPTER 34

RangeDB

34.1 Description

`RangeDB` is a database abstraction that makes it easy to map ranges, defined by a `start` and an `end`, to arbitrary values.

34.2 API

34.2.1 Structs

RangeEntry

```
interface RangeEntry {
  start: number
  end: number
  value: Buffer
}
```

Description

Represents a value in the database. All values are constructed with respect to some range, defined by `start` and `end`.

Fields

1. `start` - `number`: Start of the range described by this entry.
2. `end` - `number`: End of the range described by this entry.

3. `value` - `Buffer`: Value at this specific range.

34.2.2 Methods

get

```
async function get(start: number, end: number): Promise<RangeEntry[]>
```

Description

Queries `RangeEntry` values that intersect with a given `start` and `end`. Ranges **must** be treated as start-inclusive and end-exclusive.

Parameters

1. `start` - `number`: Start of the range to query.
2. `end` - `number`: End of the range to query.

Returns

`Promise<RangeEntry[]>`: All `RangeEntry` objects in the database such that `entry.start` and `entry.end` intersect with the given `start` and `end`.

---

put

```
async function put(start: number, end: number, value: Buffer): Promise<void>
```

Description

Adds a value to the database at a given range. Overwrites all existing `RangeEntry` objects that overlap with the range. `put` **MUST** modify or break apart existing `RangeEntry` objects if the given range only partially overlaps with the object. For example, if we currently have a `RangeEntry` over the range `(0, 100)` and call `put(25, 75, "some value")`, this function must break the existing `RangeEntry` into new entries for `(0, 25)` and `(75, 100)`.

Parameters

1. `start` - `number`: Start of the range to insert into.
2. `end` - `number`: End of the range to insert into.
3. `value` - `Buffer`: Value to insert into the range as a Buffer.

Returns

`Promise<void>`: Promise that resolves once the range has been inserted.

---

del

```
async function del(start: number, end: number): Promise<void>
```

Description

Deletes the values for the given range. Will insert new `RangeEntry` objects or modify existing ones when the given range only partially overlaps with those already in the database. This method **MUST** delete objects under the same scheme described above for `put`.

Parameters

1. `start` - `number`: Start of the range to delete.
2. `end` - `number`: End of the range to delete.

Returns

`Promise<void>`: Promise which resolves once the range has been deleted.

CHAPTER 35

ContractWrapper

35.1 Description

`ContractWrapper` provides an interface for interacting with the various plasma chain smart contracts. Other components are expected to go through `ContractWrapper` whenever they want to interact with smart contracts.

35.2 API

35.2.1 Methods

Todo: Add ContractWrapper methods.

CHAPTER 36

EventWatcher

36.1 Description

Plasma chain contracts emit various important Ethereum contract events that the client needs to be aware of. These events include things like notifications of deposits and exits. `EventWatcher` provides the necessary functionality to watch for these events.

36.1.1 Event Uniqueness

It's often important that an event only be handled **once**. We might otherwise end up in a situation in which, for example, a client attempts to credit the same deposit twice. We also don't want to increase client load by querying the same events multiple times.

`EventWatcher` is therefore slightly more complicated than a basic Ethereum event watcher. Each `EventWatcher` instance has access to a local database that it **MUST** use to store the hash of any event that's already been seen. The hash of an event is computed as the keccak256 hash of the hash of the ABI encoded transaction that emitted event prepended to the index of the event.

`EventWatcher` will also store the current block up to which it has checked for a given event on a given contract. `EventWatcher` **MUST** store a different block for each tuple of `(contract, event name, event filter)`.

36.2 API

36.2.1 Methods

on

```
function on(
  contractAddress: string,
  contractAbi: any,
  eventName: string,
  eventFilter: any,
  callback: (eventName: string, eventData: any) => void
): void
```

Description

Starts watching an event for a given contract.

Parameters

1. `contractAddress` - `string`: Address of the contract to watch.
2. `contractAbi` - `any`: JSON ABI of the contract to watch.
3. `eventName` - `string`: Name of the event to watch.
4. `eventFilter` - `any`: An event filter for the event.
5. `callback - (eventName: string, eventData: any) => void`: Callback to be triggered whenever a matching event is detected in the smart contract.

---

off

```
function off(
  contractAddress: string,
  contractAbi: any,
  eventName: string,
  eventFilter: any,
  callback: (eventName: string, eventData: any) => void
): void
```

Description

Stops watching an event for a given contract. Will only remove the listener that corresponds to the specific event name, filter, and callback provided.

Parameters

1. `contractAddress` - `string`: Address of the contract to watch.
2. `contractAbi` - `any`: JSON ABI of the contract to watch.
3. `eventName` - `string`: Name of the event to watch.
4. `eventFilter` - `any`: An event filter for the event.

5. `callback` - `(eventName: string, eventData: any) => void`: Callback to be triggered whenever a matching event is detected in the smart contract.

CHAPTER 37

WalletDB

37.1 Description

`WalletDB` stores and manages access to standard keystore files. We require that keystore files **MUST** be encrypted for the safety of user funds.

37.2 API

37.2.1 Structs

Keystore

```
interface Keystore {
  address: string
  crypto: {
    cipher: string
    ciphertext: string
    cipherparams: {
      iv: string
    }
    kdf: string
    kdfparams: {
      dklen: number
      n: number
      p: number
      r: number
      salt: string
    }
    mac: string
```

(continues on next page)

(continued from previous page)

```
  }
  id: string
  version: number
}
```

Description

Standard format for storing keystore objects in Ethereum.

