Disperse Protocol
Optimizing gas costs of ERC-20 token transfers

Artem Kharlamov
banteg@protonmail.com

November 13, 2018

Abstract

This paper dissects the gas costs incurred when transferring tokens on EVM[1]-based networks and discusses the possible optimization strategies. It presents a concise smart contract for batch sending both native and ERC-20 tokens. It argues about the design decisions that allow to fit 2–3x more transfers in a single block. The paper also discusses how the token developers optimize for gas usage and explains what the upcoming Constantinople hard fork means for token transfers and this algorithm.

1 Background

Ethereum is a decentralized ledger that allows arbitrary programs to run on its virtual machine. These programs are called smart contracts. Program execution is instrumented and each opcode has its own cost. The amount of computation is limited by the user-provided gas limit. A contract can chain the call to other contracts allowing for complex interactions.[1]

ERC-20 is a community-developed standard interface for fungible tokens[2]. Such contracts keep track of user **balances** in its storage and make the corresponding updates on each successful call of **transfer** function.

[1]Ethereum Virtual Machine

2 Sending Tokens

2.1 Transaction data

The simplest token contract consists of a mapping containing user balances and a **transfer** function that updates it:

$$\text{balances} = (address \rightarrow uint256)$$

Sending a transaction with the attached data to the address that contains code is used to invoke a program. Programs written in Solidity expect data to begin with a four-byte function signature with the rest being passed to the function.

The signature is computed by taking the first four bytes of the keccak hash of the function name and argument types. In this case:

$$\text{bytes4(keccak(“transfer(address,uint256)”)} = \text{a9059cbb}$$

The arguments are tightly packed and are usually padded to the EVM word size of 32 bytes, so a 20-byte address is encoded with 12 leading zero bytes and a number is encoded as a 256-bit big-endian unsigned integer.

Each successful token transfer updates the balances for both the sender and the recipient and also emits a **Transfer** event. The event contains three indexed topics (event signature, sender and recipient) and the raw data field with the value sent. The topics are added to the block's bloom filter, which allows for efficient search and filtering.

The event signature is computed like this:

$$\text{keccak(“Transfer(address,address,uint256)”)} = \text{ddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef}$$

2.2 Estimating gas costs

Here is a breakdown of the most expensive operations during a token transfer. For gas costs see Table 1.

1. Invoke a transaction: $G_{transaction}$
2. Cost of the attached data: $68 \cdot G_{txdata}$
3. Load the balances: $2 \cdot G_{sload}$
4. Store the updated balances: $2 \cdot G_{sstore}$
5. Emit the Transfer event: $G_{log} + 3 \cdot G_{logtopic} + 32 \cdot G_{logdata}$

Table 1: Ethereum gas schedule[4]

Name	Value	Description
$G_{transaction}$	21000	
G_{sload}	200	
G_{sstore}	5000	if value was non-zero
G_{sstore}	20000	if value was zero
G_{log}	375	
$G_{logtopic}$	375	
$G_{logdata}$	8	per byte
G_{txdata}	4	per zero byte
G_{txdata}	68	per non-zero byte

There is more logic involved, but the rest is cheap enough to ignore in this estimation.

We assume that the address contains no zero bytes which is the most common case.

Balances are stored in the lowest denomination as unsigned 256-bit integers. Most tokens go for 18 decimal points taking after the native currency (1 ether = 10^{18} wei). The median transfer value across the recent transfers on Ethereum mainnet is 10^{20} wei (100 tokens), we'll use that value in our tests.

With these assumptions the data passed along with the transaction has 31 non-zero and 37 zero bytes which results in 2256 gas cost.

It's safe to assume that the sender has had a non-zero balance, so the only variable left is whether the recipient's balance was zero. This leaves us with a ballpark estimation of 35,412–50,412 gas:

$$G^{21000}_{transaction} + G^{2256}_{data} + G^{400}_{sload} + G^{10000}_{sstore} + G^{1756}_{log} = 35412 \quad (1)$$

$$G^{21000}_{transaction} + G^{2256}_{data} + G^{400}_{sload} + G^{25000}_{sstore} + G^{1756}_{log} = 50412 \quad (2)$$

This is not far from truth, the most popular implementation of ERC-20 standard by OpenZeppelin[3] consumes 36,947–51,947 gas. It is possible to fit 154–216 transfers in a block with the current block gas limit of 8,000,000.

