Gas - Advanced

Costs of complex actions

Let's cover some more complex gas calculations: deploying contracts and function calls.

Deploying Contracts

The basic action cost includes two different values for deploying contracts. Simplified, these are:

deploy_contract_cost: 184765750000, deploy_contract_cost_per_byte: 64572944,

These values can be queried by using theprotocol_config RPC endpoint.

The first is a baseline cost, no matter the contract size. Keeping in mind that each need to be multiplied by two, for bothsend andexecute costs, and will also require sending & executing a receipt, the gas units comes to:

2 * 184765750000 + 2 * contract_size_in_bytes * 64572944 + 2 * 108059500000

(Divide the resulting number by 10^{12} to get to TGas!)

Note that this covers the cost of uploading and writing bytes to storage, but doesnot cover the cost of holding these bytes in storage. Long-term storage is compensated via storage staking, a recoverable cost-per-byte amount that will also be deducted from your account during contract deployment.

Deploying a 16kb contract requires2.65 TGas (and thus 0.265mN at minimum gas price) for the transaction fee, while 1.5N will be locked up for storage staking.

Function calls

Given the general-purpose nature of NEAR, function calls win the award for most complex gas calculations. A given function call will use a hard-to-predict amount of CPU, network, and IO, and the amount of each can even change based on the amount of data already stored in the contract!

With this level of complexity, it's no longer useful to walk through an example, enumerating each (seeext_costs underwasm_config using theprotocol_config RPC endpoint) of the gas calculations as we go (you can research this yourself,if you want). Instead, let's approach this from two other angles: ballpark comparisons to Ethereum, and getting accurate estimates with automated tests.

How much of the gas fee goes as a 30% reward to the smart contract account?

The NEAR Whitepaper mentions that30% of all gas fees go to smart contract accounts on which the fees are expensed.

This amount can be calculated for function calls in two ways:

1. Summing all values in the gas profile
2. Taking the total gas burnt for the transaction and subtract the static execution gas (which is equal to the amount of gas spent on sending the receipt(s)) from it. Both these numbers are available on theNEAR Explorer
3. overview page for a transaction.

The second approach is shorter, and quite possibly easier to remember. So here's an example:

- An account calls the methodsubmit
- onaurora
-
    - Converting the transaction to receipt burned a total of ~0.00024Ⓝ
-
    - Executing the receipt burned a total of ~0.00376Ⓝ

The 30% reward for the smart contract owner (in this case aurora) would be: (0.00376Ⓝ - 0.00024Ⓝ) * 0.3 = 0.001056Ⓝ

This transaction can also be foundhere on NEAR Explorer, feel free to have a look around!

For calls involving multiple contracts, calculating the reward for each contract with this method would not be possible with the data shown on NEAR Explorer (June 2022) as the explorer does not show the conversion cost for the second (and other) receipt(s).

Ballpark Comparisons to Ethereum

Like NEAR, Ethereum uses gas units to model computational complexity of an operation. Unlike NEAR, rather than using a predictable gas price, Ethereum uses a dynamic, auction-based marketplace. This makes a comparison to Ethereum's gas prices a little tricky, but we'll do our best.

Etherscan gives ahistoric Ethereum gas price chart . These prices are given in "Gwei", or Gigawei, where a wei is the smallest possible amount of ETH, 10^{-18}. From November 2017 through July 2020, average gas price was 21Gwei. Let's call this the "average" gas price. In July 2020, average gas price went up to 57Gwei. Let's use this as a "high" Ethereum gas fee.

Multiplying Ethereum's gas units by gas price usually results in an amount that's easy to show in milliETH (mE), the same way we've been converting NEAR's TGas to milliNEAR. Let's look at some common operations side-by-side, comparing ETH's gas units to NEAR's, as well as converting to both the above "average" & "high" gas prices.

Operation ETH gas units avg mE high mE NEAR TGas mN Transfer native token (ETH or NEAR) 21k 0.441 1.197 0.45 0.045 Deploy & initialize afungible token contract 1.1M 23.3 63.1 9 † 0.9 (plus 1.5Ⓝ instorage staking) Transfer a fungible token ~45k 0.945 2.565 14 1.4 Setting an escrow for a fungible token44k 0.926 2.51 8 0.8 Checking a balance for a fungible token 0 0 0 0 0 † Function calls require spinning up a VM and loading all compiled Wasm bytes into memory, hence the increased cost over base operations; this isbeing optimized .

While some of these operations on their surface appear to only be about a 10x improvement over Ethereum, something else to note is that the total supply of NEAR is more than 1 billion, while total supply of Ethereum is more like 100 million. So as fraction of total supply, NEAR's gas fees are approximately another 10x lower than Ethereum's. Additionally, if the price of NEAR goes up significantly, then the minimum gas fee set by the network can be lowered.

You can expect the network to sit at the minimum gas price most of the time; learn more in theEconomics whitepaper .

Estimating Gas Costs with Automated Tests

Gas unit expense for running smart contract functions can be accurately estimated by running these intestnet . Generally,testnet runs a higher version of the protocol thanmainnet . However, gas expense calculations do not change often making this is a good way to get a sense of how much gas a function will cost onmainnet .

To estimate gas costs, you can use thenear-workspaces crate in Rust or similarly namedpackage in JavaScript .

