Plasma EVM

Release 0.0.1

Onther Inc.

Jun 14, 2019

CONTENTS

Plasma EVM provides a EVM-compatitable plasma chain with scalability and security. Plasma is scalability solution for Ethereum, making layered blockchains that data in child chain is summarized and committed to root chain.

Plasma EVM can reuse any solutions and framworks on Ethereum like Soldiity, Truffle, Metamask, and even MyEther-Wallet.

The major differences from other plasma proposals are as follows:

- Account-based blockchain
- Supports EVM computation
- Supports global state for a single transaction

You can follow how this design came out from scratch *here*.

CHAPTER

ONE

CONTENTS

Keyword Index, Search Page

1.1 Overview

This section is divided into 2 main parts, *RootChain Manager Contract* and *Requesrtable Contract*.

1.1.1 RootChain Manager Contract

RootChain contract is a manager contract for Plasma EVM. Operator submits plasma block data to this contract. Users also send transaction to make *request* for a requestable contract.

The Units

Plasma blocks are abstracted in many way depending on each contexts. `Block` is the smallest unit that operator commits to RootChain contract. `Epoch` includes many blocks. `Cycle` includes many epochs.

3 Types of Block

In Plasma EVM, There are many types of block; Non-Request Block, Request Block, Escape Block. **Non-Request Block** (NRB) is a regular block that users send transaction. It is same as block of Ethereum, Bitcoin, and other blockchain. **Request Block** (RB) is a block to apply *request* in child chain. Users create request and operator mines request block to apply the request in child chain. It is enforced what transactions must be included in request block by RootChain contract calculating transactions root. **Escape Block** (EB) is kind of request block to provide user to deal with block withholding attack by operator. For escape block, see continuous rebase.

NRB data can be withheld, but RB and EB cannot because anyone can get data from RootChain contract to mine RB and EB.

3 Types of Epoch

Epoch is a period of blocks. There are also **Non-Request Epoch** (NRE), **Request Epoch** (RE) and **Escape Epoch** (EE) for each block type. Epoch is required to determine when other type of block should be placed. Also a single request block cannot take all requests due to block gas limit and block size of child chain (and block gas limit of root chain to computate transactions root of request block).

The length of epoch is the number of blocks in the epoch. So RE and EE can have length of 0 if no RB or EB exists.

Note: The length of NRE is a constant that is provided when RootChain contract is deployed.

Cycle

Continuous Rebase requires many steps to commit a single block to root chain. And Pre-commit and Commit step incldue fixed number of NRBs. **Cycle** covers Pre-commit, DA check, Commit, Challenge steps and the length of cycle is `the number of NREs + the number of OREs + 1 (the number of EE)`

Note: The length of cycle is a constant that is provided when RootChain contract is deployed. But the number of blocks in a cycle may change depending on the number of requests.

Block Mining

In 1 cycle, child chain block is mined as below:

- In pre-commit step, the first epoch is NRE. operator mines NRB with transactions from users.
- If users create enter requests and exit requests, they are reserved to be included in ORE after next NRE. For example, requests created in `NRE#1` or `ORE#2` are applied in `ORE#4`. If no request created, ORE is empty.
- Pairs of `NRE` - `ORE` are repeated `(size(cycle)) - 1 / 2` times.
- In DA check step, user can create escape requests and undo requests.
- In Commit step, the first epoch is EE. Operator mines escape blocks if any request created in DA step. The parent of first block is the last block of previous cycle's last block.
- Operator rebases pre-committed block and commit them.

How to Handle Request

This explains how the request created and applied in child chain as request trasaction.

Addresses of Requestable Contracts in Both Chain

Reqeustable contracts must be deployed in both chain and 2 addresses must be mapped in RootChain contract. Then user can make *requests* for requestable contracts.

Create Enter Request

User can send transaction to RootChain contract to create enter request.

1. User send transaction to RootChain contract to call `RootChain.startEnter()`.
2. RootChain contract apply the request to the corresponding requestable contract. Those happens in root chain.
3. If step 2 is not reverted, RootChain contract record the request.
4. In request epoch, operator mines request block with request transactions. See how request is converted into reqeust transaction *here*.

```
function startEnter(address _to,bytes32 _trieKey,bytes _trieValue)
```

`to` is the address of target reqeustable contract in root chain. `trieKey` and `trieValue` is parameters for the request.

Create Exit Request

User also can send transaction to RootChain contract to create exit request.

1. User send transaction to RootChain contract to call `RootChain.startExit()`.
2. Unlike enter request, exit request is immediately recorded and mined in reuqest block with reqeust transactions. See how request is converted into reqeust transaction *here*.
3. After challenge period for the requst block, challenge period for exit request starts. If the request transaction in step 2 is reverted, anyone can challenge on this by calling `RootChain.challengeExit()` with the transaction inclusion proof and receipt data.
4. If there is no successful challenge, User finalize the request by calling `RootChain.finalizeRequest()`. In the function, RootChain contract apply the request to the corresponding requestable contract in root chain.

```
function startExit(address _to,bytes32 _trieKey,bytes _trieValue)
```

Pamateres are same as `startEnter`.

Apply Request in Child Chain

A request has four important fields, `requestor` is a address who made the request, `to` is a address of requestable contract deployed in root chain, `trieKey` is a identifier for request type, and `trieValue` is the value of request.