37.2.2 Methods

putKeystore

```
async function setKeystore(address: string): Promise<void>
```

Description

Sets the keystore file for a given address.

Parameters

1. `address` - `string`: Address to set a keystore for.

Returns

`Promise<void>`: Promise that resolves once the keystore has been inserted.

---

getKeystore

```
async function getKeystore(address: string): Promise<Keystore>
```

Description

Pulls the keystore file for a given address.

Parameters

1. `address` - `string`: Address to query a keystore for.

Returns

`Promise<Keystore>`: The *Keystore* object associated with that address.

---

listAddresses

```
async function listAddresses(): Promise<string[]>
```

Description

Queries the list of all available account addresses with keystore files.

Returns

`Promise<string[]>`: List of account addresses where the DB has a keystore file.

CHAPTER 38

Wallet

38.1 Description

Clients often need to create signatures in order to authenticate transactions. The `Wallet` component provides a standard interface for creating new accounts, querying existing accounts, and signing arbitrary data.

38.2 API

38.2.1 Methods

listAccounts

```
async function listAccounts(): Promise<string[]>
```

Description

Queries the addresses of all accounts stored in the wallet.

Returns

`Promise<string[]>`: List of all account addresses in the wallet.

createAccount

```
async function createAccount(password: string): Promise<string>
```

Description

Creates an account and returns the new account’s address. Encrypts the account with a given password.

Parameters

1. `password` - `string`: Password used to encrypt the account.

Returns

`Promise<string>`: Address of the newly created account.

CHAPTER 39

PredicatePlugin

39.1 Description

Clients need a way to execute state transitions for a given predicate. However, it's too slow and complex to execute these state transitions in a virtual version of the EVM. As a result, Predicates **MUST** supply client-side code, called a *predicate plugin*, that can compute state transitions. Predicate plugins **MUST** conform a standard interface as described below.

39.2 Information Sources

Predicate plugins have access to various external sources of information. Plugins can use this information for various reasons, including the execution of state transitions. All information available to a plugin **MUST** also be available to the corresponding predicate contract on Ethereum.

39.2.1 Ethereum Contract Queries

Predicate plugins have access to any information available on Ethereum. As plugins are effectively native implementations of their contract counter-parts, plugins should be careful not to rely on information not available to the contract (like event logs).

39.2.2 Plasma State Queries

Predicate contracts on Ethereum can be fed information about the state of the plasma chain. Predicate plugins are therefore given a reference to StateManager and HistoryManager that permit the plugin to make queries about the existence (or non-existence) of a given StateUpdate in the plasma chain.

39.2.3 Plasma State Updates

Predicates **MUST** be able to insert state updates into the local plasma chain. Most predicates **SHOULD NOT** use this behavior, but certain predicates may require to function correctly.

39.3 API

39.3.1 Methods

executeStateTransition

```
async function executeStateTransition(
  input: StateUpdate,
  transaction: Transaction
): Promise<StateUpdate>
```

Description

Executes a transaction and returns the resulting state upate.

Parameters

1. `input` - `StateUpdate`: Previous StateUpdate that the transaction acts upon.
2. `transaction` - `Transaction`: Transaction to execute.

Returns

`Promise<StateUpdate>`: Resulting StateUpdate created by the application of the transaction.

---

queryState

```
async function queryState(stateUpdate: StateUpdate, method: string, parameters:␣
↪string[]): Promise<string[]>
```

Description

Performs a query on a given state update.

Parameters

1. `stateUpdate` - `StateUpdate`: The StateUpdate object to perform a query on.
2. `method` - `string`: Name of the query method to call.
3. `parameters` - `string[]`: Additional parameters to the query.

Returns

`string[]`: List of return values based on the predicate's API.

---

getAdditionalHistoryProof

```
async function getAdditionalHistoryProof
  transaction: Transaction
): Promise<HistoryProof>
```

Description

Predicates may specify rely on the existence (or non-existence) of a given StateUpdate in the plasma chain. Whenever this is the case, the client must verify the history proof for that `StateUpdate`. This method allows a predicate to specify any additional history proof information that may be necessary to verify these extra `StateUpdate` objects.

Parameters

1. `transaction` - `Transaction`: The Transaction that may require additional proof data.

Returns

`Promise<HistoryProof>`: The HistoryProof object that contains the extra proof data. May be an empty array if the transaction requires no additional history proof data.

---

canReplaceTransaction

```
async function canReplaceTransaction(
  oldTransaction: Transaction,
  newTransaction: Transaction
): Promise<boolean>
```

Description

Plasma blocks are composed of commitments to StateUpdate objects. Each `StateUpdate` is computed from a previous `StateUpdate` and a Transaction. It's possible for one transaction to generate the same `StateUpdate` as another transaction, and therefore still be a valid component of a history proof, but have significantly less overhead than the other. Clients may wish to "replace" one transaction with another to reduce proof overhead.

Predicates can define an arbitrary heuristic within this method to determine if one transaction is preferable to another.

Parameters

1. `oldTransaction` - `Transaction`: Original Transaction to be replaced.
2. `newTransaction` - `Transaction`: New Transaction to replace the original.

Returns

`boolean`: `true` if the newer transaction should replace the older one, `false` otherwise.