You may extract thetotal_gas_burnt field from theCallExecutionDetails struct returned by thecall method. (Read more)

println! ("Burnt gas (all): {}" , res . total_gas_burnt) ; In JS, you can calculate this value by addingresult.receipts_outcome[0].outcome.gas_burnt with the amount of gas units consumed for receipt execution inresult.transaction_outcome.outcome.gas_burnt .

Gas Cost Estimation REST API You may obtain gas cost estimates for a given function call usingapi.gasbuddy.tech . This API is experimental and may be removed in the future. One can obtain a gas cost estimate for a given function call by sending a POST request tohttps://api.gasbuddy.tech/profile with the following JSON body:

{ "contract_id": "", "method": "", "args": { "arg1": "value1", "arg2": "value2" } }

Gas Cost Estimation in the SDK

OurSDK environment exposes theused gas method, which lets you know how much gas was used so far.

You can benchmark how much gas a method (or a portion) uses by simply computing the difference in gas used between two points:

function

myMethod ( ) { // take gas usage const used_gas_point_A = environment . used_gas ( )

// --- some code goes here ---

const used_gas_point_B = environment . used_gas ( )

log ("Used gas" , used_gas_point_B - used_gas_point_A) }

Pessimistic gas price inflation

A transactions may take several blocks before it completes. Due to dynamic gas price adjustments, later blocks may have a higher gas price than when the transaction was signed. To guarantee that the transaction can still finish, the amount of tokens reserved when starting a transaction is increased by thepessimistic-inflation rule .

The pessimistic inflation rule means that the gas has to be purchased at the highest theoretical gas price that the transaction could reach. The extra spending is only temporary, the difference between the pessimistic and actual price is refunded when

the transaction finishes. This is the reason why in the explorer, virtually every transaction that spans more than one block contains a refund, even if all the gas has been spent.

By how much is the price inflated? It depends on how many blocks a transaction may take. A simple transaction that only sends tokens from one account to another can take between 2-3 blocks.

- One block to subtract the money from the signer's account
- One block to add it to the receivers account
- Potentially another block if the receiver is on another shard and the receipt application gets delayed.

Therefore, the pessimistically inflated price is increased by 3% or calculated asgas_price × 1.03. Every additional cross-shard communication adds another factor of 1.03.

For a function call, the maximum block delay is computed as the total gas attached divided by the minimum amount required to call another function. Therefore, the more gas you attach to a transaction, the higher your gas price. But again, the increased price is temporarily and will be refunded unless the network actually becomes that congested. Prices would have to go up by the maximum every block and your receipts would need to be very unlucky to have extra delays every time.

What's the price of gas right now?

You can directly query the NEAR platform for the price of gas on a specific block using the RPC methodgas_price . This price may change depending on network load. The price is denominated in yoctoNEAR (10^-24 NEAR)

1. Take any recent block hash from the blockchain usingNearBlocks Explorer
2. At time of writing,SqNPYxdgspCT3dXK93uVvYZh18yPmekirUaXpoXshHv
3. was the latest block hash
4. Issue an RPC request for the price of gas on this block using the methodgas_price
5. documented here
6. http post https://rpc.testnet.near.org jsonrpc=2.0 method=gas_price params:='["SqNPYxdgspCT3dXK93uVvYZh18yPmekirUaXpoXshHv"]' id=dontcare
7. Observe the results
8. {
9. "id": "dontcare",
10. "jsonrpc": "2.0",
11. "result": {
12. "gas_price": "5000"
13. }
14. }

The price of 1 unit of gas at this block was 5000 yoctoNEAR (10^-24 NEAR).

Some closing thoughts from the whitepaper

Fundamentally, the NEAR platform is a marketplace between willing participants. On the supply side, operators of the validator nodes and other fundamental infrastructure need to be incentivized to provide these services which make up the “community cloud.” On the demand side, the developers and end-users of the platform who are paying for its use need to be able to do so in a way which is simple, clear and consistent so it helps them.

A blockchain-based cloud provides several specific resources to the applications which run atop it:

- Compute (CPU)
- : This is the actual computer processing (and immediately available RAM) which run the code in a contract.
- Bandwidth ("Network")
- : This is the network traffic between participants and users, including messages which submit transactions and those which propagate blocks.
- Storage
- : Permanent data storage on the chain, typically expressed as a function of both storage space and time.

Existing blockchains like Ethereum account for all of these in a single up front transaction fee which represents a separate accounting for each of them but ultimately charges developers or users for them only once in a single fee. This is a high volatility fee commonly denominated in “gas”.

Developers prefer predictable pricing so they can budget and provide prices for their end users. The pricing for the above-mentioned resources on NEAR is an amount which is slowly adjusted based on system usage (and subject to the smoothing effect of resharding for extreme usage) rather than being fully auction-based. This means that a developer can more predictably know that the cost of running transactions or maintaining their storage. To dig deeper into how and why gas works the way it does on NEAR, check out theEconomics section of the main whitepaper and theTransaction and Storage Fees section of the economics whitepaper.

Got a question? Ask it on StackOverflow! Edit this page Last updatedonFeb 15, 2024 bygagdiez Was this page helpful? Yes No

Previous Gas Next NEAR Data Flow