When a request is transformed into **request transaction**, the transaction has those fields as follow:

- `msg.sender`: it is always `0x00`. It prevents other from creating request transaction because nobody know the private key of address `0x00`. Due to this, signature of request transaction is zero , `v = r = s = 0`.
- `msg.to`: requestable contract **deployed in child chain**. RootChain contract must know it.
- `msg.value`: it is always `0`.
- `msg.data`: To invoke message-call in transaction, this field must contain function signature and parameters for `applyRequestInChildChain` function. RootChain contract always knows what bytes should be in this field. See also solidity code here.

When the current epoch is RE, operator mines request block with request transactions to transit state of child chian. RootChain contract enforces operator to include what request transactions should be in the request block by calculating transactions root of the block.

Those request transactions are applied to requestable contract by *apply request functions*

1.1.2 Requestable Contract

Requestable is a interface to be able to adapt Plasma EVM. Any contract implementiong `requestable` can accept enter and exit reqeust from RootChain contract.

Request

Request is an entity that makes users to interact with contracts. If user creates request, it is recorded in *RootChain* manager contract. A request is appiled in child chain as a **request transaction**. The sender of request transaction is `0x00` and it is included in **request block** to change the state of child chain. And it is enforced by RootChain contract to mine specific request block.

Before go further, it is recommended to see how `RootChain` contract handle request *here*.

Enter and Exit

Enter is "moving something from root chain to child chain". **Exit** is "moving something from child chain to root chain". The most intuitive example is token transfer. Depositing ERC20 to child chain is enter, and withdrawing it from child chain is exit.

Enter request is applied in root chain, then applied in child chain through request transaction. If applying in root chain is invalid, it **MUST** be reverted to prevent invalid enter request from being created.

Exit request is applied in child chain through request transaction, then applied in root chain. If the request is invalid, anyone can challenge on the invalid exit **with transaction receipt as proof**. If exit request is not challenged, anyone can *finalize* the reqeust and apply it to the requestable contract in root chain.

`ApplyRequestIn*Chain` Functions

If user wants to `enter` or `exit`, he sends a transaction to `RootChain` contract to make **enter request** or **exit request**. `RootChain.startEnter()` and `RootChain.startExit()` make user to create enter or exit request.

To accept those requests, contracts must implement **Requestable** interface.

See more how those requests are converted into request transaction and applied in child chain *here*.

```
interface RequestableI {
  /// @notice Apply exit or enter request to requestable contract
  ///         deployed in root chain.
  function applyRequestInRootChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success);

  /// @notice Apply exit or enter request to requestable contract
  ///         deployed in child chain.
  function applyRequestInChildChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success);
}
```

`applyRequestIn*Chain` functions have common parameters.

- `isExit`: `true` if the request is exit.

- `requestId`: Identifier for the request. `RootChain` contract assigns it.
- `requestor`: Address who made the request.
- `trieKey`: Identifier for request type. `trieKey` tells the contract **what state variable should be changed** for this request
- `trieValue`: Value of the request. `trieValue` tells the contract **how state should be changed**.

See more *examples*.

1.2 Getting Started

If feels complexity yourself or Could have time to do step-by-step,
There is Dockerized section at the end of this document. You can go directly goto.

Let considering Two environments. One is Private Environment, cannot connected outside of host, For plasma-chain test.
Another one is Publically opened Environment almost production level.
You can Mixing those two environments if fully used to.

1.2.1 Setup Private Environment

There are two essential steps.

1. Run **go-ethereum** as RootChain.
2. Run **Plasma-evm** as ChildChain with RootChain infomation.

Follow Instructions are tested on MacOSX.

1. Run RootChain

Use forked version of go-ethereum v1.8.20 as RootChain.

Building geth requires both a GO (version 1.11 or later) and C compiler.

1.1. Clone go-ethereum Repository

```
git clone http://github.com/onther-tech/go-ethereum
```

You dont need generate genesis file for eth balance allocation. It is going to use fake ethash as dev mode.
It forked from geth v1.8.20 then added some features pre-founded, other things via flags.
Use can use *run.rootchain.sh* running script instead geth run command like as below 1.3.

1.2. Build the source

```
make geth
```

If you want do running script, can skip this command.

1.3. Run ge-ethereum with flags.

Plasma-evm subscribe RootChain Events via rootchain's websocket, must be open to ChildChain.

```
bash run.rootchain.sh
```

Or You can directly run with this commands

```
build/bin/geth --dev --dev.period 1 --dev.faucetkey
"b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291,
↪78ae75d1cd5960d87e76a69760cb451a58928eee7890780c352186d23094a115,
↪bfaa65473b85b3c33b2f5ddb511f0f4ef8459213ada2920765aaac25b4fe38c5,
↪067394195895a82e685b000e592f771f7899d77e87cc8c79110e53a2f0b0b8fc,
↪ae03e057a5b117295db86079ba4c8505df6074cdc54eec62f2050e677e5d4e66" --miner.gastarget␣
↪7500000 --miner.gasprice "10" --rpc --rpcport 8545 --rpcapi eth,debug,net --ws --
↪wsport 8546
```

In this case, 5 privateKeys generate accounts files under datadir path.

2. Run ChildChain

We currently working on Plasma-evm running stable. Suggest, Clone master branch instead develop which is default.