---

onTransitionFrom

```
async function onTransitionFrom(
  transaction: Transaction,
  from: StateUpdate,
  to: StateUpdate,
  verifiedRanges: Range[]
): Promise<void>
```

Description

Hook called whenever a StateUpdate locked by the predicate has been transitioned away from. Predicates may wish to use this hook to carry out some internal logic.

Parameters

1. `transaction` - `Transaction`: The Transaction which executed a state transition.
2. `from` - `StateUpdate`: The old StateUpdate transitioned away from by the transaction.
3. `to` - `StateUpdate`: The new StateUpdate created by the transaction.
4. `verifiedRanges` - `Range[]`: Parts of the range described by `to` with a fully verified history. It's possible that a transaction creates a StateUpdate with only a partially verified history. For example, we may have a transaction that sends state objects `(0, 100)` but have only verified `(0, 50)`. This is considered valid behavior as we simply ignore `(50, 100)` until we have its full history.

Returns

`Promise<void>`: Promise that resolves once the predicate has executed some logic for the hook.

---

onTransitionTo

```
async function onTransitionTo(
  transaction: Transaction,
  from: StateUpdate,
  to: StateUpdate,
  verifiedRanges: Range[]
): Promise<void>
```

Description

Hook called whenever a Transaction creates a new StateUpdate locked by the predicate. Predicates may wish to use this hook to carry out some internal logic.

Parameters

1. `transaction` - `Transaction`: The Transaction which executed a state transition.
2. `from` - `StateUpdate`: The old StateUpdate transitioned away from by the transaction.
3. `to` - `StateUpdate`: The new StateUpdate created by the transaction.
4. `verifiedRanges` - `Range[]`: Parts of the range described by `to` with a fully verified history. It's possible that a transaction creates a StateUpdate with only a partially verified history. For example, we may have a transaction that sends state objects `(0, 100)` but have only verified `(0, 50)`. This is considered valid behavior as we simply ignore `(50, 100)` until we have its full history.

Returns

`Promise<void>`: Promise that resolves once the predicate has executed some logic for the hook.

CHAPTER 40

PluginManager

40.1 Description

The `PluginManager` handles loading and accessing PredicatePlugin objects. Other components are expected to go through the `PluginManager` whenever they intend to access a specific `PredicatePlugin`.

40.2 API

40.2.1 Methods

loadPlugin

```
async function loadPlugin(
  address: string,
  path: string
): Promise<PredicatePlugin>
```

Description

Loads a plugin at a given file path and assigns it to a specific address.

Parameters

1. `address` - `string`: Address of the predicate to load a plugin for.
2. `path` - `string`: Path to the predicate plugin to load.

Returns

`Promise<PredicatePlugin>`: The loaded PredicatePlugin.

---

getPlugin

```
async function getPlugin(address: string): Promise<PredicatePlugin>
```

Description

Returns the plugin for the predicate with the given address.

Parameters

1. `address` - `string`: Address of the predicate to get a plugin for.

Returns

`Promise<PredicatePlugin>`: The PredicatePlugin that corresponds to the given address.

CHAPTER 41

History Proof Structure

41.1 Description

In every plasma block, a range of state objects can either be deposited, transacted, or not transacted. Whenever clients want to verify a **'transaction'_** on a specific range, they need to verify the entire "history" of what happened to the range between the block in which it was first deposited and the block in which the transaction occurred.

For example, let's imagine that a range `(0, 100)` was deposited in block 1, not transacted in block 2, and then transacted in block 3. The history proof for a transaction in block 4 would contain the deposit, a proof that the range wasn't transacted in block 2, and a proof that the range was transacted in block 3.

This page will describe the data structure that make up a history proof. We describe the history proof process in more detail separately.

41.2 Data Structures

41.2.1 DepositElement

```
interface DepositElement {
  block: number
  depositId: string
}
```

Description

Proof element for importing deposits.

Fields

1. `block` - `number`: Block in which the deposit was included.
2. `depositId` - `string`: ID of the deposit.

41.2.2 StateUpdateElement

```
interface StateUpdateElement {
  block: number
  transactions: Transaction[]
  inclusionProof: InclusionProof
}
```

Description

Proof element that transitions an existing state update with some given transactions.

Fields

1. `block` - `number`: Block in which the new state update was included.
2. `transactions` - `Transaction[]`: List of **'Transaction'_** objects that generated the new state update.
3. `inclusionProof` - `InclusionProof`: An InclusionProof for the generated state update.

41.2.3 NonInclusionElement

```
interface NonInclusionElement {
  block: number
  stateUpdate: StateUpdate
  inclusionProof: InclusionProof
}
```

Description

Proof element that shows a given range was **not** spent in a specific block.

Fields

1. `block` - `number`: Block in which the state update was included.
2. `stateUpdate` - `StateUpdate`: State update whose implicit range proves that a specific range of state objects were not spent in a specific block.
3. `inclusionProof` - `InclusionProof`: An InclusionProof that shows the state update was included in the specified block.

41.2.4 HistoryProof

```
type HistoryProof = Array<DepositElement | StateUpdateElement | NonInclusionElement>
```

Description

A list of `DepositElement`, `StateUpdateElement`, and `NonInclusionElement` objects.

CHAPTER 42

HistoryDB

42.1 Description

`HistoryDB` provides a simple interface to the set of historical StateUpdate objects. `HistoryDB` also stores all values necessary to prove the history of a given state object, including the Merkle Interval Tree inclusion proofs for each state update and the transactions that created those state updates.

