Enhanced Transaction Confirmation Performances without Gas by Using Ethereum Blockchain

B. Sriman
Faculty of Computer Science and Engineering, SRM Institute of Science and Technology (SRM IST), Kattankulathur, Chennai, India.
E-mail: srimanb@srmist.edu.in

S. Ganesh Kumar
Faculty of Computer Science and Engineering, SRM Institute of Science and Technology (SRM IST), Kattankulathur, Chennai, India.
E-mail: ganeshk1@srmist.edu.in

Received October 04, 2021; Accepted December 23, 2021
ISSN: 1735-188X
DOI: 10.14704/WEB/V19I1/WEB19357

Abstract

Blockchain is the distributed decentralized application for developing many of the use cases. In blockchain, the applications are transparent to all the nodes in the decentralized network and the application may be private or public depends upon the use cases. The nodes in the distributed network shares all the data to the other nodes without any modification in the blockchain technology. The transaction by the node is secured through the digital signatures. In this paper, we implemented the application by confirming the transaction without paying the gas fee by deploying the smart contract. The solidity is a JavaScript programming language for writing our smart contract. We explained the gas optimization technique used by the users and the gas station network in a decentralized application and also explained about the parameters in the Ethereum transactions. We used the open zeppelin library, which allows us to run our Dapp to implement the gasless transaction.

Keywords

Blockchain, Ethereum Platform, Gas Price, Gas Limit, Transaction without Gas.

Introduction

In the peer-to-peer network, where the Ethereum platform (Ethereum, Online) depends on blockchain network (Buterin, 2014; Di Martino, et al, 2011). Where, the data is immutable and the transactions are recorded in a public ledger named as blockchain. The blockchain technology that prompts the communication among the nodes with the
decentralized network (Buterin, 2014). Blockchain is being used in many of the areas with the broader application domains. The turning complete programs can be completed easily by the smart contracts in the Ethereum platform.

In blockchain, the blocks are made up of blocks and each block contains the data. The data is composed of header and the details of the transactions. In the peer-to-peer network, all nodes share their records to every node in the distributed network (Nakamoto, 2008). A copy of one node’s data will be shared to all the other nodes in the blockchain network. Here (Lin, et al, 2018), there is no central authority given to any of the node. As it is decentralized (Khan & Alghathbar, 2010), anyone in the network can verify the other node, whether it is valid or not. The node which is verified by the participant is called the miner (Nakamoto, 2008). The miner is the participant, who does the mining work. Here, mining is the process of solving the mathematical crypto-puzzle (Liu, et al, 2019). For every transaction confirmation, the fee should be paid for it and are called “Gas”. It is bought by using the ethers through the ERC20 tokens in the crowd sale (Wood, 2014).

Now, lets us see about gas, gas concepts and the purpose of gas in Ethereum blockchain (Li, et al, 2016).

1. GAS

The amount paid for confirming and validating the transaction depends upon the smart contract in the Ethereum blockchain is called the GAS. To include a block in the blockchain, the gas price is fixed by the miner as the amount of work done for confirming a transaction. We should first understand that some kinds of transaction require various amount of gas price for completing. Example, the transaction of sending the ETH from one node to another node with the cost of 21,000 gas, while sending the ICO (initial coin offering) tokens from our MyEtherWallet (MEW) may cost higher due to the high-level computational power (Nodejs, Online 2020). Let us see some questions about gas like when, where, who, how:

- **When the gas is paid**: The gas is paid, when a miner mines a transaction in the Ethereum blockchain and adds a block into the blockchain. Also, it is possible to add a block without paying the gas fee. We will explain in detail about it in the upcoming topics.
- **Where to get gas**: The gases were bought through the exchange of ethers in the ICO sale.
- **How to pay a gas**: From our Ether wallet, the gas fee will gets reduced depends upon the contracts.
• **Who pays for the gas**: the nodes which signs for a transaction through the smart contract pays to the node who verifies the transaction.

• **Who receives the gas**: the node which verifies the transaction receives the gas and also named as miner.

