Abstract

EVM (Ethereum Virtual Machine) provides such a powerful turing-complete computation so that ethereum can run a general program, also known as a smart contract. Plasma EVM is a fully generalized Plasma that can execute EVM in Plasma chain, and it is based on current Ethereum clients (go-ethereum, py-evm, parity). We propose state-enforceable Plasma construction to guarantee that only valid state can be finalized in root chain with a TrueBit-like verification game, providing a way to enter and exit account storage between two chains. This can be done because both chains have identical architecture. Plasma EVM can improve decentralization, performance, stability, and usability of Dapps (Decentralized Applications), by migrating from the current Ethereum chain into the Plasma chain.

1 Introduction

Plasma EVM consists largely of RootChain contract, Operator, User, and Requestable contract. The operator operates a child chain, mines blocks, and submits them to RootChain contract, which manages the Plasma EVM, and users receive block data from the operator and can run individual nodes. Requestable contract is a contract to apply Request which enforces state transition of the child chain. Users generate Request to move the state of Requestable contract between the root chain and the child chain, and the operator applies Request to the state of the child chain by mining blocks. Applying Request is enforced by RootChain contract and it can be Challenged if it is incorrectly applied. Plasma EVM also resolves data unavailability through Continuous Rebase.

2 Related Works

Plasma MVP Plasma MVP is the first Plasma implementation since 1. It uses transactions with only two outputs and UTXO (Unspent transaction output) to represent the state of the child chain as in the name 'Minimal Viable Plasma'. To deal with invalid block and block withholding attacks, users make a confirmation signature to finalize transactions only if their transaction is correctly
included in the block. Because the finality of each UTXO is guaranteed through confirmation signature, users can safely exit their UTXOs from the child chain to the root chain through Exit Game.

**Plasma Cash** Plasma Cash represents the state (UTXO) of the child chain using the Sparse Merkle Tree (SMT) to replace the confirmation signature, which was the disadvantage of Plasma MVP. Using inclusion-exclusion proof of SMT, Exit Game can prevent an attacker from attempting to exit 1) with unowned UTXO, 2) double-spent UTXO, or 3) when UTXO is based on a double-spending UTXO.

**More Viable Plasma** In Plasma MVP, the criteria for the exit priority were determined by UTXO’s age of output. More Viable Plasma proved that they can build the existing valid Exit Game schema excluding the confirmation step by changing the exit priority rule from the age of output of UTXO to the age of the youngest input.

**Plasma Leap** Plasma Leap is a partially generalized Plasma implemented using Spending Conditions, which breaks down smart contracts into smaller programs, and Non-fungible storage token (NST), which stores the state. It uses the Truebit-like verification game, as Plasma EVM does, to verify the correct execution of Spending Conditions. Also, it is easy to resolve data unavailability because it is still possible to use Exit Game of More Viable Plasma. However, unlike Plasma EVM, it is not possible to provide the same level of functionality as Ethereum because it reduces the state representation, from account-based to UTXO-based, and executable EVM opcodes.

### 3 Plasma EVM

Plasma EVM client is based on existing Ethereum clients, and most of the mechanisms are the same as them. The data structure of child chains such as block, transaction, and receipt is identical to Ethereum. However, there are some changes in order to apply generalized Plasma protocols, unlike the existing Ethereum.

**Rootchain Network Assumption** In Ethereum, PoW(Proof of Work) can cause problems such as chain reorganization or network congestion. However, this article assumes that the Ethereum network is not in this situation.

#### 3.1 3 Types of Block and Epoch

Plasma EVM has different types of blocks, including *Non-Request Block*, *Request Block* and *Escape Block*. *(NRB)* is a common block that includes user’s normal transactions. *NRB* is the same as the block in Ethereum, Bitcoin, and other blockchains.
Request Block (RB) is the block for applying Requests to the child chain. Users create Requests and the operator generates RB and applies these Requests to the child chain. Because the RootChain contract specifies transactionsRoot for that RB, it can enforce which transactions should be included in RB.

Escape Block (EB) is a kind of RB for users to protect themselves from block withholding attacks by the operator. For more information of Escape and Rebase, see 3.5 Continuous Rebase.

