PoC1 : EVM Compatible Transaction Fee(GAS) Delegated Execution Architecture

Kevin Jeong(kevin.j@onther.io, Onther Inc.)

Thomas Shin(Onther Inc.)

Jason Hwang(Onther Inc.)

Carl Park(Onther Inc.)

Abstract

PoC1 is the transaction execution model which is competible with Ethereum Virtual Machine. The purpose of this project is to provide the transaction processing procedure, maintaining ethereum's GAS system, delegates GAS from transactor to decentralized autonomous agent(smart contract) defined in blockchain.

In PoC1, we achieved an improvement from a simple structure, GAS and value(ETH) paid from only to transactor, to multiple structure the burden of fee is supported by "stamina contract" in a means of "Stamina". Stamina is periodic-rechargable state variable and it's comsumption is keep tracked and managed by blockchain rules.

As a result, users do not need to concern about the "keeping balance maintanance activities". It also facilitates service provider to create a more user-friendly decentralized application(dApp). This will accelerate the emergence of killer apps, pushing to expand the decentralized ecosystem futher.

1. Introduction

1.1 Project Background and Purpose

Ethereum imposes fee on all computing operations to prevent unauthorized use of the network. This fee is calculated as a special unit called Gas and it is a time-space cost[2] on the Ethereum network.

The Ethereum blockchain avoids denial of service attacks through this commission model and solves the problem of distributing limited resource; 'Right of state transitions in blockchain'. The important point is that the transactor always maintains a certain level of balance to generate the transaction paying the GAS Limit * GAS Price. Transactor sends a transaction "always" has to bear these costs. It extremely reduces the usability of DApps implemented on Ethereum blockchain.

In PoC1, we want to design an Execution Model which enhances the user experience. It is expected to provide a ground for user-friendly (accessibility / usability) application development environment.

1.2 Denotation and Grossary

The words defined in PoC1 are as follows(Some of denotations came from ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER[1]):

- ERC20: ERC20 is a smart contract technology standard for implementing tokens in the Ethereum blockchain.
- State balance: The balance recorded in the Ethereum State.
- Account: User identification unit in the Ethereum blockchain.
- NULL_ADDRESS: 0x00

or 0xFF...FF

(2^160 - 1

) with signature v = r = s = 0

.

- Transactor: The account which sends transaction.
- Gas: Ethereum's fee unit.

- Gas limit: the maximum gas consumption specified in transaction.
- Upfront cost: The cost to be charged before the EVM runs. It is calculated as value + gasLimit * gasPrice

(Expenses that are charged before executing EVM.).

- Gas-upfront cost: Upfront cost - Value. Calculated as tx.gasLimit * tx.gasPrice

.

- Value-upfront cost: Upfront cost - gas-upfront cost = value.
- Refunded gas: Gas Limit - the gas actually consumed by the EVM.
- Delegatee: An account that pays the transaction fee of another account in stamina contract.
- Delegator: An account that delegates the transaction fee to the delegatee in stamina contract.
- stamina pair: If delegatee designate delegator, they are pair. It is called .
- Stamina: State variable deposited to delegatee's balance in stamina contract.
- Stamina Contract: A decentralized contract code containing procedures for handling delegated gas with stamina.
- Depositor account (depositor): An account who pours asset into the delegatee's stamina bucket.
- Stamina deposit: An action of charging stamina to somebody's account.
- Stamina withdraw: Removing charged stamina and filling other account balances accordingly.
- Stamina refund: Stamina refund is equals to refunded Gas in stamina contract.
- Stamina substract: An action of deducting delegatee's stamina.
- Stamina recover: Refreshment of stamina.
- Stamina Epoch Length: It takes a certain period of time to recover stamina, which is called stamina epoch length.
- Epoch: Unit of stamina recovery cycle.

2. System Architecture

The system we design in this chapter is largely divided into two parts:

1. A contract dealing with the delegated gas costs
2. The procedure of transaction processing.

2.1 Delegated Gas Treatment

2.1.1 Stamina

The transaction fee of the delegator is purchased by the delegatee in the form of stamina, alleviating the delegator gas burden. A stamina can be obtained by depositing asset on a stamina contract

, and an account that deposits the stamina of a delegatee is called a depositor

.

The deduced stamina does not mean disappearing forever, but it will be recovered after a certain period of time. It is stamina recover

in PoC1. The stamina requires a period of time to recover, and it is recover epoch length

. Every epoch, stamina is newly charged as much as amount deposited.