2.1. Clone Plamsa-evm Repository

```
git clone http://github.com/onther-tech/plasma-evm
```

2.2. Build the source

Do as same as go-ethereum

```
make geth
```

2.3. Run Plasma-evm with flags

There are additional params to run Plasma-chain through flags. No need genesis file. It going to automatically generated.
We added some flags for get params to run plasma-evm.
You can get some information about the flags for plasma-evm as like below, using `geth --help`.

```
MISC OPTIONS:
 --operator.minether value                  Plasma operator minimum balance (default␣
↪= 0.5 ether) (default: "0.5")
 --operator value                           Plasma operator address as hex. The␣
↪account should be unlock by using --unlock
```

(continues on next page)

(continued from previous page)

```
 --operator.key value                          Plasma operator key as hex(for dev)

 --dev.key value                               Comma seperated developer account key as␣
↪hex(for dev)

 --rootchain.url value                         JSONRPC endpoint of rootchain provider␣
↪(default: "ws://localhost:8546")
 --rootchain.contract value                    Address of the RootChain contract
 --rootchain.challenger value                  Address of challenger account

 --tx.mingasprice "1000000000"                 Minimum gas price for submitting a block␣
↪(default = 1 Gwei)
 --tx.maxgasprice "100000000000"               Maximum gas price for submitting a block␣
↪(default = 100 Gwei)
 --tx.interval value                           Pending interval time after submitting a␣
↪block (default = 10s). If block submit transaction is not mined in 2 intervals, gas␣
↪price will be adjusted. See https://golang.org/pkg/time/#ParseDuration (default:␣
↪10s)

 --stamina.mindeposit "500000000000000000"  MinDeposit variable state of stamina␣
↪contract
 --stamina.recoverepochlength "10080"       RecoverEpochLength variable state of␣
↪stamina contract
 --stamina.withdrawaldelay "30240"          WithdrawalDelay variable state of stamina␣
↪contract
```

dev.key, *operator.key*, *rootchain.challenger* Are for testing.

If there is already deployed *rootchain* contract on RootChain Network,
you can use like `--rootchain.contract 0x123456789aa` instead `--dev` mode. Cannot use *–dev* and *–rootchain.contract* at the same time.
In plasma-evm *dev* mode has additional features, which automatically deploying *rootchain* contract if no have rootchain address via *–roothchain.contract*.

In this case for testing. use dev mode.

[Important Notice] Operator Account must have some ether balance at RootChain. If Does not have, Could not start ChildChain.

```
geth --miner.etherbase 0x71562b71999873DB5b286dF957af199Ec94617F7 --dev --rpc --
↪rpcaddr 0.0.0.0 --rpcport 8547 --port 30307 --dev.key␣
↪b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291 --operator␣
↪0x71562b71999873DB5b286dF957af199Ec94617F7 --tx.interval "300ms" --rootchain.url
↪"ws://127.0.0.1:8546"
```

If you consider to run in production level, Recommand raise *tx.interval* time, at least 20s.

Finally, You can see like Plasma-chain running!

Remember deployed contract addresses! It can be use for later.

RootChain contract address is most Important contract addresses on below. In here, address start with 0x880EC...

```
INFO [04-04|01:02:11.897] Persisted trie from memory database      nodes=13 size=5.
↪80kB time=66.292µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [04-04|01:02:11.897] Deploying contracts for development mode
INFO [04-04|01:02:11.908] Deploy MintableToken contract
↪hash=2febea...3b8f02 address=0x3A220f351252089D385b29beca14e27F204c296A
INFO [04-04|01:02:11.908] Wait until deploy transaction is mined
INFO [04-04|01:02:12.920] Deploy EtherToken contract
↪hash=b48a9b...0e82f3 address=0xdB7d6AB1f17c6b31909aE466702703dAEf9269Cf
INFO [04-04|01:02:12.921] Wait until deploy transaction is mined
INFO [04-04|01:02:19.953] Deploy EpochHandler contract
↪hash=f30f84...337aa6 address=0x537e697c7AB75A26f9ECF0Ce810e3154dFcaaf44
INFO [04-04|01:02:19.953] Wait until deploy transaction is mined
INFO [04-04|01:02:22.983] Deploy RootChain contract
↪hash=175321...ff616f address=0x880EC53Af800b5Cd051531672EF4fc4De233bD5d
INFO [04-04|01:02:30.012] Initialize EtherToken
↪hash=584c83...0a41e1
INFO [04-04|01:02:32.019] Set options for submitting a block
↪mingaspirce=1000000000 maxgasprice=300000000000 interval=10s
INFO [04-04|01:02:32.019] Starting peer-to-peer node               instance=Geth/v1.8.
↪20-stable-3a343606/darwin-amd64/go1.9.5
INFO [04-04|01:02:32.019] Allocated cache and file handles         database=/Users/
↪jins/.pls.dev/geth/chaindata cache=512 handles=4611686018427387903
INFO [04-04|01:02:32.026] Writing custom genesis block
↪rootChainContract=0x880EC53Af800b5Cd051531672EF4fc4De233bD5d
INFO [04-04|01:02:32.027] Persisted trie from memory database      nodes=13 size=5.
↪80kB time=124.834µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [04-04|01:02:32.027] Initialised chain configuration          config="{ChainID:
↪16 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: 0 EIP155: 0 EIP158: 0
↪Byzantium: 0 Constantinople: <nil> Engine: ethash}"
WARN [04-04|01:02:32.027] Ethash used in fake mode
INFO [04-04|01:02:32.027] Initialising Plasma protocol             versions="[63 62]"
↪network=1337
INFO [04-04|01:02:32.048] Loaded most recent local header          number=0
↪hash=e413e8...e44af1 td=1 age=49y11mo2w
INFO [04-04|01:02:32.048] Loaded most recent local full block      number=0
↪hash=e413e8...e44af1 td=1 age=49y11mo2w
INFO [04-04|01:02:32.048] Loaded most recent local fast block      number=0
↪hash=e413e8...e44af1 td=1 age=49y11mo2w
INFO [04-04|01:02:32.049] Regenerated local transaction journal    transactions=0
↪accounts=0
INFO [04-04|01:02:32.051] Rootchain provider connected             url=ws://
↪localhost:8546
INFO [04-04|01:02:32.061] New local node record                    seq=1
↪id=df4cc248d21c5db6 ip=127.0.0.1 udp=0 tcp=55563
INFO [04-04|01:02:32.061] Started P2P networking                   self="enode://
↪6f7ff81c34959c797e96704e5082fab0550ba603c5dec6825fc1b31f85f1a441303eb94af46ca2ab36165bd0f9738b3337e
↪0.0.1:55563?discport=0"
INFO [04-04|01:02:32.063] Iterating epoch prepared event
INFO [04-04|01:02:32.063] RootChain epoch prepared                 epochNumber=1
↪epochLength=2 isRequest=false userActivated=false isEmpty=false ForkNumber=0
↪isRebase=false
INFO [04-04|01:02:32.063] NRB epoch is prepared, NRB epoch is started NRBepochLength=2
INFO [04-04|01:02:32.064] Iterating block finalized event
INFO [04-04|01:02:32.064] RootChain block finalized                forkNumber=0
↪blockNubmer=0
```

