Disperse Protocol
Optimizing gas costs of ERC-20 token transfers

Artem K
banteeg@gmail.com

November 27, 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 would allow to fit 2–3x more transfers in a single block. It also discusses how the token developers optimize for gas usage and provides benchmarks against Byzantium and Constantinople virtual machines.

1 Background

Ethereum is a decentralized ledger that allows arbitrary programs to run on its virtual machine. These programs are called smart contracts. In addition to balance and nonce, contract accounts have an associated EVM code and storage. A contract can chain calls to other contracts allowing for complex interactions.[1]

The base unit of computation is called gas. Program execution is instrumented with each opcode having its own gas cost. The user provides a gas limit and a gas price, the maximum amount of computation allowed and the cost per unit of gas respectively. Transaction fee is the product of the actual gas used and the gas price.

ERC-20 is a community-developed standard interface for fungible tokens[2]. Such contracts keep track of user balances in its storage and

[1]Ethereum Virtual Machine

make the corresponding updates on each successful **transfer** function call. The state-changing operations are are disincentivized with higher gas costs.

A lot of effort is currently directed at scaling Ethereum. This paper proposes a ready-to-use batching protocol for both native (ETH) and ERC-20 tokens which allows to fit 2–3x more transfers in a single block. It can aid heavy gas consumers like centralized exchanges to significantly cut their costs.

2 Sending tokens

2.1 Single transfers

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 the function arguments.

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

$$\text{function} = \text{“transfer(address,uint256)”}$$
$$\text{signature} = \text{bytes4(keccak256(function))} = \text{a9059cbb}$$

The arguments are tightly packed and padded to 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 emits a **Transfer** event. The event contains three indexed log topics (event signature, sender and recipient) and the raw data field with the amount sent. The topics are added to the block's bloom filter which allows for efficient search and filtering.

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 stored as a map of addresses to spenders to their remaining allowance:

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

A user invokes **approve** function with a 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.

2.2 Estimating gas costs

Table 1: Ethereum gas schedule[3]

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
$G_{txdatazero}$	4	per zero byte
$G_{txdatanonzero}$	68	per non-zero byte

Let's break down the most expensive operations during **transfer**. For gas costs see Table 1.

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

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:

$$G_{data} = 31 \cdot G_{txdatanonzero} + 32 \cdot G_{txdatazero} = 2256 \text{ gas} \quad (1)$$

It's safe to assume that the sender's token balance is non-zero, 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:

$$G^{21000}_{transaction} + G^{2256}_{data} + G^{400}_{sload} + G^{10000}_{sstore} + G^{1756}_{log} = 35412 \text{ gas} \quad (2)$$

$$G^{21000}_{transaction} + G^{2256}_{data} + G^{400}_{sload} + G^{25000}_{sstore} + G^{1756}_{log} = 50412 \text{ gas} \quad (3)$$

This is not far from truth, the widely-used OpenZeppelin[4] ERC-20 implementation 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.

The gas cost for **transferFrom** function is in range of 44,400–59,400, or up to 134–180 transfers per block. The biggest addition is 5000 gas for updating allowance.

3 Batch transfers

3.1 Simple approach

The gas costs can be reduced by invoking multiple transfers at once. A simple approach is to create a contract that receives a token address, a list of recipients and a list of values and calls token's **transferFrom** while iterating over recipients and values pairwise. This allows to to pay $G_{transaction}$ just once which makes a significant difference when approaching the block gas limit.

Algorithm 1 is based on a simplified Multiplexer contract[5]. It removes the unnecessary assumptions about the recipients and values arrays as transaction reverts on out of bounds access anyway[6]. It also uses uint256 instead of uint8 for iterator variable which saves some gas because no additional type conversion is required.

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.

Algorithm 1: Disperse ERC-20 tokens using transferFrom

input: token, recipients, values

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

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, Algorithm 2 saves 5000 gas on each transfer because **allowance** is only touched once.

This approach allows to fit 242–449 transfers in a block and reduces 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

Table 2 and Figure 1 present a comparison of two methods with n **transfer** transactions as a baseline. The areas on the chart show the bounds of the worst (all recipients had zero token balances) and the best cases (everyone had a non-zero balance).