42.2 API

42.2.1 Methods

getStateUpdate

```
async function getStateUpdate(
  stateUpdateHash: string
): Promise<StateUpdate>
```

Description

Queries a full state update from the hash of the state update.

Parameters

1. `stateUpdateHash` - `string`: keccak256 hash of the state update.

Returns

`Promise<StateUpdate>`: Full state update that corresponds to the given hash.

---

putStateUpdate

```
async function putStateUpdate(stateUpdate: StateUpdate): Promise<void>
```

Description

Adds a state update to the database.

Parameters

1. `stateUpdate` - `StateUpdate`: StateUpdate to add to the database.

Returns

`Promise<void>`: Promise that resolves once the state update has been added to the database.

---

getStateUpdateTransactions

```
getStateUpdateTransactions(
  stateUpdateHash: string
): Promise<Transaction[]>
```

Description

Queries the list of Transaction objects that created the given state update.

Parameters

1. `stateUpdateHash` - `string`: keccak256 hash of the state update to query.

Returns

`Promise<Transaction[]>`: List of Transaction objects that created the given state update.

---

putStateUpdateTransactions

```
async function putStateUpdateTransactions(
  stateUpdateHash: string,
  transactions: Transaction[]
): Promise<void>
```

Description

Stores the set of transactions that created a given state update.

Parameters

1. `stateUpdateHash` - `string`: keccak256 hash of the state update to set transactions for.
2. `transactions` - `Transaction[]`: List of transactions that created the given state update.

Returns

`Promise<void>`: Promise that resolves once the transactions have been stored.

---

getStateUpdateLeafPosition

```
async function getStateUpdateLeafPosition(
  stateUpdateHash: string
): Promise<number>
```

Description

Gets the **'leaf position'_** of a given state update within the Merkle Interval Tree of the block in which the state update was included.

Parameters

1. `stateUpdateHash` - `string`: keccak256 hash of the state update to query.

Returns

`Promise<number>`: Leaf position of the given state update.

---

putStateUpdateLeafPosition

```
async function putStateUpdateLeafPosition(
  stateUpdateHash: string,
  leafPosition: number
): Promise<void>
```

Description

Sets the leaf position for a given state update within the Merkle Interval Tree of the block in which the state update was included.

Parameters

1. `stateUpdateHash` - `string`: keccak256 hash of the state update.
2. `leafPosition` - `number`: Leaf position for the state update.

Returns

`Promise<void>`: Promise that resolves once the leaf position has been set.

---

getBlockStateUpdateCount

```
async funtion getBlockStateUpdateCount(
  block: number
): Promise<number>
```

Description

Gets the number of state updates that occurred within a given block.

Parameters

1. `block` - `number`: Block to query.

Returns

`Promise<number>`: Number of state updates that occurred within the given block.

---

putBlockStateUpdateCount

```
async function putBlockStateUpdateCount(
  block: number,
  stateUpdateCount: number
): Promise<void>
```

Description

Sets the number of state updates that were included within a given block.

Parameters

1. `block` - `number`: Block to set a count for.
2. `stateUpdateCount` - `number`: Number of state updates included within the specified block.

Returns

`Promise<void>`: Promise that resolves once the state update count has been stored.

---

getStateTreeNode

```
async function getStateTreeNode(
  block: number,
  nodeIndex: number
): Promise<MerkleIntervalStateTreeNode>
```

Description

Queries a node in the state tree.

Parameters

1. `block` - `number`: Block for which to query a node.
2. `nodeIndex` - `number`: Index of the node to query.

Returns

`Promise<MerkleIntervalStateTreeNode>`: The MerkleIntervalStateTreeNode at the given block and node index.

---

putStateTreeNode

```
async function putStateTreeNode(
  block: number,
  nodeIndex: number,
  node: MerkleIntervalStateTreeNode
): Promise<void>
```

Description

Adds a node to the state tree for a given block.

Parameters

1. `block` - `number`: Block to add a state tree node for.
2. `nodeIndex` - `number`: Index of the node to insert.
3. `node` - `MerkleIntervalStateTreeNode`: State tree node to add to the tree.

Returns

`Promise<void>`: Promise that resolves once the node has been inserted into the tree.

---

getAddressTreeNode

```
async function getAddressTreeNode(
  block: number,
  nodeIndex: number
): Promise<MerkleIntervalAddressTreeNode>
```

Description

Gets a node in the address tree of a given block.

Parameters

1. `block` - `number`: Block for which to query an address tree node.
2. `nodeIndex` - `number`: Index of the node to query.

Returns

`Promise<MerkleIntervalAddressTreeNode>`: The MerkleIntervalAddressTreeNode at the given index for the specified block.

---

putAddressTreeNode

```
async function putAddressTreeNode(
  block: number,
  nodeIndex: number,
  node: MerkleIntervalAddressTreeNode
): Promise<void>
```

Description

Sets a node in the address tree of a given block.

Parameters

1. `block` - `number`: Block for which to set an address tree node.
2. `nodeIndex` - `number`: Index of the node in the address tree.
3. `node` - `MerkleIntervalAddressTreeNode`: Node to insert into the tree.

Returns

`Promise<void>`: Promise that resolves once the node has been added to the tree.

CHAPTER 43

HistoryManager

43.1 Description

`HistoryManager` is a wrapper around HistoryDB that provides easy access to specific historical data. `HistoryManager` handles things like generating history proofs, creating inclusion proofs, and inserting historical state updates.

We've separated `HistoryManager` from `HistoryDB` to avoid coupling the underlying data storage with more complex queries.