Epoch is a period of several blocks. Depending on the type of block, Epoch is divided into Non-Request Epoch (NRE), Request Epoch (RE), and Escape Epoch (EE). The length of Epoch is the number of blocks included in each Epoch. The length of NRE is a predefined constant, so it cannot be changed. However, for RE and EE, the length is variable so that it can be increased or decreased according to how many users create Request or Escape request. However, the Genesis block is included in the 0th NRE and the length of NRE#0 is always 1.

RootChain contract enforces state transition of the child chain by specifying which type of Epoch should be placed. It causes ordinary transactions of the child chain to be included through NRE. If Request is generated, it places RE after NRE, and if Escape request is generated, it enforces that existing NRE and RE are Rebased after EE.

3.2 Block Mining

RootChain contract can apply transactions in the child chain and requests generated in root chain by placing Epoch in the following order:
1. NRE#1 is placed after NRE#0.
2. After NRE#N, RE#(N+1) is always placed. Similarly, after RE#N, NRE#(N+1) is placed.
3. Request generated when current Epoch is NRE#N or RE#(N+1), is applied in RE#(N+3). If the Request is not generated, the length of the RE becomes 0.

The state of RootChain contract changes to Accept NRB and Accept RB in order to receive two types of block like above. In Accept NRB, only Non-Request Block can be submitted and only Request block can be submitted in Accept RB. The operator must submit Non-Request Block or Request block according to the state of RootChain contract.

Figure 1 shows how the state of RootChain contract changes. However, the content covered here does not include Continuous Rebase of the Plasma EVM. The overall operation process is covered later in 3.5 Continuous Rebase.

3.3 Block Submission

The operator should submit three Merkle roots, stateRoot, transactionsRoot and receiptsRoot for each block. However, there are situations in which it is possible to submit only transactionsRoot, further details are covered in 3.5 Continuous Rebase. If computation of state transition is not executed properly and an incorrect Merkle roots have been submitted, the invalid block cannot be
Figure 1: Simplified state transition of RootChain contract

finalized by Computation Challenge. It is similar to TrueBit’s verification game, and resolved using preStateRoot and postStateRoot to verify the State transition function for each block.

3.4 Apply Request

Request is created by users sending transactions to RootChain contract. Enter request and Exit request are included in RB. Escape request and Undo request are included in EB. Enter request changes state in the root chain and then it is included in block in form of Request transaction. For Exit, Escape, and Undo request, they are included in the block in the form of Request transaction, and then applied in the root chain. If the Request transaction has been reverted, Exit Challenge can prevent it from being applied in the root chain.

Contract, which can apply Request, is Requestable contract. It is possible to apply Request in both chains by calling a specific function of Requestable contract. By implementing Requestable interface for each contract, the general-purpose computing platform oriented by Ethereum can be 'Plasmafied'.

The operator must map the addresses of Requestable contracts in each chain so that RootChain contract can create Request. Requestable contracts in each chain must have the same codeHash, which means that both contracts have identical storage layout.

3.4.1 Request and Request Transaction

Request consists of following 4 parameters.

- requestor: account who generated Request
- to: an address of Requestable contract deployed on the root chain
- trieKey: the identifier of Request
- trieValue: the value of Request
It is possible to create each Request using startEnter, startExit, startEscape, and startUndo functions of RootChain contract. All four functions have the same parameters of to, trieKey, and trieValue.

Request transaction is included in Request block and is generated with Null Address as Transactor so that anyone can mine that block. Null Address (NA) has no private key and the address is 0x00. Request transaction consists of following 5 parameters.

- sender: NA
- to: an address of Requestable contract deployed on the child chain
- value: 0
- function signature: a specific function signature of Requestable contract
- parameters: parameters for calling the function

RootChain contract can enforce RB to include Request by computing the hash of Request transactions and transactionsRoot of RB. Users can also get Request data from RootChain contract, so the operator cannot withhold RB data. This is the same for EB.