(continues on next page)

(continued from previous page)

```
INFO [04-04|01:02:32.064] Watching epoch prepared event           start block␣
↪number=0
INFO [04-04|01:02:32.065] Watching block finalized event         start block␣
↪number=0
INFO [04-04|01:02:32.065] Updated mining threads                 threads=8
INFO [04-04|01:02:32.065] started whisper v.6.0
INFO [04-04|01:02:32.068] IPC endpoint opened                    url=/Users/jins/.
↪pls.dev/geth.ipc
INFO [04-04|01:02:32.068] HTTP endpoint opened                   url=http://127.0.0.
↪1:8547         cors= vhosts=localhost
INFO [04-04|01:02:34.312] Mapped network port                    proto=tcp␣
↪extport=55563 intport=55563 interface="UPNP IGDv2-IP1"
```

Looks like stop, but It Just waiting Tx! In dev mode, Start block mine when transaction has on txpool.

1.2.2 Setup Public Environment

1. Run RootChain

Recommand to use go-ethereum v1.8.23 as RootChain.

Building geth requires both a GO (version 1.11 or later) and C compiler.

1.1. Clone go-ethereum Repository

```
git clone -b v1.8.23 http://github.com/ethereum/go-ethereum
```

You should generate genesis file via puppeth. Recommand consensus ethash, not Clique.

1.2. Build the source

```
make geth
```

1.3. Run ge-ethereum with flags.

If you want to run with Ropsten testnet in here, Add *–testnet* then this geth going to have chainId 3.
Plasma-evm subscribe RootChain Events via rootchain's websocket, must be open to ChildChain.
If you want to running on Ropsten network, use **–testnet** flag instead –dev flag.

geth initialize with genesis file

```
geth init --datadir data genesis.json
```

then

```
geth --datadir data --mine --miner.etherbase␣
↪0x71562b71999873DB5b286dF957af199Ec94617F7 --miner.gastarget 7500000 --miner.
↪gasprice "10" --rpc --rpcaddr 0.0.0.0 --rpcport 8545 --rpcapi web3,eth,personal,
↪miner,net,txpool --ws --wsaddr 0.0.0.0 --wsport 8546 --wsorigins="*" --unlock␣
↪0x71562b71999873DB5b286dF957af199Ec94617F7,
↪0x3d[illegible]0b90a3d7ed9a83203d1f98a811d,
↪0x3cd9f729c8d882b851f8c70fb36d22b391a288cd --password ./signer.pass
```

(continues on next page)

(continued from previous page)

In this case, Inserted 3 Keyfiles already in *data* path.

2. Run ChildChain

We currently working on Plasma-evm running stable. Suggest, Clone master branch instead develop which is Default.

Important Notice, Every User has own node in plasma-evm which syncing operator's one. If user does not, It is securely vulnerable by Data Availability.

And also, Operator's node has own private key for commit transactions to RootChain. Have to properly secure action via firewall etc.

2.1. Clone Plamsa-evm Repository

```
git clone -b master http://github.com/onther-tech/plasma-evm
```

2.2. Build the source

Do as same as go-ethereum

```
make geth
```

2.3. Run Plasma-evm as Operator

If there is already deployed *rootchain* contract on RootChain Network,
you can use like `--rootchain.contract 0x123456789aa` instead `--dev` mode. Cannot use *–dev* and *–rootchain.contract* at the same time.
In plasma-evm *dev* mode has additional features, which automatically deploying *rootchain* contract if no have rootchain address via *–roothchain.contract*.