2.3 Advanced transfers

A user can give another account the right to spend up to a certain amount on her behalf. This concept is called allowance. Allowances are kept in the contract storage in a mapping that maps the addresses to spenders to their remaining allowance:

$$\text{allowed} = (address \rightarrow (address \rightarrow uint256))$$

A user invokes **approve** passing the spender address and a maximum allowance as the arguments. Then the spender can transfer the tokens by calling **transferFrom** passing the user address, a recipient and a value.

This is a common pattern for interacting with smart contracts. Instead of calling **transfer** function which doesn't notify the recipient, the user calls a contract which chains the call to the token contract's **transferFrom** and makes the operation on behalf of the user.

The gas costs for this function are in range of 44,400–59,400, or up to 134–180 transfers per block. The main addition is 5000 gas for updating allowance.

3 Batch transfers

3.1 Simple approach

A simple approach would be to create a contract that receives a token address, a list of recipients and a list of values and calls **transferFrom** while iterating over (recipients, values) pairwise. This would allow to to pay $G_{transaction}$ just once which makes a significant difference when approaching the block gas limit.

Algorithm 1: Disperse ERC-20 token using transferFrom

input: token, recipients, values

for $i \in [0 \ldots recipients.length)$ **do**
  require(token.transferFrom(sender, recipients[i], values[i]))
end

This function is based on a simplified Multiplexer contract[7]. It removes the unnecessary assumptions about the recipients and values arrays as transaction reverts on out of bounds access anyway[8]. It

also uses uint256 instead of uint8 for iterator variable, which saves some gas because no additional type conversion is needed.

One such transaction could accommodate 206–337 transfers with an average gas cost of 23,694–38,739 per transfer. This is already a 1.34–1.56x improvement over direct transfers.

3.2 Optimized approach

A contract can use just one **transferFrom** to itself and then distribute the tokens from its own address using **transfer**. We can ensure atomicity by using a **require** statement that reverts the transaction if any of the transfers fails.

Algorithm 2: Disperse ERC-20 tokens using transfer

input: token, recipients, values
$total \leftarrow 0$
for $i \in [0 \dots recipients.length)$ **do**
  $total \leftarrow total + values[i]$
end
require(token.transferFrom(sender, this, total))
for $i \in [0 \dots recipients.length)$ **do**
  require(token.transfer(this, recipients[i], values[i]))
end

Despite the need to iterate over the list twice, this approach saves 5000 gas on each transfer because **allowance** is only touched once. The side effect of this is that the Transfer events will have the contract's address as a sender.

This approach allows to fit 242–449 transfers in a block and averages down the cost of one transfer to 17,813–32,928 gas. This is a 1.58–2.07x improvement over regular transfers and 1.18-1.33x improvement over a simpler approach.

3.3 Visual comparison

The chart below presents a comparison of two methods with n **transfer** transactions. The area shows the bounds of the worst (all recipients had zero token balances) and the best cases (everyone had a non-zero balance).