### 3.4.2 Requestable Contract

The Requestable interface has the following functions, and the Requestable contract implements the Requestable interface below.

```solidity
interface Requestable {
    function applyEnter(bool isRootChain, uint256 requestId, address requestor, bytes32 trieKey, bytes trieValue) external returns (bool success);
    function applyExit(bool isRootChain, uint256 requestId, address requestor, bytes32 trieKey, bytes trieValue) external returns (bool success);
    function applyEscape(bool isRootChain, uint256 requestId, address requestor, bytes32 trieKey, bytes trieValue) external returns (bool success);
    function applyUndo(bool isRootChain, uint256 requestId, address requestor, bytes32 trieKey, bytes trieValue) external returns (bool success);
}
```

An variable of Requestable contract that can be changed by Request is Requestable variable. Not all variables in Requestable contract need to be Requestable. This is because some variables do not
require any feature related to Request. For a particular variable, it needs to check the permission of Request according to who the requestor is. For example, if Request is available for someone else’s token balance, this would not be desirable. Therefore, we included specific logic in Requestable contract to solve these issues in advance. This is possible because trieKey can be used to identify the permission of Request for specific variables.

Depending on the type of Request, Request transaction has a different function signature. See also B Requestable Contract Example for an example of Requestable contract implementation.

3.4.3 Apply Enter Request

RootChain contract applies Enter request to the Requestable contract deployed on each chain as follows:

1. The user sends a transaction calling RootChain.startEnter() to RootChain contract.
2. RootChain contract applies Enter request to Requestable contract in the root chain. Enter request will not be generated if the transaction is reverted in this process.
3. If step 2 has been processed successfully, the RootChain contract records Enter request.
4. In Request epoch, the operator mines Request Block including Request transaction.
5. Request transaction changes the state of the child chain according to Enter request.

![Figure 2: Enter Diagram](image)

3.4.4 Apply Exit Request

RootChain contract applies Exit request to the Requestable contract deployed on each chain as follows:

1. User sends transaction calling RootChain.startExit() to RootChain contract.
2. Unlike Enter request, Exit request is immediately recorded and included in RB in the form of Request transaction.

3. After the Challenge period of RB has ended, Challenge period of Exit request is started. If the Request transaction in step 2 is reverted, anyone can execute Exit Challenge to it by calling the RootChain.challengeExit() function.

4. After Challenge in step 3, Exit request is Finalized by calling RootChain.finalizeRequest(). It will apply Exit request to Requestable contract deployed on the root chain.

Figure 3: Exit Diagram

3.4.5 Apply Escape and Undo request

Escape request and Undo request are processed the same as Exit request, but Request transaction is included in Escape Block instead of Request Block.

3.5 Continuous Rebase

Plasma EVM sets Rebase\(^1\) as default behavior. It makes the Plasma chain apply continuously and periodically Escape request. It allows users to safely escape based on the last finalized block by submitting an Escape request if the block data is withheld.

\(^1\)It is named after git’s rebase.
3.5.1 Rebase

Rebase is re-mining blocks based on another block. In other words, transactions are re-executed based on other parent block. Even After Rebase, transactionsRoot of block remains to be unchanged. However, stateRoot and receiptsRoot can be changed because the specific results of each transaction may vary.

Rebased Non-Request Epoch (NRE') and Rebased Request Epoch (RE') are Epochs which Rebase existing NRE and ORE. Similarly, Rebased Non-Request Block (NRB') and Rebased Request Block (RB') are Rebased blocks.

3.5.2 Cycles and Stages

Cycle is an entire operational period of the Plasma EVM. Cycle consists of a total of four Stages. The Stages are Pre-commit, DA-check, Commit, and Challenge in that order. When all Stages are completed in sequence, the Cycle is Finalized. See Figure 5 to know how they work.
3.5.3 Pre-commit

Pre-commit is a kind of preliminary submission step, and the operator mines blocks as described in 3.2 and submits transactionsRoot of those blocks to the root chain. The operator also propagates those blocks to users at the same time. If the operator sends incorrect block data at the Pre-commit stage, or if the entire block data is not fully transferred, users can submit an Escape request. However, for RB, the transactionsRoot is determined by RootChain contract so it is not necessary to submit it to the root chain.