The stamina balance expressed as follows:

[a]_{stm} \tag{1}

2.1.2 Stamina Pair

A delegator is denoted as D_{tor}

and the delegatee as D_{tee}

. The relationship between delegator and delegatee is defined as ST_{pair}F()

.

ST_{pair}F(D_{tor}) = D_{tee} \tag{2}

When the delegatee and the delegator satisfy the above equation, they become a stamina pair

.

2.1.3 Stamina Contract

Stamina contract provides the following functions:

1. Designate delegator
2. Stamina Increase/Decrease
3. Stamina Refund/Substract/Recover

2.1.3.1 Designate the Delegator

A delegatee can designate a delegator. Only a delegatee can designate, whereas a delegator can not. In addition, a delegatee can designate multiple delegators.

The delegatee calls stamina contract's setDelegator()

function to designate the delegator.

function setDelegator(address delegator) external onlyInitialized returns (bool) { address oldDelegatee = _delegatee[delegator];

_delegatee[delegator] = msg.sender;

emit DelegateeChanged(delegator, oldDelegatee, msg.sender);
return true;

}

2.1.3.2 Stamina Increase & Decrease

To increase the stamina, the depositor must call the deposit()

function of the stamina contract.

For example, let's assume account A, B exist. If account A calls deposit()

function to add stamina of B, B's stamina increased.

function deposit(address delegatee) external payable onlyInitialized returns (bool) { require(msg.value >= MIN_DEPOSIT);

uint totalDeposit = _total_deposit[delegatee];
uint deposit = _deposit[msg.sender][delegatee];
uint stamina = _stamina[delegatee];

require(totalDeposit + msg.value > totalDeposit);
require(deposit + msg.value > deposit);
require(stamina + msg.value > stamina);

_total_deposit[delegatee] = totalDeposit + msg.value;
_deposit[msg.sender][delegatee] = deposit + msg.value;
_stamina[delegatee] = stamina + msg.value;

if (_last_recovery_block[delegatee] == 0) {
  _last_recovery_block[delegatee] = block.number;
}

emit Deposited(msg.sender, delegatee, msg.value);

return true;

}

In contrast, stamina contract's requestWithdrawl()

, withdraw()

function called to reduce the stamina of the delegatee. At this point stamina shrinks and depositor's balance is increased as much as stamina reduced, accordingly.

In order for the depositor to withdraw, it goes through two steps:

1. Call the requestWithdrawl()

function

: First, the depositor calls stamina contract's requestWithdrawal()

function. When it is called, the stamina of the delegatee is deducted by the requested amount, and this record remains in stamina contract.

1. Call the withdraw()

function

: depositor can call the withdraw()

function to withdraw the account balance as much as the stamina recorded in previous step.

function requestWithdrawal(address delegatee, uint amount) external onlyInitialized returns (bool) { require(amount > 0);

```
uint totalDeposit = _total_deposit[delegatee];
uint deposit = _deposit[msg.sender][delegatee];
uint stamina = _stamina[delegatee];

require(deposit > 0);

require(totalDeposit - amount < totalDeposit);
require(deposit - amount < deposit);

_total_deposit[delegatee] = totalDeposit - amount;
_deposit[msg.sender][delegatee] = deposit - amount;

if (stamina > amount) {
  _stamina[delegatee] = stamina - amount;
} else {
  _stamina[delegatee] = 0;
}

Withdrawal[] storage withdrawals = _withdrawal[msg.sender];

uint withdrawalIndex = withdrawals.length;
Withdrawal storage withdrawal = withdrawals[withdrawals.length++];

withdrawal.amount = uint128(amount);
withdrawal.requestBlockNumber = uint128(block.number);
withdrawal.delegatee = delegatee;

emit WithdrawalRequested(msg.sender, delegatee, amount, block.number, withdrawalIndex);
return true;
```

}

function withdraw() external returns (bool) { Withdrawal[] storage withdrawals = _withdrawal[msg.sender]; require(withdrawals.length > 0);

```
uint lastWithdrawalIndex = _last_processed_withdrawal[msg.sender];
uint withdrawalIndex;

if (lastWithdrawalIndex == 0 && !withdrawals[0].processed) {
  withdrawalIndex = 0;
} else if (lastWithdrawalIndex == 0) { // lastWithdrawalIndex == 0 && withdrawals[0].processed
  require(withdrawals.length >= 2);

  withdrawalIndex = 1;
} else {
  withdrawalIndex = lastWithdrawalIndex + 1;
```

```
}

require(withdrawalIndex < withdrawals.length);

Withdrawal storage withdrawal = _withdrawal[msg.sender][withdrawalIndex];

require(!withdrawal.processed);
require(withdrawal.requestBlockNumber + WITHDRAWAL_DELAY <= block.number);

uint amount = uint(withdrawal.amount);

withdrawal.processed = true;
_last_processed_withdrawal[msg.sender] = withdrawalIndex;

msg.sender.transfer(amount);
emit Withdrawn(msg.sender, withdrawal.delegatee, amount, withdrawalIndex);

return true;

}
```

2.1.3.3 Stamina Subtract, Refund and Recover

The stamina of the delegatee is subtracted to purchase the delegator gas cost, and the remaining gas is returned to the delegatee after the transaction execution.