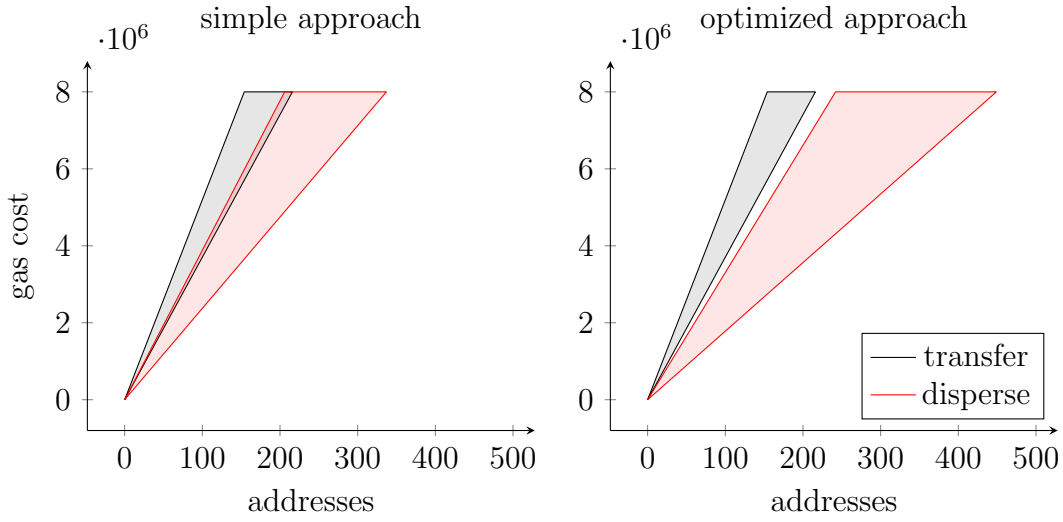

Figure 1: Gas profile of ERC-20 token transfers

3.4 Caveats

ERC-20 interface dictates that all transfer functions should return *true* on successful transfer.

However, there are at least 130 live tokens that don't follow this rule and return nothing. The problem wasn't discovered for a long time because these tokens behaved as expected[5]. By pure coincidence the memory slot where the return value was expected contained the function selector which has always evaluated to *true*.

Byzantium hard fork in November 2017 introduced a new opcode (RETURNDATASIZE) that enables reading of dynamically-sized data. Contracts compiled with solc[2] $\geq$ 0.4.22 are checking the size of the returned data instead or blindly reading a chunk of memory. Such contracts won't be able to interact with these tokens. Some of the more notable tokens affected are BNB and OMG.

The problem is fixable with inline assembly that handles both cases, but this fix is not a part of the contract presented for the sake of code simplicity.

[2]Solidity compiler

4 Further Optimization

4.1 Unlimited Allowance

Allowance is oftentimes used as a binary switch. A common pattern is to **approve** uint256_{max} to indicate unlimited allowance:

Allowance	Interpretation
0	disabled
$2^{256} - 1$	enabled

There are at least two live ERC-20 tokens that utilize this pattern to optimize for gas usage. 0x protocol's own ZRX token interprets $2^{256} - 1$ as a magic value meaning "unlimited allowance" and skips updating the allowance altogether in this case[9]. This saves 5000 gas by skipping one SSTORE and equates the cost of **transfer** and **transferFrom**.

Another example is Wrapped Ether (WETH), which primary function is to trustlessly swap the native token to its ERC-20 form or back.

4.2 Constantinople

Constantinople is a code name for Ethereum hard fork which is set to go live mid-January 2019. It is already live on Ropsten testnet at the time of publication. Among other improvements, the fork changes the gas metering rules for SSTORE, discussed in EIP-1283[6].

The new gas costs and refunds for storage writing are based on the combination of three values: original, current and new. The logic became much more complex, but the gist is that the subsequent writes to the same storage slot in scope of a single transaction became much cheaper.

This affects the proposed contract in a major way. Consider the transitions of the contract token balance during the execution of an example transaction transferring three tokens to three recipients.

The number shows the balance value in storage. The upper number shows the gas cost of the SSTORE and the lower number shows the refund for clearing storage.

$$0 \xrightarrow{20000} 3 \xrightarrow{5000} 2 \xrightarrow{5000} 1 \xrightarrow[-15000]{5000} 0 = 20000 \text{ gas} \tag{3}$$

This is how it looks with EIP-1283 metering:

$$0 \xrightarrow{20000} 3 \xrightarrow{200} 2 \xrightarrow{200} 1 \xrightarrow[-19800]{200} 0 = 800 \text{ gas} \tag{4}$$

In generalized form with n recipients:

$$G_{sset}^{20000} + n \cdot G_{sreset}^{5000} - R_{sclear}^{15000} = 5000 \cdot (n+1) \tag{5}$$

$$G_{sset}^{20000} + n \cdot G_{snoop}^{200} - R_{sresetclear}^{19800} = 200 \cdot (n+1) \tag{6}$$

The new metering rules result in a 25x improvement of the total cost of updating the contract token balance.

Below is a comparison of how this change affects the performance of the proposed contract.