In this case for testing. use dev mode.

[Important Notice] Operator Account must have some ether balance at RootChain. If Does not have, Could not start ChildChain.

```
geth --miner.etherbase 0x71562b71999873DB5b286dF957af199Ec94617F7 --dev --port 30307 -
↪-dev.key b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291 --
↪operator 0x71562b71999873DB5b286dF957af199Ec94617F7 --tx.interval "300ms" --
↪rootchain.url "ws://127.0.0.1:8546"
```

If you consider to run in production level, Recommand raise *tx.interval* time, at least 10s.

Remember deployed contract addresses! It can be use for later.

2.4. Run Plasma-evm as User

There are different between Operator Node and User Node.
User Node Has no Own Private Key and No Mining.
But Need same networkId, genesis hash at start.

There is two way to connect Operator Node and User Node Each other.
One is that Using bootnode, It is simpler than other way. run bootnode then insert bootnode address on flag *–bootnodes*.

The Other is that add peer, Operator Node, manually at User Node.

Important thing is that Exactly Match networkid, genesis block hash between User and Operator Node.
If does not, Cannot find each other forever.

There is one Problem, has different genesis block hash when run plasma-evm everytime. Must fix one of Two.
So, If you want to run User node, use other plasma-evm branch, p2p-in-dev-mode, still not merged soon.

```
git clone -b p2p-in-dev-mode http://github.com/onther-tech/plasma-evm
```

then you should check *rootchain*, *operator* addresses as same as Operator's.

```
geth --dev --dev.p2p --networkid 1337 --rpc --rpcaddr 0.0.0.0 --rpcport 8549 --port␣
↪30307 --rootchain.url "ws://127.0.0.1:8546"  --dev.key␣
↪b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291 --dev.rootchain␣
↪0x880EC53Af800b5Cd051531672EF4fc4De233bD5d --operator␣
↪0x71562b71999873DB5b286dF957af199Ec94617F7
```

In here, *–dev.p2p* mode make turn on p2p networking. so Please do not forgot. And others, *dev.rootchain*, *dev.operator*, are need to generate same genesis block hash.

Add *–bootnode [bootnode key@ip:port]* flag if you using Bootnode. or Add peer using geth console.

Remember, have to same Networkid and Genesis hash between User & Operator Node.

1.2.3 Quick Start with Docker

It is Private Environment for test.

1. Clone dockerize branch Plasma-evm

```
git clone -b dockerize http://github.com/onther-tech/plasma-evm
```

2. Update Submodules

```
git submodule update --init --recursive
```

3. Up docker-compose

```
docker-compose up
```

- If you turn down containers *docker-compose down* on plasma-evm path.

1.2.4 Quick Start with Truffle

You can use the truffle framework to deploy contracts to the tokamak testnet, faraday. To use the faraday network, you need to specify the network in truffle-config.js as follows.

```
module.exports = {
networks: {
  development: {
    host: 'localhost',
    port: 8545,
    network_id: '*'
  },
  ropsten: {
    provider: ropstenProvider,
    network_id: 3
  },
  faraday: {
    host: "112.169.69.41",
    port: 48549,
    network_id: "*"
  }
}
```

If you want to test on the faraday network, you can use faucet at here.

Note: Faraday is the Tokamak network used as a testing environment. Faraday is connected to the ropsten network as a root chain.

Enter and exit token at faraday network

We will enter and exit token using the *RequestableSimpleToken* without ownership, RequestableSimpleTokenWithoutOwnership. Anyone can mint the token because there is no ownership.

RequestableSimpleTokenWithoutOwnership contract and RootChain contract are already deployed, so you can use it to test enter and exit. RequestableSimpleTokenWithoutOwnership contract address at Ropsten is `0x6B27C38e3376C4E8B29cFbB3986f00676267D489` and contract address at Faraday is `0x1d93d7bd7d820ac7691109ace371e42d5004e1c1`. *RootChain* contract address at Ropsten is `0x3122546c1544FD0F910A423A8c80fdCD48d742Fd`.

The scenario will work as follows:

1. Alice mint `RequestableSimpleTokenWithoutOwnership` at the Ropsten.
2. Alice get her trieKey by using `RequestableSimpleTokenWithoutOwnership.getBalanceTrieKey()` and trieValue.
3. Alice call `RootChain.startEnter()` method to start entering process to Faraday.

```
truffle(ropsten)> alice = '0xb60e8dd61c5d32be8058bb8eb970870f07233155'
truffle(ropsten)> rootchain = RootChain.at('0x3122546c1544FD0F910A423A8c80fdCD48d742Fd
↪')
truffle(ropsten)> token = RequestableSimpleTokenWithoutOwnership.at(
↪'0x6B27C38e3376C4E8B29cFbB3986f00676267D489')

truffle(ropsten)> token.mint(alice, 10000000)
truffle(ropsten)> key = token.getBalanceTrieKey(alice)
truffle(ropsten)> value =
↪'0x0000000000000000000000000000000000000000000000000000000000000100'
truffle(ropsten)> rootchain.startEnter(alice, key, value)
```