43.2 API

43.2.1 Methods

getHistoryProof

```
async function getHistoryProof(
  start: number,
  end: number,
  startBlock: number,
  endBlock: number,
  plasmaContract: string
): Promise<HistoryProof>
```

Description

Generates a proof for the history of a given range of state objects. Only returns the history for a specified block range so that a user with a known state doesn't need to receive a significant amount of duplicate information.

Parameters

1. `start` - `number`: Start of the range of state objects to query.
2. `end` - `number`: End of the range of state objects to query.
3. `startBlock` - `number`: Block number to start querying history from.
4. `endBlock` - `number`: Block number to query history to.
5. `plasmaContract` - `string`: Address of the specific plasma contract to query.

Returns

`HistoryProof`: A HistoryProof composed of proof elements that, when applied sequentially, build a valid history for the given range.

---

getInclusionProof

```
function getInclusionProof(
  stateUpdate: StateUpdate
): Promise<InclusionProof>
```

Description

Generates an inclusion proof for a given state update. Creates a proof for both the upper-level address tree and the lower-level state tree. Will throw an error if the client does not have enough information to generate the proof.

Parameters

1. `stateUpdate` - `StateUpdate`: The `StateUpdate` object to generate an inclusion proof for.

Returns

`InclusionProof`: An `InclusionProof` that contains Merkle Interval Tree inclusion proofs for both the address tree and the state tree.

CHAPTER 44

StateDB

44.1 Description

`StateDB` handles storage and modification of the local verified state. `StateDB` is frequently used by StateManager to keep track of the known state and to verify incoming transactions.

44.2 API

44.2.1 Structs

VerifiedStateUpdate

```
interface VerifiedStateUpdate {
  start: number
  end: number
  verifiedBlockNumber: number
  stateUpdate: StateUpdate
}
```

Description

Represents a StateUpdate that has been correctly verified up to a specific block. `verifiedBlockNumber` is updated whenever a exclusion proof implicitly demonstrates that the given state update is valid for `verifiedBlockNumber + 1`.

Fields

1. `start` - `number`: Start of the range for which this state update is still valid.
2. `end` - `number`: End of the range for which this state update is still valid.
3. `verifiedBlockNumber` - `number`: Plasma block number up to which this state update has been verified.
4. `stateUpdate` - `StateUpdate`: Full original state update.

44.2.2 Methods

getVerifiedStateUpdates

```
async function getVerifiedStateUpdates(start: number, end: number): Promise
↪<VerifiedStateUpdate[]>
```

Description

Pulls all `VerifiedStateUpdate` objects such that the range described by `start` and `end` intersects with the `VerifiedStateUpdate`. Intersection **MUST** be computed as start-inclusive and end-exclusive, i.e. `(start, end]`.

Parameters

1. `start` - `number`: Start of the range to query.
2. `end` - `number`: End of the range to query.

Returns

`Promise<VerifiedStateUpdate[]>`: List of `VerifiedStateUpdate` objects that intersect with the given range.

---

putVerifiedStateUpdate

```
async function putVerifiedStateUpdate(
  verifiedStateUpdate: VerifiedStateUpdate
): Promise<void>
```

Description

Adds a new `VerifiedStateUpdate` object to the database. Overwrites, modifies, or breaks apart any existing objects in the database that intersect with the given one.

Parameters

1. `verifiedStateUpdate` - `VerifiedStateUpdate`: The `VerifiedStateUpdate` object to insert into the database.

Returns

`Promise<void>`: Promise that resolves once the object has been added to the database.

CHAPTER 45

StateManager

45.1 Description

`StateManager` primarily handles incoming transactions that modify the current and historical state. `StateManager` effectively acts as a wrapper around StateDB but also makes some calls to HistoryManager.

45.2 API

45.2.1 Data Structures

StateQuery

```
interface StateQuery {
  plasmaContract: string
  predicateAddress: string
  start?: number
  end?: number
  method: string
  params: string[]
  filter?: Expression
}
```

Description

Represents a query for some information about the current state.

Fields

1. `plasmaContract` - `string`: Address of the plasma contract to query. Clients may track multiple plasma contracts, so this parameter is necessary to resolve the correct data.
2. `predicateAddress` - `string`: Address of the predicate to query.
3. `start?` - `number`: Start of the range to query. If not provided, will default to the 0.
4. `end?` - `number`: End of the range to query. If not provided, will default to the max range value.
5. `method` - `string`: Name of the method to call.
6. `params` - `string[]`: List of parameters to the call.
7. `filter?` - `Expression`: An Expression to use to filter results. May be omitted to return all results.

StateQueryResult

```
interface StateQueryResult {
  stateUpdate: StateUpdate
  result: string[]
}
```

Description

Element of the list of results returned when a client makes a state query.

Fields

1. `stateUpdate` - `StateUpdate`: `StateUpdate` object to which the result pertains.
2. `result` - `string[]`: Result values of the query corresponding to the output values described in the Predicate API.

45.2.2 Methods

executeTransaction

```
async function executeTransaction(
  transaction: Transaction
): Promise<{ stateUpdate: StateUpdate, validRanges: Range[] }>
```

Description

Executes a **'transaction'_** against the current verified state and returns the resulting state update.

Transactions reference a range of state objects that they're attempting to modify. It's possible that we only have the full valid history for some of the referenced state objects but not others. This behavior is allowed by construction. As a result, this method also returns the list of ranges over which the transaction can be considered "valid".