2. **GAS Limit**

The maximum amount of gas sent for confirming a transaction is called the Gas limit. If the computation power gets more complicated, then the limit of gas will also get increased. Now, let us assume that everybody is free to send transaction in Ethereum network and transaction is confirmed by a miner without any gas fee. But in such a case, the group of people may do some attack in the Ethereum, so that they have to use high computational power (Li, 2015). By using this computational power, it is very much easier for the attackers to do spam in the network and no one could use the network anymore. For avoiding this type of attack, the transaction fee is introduced to avoid the spam in the network.

Assume that there is no transaction fee and all the nodes are free to send the transaction in the Ethereum network and the miner will then mine. Here what happens, if these is a group of people that doesn’t think that Ethereum can send or buy transaction that uses lots of computational power and then the miner has to process the transaction. So, it is very easy for the attackers to spam the network and nobody can use the network anymore (Bhargavan, et al, 2016; Atzei, Bartoletti, & Cimoli, 2017), if we want to avoid this, we need to introduce the transaction fee. Now, the attacker wants to spam the network and going to cost lots of money. When two types of transactions occur, it is really hard for the miners to do the mining process.

• The Simple transaction that sends the executable smart contract.

• The complex transaction that executes some smart contract.

Now, the miner is going to get the same transaction fee for these two transactions. The miner has to go for the priority to pick the first smart contract transaction (Bhargavan, et al, 2016). The network is really congested for having lots of transactions, so the complexity of processing the transaction in the network is really hard and cannot be executed sometime for some smart contracts in the Ethereum blockchain. The solution for this type of problem is making the variables for every smart contract that we are going to execute (Lee, et al, 2017). Here, the difficulty will be based on the computational power of the smart contract execution.
In this paper, our aim is to implement the gasless transaction by deploying a smart contract in the Ethereum blockchain platform. For the upcoming section II, explains about the Ethereum gas operations, section III explains about the parameters in gas optimization technique, section IV explains about the gas optimization in smart contract, section V explains about the implementation of transaction without gas fee, section V is about building the decentralized application powered by GSN (Gas Station Network) and section VI finally the conclusion.

**Ethereum Gas Operation**

The computational difficulty of every operation in the EVM (Ethereum Virtual Machine) are measured to an absolute price but also gives a relative price. Means, that it takes computational power twice when it goes for add operation with the relative price in ethers shown in figure 1. The unit of ether is *wei, gwei, etc.*

![Figure 1 Sequence diagram of Gas](http://www.webology.org)

**Gas Price in Units**

Here, the gas price refers to the ether, in which we pay for the transaction in units is gas (Wood, 2014). It is measured in “gwei”. The gas price analogy depends on the previous gas limit paid and are explained in the table 1.

1. One unit of *wei* is equal to: $1 \text{wei} = 10^{(-18)} \text{ether ETH}$
2. One unit of *mwei* is equal to: $1 \text{mwei} = 10^{(-15)} \text{ether ETH}$
3. One unit of *gwei* is equal to: $1 \text{gwei} = 10^{(-9)} \text{ether ETH}$
Table 1 Measure of gas in units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Wei Value</th>
<th>Ether Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wei</td>
<td>1 wei</td>
<td>1</td>
</tr>
<tr>
<td>K weí</td>
<td>10 ∧ 3 weí</td>
<td>1,000</td>
</tr>
<tr>
<td>M weí</td>
<td>10 ∧ 6 weí</td>
<td>1,000,000</td>
</tr>
<tr>
<td>G weí</td>
<td>10 ∧ 9 weí</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Micro ether</td>
<td>10 ∧ 12 weí</td>
<td>1,000,000,000,000</td>
</tr>
<tr>
<td>Milli ether</td>
<td>10 ∧ 14 weí</td>
<td>1,000,000,000,000,000</td>
</tr>
<tr>
<td>Ether</td>
<td>10 ∧ 16 weí</td>
<td>1,000,000,000,000,000,000</td>
</tr>
</tbody>
</table>

The most used unit to exchange the gas price is gwei. As mentioned above in the table gwei, wei, kwei are the three important units in ether. One ether is divided into 1080 wei units, which is equal to 1,000,000,000,000,000,000 (i.e., Equal to one ether). And one ether is equal to 135 USD. In Indian rupees it is equal to ₹ 10,398.