The reason why only transactionsRoot is submitted to the root chain is because stateRoot and receiptsRoot are determined later in the Rebase stage. In addition, the reason why the operator must submit the transactionsRoot is because users can know the correct stateRoot of the block if they know which transactions are to be included in that block. Thus, users who receive block data that matches the transactionsRoot submitted to RootChain contract, can challenge even if the operator withheld the data and submitted the wrong stateRoot.

![Figure 6: Pre-commit](image)

3.5.4 DA-check

DA-check is a stage in which users check data availability in Pre-commit and the size of the transaction. The reason why users should check the size is that transactions whose size exceeds block gas limit of the root chain cannot be verified. If the operator is determined to be byzantine during DA-check, users must escape from that chain to protect themselves.

1) Submit Escape request  Escape request is a means that allows users to safely escape from the Plasma chain in case of block withholding attack. Its structure and feature are the same as Exit request, but there is a difference in that it is always processed based on the last block of the Commit stage of the previous Cycle. This is to ensure that users are always able to escape from the chain based on the final state of the last Cycle. If users did not escape from the previous Cycle, it means that those users received all the data of the previous Cycle correctly. This means that they can know what the correct state of the Cycle is.

2) Submit Undo request if users enter that Cycle  Because Escape request is processed based on the last block of the previous Cycle, it cannot rollback Enter request of the current Cycle. For this reason, users must submit Undo request, which cancels Enter request of the current Cycle, together with Escape request. Undo request prevents Enter request from being applied in the child chain.
After the *Undo request* is finalized, users can undo the state transition of the *Enter request* in the root chain.

3) **Cancel Exit request if users make Exit request for that Cycle**  If *Exit request* is executed again after all states have been exited with *Escape request*, the *Request* can be reverted and challenged with *Exit challenge*. As a result, users who submitted *Escape request* should cancel all *Exit requests* of that *Cycle* in *RootChain contract*. Cancelled *Exit request* cannot be subject to *Exit Challenge* regardless of its execution result. In addition, cancelled *Exit requests*, even if finalized, cannot cause corresponding state transition in the root chain. Thus, the canceled *Exit request* may result in no inappropriate gain or loss for users.

The reason why the *DA-check* stage should be placed explicitly is that operator may not transfer data correctly at the *Pre-commit* stage to attack users. For example, if the operator withholds the block data at the end of a *Pre-commit* stage, it provides enough time for users to recognize it and take appropriate action. Users can immediately submit an *Escape request* without waiting for the *DA-check* stage if the block data is withheld from the beginning of the *Pre-commit* stage.

![Figure 7: DA-check](image)

3.5.5 **Commit**

*Commit* is the stage in which the operator submits *Escape block*(*EB*) applying *Escape request* and *Undo request* submitted in *Pre-commit* and *DA-check*, *Rebasing* all the blocks submitted in the *Pre-commit* stage based on the *EB* and submitting *stateRoot*, *transactionsRoot* and *receiptsRoot*.

At this time, the parent block of *EB* is the last committed block of the previous *Cycle*. However, if none of the *Escape request* is submitted, submission of *EB* and *Rebase* is skipped and the operator just submits *stateRoot* and *receiptsRoot* of mined blocks in *Pre-commit* stage.

![Figure 8: Commit](image)
3.5.6 Challenge

In the Challenge stage, users can submit a Challenge for which the blocks submitted in Commit are invalid. However, the available Challenge is Request Challenge, Null Address Challenge, and Computation Challenge except Exit Challenge.

If the number of Challenges submitted is large, the other Challenges will be canceled as soon as the challenger wins. Besides, if the challenger wins, all subsequent blocks including that Cycle will be canceled, and that Cycle will go back to the DA Check stage.

![Figure 9: Challenge](image)

3.6 Finalize

If there is no successful challenge, the corresponding Cycle will be Finalized, and all blocks submitted in Commit will be Finalized simultaneously. However, the Challenge stage will not be completed until all submitted challenges are completed, and all subsequent Cycles, including challenged Cycle, will not be Finalized.

3.7 Stage Length

Stage length is the number of Epoch to be processed in Pre-commit and Commit. Pre-commit length is $N_{NRE} \times 2$ and Commit length is $N_{NRE} \times 2 + 1$. Additional one Epoch in Commit length means Escape Epoch. Pre-commit length and Commit length are determined by the operator when the RootChain contract is deployed.