For example, we may have a valid history for the ranges `(0, 50)` and a transaction that sends `(0, 100)`. We can assert that the transaction is valid for `(0, 50)` but cannot make the same assertion for `(50, 100)`.

Parameters

1. `transaction` - `Transaction`: **'Transaction'_** to execute against the verified state.

Returns

`Promise<{ stateUpdate: StateUpdate, validRanges: Range[] }>`: The StateUpdate created as a result of the transaction and the list of ranges over which the state update has been validated.

---

ingestHistoryProof

```
async function ingestHistoryProof(
  historyProof: HistoryProof
): Promise<void>
```

Description

Validates a given `HistoryProof`, which consists of elements that are either deposits ("Deposit Proof Elements"), transactions ("State Update Proof Elements"), or state updates that prove a given range was *not* included in a specific block ("Exclusion Proof Elements").

Parameters

1. `historyProof` - `HistoryProof`: A `HistoryProof` to validate.

Returns

`Promise<void>`: Promise that resolves once the proof has been applied or rejected.

---

queryState

```
async function queryState(query: StateQuery): Promise<StateQueryResult[]>
```

Description

Performs a query on the local state.

Parameters

1. `query` - `StateQuery`: A *StateQuery* object with information about what state to query.

Returns

`Promise<StateQueryResult[]>`: A *StateQueryResult* object for each state update that passed the filter provided in the query.

CHAPTER 46

SyncDB

46.1 Description

`SyncDB` is used by the SyncManager to store information about current synchronization statuses.

46.2 API

46.2.1 Methods

getCommitmentContracts

```
async function getCommitmentContracts(): Promise<string[]>
```

Description

Gets the list of commitment contracts the client is currently watching.

Returns

`Promise<string[]>`: List of commitment contract addresses to synchronize with.

getDepositContracts

```
async function getDepositContracts(
  commitmentContract: string
): Promise<string[]>
```

Description

Gets the list of deposit contracts that are connected to a given commitment contract.

Parameters

1. `commitmentContract` - `string`: Address of the commitment contract to get deposit contracts for.

Returns

`Promise<string[]>`: List of deposit contract addresses connected to a given commitment contract.

addDepositContract

```
async function addDepositContract(
  commitmentContract: string,
  depositContract: string
): Promise<void>
```

Description

Connects a deposit contract to a commitment contract.

Parameters

1. `commitmentContract` - `string`: Address of the commitment contract to connect to.
2. `depositContract` - `string`: Address of the deposit contract to connect.

Returns

`Promise<void>`: Promise that resolves once the contracts have been connected.

removeDepositContract

```
async function removeDepositContract(
  commitmentContract: string,
  depositContract: string
): Promise<void>
```

Description

Removes a connection between a deposit contract and a commitment contract.

Parameters

1. `commitmentContract` - `string`: Commitment contract to remove the connection from.
2. `depositContract` - `string`: Deposit contract to remove.

Returns

`Promise<void>`: Promise that resolves once the contracts have been disconnected.

putLastSyncedBlock

```
async function putLastSyncedBlock(
  plasmaContract: string,
  block: number
): Promise<void>
```

Description

Sets the last block up to which the client has synchronized with a given plasma contract.

Parameters

1. `plasmaContract` - `string`: ID of the plasma chain to set last block for.
2. `block` - `number`: Last block up to which the client has synchronized.

Returns

`Promise<void>`: Promise that resolves once the block has been set.

getLastSyncedBlock

```
async function getLastSyncedBlock(plasmaContract: string): Promise<number>
```

Description

Gets the last block up to which the client has synchronized with a given plasma contract.

Parameters

1. `plasmaContract` - `string`: Contract to query the last synced block for.

Returns

`Promise<number>`: Block up to which the client has synchronized.

addSyncQuery

```
async function addSyncQuery(
  plasmaContract: string,
  stateQuery: StateQuery
): Promise<void>
```

Description

Adds a StateQuery to the list of queries to execute for a given plasma contract.

Parameters

1. `plasmaContract` - `string`: Contract to add a query for.
2. `stateQuery` - `StateQuery`: Query to add for the contract.

Returns

`Promise<void>`: Promise that resolves once the query has been added.

removeSyncQuery

```
async function removeSyncQuery(
  plasmaContract: string,
  stateQuery: StateQuery
): Promise<void>
```

Description

Removes a StateQuery from the list of queries to execute for a given plasma contract.

Parameters

1. `plasmaContract` - `string`: Contract to remove a query for.
2. `stateQuery` - `StateQuery`: Query to remove for the contract.

Returns

`Promise<void>`: Promise that resolves once the query has been removed.

getSyncQueries

```
async function getSyncQueries(
  plasmaContract: string
): Promise<StateQuery[]>
```

Description

Returns the StateQuery objects to execute for a given plasma contract.

Parameters

1. `plasmaContract` - `string`: Contract to get sync queries for.

Returns

`Promise<StateQuery[]>`: List of queries to execute for a given contract.

CHAPTER 47

SyncManager

47.1 Description

`SyncManager` runs in the background and automatically synchronizes the client's state with the operator's state. It watches for new block submissions and queries the operator for any relevant state updates.

47.2 API

47.2.1 Methods

addDepositContract

```
async function addDepositContract(
  commitmentContract: string,
  depositContract: string
): Promise<void>
```

Description

Connects a deposit contract to a commitment contract.