Parameters in Ethereum Transaction

There are two parameters in the Ethereum transactions. They are:

a. Gas price
b. Gas limit

1. Gas Price

The gas price is calculated by multiplying the gas price (gas spent) with the gas limit (in gas) (Easley, O’Hara, Basu, 2019).

\[
\text{Gas cost (ether)} = \text{gas price} \times \text{gaslimit(gas)}
\]

Before entering into the method, we have to set the current gas price from the ether gas station called “ethgasstation.info” (Ethereum, 2018). Where, the gas station is continually monitoring the blockchain and calculates the gas price that are mostly accepted by the miners. In the peer-to-peer transaction, the three types of gas prices as recommended.

1. Fast (40 gwei): our transactions will be picked up immediately after the blocks.
2. Average (4 wei): our transaction will be picked up after next few blocks.
3. Slow (0.6 wei): our transaction will be picked up after many of the blocks.

Example

Fast < 2M
$ 0.038 /transfer
TX CAL
2. Gas Limit

The usage of gas is set to be in limit per transaction. Let us see the methods as follows:

\[
Gas \cost \( ether \) = gas \ price \ast \min \( gas \ cost, \ gas \ limit \)
\]

- **Method 1: The Estimate Gas () Method of Web3 Library**

  The truffle contract automatically estimates the gas of web3 that is necessary for the transaction and the parameter for the transactions (Web3ETHlibrary). So, the different gas amount is needed for every transaction. We have to turn it off, when it is for different transactions. It will be like methods.my, *Method estimate gas.*

- **Method 2: Formula for Gas Limit**

  According to Ethereum Yellow Paper (Wood, 2014), in order to calculate gas limit for transactions with non-zero data, we need to use the following formula:

  \[
  Gas \ Limit = G_{\text{transaction}} + G_{\text{tx data non zero}} \ast dataByte \ Length
  \]

  Where

  \[
  G_{\text{transaction}} = 21000 \ gas
  G_{\text{txdata non zero}} = 68 \ gas
  \]

  \[
  dataByte length \rightarrow your \ data \ size \ in \ bytes
  \]

  So, the final formula would look like this:

  \[
  GasLimit = 21000 + 68 \ast dataByte length
  \]

  If your data has 10 bytes, the gas limit would be:

  \[
  GasLimit = 21000 + 68 \ast 10 = 21000 + 680 = 21680
  \]

**Metamask in DAPP**

Here, we are connected with the metamask (Metamask, 2020) for the distributed network. It gets the notification that when the contracts are deployed, the gas price got deducted and the gas limit of web3.

3. **Ethereum Gas Cost for each Opcode**

In the Ethereum, every transaction gas cost is spent in the form of ether (Buterin, 2014). The consumption of gas depends upon the opcodes, that are executed by the Ethereum
Virtual Machine (Ethereum Foundation, 2013). Some of the opcodes and their gas values are mentioned below in the table 2.