3.8 Stage Period

Stage period is the time allocated for each Stage. Pre-commit period and Commit period are the criteria for fulfilling Halting condition to be covered below. DA-check period and Challenge period is simply the time to process each Stage.

All Stage periods are determined by the operator when the RootChain contract is deployed. Pre-commit and Commit can be completed regardless of corresponding Stage period. However, DA-check and Challenge is completed only after corresponding Stage period. For Challenge period, it can be extended depending on the progress of Computation Challenge.
3.9 Halting Condition

*Halting condition* is fulfilled if the operator does not properly perform the prescribed procedures of each *Stage* of *Cycle* for any reason. The *Stage* where the *Halting condition* can be fulfilled is *Pre-commit*, *Commit*, and the specific conditions and results are as follows.

**Pre-commit** When the operator fails to submit all *Epochs of Pre-commit length* within *Pre-commit period*, the *Halting condition* is fulfilled. In this case, the chain is in *Shutdown*.

**Commit** When the operator fails to submit all *Epochs of Commit length* within *Pre-commit period*, the *Halting condition* is fulfilled. This will suspend progress of all subsequent *Cycles*, including the corresponding *Cycle*. If the *EB* has not yet been submitted, anyone can submit *EB*. After *EB* is submitted, the suspended *Cycle* can be resumed. If it is halted in the progress of *Rebase*, the chain will be in *Shutdown*.

**Shutdown** *Shutdown* is a kind of Plasma chain closing procedure. When *Shutdown* is started, no further *Cycle* in that chain can proceed. *Cycle* can only repeat following procedure: Submit *Escape request* based on last finalized *Cycle* - Submit *EB* - *Challenge*, to ensure that all users can safely exit from the chain. In the *Shutdown*, anyone can submit *EB*, which allows users to exit for themselves.

The operating process of one *Cycle* discussed so far can be represented by Figure 10.

![Figure 10: Continuous rebase (Sequential cycle)](image-url)
3.10 Overlapping of Cycles

What we have discussed so far is the operational process of only one Cycle. When several Cycles are overlapped, the Pre-commit of the following Cycle can be started, regardless of whether the Challenge of previous Cycle is completed or not.

In Figure 11, as soon as the Pre-commit of Cycle2 is completed, DA-check of Cycle2 and Pre-commit of Cycle3, the next Cycle, is started. At this point, the first block of Pre-commit of Cycle3 is computed based on the last Pre-commit block of Cycle2. In order for Cycles to be overlapped together like this, the following conditions must be fulfilled:

1. As soon as the Pre-commit of the current Cycle is completed, the Pre-commit of next Cycle must begin.
2. As soon as Pre-commit of the current Cycle is completed, DA-check of the current Cycle must start.
3. The Commit of the current Cycle must start only after completing the Commit of pre-Cycle.

These rules will allow users to continue to use the chain if the operator normally operates the child chain. Note that further Cycles can be progressed regardless of the completion of Challenge stage in the last Cycle. However, even if a Challenge is completed for a subsequent Cycle, it cannot be Finalized until the previous Cycle is Finalized.

![Figure 11: Overlapping of Cycles](image)

3.11 Finality

Block Plasma EVM can specify the checkpoints of blocks like Plasma XT, for blocks that have passed the challenge period without any valid challenge.
Apart from finality of blocks, there is also finality for all Requests except Enter requests. When a block containing a Request is Finalized, Request challenge period which is challenge period for that Request begins. If no valid challenge is submitted during this time, the Request will be Finalized.

4 Challenge

The block and Exit request can be verified with the following Challenges.

4.1 Null Address Challenge

Null address challenge checks for transactions with the Transactor of NA in NRB. Such transactions mean Request transaction, so if they are included in NRB instead of RB and EB, the operator is byzantine.

4.2 Exit Challenge

If an Exit request and Escape request is invalid, the corresponding Request transaction will be reverted, and also the Request must be deleted from the RootChain contract. The Exit challenge verifies invalid Exit request with the reverted Request transaction as proof. However, before starting challenge for Exit request, the block including that Request must be Finalized. This is because Request transactions that are not reverted in valid block can be reverted in invalid block. Therefore, Exit Challenge can start after the block including target Request is Finalized.