When the delegator purchases the delegatee gas cost, the stamina of the delegatee is deducted. The stamina deduction is made by calling subtractStamina()

function from the NULL_ADDRESS.

function subtractStamina(address delegatee, uint amount) external onlyChain returns (bool) { uint stamina = _stamina[delegatee];

```
require(stamina - amount < stamina);
_stamina[delegatee] = stamina - amount;
return true;

} }
```

The subtractStamina()

function has an onlyChain

modifier. Functions with this onlyChain modifier can only be called directly by the NULL_ADDRESS, and can not be called by other accounts.

modifier onlyChain() { require(msg.sender == address(0)); _; }

After purchasing the gas-upfront cost with stamina and processing the transaction, the remaining gas is refunded to the delegatee. This is also done by calling the stamina contract function from the NULL_ADDRESS which will call the addStamina()

function. Futhermore, the addStamina()

function includes logic to check recovery epoch

.

function addStamina(address delegatee, uint amount) external onlyChain returns (bool) { if (_last_recovery_block[delegatee] + RECOVER_EPOCH_LENGTH <= block.number) { _stamina[delegatee] = _total_deposit[delegatee]; _last_recovery_block[delegatee] = block.number; _num_recovery[delegatee] += 1;

```
  return true;
}

uint totalDeposit = _total_deposit[delegatee];
uint stamina = _stamina[delegatee];

require(stamina + amount > stamina);
uint targetBalance = stamina + amount;

if (targetBalance > totalDeposit) _stamina[delegatee] = totalDeposit;
else _stamina[delegatee] = targetBalance;

return true;
```

}

The stamina of the delegatee will eventually be exhausted. But the depleted stamina can be recharged in the next epoch. It means the delegatee's stamina is reusable.

Stamina recovery condition appears at the beginning of addStamina()

function. If current epoch meets recovery condition, then total amount deposited will be regained.

if (_last_recovery_block[delegatee] + RECOVER_EPOCH_LENGTH <= block.number) { _stamina[delegatee] = _total_deposit[delegatee]; _last_recovery_block[delegatee] = block.number; _num_recovery[delegatee] += 1;

return true;

}

2.2 Transaction Execution

2.2.1 Transaction Validity

Gav Wood defined five conditions for testing the intrinsic validity of a transaction in ETHEREUM: A SECURE DECENTRALISED GENERALIZED TRANSACTION LEDGER [1].

1. The transaction is well-formed RLP, with no additional trailing bytes;
2. the transaction signature is valid;
3. the transaction nonce is valid (equivalent to the transactor account's current nonce);
4. the gas limit is no smaller than the intrinsic gas, g_0

, used by the transaction;

1. the transactor account balance contains at least the cost, v_0

, required in up-front payment.

In this transaction model, we re-defined condition 5:

1. the transactor account balance contains at least the cost, v_{v0}

, required in value-upfront payment.

1. transactor's balance is contained at least the cost v_{0}

or delegatee's stamina contain at least the cost, v_{g0}

.

If upfront cost is defined as:

v_0 \equiv T_g T_p + T_v \tag{3}

The above equation is divided into gas-upfront cost and value-upfront cost.

\begin{align} v_{g0} \equiv T_g T_p \tag{4} \ v_{v0} \equiv T_v \tag{5} \end{align}

In this case, the transaction validity can be expressed as follows:

\begin{align} S(T) \neq \varnothing \quad \land \tag{6} \ T_n = \sigma[S(T)]*n* *quad \land \tag{7}* \ *g_0 \leqslant T_g \quad \land \tag{8}* \ *v*{v0} \leqslant \sigma[S(T)]*b* *quad \land \tag{9}* \ *\bigl((v*{0} \leqslant \sigma[S(T)]*b)* *\lor (v*{g0} \leqslant [ST_{pair}F(S(T))]_{stm})\bigr) \quad\tag{10} \end{align}

2.2.2 Traditional Execution

The non-delegated execution is the same as the transaction processing in Ethereum, and it is followed:

1. get balance of transactor
2. compare with upfront cost.
3. buy upfront cost

4. execute EVM
5. refund gas

2.2.3 Delegated Execution

First of all, check whether the delegator is included in the stamina pair.