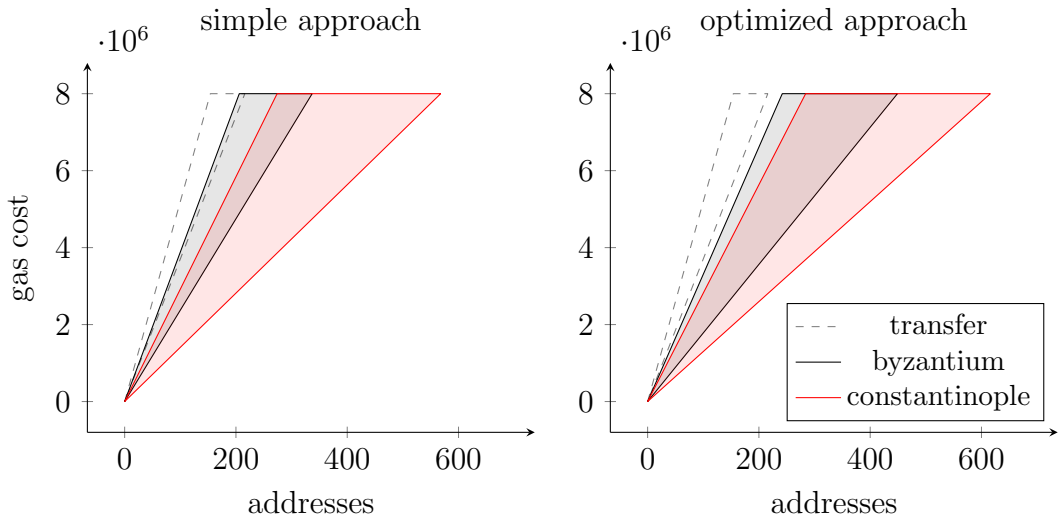

Figure 2: EIP-1283 effect on algorithms 1 and 2

The overall effect is improvement from 242–449 transfers to 284–616 for optimized approach, which is a 1.17–1.37x better result and a 1.84–2.85x improvement over non-batched transfers.

5 Native Token

Sending native token requires a slightly different approach. Any value sent along with the transaction immediately becomes available for the contract to use as its own balance. That's why it's important to keep track of how much was already sent to refund the excess value.

One way to approach this would be to calculate the total, similar to Algorithm 2, but a more efficient method is to always ensure that the final contract balance remains zero after transaction.

Algorithm 3: Disperse native token

input: recipients, values
for $i \in [0 \dots recipients.length)$ **do**
  recipients[i].transfer(values[i])
end
$balance \leftarrow address(this).balance$
if $balance > 0$ **then**
  msg.sender.transfer(balance)
end

The main gas costs involved are:

1. Invoke a transaction: $G_{transaction}$
2. Cost of the attached data: $64 \cdot G_{txdata}$
3. Transfer (CALL): $G_{call} + G_{callvalue} + G_{newaccount} - G_{callstipend}$
4. Get account balance: $G_{balance}$

Table 2: Ethereum gas schedule

Name	Value	Description
$G_{transaction}$	21000	
G_{call}	700	
$G_{callvalue}$	9000	if $value > 0$
$G_{newaccount}$	25000	if creates new account
$G_{balance}$	400	
G_{txdata}	4	per zero byte
G_{txdata}	68	per non-zero byte

The gas cost of each CALL[10] is in range of 7,400–32,400 gas, whereas a regular transfer costs a flat fee of 21,000 gas.

$$700 + 9000 - 2300 = 7400 \qquad \text{best case}$$
$$700 + 9000 - 2300 + 25000 = 32400 \qquad \text{worst case}$$

The $G_{newaccount}$ penalty makes it impractical to create new accounts using this algorithm, limiting the number of transfers in a block

to 230, as opposed to 380 regular transfer. This is a 1.65x worse result in terms of gas, but it's still one transaction instead of n.

Calling existing accounts from a contract allows to fit 830 transfers in a block, which is a massive 2.18x improvement over regular transfers. The resulting cost per transfer is 9,629–34,701 gas.

Constantinople hard fork has no effect on sending native tokens.