### Table 2 Ethereum gas cost for each opcode

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>Gas</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/SUB</td>
<td>3</td>
<td>Arithmetic operation</td>
</tr>
<tr>
<td>ADDMOD/MULMOD</td>
<td>8</td>
<td>Bitwise operations</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>5</td>
<td>Arithmetic operation</td>
</tr>
<tr>
<td>POP</td>
<td>2</td>
<td>Arithmetic operation</td>
</tr>
<tr>
<td>PISH/ DIP</td>
<td>3</td>
<td>Stack memory</td>
</tr>
<tr>
<td>AND/ OR</td>
<td>3</td>
<td>Stack memory</td>
</tr>
<tr>
<td>MLOAD/ MSTORE</td>
<td>3</td>
<td>Memory operation</td>
</tr>
<tr>
<td>CALL</td>
<td>25,000</td>
<td>Create a new account using CALL</td>
</tr>
<tr>
<td>CREATE</td>
<td>32,000</td>
<td>Create a new account using CREATE</td>
</tr>
<tr>
<td>BALANCE</td>
<td>400</td>
<td>Get balance of an account</td>
</tr>
<tr>
<td>SLOAD</td>
<td>200</td>
<td>Storage operation</td>
</tr>
<tr>
<td>SSTORE</td>
<td>5000/20,000</td>
<td>Storage operation</td>
</tr>
<tr>
<td>LE/ LT/ GE/ EQ</td>
<td>3</td>
<td>Comparison operation</td>
</tr>
<tr>
<td>JUMP</td>
<td>3</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>JUMPI</td>
<td>10</td>
<td>Conditional jump</td>
</tr>
</tbody>
</table>

4. **Ethereum Transaction Summary**

After the completion of the transaction and the gas fee, the summary of those transaction is shown in figure 2 etherscan.io website (Ethereum, 2018).

![Ethereum Transaction Summary](http://www.webology.org)
Gas limit: the amount is fixed as a limit for the users to the transactions. The gas amount 21,000 is fixed as a default amount.

Gas used for transaction: this is the amount that is used by the user to execute their transaction.

Gas price: the unit of gas paid by the user through the ethers as *gwei*. The users are preferred to pay 8 *gwei* for the faster and high prioritized transactions.

Actual transaction cost fee: the amount that is actually paid by the users for the transaction in the ethers (usually calculated in USD). The users usually pay 14 USD via ethers in less than two minutes.

**Gas Optimization Techniques in Smart Contract**

The consumption of gas cost sometimes goes up and sometimes falls down, so the deployment cost varies depends upon the objective of the gas optimization (Solidity, 2019).

1. **Minimization of Chain Data**

   In the smart contract (Bhargavan, et al., 2016), we are not using 100% of data in our project. We also used our systems (*i.e.*, Central server) for interacting with the smart contract. Like that, few of the process used in our application are explained as follows:

2. **Using the Libraries**

   To deploy our application, we need few of the functionalities that runs our application. Those functions will have all the dependencies in a single function for the particular application. Our smart contract will point to the single library to execute our shared functionality (Jiang, Liu, & Chan, 2018). The two kinds of libraries are embedded and deployed. In order to have our deployed library, we should make our library as an external library.

3. **Using ERC1167 Token**

   If we are deploying our smart contract many times, we need to use ERC1167 tokens (Sovbetov, 2018). It makes the system as a proxy smart contract, where we have to deploy only the actual code once and our subsequent smart contract keeps on deploying by using the code deployed first. But it points only one of the initially deployed smart contract and uses its own data of our smart contract on the blockchain.
4. Solidity Optimization

In our solidity smart contract, we have an optimization flag. That tells the solidity compiler to produce the very high optimization bytecode. If we use the bytecode, our gas price will consume less gas. In our code, we should turn on the optimization, by enabling it as true (Fraser, & Arcuri, 2012). It is because the normal optimization takes more time that in computing this type of optimization.

```solidity
Module.exports
{

|

{solc:

Optimizer:

Enable: true,
Runs: 200
}
}
```

5. Using the Events

If we are having a good event, it should have the data that is created only once and no need to get updated and also no need to read from the smart contract. This shows the evidence that it consumes only less amount of gas than the normal variables. So, if we are going to verify the solidity variable, we should make sure that we are using this instead of normal standards.

6. Using the Literal Values Instead of Using Computed Values

Here, we know that the codes of our smart contract will be stored inside the case of hashes. If we want to compile the data, we have to give the values inside the functions. Which means with the static string to instantiate the variable with the hash value. This will reduce the consumption gas during compilation.