4. After entering process is finished, you can check entered token balance by using `RequestableSimpleTokenWithoutOwnership.balances()` at Faraday.

```
truffle(faraday)> alice = '0xb60e8dd61c5d32be8058bb8eb970870f07233155'
truffle(faraday)> token = RequestableSimpleTokenWithoutOwnership.at(
↪'0x1d93d7bd7d820ac7691109ace371e42d5004e1c1')
truffle(faraday)> token.balances(alice)
```

5. If Alice wants to use his token at rootchain, then start exit process to rootchain by using `RootChain.startExit()`.

```
truffle(ropsten)> alice = '0xb60e8dd61c5d32be8058bb8eb970870f07233155'
truffle(ropsten)> rootchain = RootChain.at('0x3122546c1544FD0F910A423A8c80fdCD48d742Fd
↪')

truffle(ropsten)> key = token.getBalanceTrieKey(alice)
truffle(ropsten)> value =
↪'0x0000000000000000000000000000000000000000000000000000000000000100'
truffle(ropsten)> rootchain.startExit(alice, key, value)
```

1.3 Design

In this journey, we describe abstractly how Plama EVM is designed from Requetable to Continuous Rebase. See more details in technical paper.

1.3.1 Principle

The fundamental pricinple is making plasma chain client by forking current Ethereum client (go-ethereum, pyethereum, parity). It provides many benefits like secure P2P communicartion, robustness, key management, and other things. But the most important thing is that other programming languages and tools for Ethereum (Solidity, Truffle, and so on) can be reused. It is wasteful to invent the wheel from scratch.

So the basic concept of Plasma EVM, Requestable, comes out. If we have 2 Ethereum networks and if they have to interoprate (like transfer *ERC20* or *(P)ETH* to another chain) in a truestless way, they need kind of **receipts** saying "Something is changed in chain A. So corresponding change is required in chain B". (e.g. transfer ERC20 from chain A to chain B by burning in chain A and minting in chain B).

Plasma EVM manager contract in root chain (a.k.a `RootChain` contract) needs a common interface for contracts that are compatible with Plasma EVM. We define those contracts as Requestable contract.

To accept the change in both chains, same requestable contracts must be deployed in both chain.

1.3.2 Deal with Data Availability

Exit Game

UTXO-based plasma proposals have a mechanism to protect assets in plasma chain. Plasma MVP uses confirmation model and exit game. Plasma Cash removes confirmation by introducing Sparse Merkle Tree to describe plasma chain state. Even if operator withholds block data, any user can exit their assets through exit game with proofs. They are absolutely secure models. Plasma EVM also needs those mechamism.

But Plasma EVM is account-based and storage of contract account is same as Ethereum, patricia trie. So we cannot adopt exit game. Instead, we developped user-activated fork.

User-activated Fork

Basic concept is that user can make special kind of request, `exit request for URB`. And another type of block, user-submitted request block (`URB`), can be submitted to `RootChain` contract by any user if he can afford it. User-submitted request block has the last finalized block as parent.

So any user can make a new fork without fear of data unavailability. If they notice unavailability, then will make a new fork before unavailable block is finalized. But **to give higher priority on user-submitted request block than operator-submitted block**, the block number of user-submitted block has to be less than those of operator-submitted block.

We call this updating priority by **rebase**, named after git's rebase. Any operator-submitted blocks are placed after user-submitted block by rebasing. Rebase requires many computations in plasma chain and transactions in root chain, so we considered "making fork even when data is available" as a attack.

But there is no concrete structural solution to prevent this attack. Our first naive approach is to charge higher cost to user, but this naive and finantial solution cannot be considerred as a soultion.

So we set rebase as the default behavior.

Continous Rebase

In this model, rebase is considered as the default behavior. URB is renamed `escape block` and exit request for URB as `escape request`.

Below 4 stages are processed sequentially in one **cycle**.

- In **pre-commit** stage, operator commits only transactions root to `RootChain` contract. If operator stop committing in this stage, **halting condition** is fulfilled. Halting condition forces to fully execute incomplete stage.
- In **DA check** stage, if users notice unavailability for the root, they make a escape request. User also can make it in pre-commit stage.
- In **commit** stage, operator mines and commits escape blocks first whose parent is last block of previous cycle. And rebase blocks which were commited in pre-commit stage.
- In **challenge** stage, anyone can challenge on blocks commited in commit stage by computation challenge.

1.4 Requestable Contract Examples

You can see all the source codes in this repository.

1.4.1 Counter

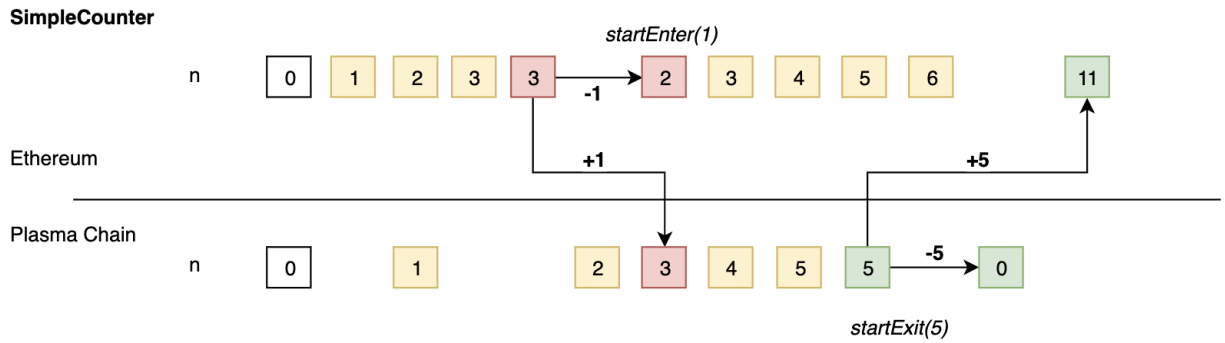
BaseCounter

BaseCounter is a simple base counter contract. Other requestable counter contracts inherit this base contract and implement requestable interface.

```
pragma solidity ^0.4.24;


contract BaseCounter {
  uint n;

  event Counted(uint _n);

  function count() external {
    n++;
    emit Counted(n);
  }

  function getCount() external view returns (uint) {
    return n;
  }
}
```