Parameters

1. `commitmentContract` - `string`: Address of the commitment contract to connect to.
2. `depositContract` - `string`: Address of the deposit contract to connect.

Returns

`Promise<void>`: Promise that resolves once the contracts have been connected.

removeDepositContract

```
async function removeDepositContract(
  commitmentContract: string,
  depositContract: string
): Promise<void>
```

Description

Removes a connection between a deposit contract and a commitment contract.

Parameters

1. `commitmentContract` - `string`: Commitment contract to remove the connection from.
2. `depositContract` - `string`: Deposit contract to remove.

Returns

`Promise<void>`: Promise that resolves once the contracts have been disconnected.

getLastSyncedBlock

```
async function getLastSyncedBlock(plasmaContract: string): Promise<number>
```

Description

Gets the last block up to which the manager has synchronized for a given plasma chain.

Parameters

1. `plasmaContract` - `string`: ID of the plasma chain to get the last synced block for.

Returns

`Promise<number>`: Block up to which the manager has synchronized.

addSyncQuery

```
async function addSyncQuery(
  plasmaContract: string,
  stateQuery: StateQuery
): Promise<void>
```

Description

Adds a **‘StateQuery‘_** to the list of queries to call on a specific plasma chain when the synchronization loop triggers. Necessary because different predicates can be parsed in different ways and the manager needs to know what to look for.

Parameters

1. `plasmaContract` - `string`: ID of the plasma contract to add a query for.
2. `stateQuery` - `StateQuery`: **‘StateQuery‘_** to add for that contract.

Returns

`Promise<void>`: Promise that resolves once the query has been added.

removeSyncQuery

```
async function removeSyncQuery(
  plasmaContract: string,
  stateQuery: StateQuery,
): Promise<void>
```

Description

Removes a **‘StateQuery‘_** to the list of queries to call on a specific plasma chain when the synchronization loop triggers.

Parameters

1. `plasmaContract` - `string`: ID of the plasma contract to remove a query for.
2. `stateQuery` - `StateQuery`: **‘StateQuery‘_** to remove for that contract.

Returns

`Promise<void>`: Promise that resolves once the query has been added.

getSyncQueries

```
async function getSyncQueries(plasmaContract: string): Promise<StateQuery[]>
```

Description

Returns the list of active **'StateQuery'_** objects the manager is using when the synchronization loop triggers.

Parameters

1. `plasmaContract` - `string`: Contract to get active queries for.

Returns

`Promise<StateQuery[]>`: A list of **'StateQuery'_** objects the manager is using.

CHAPTER 48

RpcClient

48.1 Description

`RpcClient` is simple client wrapper for any JSON RPC server. `RpcClient` can be used to make requests to the operator or to any other client. A list of available RPC methods are specified separately.

48.2 API

48.2.1 Methods

send

```
async function send(request: JsonRpcRequest): Promise<JsonRpcResponse>
```

Description

Sends a JSON RPC request to the client's specified server and returns the corresponding JSON RPC response.

Parameters

1. `request` - `JsonRpcRequest`: A JSON RPC request object to send to the server.

Returns

`JsonRpcResponse`: A JSON RPC response object sent back by the server.

CHAPTER 49

RpcServer

49.1 Description

Each client **MUST** provide a JSON RPC server so that other clients can interact with it. `RpcServer` is a standard module that exposes a JSON RPC server over HTTP. A list of required RPC methods are specified separately.

49.2 API

49.2.1 Methods

serve

```
async function serve(): Promise<void>
```

Description

Starts the server.

Returns

`Promise<void>`: Promise that resolves once the server has started.

close

```
async function close(): Promise<void>
```

Description

Shuts down the server.

Returns

`Promise<void>`: Promise that resolves once the server has been shut down.

CHAPTER 50

Introduction

CHAPTER 51

BlockDB

51.1 Description

Todo: Add description for BlockDB.

51.2 API

51.2.1 Methods

putStateUpdate

Description

Parameters

Returns

CHAPTER 52

BlockManager

52.1 Description

Todo: Add description for block manager.

52.2 API

52.2.1 Methods

getNextBlockNumber

```
async function getNextBlockNumber(): Promise<number>
```

Description

Gets the number of the **next** block to be published.

Returns

`Promise<number>`: Number of the next block to be published.

enqueueStateUpdate

```
async function enqueueStateUpdate(
  stateUpdate: StateUpdate
): Promise<void>
```

Description

Adds a state update to the queue of updates to be added to the next block.

Parameters

1. `stateUpdate` - `StateUpdate`: State update to add to the queue.

Returns

`Promise<void>`: Promise that resolves once the state update has been added to the queue.

---

getPendingStateUpdates

```
async function getPendingStateUpdates(): Promise<StateUpdate[]>
```

Description

Gets the list of state updates that are pending for inclusion in the next block.

Returns

`Promise<StateUpdate[]>`: List of state updates queued for inclusion in the next block.

---

submitNextBlock

```
async function submitNextBlock(): Promise<void>
```

Description

Merklizes the next block and sends the header to Ethereum.

Returns

`Promise<void>`: Promise that resolves once the block has been submitted.

CHAPTER 53

OperatorStateManager

53.1 Description

Todo: Add description for OperatorStateManager.

53.2 API

53.2.1 Methods

ingestTransaction

```
async function ingestTransaction(
  transaction: Transaction
): Promise<string>
```

Description

Ingests an incoming transaction and enqueues it for inclusion in the next block.

Parameters

1. `transaction` - `Transaction`: The Transaction to ingest.