7. Avoiding the Array Copies in the Memory

In our smart contract, we will be using the array for storage and we manipulate it. But, if we use storage to point an array, it will reduce the space and also the gas spent for searching it. So, we should avoid copying the arrays in the memory.
8. Avoiding Loop Over Dynamic Ranges

Let us consider, that our smart contract which has a function of full loop inside and it iterates over a value named as dynamic. When we deploy our smart contract, initially it takes more time to interact. After deployment few of the data will be consumed is higher in our smart contract. So, our data will also get iterated multiple times for first time. Eventually, the gas price will also be increased at a point. So, we have to avoid using the looping over the data in dynamic ranges.

9. Optimizing the Order of Variable Declaration

Our smart contract in the Ethereum virtual machine stores the data in the slots of 256 bits (Ethereum virtual machine, Online). So, the elementary types like integers can fit in one single slot. For the other types, it spans for several slots. But, if we have several elementary types, which can adopt in a single 256 bit (Lin, et al., 2018), it will do optimization of assigning it in single slot. In the example given below, it is shown that the two 128 bits can be assigned into a single slot as 256 bits. But, in the second code, we cannot assign the values in a single slot because, it comes in the order of 128 bits and 256 bits. So, it is not possible to assign it in a single slot.

Example

<table>
<thead>
<tr>
<th>Slot 1</th>
<th>Uint256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot 2</td>
<td></td>
</tr>
<tr>
<td>Slot 3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slot 1</th>
<th>Uint128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot 2</td>
<td>Uint256</td>
</tr>
<tr>
<td>Slot 3</td>
<td>Uint128</td>
</tr>
</tbody>
</table>

Code

```solidity
pragma solidity ^0.5.0;
{
    contract {
        Uint 128 a;
        Uint 128 b;
        Uint 256 c;
    }
}
```