SimpleCounter

When there is a counter contract that can be increased by anyone, the most intuitive enter is to reduce the number in the root chain and increase the value by that much in the child chain. Exit, in contrast, reduces the value first in the child chain and increase the value in the root chain. SimpleCounter that implements this is illustrated as follows.



A yellow box means that the `counter()` has increased the status variable `n` by 1, a red box means entering the request changes `n`, and a green box means exiting the request changes `n`.

```
pragma solidity ^0.4.24;

import {SimpleDecode} from "../lib/SimpleDecode.sol";
import {RequestableI} from "../lib/RequestableI.sol";
import {BaseCounter} from "./BaseCounter.sol";
import {SafeMath} from "openzeppelin-solidity/contracts/math/SafeMath.sol";
```

(continues on next page)

(continued from previous page)

```
/// @notice A request can decrease `n`. Is it right to decrease the count?
contract SimpleCounter is BaseCounter, RequestableI {
  // SimpleDecode library to decode trieValue.
  using SimpleDecode for bytes;
  using SafeMath for *;

  // trie key for state variable `n`.
  bytes32 constant public TRIE_KEY_N = 0x00;

  // address of RootChain contract.
  address public rootchain;

  mapping (uint => bool) appliedRequests;

  constructor(address _rootchain) {
    rootchain = _rootchain;
  }

  function applyRequestInRootChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    require(!appliedRequests[requestId]);
    require(msg.sender == rootchain);

    // only accept request for `n`.
    require(trieKey == TRIE_KEY_N);

    if (isExit) {
      n = n.add(trieValue.toUint());
    } else {
      n = n.sub(trieValue.toUint());
    }

    appliedRequests[requestId] = true;
  }

  function applyRequestInChildChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    require(!appliedRequests[requestId]);
    require(msg.sender == address(0));

    // only accept request for `n`.
    require(trieKey == TRIE_KEY_N);

    if (isExit) {
      n = n.sub(trieValue.toUint());
    } else {
      n = n.add(trieValue.toUint());
```

(continues on next page)

(continued from previous page)

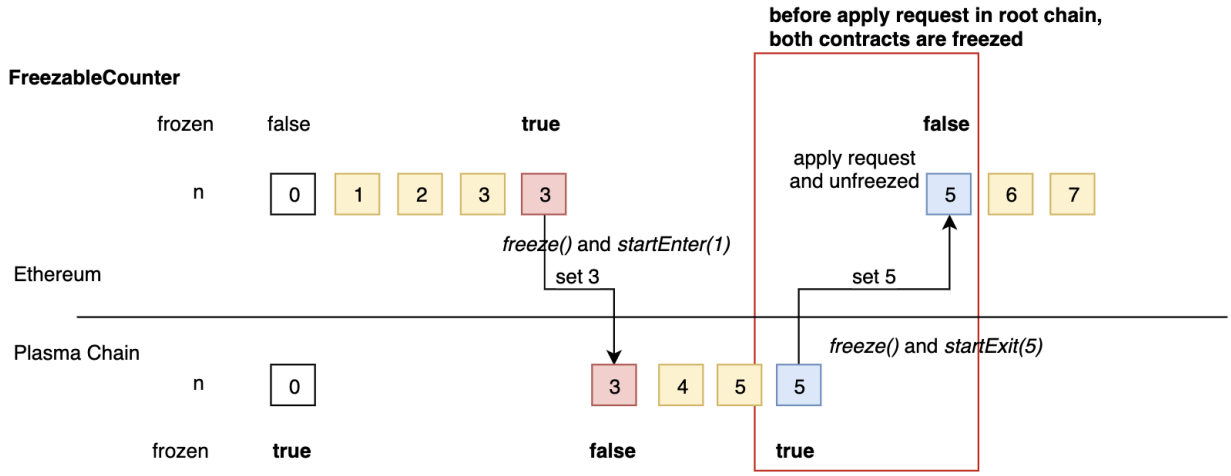
```
    }

    appliedRequests[requestId] = true;
  }
}
```

Let's read the code one by one.

However, SimpleCounter may decrease with variable `n` due to enter and exit. If this is not desired, you can implement counter contract as below.