4.3 Computation Challenge

For the NRB, RB and EB, Computation Challenge verifies that the operator has executed transactions correctly. If the operator submits an invalid stateRoot, he can be challenged through TrueBit-like Verification game with blockData, preStateRoot, and postStateRoot.

\[
\text{preState} = \text{commitedStateRoots}[i - 1]
\]

\[
\text{postState} = \text{commitedStateRoots}[i] = \text{STF}_{\text{block}}(\text{preState}, \text{Block}_i)
\]

By comparing the output of STF_{block} executed in RootChain contract with the output submitted previously, the state transition of the block can be verified.

4.4 Verification Game

TrueBit proposed a verification game as a way to verify the output of outsourced computation. Its final step uses a method of executing computation once in Ethereum and comparing the actual output with the expected output. We will verify the computation with solEVM 12, a smart contract that runs EVM inside the EVM that has been implemented by Ohalo and Parsec Labs. However,
to use EVM Compatible Transaction Fee (GAS) Delegated Execution Architecture, this must be adopted to solEVM.

5 Attack Scenarios

5.1 Block Withholding Attack

If the operator withholds block data in Pre-commit, users can submit an Escape request in Pre-commit and DA-check to safely exit from the chain.

5.2 Invalid Block

The operator can submit invalid blocks to the root chain in Commit stage. However, in this case, all users remaining in the chain have checked which transactions are included in those blocks, so they can know the correct state of them. Therefore, users can cancel invalid blocks through Computation challenge.

6 Further Research

6.1 Toward Instant Finality

By setting Rebase as the default behavior of the Plasma chain, users can exit safely even though the operator withholds the block data. But users can maliciously cancel their transactions by submitting
a *Escape request* regardless of data availability. Therefore, users must check if any *Escape request* canceling their transactions is submitted or not.

For example, the decentralized exchange(DEX) is operating on Plasma EVM. Also, it takes about 24 hours for one *Cycle* to be *Finalized*. User A purchased token T from user B with 100 ETH about an hour ago. But now the price of token T has plummeted to 1 ETH. In this case, User A will submit an *Escape request* before the transaction is *Finalized* so that he or she can withdraw all of his or her balances in advance and cancel the transaction. In this case, user B who sold the token could suffer a significant loss.

### 6.2 Light Client Support

Users should always download all block data to check data availability. In other words, all users must be running a full node on the Plasma chain. This could be a serious impediment to User Experience(UX).

This problem is not only just for Plasma EVM but also all other Plasma proposals. Most other Plasma models also have restrictions on users to run a full node to check all data. To address these problems, some research has been presented using Erasure Coding to enable checking data availability with an only light client. As with Plasma EVM, further research is needed to enable users to check data availability with an only light client rather than a full node.

### 6.3 Addressing Requestable Contracts in Both Chains

*Requestable contract* should have the same codeHash in both chains. In an existing contract creation transaction, a contract address was determined using the address of the Transactor and Nonce. The operator had to check the codeHash and address of those two contracts and map it in the *RootChain contract*. However, anyone can deploy a *Requestable contract* to both chains with identical addresses by simply executing the same transaction, which is generated in child chain, on the root chain using CREATE2 to be newly added in the Constantinople hard fork 11.

By making the address of the Factory contract, which creates contracts, the same as the address of the *RootChain contract* deployed on the root chain using CREATE2 and putting it in the genesis block of the child chain, this feature can be supported.

### 7 Conclusion

Plasma EVM makes generalized state feasible with *Request, Request block* and *Requestable contract* in Plasma chain. *Challenge* guarantees that only valid block and *Request* is finalized in the root chain. Most of all, the data unavailability, which has been the deadly vulnerability of generalized Plasma, has been resolved through Continuous Rebase. It made honest users safe from any attack situation. However, shortcomings such as not instant finality due to *Rebase* are still major challenges to address.
A Glossary