Code

```solidity
pragma solidity ^0.5.0;
{
    contract {
        Uint 128 a;
        Uint 256 b;
        Uint 128 c;
    }
}  ```
10. Using npm Packages

Gas reporter, usually called the npm package that shows all the gas consumptions of each
and every function when we run smart contracts. So, it is very important for us to monitor
the report of the npm package (Nodejs, 2020). While observing it, we can identify it
whether we have consumed the gas in higher or lower level.

In this paper, we implemented a transaction that didn’t use any amount of the gas through
the gas station network by using the OpenZepleline library (Open Zeppelin, online). In the
blockchain, we built a decentralized application that completes a transaction without using
the gas. We know that in Ethereum blockchain, it specially needs wallets to pay for it. But
we used the metamask as user account here. It will be explained in the upcoming sections
(Shie, et al., 2014).

Gasless Transaction

For implementing the gasless transaction, we have to know mainly the two transactions.
One is gasless and the another one is meta transaction.

1. Gasless transaction: where user’s no need to pay gas fees for the transactions.
2. Meta transaction: relayer – where a third party can pay another user transaction
   and pay themselves for the gas with an open zeppelin library.

![Figure 3 Gasless transaction sequence diagram](image)

a. Gasless Transactions

In blockchain Ethereum platform for every transaction, we have to pay gas for confirming
it. This method forces the users to buy the ethers before they start using the distributed
applications. This is the major issue in blockchain. In this paper, (Easley, O'Hara, & Basu, 2019) we mainly focus on the gasless transaction shown in figure 3. For this, we are using a library called OpenZeppelin (Open Zeppelin, online). Which allows us to do our transactions without any gas fee.

b. Meta-transaction

In Ethereum transaction, all the senders use the ethers to pay for transaction as gas fee. Even, when the gas costs may low, but getting the ethers is not an easy task. Few of the decentralized application users use the KYC (Know Your Key) and AML (Anti-Money Laundering) in which it not only involves in time, but also involves in holding the user proofs over the internet (Shie, et al., 2014).

Meta-transactions – The third party can send another user’s transaction and pay themselves for the gas cost and are named as relayer. In this process, the users sign the messages (not including the transactions) contains the information about the transaction and they would like to execute. Where, relayer also has the responsibility of signing themselves to the network and paying for the gas cost. The identity of the user that originally requests the transaction are preserved by the base (smart) contracts. Through this way, the users can directly interact with the smart contracts without any of their wallet or their ethers. It means that the meta-transaction in our application needs to keep a relayer.

c. The Gas Station Network

The decentralized network of relayer is called Gas Station Network. It allows us to build the decentralized application, where we can pay for our user’s transaction. So that the user need not pay for gas as ethers. The TABOOKEY designed the GSN and has the grown to encompass many companies within the Ethereum platform to solve the problems of onboard user nodes to the decentralized applications. The charities are not run by relayer in the GSN but running their business. The reason behind the business is that they will get back the gas cost in the charge of their contract. Will also get extra rewards for their services (Di Martino, et al., 2011). Here, the money spent is on free trail, discounts for new users, advertisements, etc… with all the goals of acquisition. In comparison with all those transactions, the cost is very small for the couple of Ethereum transactions. Furthermore, the GSN leverages the users to pay off-chain in advance with deduction from the user’s balance on their systems. If any of the system misbehaves, then the relayer will serve for the user’s request and there is a measure in place to penalize them.
It is runs automatically. So, we can start using is it without any fear. In the following sections, we will learn more about interacting with the RelayHub.

**Building a Gas Station Network**

Now, we are building a *dapp* leveraging the GSN and push it to a *testnet*. The procedure for building the GSN network are as follows:

1. The package named create-react-app is used to bootstrap a React application, along with *OpenZeppelin* Network JS to easily set up a web3 object with GSN support (Web3ETHlibrary).
2. To emulate the GSN in our local ganache instance the *OpenZeppelin* GSN helpers are used.
3. The smart contract library to get GSN is *OpenZeppelin /contracts-Ethereum-package* (Open Zeppelin, online).
4. To manage and deploy our contracts, we use the *OpenZeppelin* CLI.

Here, a simple contract is created and that counts the transactions sent to it. Then, it ties up into the GSN. So, the users need not pay for the transaction gas fee.

**1. Setting up the Environment**

1. Install the dependencies with node install node.js and install *npm* package (Nodejs, 2020).
2. While installing the node in our terminal, use the command “*npm install node*”.
3. To check the version, use the command “*node -v*”.
4. Create a new a directory by using the command *mkdir* name call GSN.
5. The command *npm init-y* is used to create a *package.json* file and – *y* means to say all the prompting called-y flag.
6. To install open zeppelin network, use the command “*npm install @openzeppelin/network*” (Open Zeppelin, online).
7. Next command “*npm install --save-dev @openzeppelin/gsn-helpers @openzeppelin/contracts-Ethereum-package @openzeppelin/upgrades @openzeppelin/cli ganache-cli*” Which will put in our package.json file, open zeppelin propriety upgrade, command line interface and also ganache-cli client application, after that install the dev dependencies.
2. Creating our Smart Contact

Our smart contract is written in the solidity programming language (Solidity, 2019). The version of it is used between 0.4.0 and 0.7.0 (Ethereum Community, 2016). The following are the steps for creating our smart contract. By clicking on the increase button, it takes one value and adds it to the blockchain, when a transaction is confirmed (Ethereum, 2018). These functions are written in our smart contract and executed in the Ethereum blockchain platform. In our contract, it requires extension from the gas station network recipient and implementing our acceptrelayedcall.method. This method returns whether we accept or reject to pay for the user transaction. In our contract, we extend it called GSN (Gas recipient). It is a library in which solidity is provided by OpenZeppelin that allow us to create a special function that works well with gas station network. Specifically, it implies that accept elayer call which allow us to introduce the elayer in order to pay for the user’s gas fee. Now, we have imported this contract with library OpenZeppelin gas station network recipient.