Table 2: ERC-20 transfers with gas limit → 8,000,000

Method	Transfers	Gas/transfer	Transactions
transfer	154–216	36,947–51,947	154–216
transferFrom	134–180	44,400–59,400	134–180
disperseTokenSimple	206–337	23,694–38,739	1
disperseToken	242–449	17,813–32,928	1

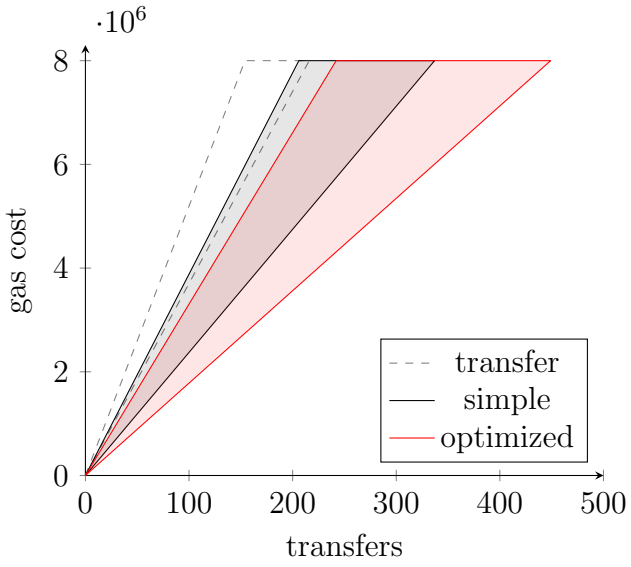

Figure 1: Gas profile of Algorithms 1 and 2

4 Further Optimization

4.1 Constantinople

Constantinople is the name of 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[7].

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

$$0 \xrightarrow{20000} 3 \xrightarrow{5000} 2 \xrightarrow{5000} 1 \xrightarrow[-15000]{5000} 0 = 20000 \text{ gas} \tag{4}$$

This is how it looks with EIP-1283 metering:

$$0 \xrightarrow{20000} 3 \xrightarrow{200} 2 \xrightarrow{200} 1 \xrightarrow[-19800]{200} 0 = 800 \text{ gas} \tag{5}$$

In generalized form with n recipients:

$$G_{sset}^{20000} + n \cdot G_{sreset}^{5000} - R_{sclear}^{15000} = 5000 \cdot (n+1) \tag{6}$$

$$G_{sset}^{20000} + n \cdot G_{snoop}^{200} - R_{sresetclear}^{19800} = 200 \cdot (n+1) \tag{7}$$

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

See Table 3 and Figure 2 for a comparison of how this change affects the performance of the proposed contract.

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

Table 3: ERC-20 transfers with Constantinople VM

Method	Transfers	Gas/transfer	Transactions
transfer	154–216	36,947–51,947	154–216
transferFrom	134–180	44,400–59,400	134–180
disperseTokenSimple	274–568	14,082–29,145	1
disperseToken	284–616	12,977–28,092	1

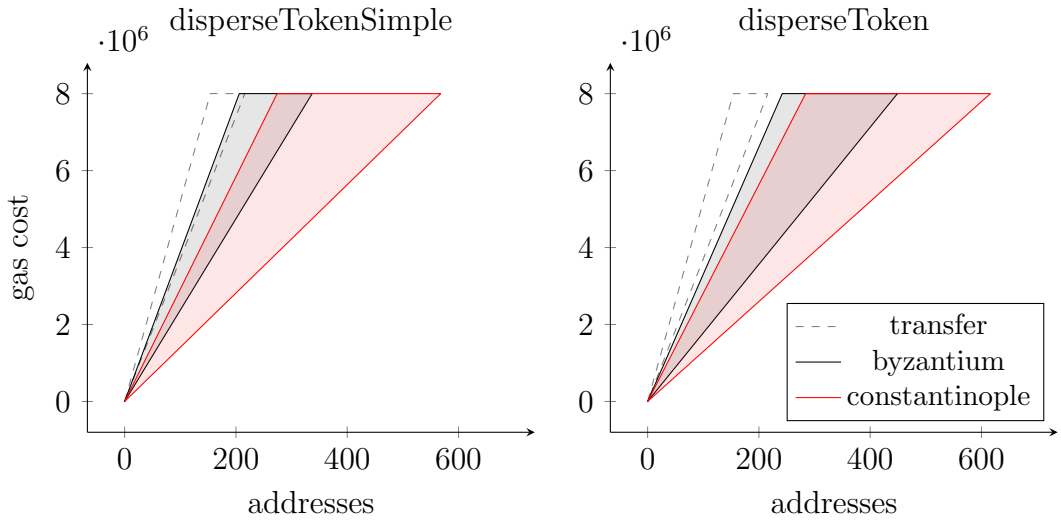

Figure 2: EIP-1283 effect on algorithms 1 and 2

4.2 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 optimize for gas usage using this pattern. 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[8]. This saves 5000

gas by skipping one SSTORE and makes the cost of **transfer** and **transferFrom** equal.