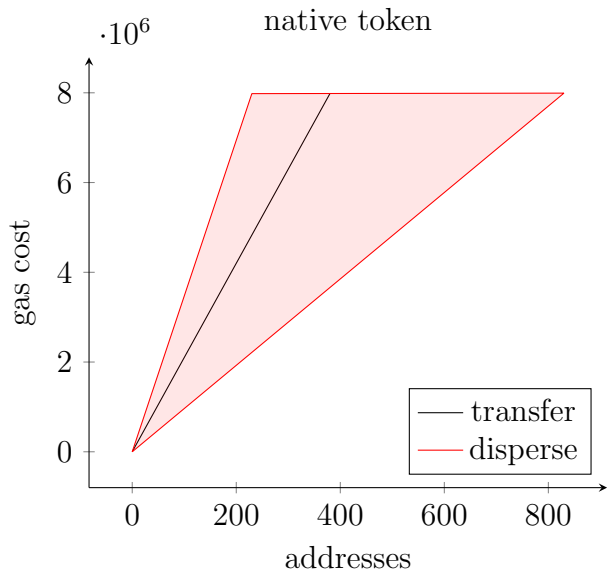

Figure 3: Gas profile of native token transfers

6 Methodology

Gas usage was instrumented using Py-EVM[11]. The benchmark setup is built from low-level primitives which include only the virtual machine and an in-memory state database. The setup allows testing against different forks by switching the virtual machine. The mining and blockchain components are not used.

This allows to have full control over the underlying state and to read and write directly to the contract storage without the need to apply hundreds of transactions to prepare for each test case.

Storage is accessed by index, in the same order as defined in the Solidity source. Mappings are accesses by keccak of their key and

index, both of which are encoded as bytes32. For example, **balances** mapping is accesses like this:

$$\text{keccak(bytes32(address) + bytes32(0))}$$

Allowances are stored in the **allowed** mapping which can be accessed by repeating that twice:

$$\text{keccak(bytes32(spender) + keccak(bytes32(address) + bytes32(1)))}$$

There is a helper function that simplifies mapping access, but be aware that it doesn't cover all the possible data types. Consult this article[12] if you plan to implement test cases for your contracts.

All tests use binary search to find the maximum number of transfers that can be fitted in a single block. The complete test suite can be found on Github.

7 Disperse

The smart contract discussed is deployed to Ethereum mainnet and all its testnets, as well as multiple other EVM-compatible networks at the same address: 0xD152f549545093347A162Dce210e7293f1452150.

There is also a client-only frontend interface hosted at disperse.app which interacts with Metamask.

References

[1] Ethereum Project, 2015

[2] Fabian Vogelsteller, Vitalik Buterin. *EIP 20: ERC-20 Token Standard*, 2015

[3] OpenZeppelin-Solidity *Standard ERC20 token*, 2018

[4] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*, Byzantium revision, 2017

[5] Lukas Cremer. *Missing return value bug–At least 130 tokens affected*, 2018

[6] Wei Tang. *EIP 1283: Net gas metering for SSTORE without dirty maps*, 2018

[7] DigixGlobal. *Multiplexer*, 2017

[8] Solidity Documentation. *Error handling in Solidity*, 2018

[9] 0x Project. *Unlimited Allowance Token v1*, 2017

[10] Vitalik Buterin. *Ethereum Subtleties*, 2015

[11] Py-EVM. *A Python implementation of the Ethereum Virtual Machine*, 2018

[12] Darius. *How to read Ethereum contract storage*, 2017

A Summary Tables

Here is a summary of the strategies outlined in this paper. All numbers assume a transaction filling the whole block gas limit.

Table 3: Gas usage per transfer

Function	Byzantium	Constantinople
transfer	36,947–51,947	36,947–51,947
transferFrom	44,400–59,400	44,400–59,400
simple	23,694–38,739	14,082–29,145
optimized	17,813–32,928	12,977–28,092
native	9,629–34,701	9,629–34,701

Table 4: Maximum transfers per block

Function	Byzantium	Constantinople
transfer	154–216	154–216
transferFrom	134–180	134–180
simple	206–337	274–568
optimized	242–449	284–616
native	230–830	230–830