3. Starting the Ganache-cli

Ganache is a user interface for the Blockchain in Ethereum platform. We started ganache in the command line by using the command npx ganache-cli. Blockchain that runs in our computer by the ganache and gives the users ten account for free with 100 ETH to test our cryptocurrency by using the private key. Now, the ganache is started on another terminal, create an instance for our smart contract by using OpenZeppelin cli library. It is created with the command npx oz create and it also follows the prompts like choosing to call the function for initializing the instance. We have to keep note on the address of our instance for future process. It is important that we should remember to call the initialize() function while creating the contract, as this will get our contract ready to be used in the GSN.

Command: Npx ganache-cli.

4. Creating the Contract Instance

Our contract is initialized by using the command npx oz init. Here, oz refers to the command line interface that we have already installed our open zeppelin library by using the node package (Ganache; Open Zeppelin). We are going to write the smart contract through the ganache development in blockchain. It will create the instance address for the given counter contract.sol file. While entering the instantiate counter contract, we pick a network to develop the network. After that counter contract will be deployed in the GSN network. All our files have been deployed, then it will call a function to initialize after
creating the counter \textit{contract.sol}. It will provide us the contract address to upgrade to the network.

Command: Npx oz create.

Now, if we deploy this contract by using the main network, \textit{rinkeby} or any of the test network, we would almost ready to start sending the gasless transactions to it. Since we are using the GSN, it is already set up on both of those networks. However, we are on a local ganache, we need to set it up by ourselves.

5. Deploying the Local GSN for our Smart Contract

The central \textit{relayhub} contract is contained by Gas Station Network and it also coordinates the transactions along with the multiple decentralized \textit{relayer}. The \textit{relayer} receives request from the relay transaction through the HTTP interface and also sends it back to the network through the relay Hub. By using the command – \textit{npx oz-gsn run- relayer} (Open Zeppelin, online), we start our \textit{relayer} by running the ganache. To take care of these steps we need to handle the local relayer up and running this command. Initially, the \textit{relayer} binary for our Ethereum platform gets started and then deploys the relayhub contracts in the ganache. After registering in the \textit{relayer} on the hub and funding, the relay transaction gets started. By this command: \textit{oz-gsn}, we run the steps individually.

6. Funding our Counter Contract

The last step is funding in our counter contract. Funding the contract means that we create the gas station network \textit{relayer}, that it requires recipient contract to move funds since they to change the cost of the \textit{relayer} transcation (plus a fee). This is how it work and we have counter contract. By sending money to this counter contract the fees are going to be taken, whenever users interact with it. This won’t have to pay directly for the gas cost but the \textit{relayer} needs to make the money bank (Amato, 2014). So, they can change the contract not only with the user directly. But, through business they may change the user’s (or) our contract might pay fees entirely depending on what our business model. Account address in ganache-cli in smart contract also has the address. They allow us to hold the crypto currency for funding by using the smart contract and charging the fees with \textit{relayer}, to recipient address. Now, we are going interact with smart address and ganache cli blockchain address by using the command: “\textit{npx oz-gsn fund-recipient}”. 
Recipient address command adds some cryptocurrency to our smart contract and that will
create the balance of one ether as \(10^{8}\) decimal. Where, 18 is one ether.

Command:

\[
npx \quad oz\text{-}gsn \quad \text{fund-recipient} \rightarrow \text{recipient}
\]

0x8649958f78A5211f2f2065378b13a474B8636d1b

The above figure 4 screenshot shows that the local network is connected with the new provider
created by using the Open Zeppelin network JS (Open Zeppelin, online). To sign the
transactions on behalf of the users, the user uses the key generated and also the GSN to
relay on the network. It allows the users to start interacting with the distributed
decentralized application even if the users are not using the metamask (Di Martino, et al., 2011). Metamask, an Ethereum account or any form of ethers. By setting up with all the
providers, we can also pass the dev (a true flag) to the gas station network. Instead of
the GSN provider, it uses the \(npx oz\text{-}GSN\text{DevProvider}\). It is a provider to for
specifically setting up the development and also it does not require a relayer to work for
running. The actual GSN experience makes us feel easier for deployment. For running the
actual relayer, the command \(npx oz\text{-}gsn\) is used to run relayer locally. We should keep in
mind to maintain both the ganache and relayer up and running. Now, we could be able
to send or receive transactions to our counter contracts without using the metamask or by
using any of the tokens at all.

7. Test Network

Now, we are moving to the test network. Well, it is not very impressive to send a local
transaction in our ganache network, where we already have bunch of fully-funded
accounts. To witness the GSN at its full potential shown in figure 5, let’s move our
application to the local host (Ethereum online). We need to create a new entry in
the networks.js file with an account that has been funded. We can now deploy
our Counter contract to localhost. We are using a real GSN provider rather than our
developer environment. So, we want to provide a configuration object, which will give us more control over things such as the gas price, we are willing to pay.