A.1 General

- Root chain: Ethereum blockchain
- Child chain: Plasma blockchain. It can also be called Plasma chain.
- Operator: Agent that operate child chain
- NULL ADDRESS: 0x00 with nonce and signature \( v = r = s = 0 \), denoted \( NA \)
- Transactor: Account which generate transaction, \( tx.origin \)
- RootChain manager contract: A Plasma contract on root chain accepting enter / exit (ETH / ERC20)
- Request: A request which enforces to apply state transition by root chain
- Requestable contract: Contracts able to accept exit / enter request in both root and child chain. 2 identical contracts should be deployed in root and child chain, and \( R \) maps two addresses.
- Enter request: A request to enter something from root chain to child chain. eg) deposit asset, move account storage variable.
- Exit request: A request to exit asset or account storage from child chain to root chain. Any exit request on the root chain immediately updates account storage in the child chain. If the update in the child chain is rejected(TX reverted), the exit can be challenged with the computation output of the update as proof.
- Escape request: A request to escape from child chain. It has an identical structure to exit request, but there is a restriction on the submission time.
- Undo request: A request to prevents future enter requests in child chain from being applied.
- Request block: A block applying state transition that is enforced by the root chain, denoted \( RB \)
- Non-Request block: A block where transactions are only related between accounts in child chain, denoted \( NRB \)
- Request block: A request block applying enter / exit request by the operator, denoted \( RB \)
- Escape request block: A request block only including escape requests and undo requests, denoted \( EB \)
- Epoch: A period in which the same block must be submitted. Epoch for each type of block is denoted NRE, ORE, and ERE respectively.
A.2 Challenge

- Null Address Challenge: challenge if NRB contains a transaction from NA.
- Computation Challenge: challenge if block have the state computed in a wrong way.
- Exit Challenge: challenge if invalid Exit request cannot be accepted in child chain (but the request should be included in RB).
- Finalized: Every block and Exit request can be deterministically finalized only if any no successful challenge exists.

A.3 Continuous Rebase

- Rebase: Re-mining blocks based on another block. As a result, stateRoot and receiptsRoot of blocks can be changed but transactionRoot should be identical to the original one.
- Cycle: The entire operational period of the Plasma EVM, one cycle consists of a total of four stages. Each stage is Pre-commit, DA-check, Commit, and Challenge.
- Pre-commit: A stage that operator mines blocks in child chain and submits transactionRoot of them. At the same time, the operator must broadcast block data to users so that they can check data availability.
- DA-check: A stage that users check data availability of blocks submitted in Pre-commit. In case of unavailability, they can submit Escape request to escape from that chain. However, Escape request can be submitted at both Pre-commit, DA-check stages.
- Commit: The stage which the operator must submit an Escape request block applying Escape request and execute Rebase blocks submitted in Pre-commit.
- Challenge: The stage in which users can challenge if blocks submitted in Commit are invalid.
- Stage length: The number of Epoch to be processed in Pre-commit and Commit.
- Stage period: The time allocated for each Stage.
- Halting condition: A specific condition to halt child chain.
- Finalize: After the Challenge stage, whole blocks in a cycle can be finalized.
- Shutdown: A closing procedure of Plasma chain. After shutdown, no more cycles can be initiated. It only allows users to escape from that chain.
B Requestable Contract Examples

B.1 Requestable token contract

// @NOTE: This implements only enter and exit requests.
contract RequestableSimpleToken is Ownable, RequestableI {
    using SafeMath for *;

    // 'owner' is stored at bytes32(0).
    // address owner; from Ownable

    // 'totalSupply' is stored at bytes32(1).
    uint public totalSupply;

    // 'balances[addr]' is stored at keccak256(bytes32(addr), bytes32(2)).
    mapping(address => uint) public balances;

    // requests
    mapping(uint => bool) appliedRequests;