Returns

`Promise<string>`: Receipt for the given transaction.

CHAPTER 54

SimpleOwnership Predicate

54.1 Overview

The ownership predicate is the one of the simplest useful predicates. It gives an address, specified as the `state.data.owner` , the ability to execute a state transition which changes any part of the owned subrange to a new `StateObject` by signing an approval transaction.

54.2 Predicate API

54.2.1 getOwner

```
{
      name: "getOwner",
      constant: true,
      inputs: [],
      outputs: [
          {
              name: "stateOwner",
              type: "address"
          }
      ]
}
```

Description

The `getOwner` API method allows the client or operator to query the current owner of the state.

Inputs

N/A

Outputs

1. *stateOwner* - *address*: the current owner based on the state object. Will equal `data.owner`.

Justification

This function allows developers to get the owner without directly dissecting the state object.

54.2.2 send

```
{
      name: "send",
      constant: false,
      inputs: [
          {
              name: "newStateObject",
              type: "StateObject"
          },
          {
              name: "originBlock",
              type: "uint"
          }
      ],
      outputs: []
}
```

Description

The `send` method is used to set the state to a new arbitrary state object, given a signature.

Inputs

1. `newStateObject` - `StateObject` : the state object that the owner desires to mutate to.
2. `originBlock` - `uint` : the maximum plasma blocknumber of the ownership `StateUpdate` s from which you are spending.

Outputs

N/A

Justification

Being able to spend to any new state is the base property of ownership. The `targetBlock` may be used to produce replay protection while allowing some level of asynchronicity between the client and operator.

54.3 State Object Specification

```
struct ownershipStateData:
  owner: address
```

54.3.1 Fields

1. *owner* - *address*: The Ethereum public address of the person who may mutate the state.

54.4 Additional Exit Game Logic

N/A

54.5 Predicate Contract Logic

54.5.1 Transition Execution

```
def verifyTransaction(preState: StateUpdate, transaction: Transaction, witness: bytes␣
↪postState: StateUpdate)
```

Requirements

1. **MUST** ensure that the `witness` is a signature by the `preState.stateObject.owner` on the `transaction`.
2. **MUST** ensure that the `preState.plasmaBlockNumber` is less than the `input.parameters.originBlock`.
3. **MUST** ensure that the `postState.range` is the same as `transaction.start` and `transaction.end`.
4. **MUST** ensure that the `transaction.parameters.newState` is the same as the `postState.state`.

Rationale

These conditions allow a signature by the sender to approve only a single output state.

Exit Finalization Logic

```
def onFinalizeExit(owner: address, ERC20Contract: address, amount: uint256)
```

Parameters

1. `owner` - `address`: the owner of the exit.
2. `ERC20Contract` - `address`: The ERC20 contract the ownership is of.
3. `amount` - `uint256`: the amount of the ERC20 token being redeemed.

Description

This function is called internally by the predicate when it needs to handle an exit, whether as a limbo target or as a regular exit.

Requirements

1. **MUST** only allow this method to be called internally.
2. **MUST** Send the total `amount` to the `owner` .

Rationale

The owner of a range gets all of the assets it corresponds to.

54.6 Limbo Exit Logic

As with all predicates in this spec, the ownership predicate supports limbo exit functionality. As such, it **MUST** fulfill all the methods and requirements outlined in the limbo standard outlined in the contracts section of this spec. Additional requirements for some of the methods in this predicate are shown below. They are quite simple as the ownership predicate is mostly pure functions.

```
function onTargetedForLimboExit(
    Checkpoint _sourceExit,
    StateUpdate _limboTarget
) public
```

N/A–just return!

There's no custom exit logic for the ownership predicate if it's a limbo exit, so no additional functionality needed.

```
function canReturnLimboExit(
    Checkpoint _limboSource,
    StateUpdate _limboTarget
    bytes _witness
) public returns (bool)
```

The `_witness` is be unused for this predicate.

- **MUST** ensure that the tx.origin is the `state.data.owner` of the `limboTarget`

We require the target owner's permission to return a limbo exit.

```
function finalizeExit(
    Checkpoint _exit
) public
```

- **MUST** fulfill the generic requirements for `finalizeExit`.
- **MUST** make an internal call to `onFinalizeExit` to send the total amount to the `_exit.stateUpdate.owner`.

The finalization logic for limbo and non-limbo exits remains the same.

```
function onFinalizeTargetedExit(
    Checkpoint _exit,
    StateUpdte _target
) public
```

Logic for the target of a limbo exit to handle the exit's finalization.

- **MUST** make an internal call to `onFinalizeExit` to send the total amount to the `_target.stateUpdate.owner`.

The finalization logic for limbo and non-limbo exits remains the same.

54.7 Verification Plugin

54.7.1 State Transitions

```
def executeStateTransition(preState: StateUpdate, transaction: StandardTransaction)
```

Requirements

1. **MUST** ensure that the `transaction.witness` is a signature by the `preState.stateObject.owner` .
2. **MUST** ensure that the `preState.plasmaBlockNumber` is less thana the `input.parameters.originBlock` .
3. **MUST** return a `StateUpdate` with a range the same as `transaction.start` and `transaction.end` .
4. **MUST** return a `StateUpdate` with `state` is the same as the `transaction.parameters.newState` .

Rationale

These steps always produce a `StateUpdate` which passes the predicate contract's `verifyStateTransition` step.

54.8 Guarding Plugin

TODO