- If delegatee exists:
- get stamina of delegatee
- compare with gas-upfront cost
- buy upfront cost with stamina
- execute EVM
- refund gas
- get stamina of delegatee
- compare with gas-upfront cost
- buy upfront cost with stamina
- execute EVM
- refund gas
- else
- run Traditional Execution
- run Traditional Execution

2.2.4 Pseudo-code

Following code block is a pseudo-code representation of 2.2.3 Delegated Execution

chapter.

The tx_traditional_execute()

function contains the pseudo-code for the non-delegated execution of the traditional ethereum, and the tx_delegated_execute()

function contains the pseudo-code for the delegation execution.

def tx_traditional_execute(from, to, value, gasLimit, gasPrice, data) : # 1. check from can pay upfront cost balance = getBalance(from) assert balance >= value + gasLimit * gasPrice

```
# 2. substract upfront cost(tx.value + tx.gasLimit * tx.gasPrice)
substractBalance(from, balance - value - gasLimit * gasPrice)

# 3. execute EVM
executeVM(from, to, value, gasLimit, gasPrice, data)

# 4. collect refunded gas
addBalance(from, gasRemained * gasPrice)
```

def tx_delegated_execute(from, to, value, gasLimit, gasPrice, data) : # 0. check if (delegator, delegatee) pair exists. execute EVM. # getDelegatee() returns # or it returns delegatee = staminaContract.getDelegateeAddress(from)

```
# 1. case where delegatee exist
# Check, if `to` has delegatee check upfront cost(only gasLimit * gasPrice)
if delegatee and staminaContract.balanceOf(delegatee) >= gasLimit * gasPrice:
    # 1-1. check if `from` can pay upfront cost(only value)
    assert getBalance(from) >= value

    # 1-2. subtract upfront cost(only gasLimit * gasPrice) from delegatee
    staminaContract.subtractBalance(delegatee, gasLimit * gasPrice)

    # 1-3. substract upfront cost(only value)
    substractBalance(from, value)
```

```
    # 1-4. execute EVM
    executeVM(from, to, value, gasLimit, gasPrice, data)

    # 1-5. collect refunded gas to delegatee
    staminaContract.addBalance(delegatee, gasRemained * gasPrice)

# 2. case where delegatee does not exist
else:
    # Execute tx tranditional way
    tx_traditional_execute(from, to, value, gasLimit, gasPrice, data)
```

2.2.5 Compare with Ethereum

Traditional Ethereum

Delegated Model

Remarks

Account to buy upfront cost

transactor

transactor or delegatee

Account to buy gas-upfront

transactor

transactor or delegatee

Account to buy value-upfront

transactor

transactor

Minimum EVM Execution

1

3

3. Implementation

3.1 Implement go-ethereum, py-evm

Link : https://hackmd.io/s/HJrS5IxIm (Korean)

4. Conclusion

In PoC 1: EVM Compatible Gas In the Delegated Transaction Execution Model, we proposed an execution model to solve the usability problem, so that users do not have to worry about transaction fees. And this model also endures network attacks such as DoS.

The main feature of PoC1 is the delegation of transaction fees between stamina pair's accounts. To delegate commission, PoC1 added a new concept "Stamina". Once the stamina runs out, delegator can pay for the transaction fee on their own state balance. And it can still prevent any network attacks, because regardless of commission type, weather is stamina or balance, fees are charged for every transaction. Also, the delegatee can prevent the malicious delegator from consuming the stamina by releasing the pair designation when delegatee detects an abnormal behavior. If there is a dapp running on this model for a specific purpose, the service provider will have a number of delegatee and set the service users as delegator. Dapp's initial users don't have to worry about the transaction fee. This suggests that the proposed model of PoC1 is much more usable than the traditional execution model of Ethereum.

As we have seen in the 2.2.5 Comparison with the existing Ethereum execution model

, this model runs EVM at least three times per transaction, so if it is applied to mainnet, it becomes slower than before. Therefore, it is more suitable to use in private chain or sidechain. In order to solve this issue, we will discuss in PoC2: Plasma EVM on how to use the proposed architecture applied in sidechain.

Reference

[1] ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BYZANTIUM VERSION e94ebda - 2018-06-05

[2] Plasma MVP Audit Transacript (Script)