```
Sucess! Created client at /Users/sriman/Desktop/client
Inside that directory, you can run several commands:

    npm start
    Starts the development server.

    npm run build
    Bundles the app into static files for production.

    npm test
    Starts the test runner.

    npm run eject
    Removes this tool and copies build dependencies, configuration files and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

    cd client
    npm start

Happy hacking!

srimans-Air:desktop sriman$
```

Figure 5 Screenshot of successfully create client dapp

```
Compiled successfully!
You can now view client in the browser.

Local: http://localhost:3000
On Your Network: http://192.168.15.15:3000

Note that the development build is not optimized. To create a production build, use npm run build.
```

Figure 6 Screenshot of run in localhost:3000

The above figure 6 and figure 7 shows that the application runs on the local host (http://localhost:3000). To run the application, we have to give the command: npm start. The dev server will launch and automatically open a new browser containing our dapp GSN and opens a new browser window with our client-side application. The core functionality of this function is to increase the counter instance by reading the counter action struct. Out of the “instance” mapping and increasing the “counter value” by 1 with the increment operation(++). Let’s look at few things that it works with:

1. This uses the GSN Dev Provider instead of regular GSN provider.
2. `relayer` library will pay for the transaction fee in counter increasing by 1.
3. It adds instance contract address account for recipients to the fund mapping that we have created. This allows us to keep track of the counter contract in the GSN network. We can access the account that calls this function with global variable “GSN recipients” provided by solidity language.

Now, note that the GSN relayer opens a counter action window and checks that our counter contract is increased by 1 instance shown in figure 7. This is what, we actually want to submit for the transcation through GSN to the Ethereum blockchain (Visual Studio Code, 2019). Moving to cost “no gas fee, gas limit or gas price” and also no conversion fee of etgers which is equal to 2.19 USD doller. Finally, we have submitted a gasless transcation and “increased the counter by 1”, in the Ethereum blockchain by using the smart contract.

Counter counterInstance

Counter Value: 3

Counter Actions

Increase Counter by 1

Figure 7 Screenshot of gasless transaction

Conclusion

Our main goal is to implement a decentralized application that runs on Ethereum platform should have a transaction without paying the gas fees. In this paper, a gasless transaction in the Ethereum blockchain has been implemented by using the open zeppelin library through the gas station network with the simple counter contract. With open zeppelin library, we have successfully confirmed our transaction in securing the data as well as keeping it transparent to safe guard the decision taken by the users in the distributed network. The actual rate of the transactions may differ from one transaction to another transactions. Estimating the rate are the challenges for us, because a different contract with different library is required, in which vulnerabilities have to be manually classified. In future work, the lags in time and location constraints by allowing the users to do gasless meta transaction from their own Ethereum blockchain can be reduced.
References


Ethereum [online]: https://ethereum.org
Solidity, 2019. [Online]. https://solidity.readthedocs.io/en/v0.5.11
Web3ETHlibrary, [Online]. https://web3js.readthedocs.io/en/v1.2.0/web3-eth.html
Open Zeppelin, [online]. https://github.com
OpenZeppelin/openzeppelin- solidity/blob/master/contracts/token/ERC20/ERC20.sol

Authors Profile

B. Sriman is a research scholar at SRM Institute of Science and Technology SRM IST, his main research interest involve trust, privacy in distributed system in mobile device and Dapp, online social Network and blockchain main characteristics and potentials to 5G. He completed B.E in Arunai college of Engineering, Thiruvannamali in 2016. Completed M.Tech in SRM Institute of Science and Technology, Chennai in 2019.