FreezableCounter

Enter and exit can be applied after freezing the contracts in each chain. FreezableCounter can be avoided if the number decreases through the request method after freezing.




```
pragma solidity ^0.4.24;

...


/// @notice Both contract may be frozen at the same time. Is it right?
contract FreezableCounter is BaseCounter, RequestableI {
  ...

  // freeze counter before make request.
  bool public frozen;

  constructor(address _rootchain) {
    rootchain = _rootchain;

    // Counter in child chain is frozen at first.
    if (_rootchain == address(0)) {
      frozen = true;
    }
  }
```

(continues on next page)

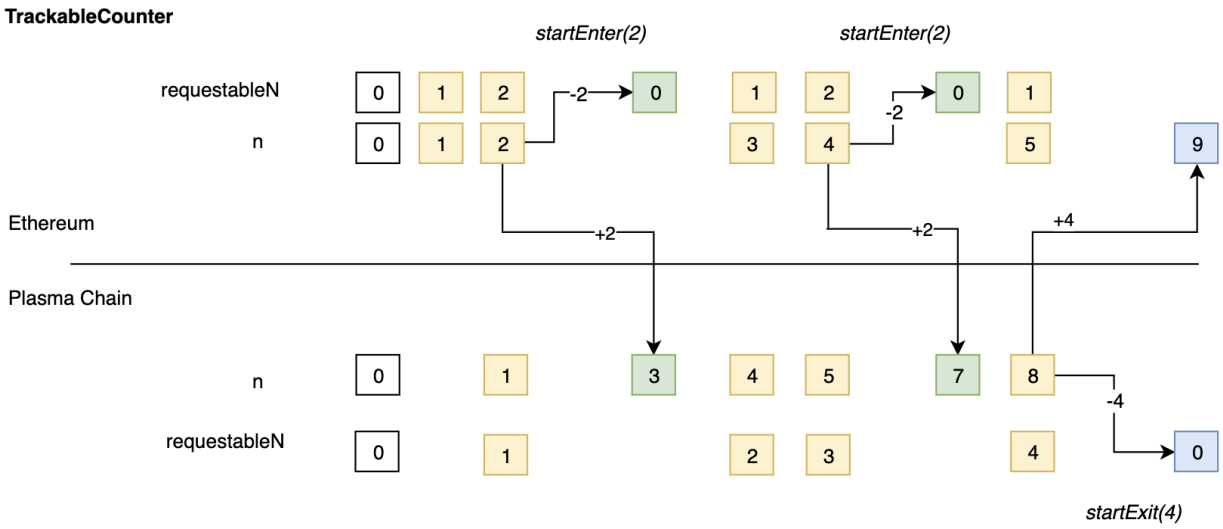
(continued from previous page)

```
  function freeze() external returns (bool success) {
    frozen = true;
    return true;
  }

  function applyRequestInRootChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    ...
    require(frozen);

    ...

    if (isExit) {
      frozen = false;
      n = trieValue.toUint();
    } else {
      require(n == trieValue.toUint());
    }

    ...
  }

  function applyRequestInChildChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    ...
    require(frozen);

    ...

    if (isExit) {
      require(n == trieValue.toUint());
    } else {
      n = trieValue.toUint();
      frozen = false;
    }

    ...
  }
}
```

However, the challenge period exists until exit is applied in root chain, for this freeze counter, all counters in each chain are frozen before the end of this challenge period. The enter is relatively short, but both are frozen. Therefore, to prevent this, the state variable used for enter and the state variable used for exit must be different.

TrackableCounter

TrackableCounter checks whether enter and exit is possible through a separate state variable `requestableN` in enter in the root chain and exit in child chain, reduces the value, and increases `n` in exit in the root chain and enter in the child chain. Both operations can prevent the reduction of `n` and apply only the correct enter and exit.




```
pragma solidity ^0.4.24;

...

contract TrackableCounter is BaseCounter, RequestableI {
  ...

  // previous count before enter request in root chain and exit request in child chain.
  uint public requestableN;

  ...

  /// @dev override BaseCounter.count function.
  function count() external {
    requestableN++;
    n++;
    emit Counted(n);
  }

  function applyRequestInRootChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    ...

    uint _n = trieValue.toUint()
    if (isExit) {
```

(continues on next page)

(continued from previous page)

```
      n = n.add(_n);
    } else {
      requestableN = requestableN.sub(_n);
    }

    ...
  }

  function applyRequestInChildChain(
    bool isExit,
    uint256 requestId,
    address requestor,
    bytes32 trieKey,
    bytes trieValue
  ) external returns (bool success) {
    ...

    if (isExit) {
      requestableN = requestableN.sub(_n);
    } else {
      n = n.add(_n);
    }

    ...
  }
}
```

1.4.2 Token

RequestableSimpleToken

RequestableSimpleToken contract is a requestable token contract. RequestableSimpleToken contract makes `balances` state variable requestable. Therefore, the `balances` state variable can enter and exit.

```
pragma solidity ^0.4.24;

...

contract RequestableSimpleToken is Ownable, RequestableI {

// `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;

function transfer(address _to, uint _value) public {}

function mint(address _to, uint _value) public onlyOwner {}

// User can get the trie key of one's balance and make an enter request directly.
```

(continues on next page)

(continued from previous page)

```
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) {
    require(!appliedRequests[requestId]);

    if (isExit) {
    // exit must be finalized.
    // TODO: adpot 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();
    }
    }
```

(continues on next page)

(continued from previous page)

```
    appliedRequests[requestId] = true;

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

    return true;
}

// 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) {
    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();
    }
```

(continues on next page)

(continued from previous page)

```
    }

    appliedRequests[requestId] = true;

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


}
```

`getBalanceTrieKey` function helps calculate `balance` state variable's trie key.