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

5 Native Token

Any value sent along with the transaction immediately becomes available as contract's 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 ∈ [0 ... recipients.length) do
  | recipients[i].transfer(values[i])
end
balance ← address(this).balance
if balance > 0 then
  | msg.sender.transfer(balance)
end
```

Contract transfers use CALL[9] opcode which costs 7,400–32,400 gas, whereas a regular transaction costs a flat fee of 21,000 gas.

$$G^{700}_{call} + G^{9000}_{callvalue} - G^{2300}_{callstipend} = 7400 \tag{8}$$

$$G^{700}_{call} + G^{9000}_{callvalue} - G^{2300}_{callstipend} + G^{25000}_{newaccount} = 32400 \tag{9}$$

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 transfers. This is 40% worse result in terms of gas usage, but it is still one transaction instead of n.

Transferring to existing accounts from a contract allows to fit 830 transfers in a block, which is a massive 2.18x improvement over regular transfers. Note that existing account may have zero balance. The resulting cost per transfer is 9,629–34,701 gas.

Constantinople hard fork has no effect on sending native tokens.

Table 4: Ethereum gas schedule

Name	Value	Description
G_{call}	700	
$G_{callvalue}$	9000	if $value > 0$
$G_{callstipend}$	2300	subtracted from $G_{callvalue}$
$G_{newaccount}$	25000	if creates new account

Table 5: Native token transfers with gas limit → 8,000,000

Method	Transfers	Gas/transfer	Transactions
call	380	21000	380
disperseEther	230–830	9,629–34,701	1

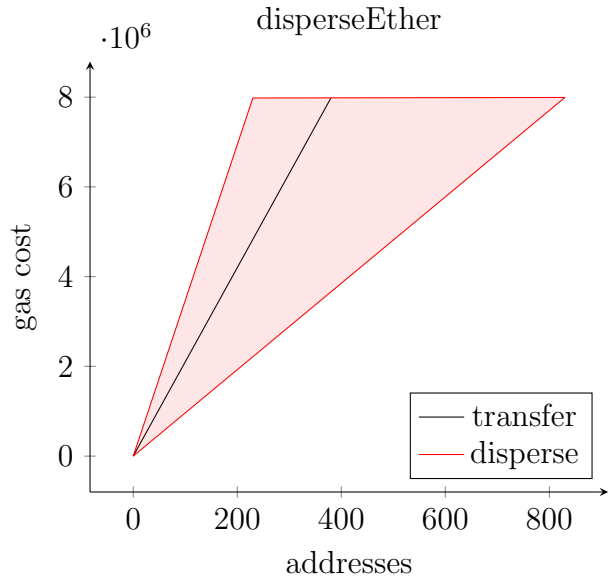

Figure 3: Gas profile of native token transfers

6 Methodology

Gas usage was instrumented using Py-EVM[10]. 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 accessed by keccak of their key and index, both of which are encoded as bytes32. For example, **balances** mapping is accesses like this[11]:

$$\text{keccak256}(\text{bytes32}(\text{address}) + \text{bytes32}(0))$$

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

$$\begin{gathered}\text{index} = \text{keccak256}(\text{bytes32}(\text{address}) + \text{bytes32}(1)) \\ \text{keccak256}(\text{bytes32}(\text{spender}) + \text{index})\end{gathered}$$

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. There are also additional tests against other similar contracts.

7 Conclusion

We presented simple and optimal algorithms for batch sending both ERC-20 and native tokens and benchmarked them.

Batching results in 1.58–2.07x (1.84–2.87x after Constantinople) improvement in gas usage for tokens and 0.61–2.18x for ether transfers, see Tables 2, 3, 5.

A smart contract implementing Algorithms 1, 2, 3 is deployed on Ethereum mainnet and all its testnets as well as multiple other EVM-compatible networks at address:

0xD152f549545093347A162Dce210e7293f1452150

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

References

[1] Ethereum Project, 2015

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

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

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

[5] DigixGlobal. *Multiplexer*, 2017

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

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

[8] 0x Project. *Unlimited Allowance Token v1*, 2017

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

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

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