    /* Events */
    event Transfer(address _from, address _to, uint _value);
    event Mint(address _to, uint _value);
    event Request(bool _isExit, address _requestor, bytes32 _trieKey, bytes _trieValue);

    function transfer(address _to, uint _value) public {
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
    }

    function mint(address _to, uint _value) public onlyOwner {
        totalSupply = totalSupply.add(_value);
        balances[_to] = balances[_to].add(_value);

        emit Mint(_to, _value);
        emit Transfer(0x00, _to, _value);
    }

    // User can get the trie key of one's balance and make an enter request directly.
    function getBalanceTrieKey(address who) public pure returns (bytes32) {
        return keccak256(bytes32(who), bytes32(2));
    }
function applyRequestInRootChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
) external returns (bool success) {
    // TODO: adapt RootChain
    // require(msg.sender == address(rootchain));
    // require(!getAppliedRequestId(requestId)); // check double applying

    require(!appliedRequests[requestId]);

    if (isExit) {
        // exit must be finalized.
        // TODO: adapt RootChain
        // require(rootchain.getExitFinalized(requestId));

        if (bytes32(0) == trieKey) {
            // only owner (in child chain) can exit 'owner' variable.
            // but it is checked in applyRequestInChildChain and exitChallenge.

            // set requestor as owner in root chain.
            owner = requestor;
        } else if (bytes32(1) == trieKey) {
            // no one can exit 'totalSupply' variable.
            // but do nothing to return true.
        } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey) {
            // this checks trie key equals to 'balances[requestor]'.
            // only token holder can exit one's token.
            // exiting means moving tokens from child chain to root chain.
            balances[requestor] += decodeTrieValue(trieValue);
        } else {
            // cannot exit other variables.
            // but do nothing to return true.
        }
    } else {
        // apply enter
        if (bytes32(0) == trieKey) {
            // only owner (in root chain) can enter 'owner' variable.
            require(owner == requestor);
            // do nothing in root chain
        }
    }
}
} else if (bytes32(1) == trieKey) {
  // no one can enter ‘totalSupply’ variable.
  revert();
} else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey) {
  // this checks trie key equals to ‘balances[requestor]’.
  // only token holder can enter one’s token.
  // entering means moving tokens from root chain to child chain.
  require(balances[requestor] >= decodeTrieValue(trieValue));
  balances[requestor] -= decodeTrieValue(trieValue);
} else {
  // cannot apply request on other variables.
  revert();
}

appliedRequests[requestId] = true;

emit Request(isExit, requestor, trieKey, trieValue);

  // TODO: adopt RootChain
  // setRequestApplied(requestId);
  return true;
}

function decodeTrieValue(bytes memory trieValue) public pure returns (uint v) {
  require(trieValue.length == 0x20);

  assembly {
    v := mload(add(trieValue, 0x20))
  }
}

  // this is only called by NULL_ADDRESS in child chain
  // when i) exitRequest is initialized by startExit() or
  // ii) enterRequest is initialized
function applyRequestInChildChain(
  bool isExit,
  uint256 requestId,
  address requestor,
  bytes32 trieKey,
  bytes trieValue
) external returns (bool success) {
}
// TODO: adopt child chain
// require (msg.sender == NULL_ADDRESS);
// require (!appliedRequests[requestId]);

if (isExit) {
    if (bytes32(0) == trieKey) {
        // only owner (in child chain) can exit 'owner' variable.
        require (requestor == owner);

        // do nothing when exit 'owner' in child chain
    } else if (bytes32(1) == trieKey) {
        // no one can exit 'totalSupply' variable.
        revert();
    } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey) {
        // this checks trie key equals to 'balances[tokenHolder]'.
        // only token holder can exit one's token.
        // exiting means moving tokens from child chain to root chain.

        // revert provides a proof for 'exitChallenge'.
        require (balances[requestor] >= decodeTrieValue(trieValue));

        balances[requestor] -= decodeTrieValue(trieValue);
    } else { // cannot exit other variables.
        revert();
    }
} else { // apply enter
    if (bytes32(0) == trieKey) {
        // only owner (in root chain) can make enterRequest of 'owner' variable.
        // but it is checked in applyRequestInRootChain.

        owner = requestor;
    } else if (bytes32(1) == trieKey) {
        // no one can enter 'totalSupply' variable.
    } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey) {
        // this checks trie key equals to 'balances[tokenHolder]'.
        // only token holder can enter one's token.
        // entering means moving tokens from root chain to child chain.

        balances[requestor] += decodeTrieValue(trieValue);
    } else {
        // cannot apply request on other variables.
        revert();
    }
}
appliedRequests[requestId] = true;

emit Request(isExit, requestor, trieKey, trieValue);
return true;
}
References


