Transaction Fees

The small fees paid to processinstructions on the Solana blockchain are known as "transaction fees ".

As each transaction (which contains one or more instructions) is sent through the network, it gets processed by the current leader validation-client. Once confirmed as a global state transaction, thistransaction fee is paid to the network to help support theeconomic design of the Solana blockchain.

Info NOTE: Transaction fees are different fromaccount rent ! While transaction fees are paid to process instructions on the Solana network, rent is paid to store data on the blockchain. You can learn more about rent here:What is rent?

Why pay transaction fees?#

Transaction fees offer many benefits in the Solanaeconomic design described below. Mainly:

- they provide compensation to the validator network for the CPU/GPU resources
- necessary to process transactions,
- reduce network spam by introducing real cost to transactions,
- and provide long-term economic stability to the network through a
- protocol-captured minimum fee amount per transaction

Info NOTE: Network consensus votes are sent as normal system transfers, which means that validators pay transaction fees to participate in consensus.

Basic economic design#

Many blockchain networks (e.g. Bitcoin and Ethereum), rely on inflationaryprotocol-based rewards to secure the network in the short-term. Over the long-term, these networks will increasingly rely ontransaction fees to sustain security.

The same is true on Solana. Specifically:

- A fixed proportion (initially 50%) of each transaction fee isburned
- (destroyed), with the remaining going to the currentleader
- processing the transaction.
- A scheduled global inflation rate provides a source forrewards
- distributed toSolana Validators
- .

Why burn some fees?#

As mentioned above, a fixed proportion of each transaction fee isburned (destroyed). This is intended to cement the economic value of SOL and thus sustain the network's security. Unlike a scheme where transactions fees are completely burned, leaders are still incentivized to include as many transactions as possible in their slots.

Burnt fees can also help prevent malicious validators from censoring transactions by being considered infork selection.

Example of an attack:#

In the case of aProof of History (PoH) fork with a malicious, censoring leader:

- due to the fees lost from censoring, we would expect the total fees burned to
- beless than
- a comparable honest fork
- if the censoring leader is to compensate for these lost protocol fees, they
- would have to replace the burnt fees on their fork themselves
- thus potentially reducing the incentive to censor in the first place

Calculating transaction fees#

Transactions fees are calculated based on two main parts:

- a statically set base fee per signature, and
- the computational resources used during the transaction, measured in
- "compute units
- "

Since each transaction may require a different amount of computational resources, they are allotted a maximum number

ofcompute units per transaction known as the "compute budget ".

The execution of each instruction within a transaction consumes a different number ofcompute units . After the maximum number ofcompute units has been consumed (aka compute budget exhaustion), the runtime will halt the transaction and return an error. This results in a failed transaction.

Info Learn more: compute units and theCompute Budget in the Runtime andrequesting a fee estimate from the RPC.

Prioritization fee#

A Solana transaction can include anoptional fee to prioritize itself against others known as a "prioritization fee ". Paying this additional fee helps boost how a transaction is prioritized against others, resulting in faster execution times.

How the prioritization fee is calculated#

A transaction'sprioritization fee is calculated by multiplying the maximum number ofcompute units by thecompute unit price (measured inmicro-lamports).

Each transaction can set the maximum number of compute units it is allowed to consume and the compute unit price by including aSetComputeUnitLimit andSetComputeUnitPrice compute budget instruction respectively.

Info Note: Unlike other instructions inside a Solana transaction,Compute Budget instructions doNOT require any accounts. If noSetComputeUnitLimit instruction is provided, the limit will be calculated as the product of the number of instructions in the transaction and the default per-instruction units, which is currently200k .

If noSetComputeUnitPrice instruction is provided, the transaction will default to no additional elevated fee and the lowest priority.

How to set the prioritization fee#

A transaction's prioritization fee is set by including aSetComputeUnitPrice instruction, and optionally aSetComputeUnitLimit instruction. The runtime will use these values to calculate the prioritization fee, which will be used to prioritize the given transaction within the block.

You can craft each of these instructions via theirrust or@solana/web3.js functions. Each of these instructions can then be included in the transaction and sent to the cluster like normal. See also thebest practices below.

Info Caution: Transactions can only containone of each type of compute budget instruction. Duplicate types will result in anTransactionError::DuplicateInstruction error, and ultimately transaction failure.

Rust#

The rustsolana-sdk crate includes functions withinComputeBudgetInstruction to craft instructions for setting thecompute unit limit andcompute unit price :

let instruction = ComputeBudgetInstruction :: set_compute_unit_limit (300_000); let instruction = ComputeBudgetInstruction :: set_compute_unit_price (1);

Javascript#

The@solana/web3.js library includes functions within theComputeBudgetProgram class to craft instructions for setting thecompute unit limit andcompute unit price :

const instruction = ComputeBudgetProgram. setComputeUnitLimit ({ units: 300_000 , }); const instruction = ComputeBudgetProgram. setComputeUnitPrice ({ microLamports: 1 , });

Prioritization fee best practices#

Request the minimum compute units#

Transactions should request the minimum amount of compute units required for execution to minimize fees. Also note that fees are not adjusted when the number of requested compute units exceeds the number of compute units actually consumed by an executed transaction.

Get recent prioritization fees#

Prior to sending a transaction to the cluster, you can use thegetRecentPrioritizationFees RPC method to get a list of the recent paid prioritization fees within the recent blocks processed by the node.

You could then use this data to estimate an appropriate prioritization fee for your transaction to both (a) better ensure it gets processed by the cluster and (b) minimize the fees paid.

Fee Collection#

Transactions are required to have at least one account which has signed the transaction and is writable. Writable signer accounts are serialized first in the list of transaction accounts and the first of these accounts is always used as the "fee payer".

Before any transaction instructions are processed, the fee payer account balance will be deducted to pay for transaction fees. If the fee payer balance is not sufficient to cover transaction fees, the transaction will be dropped by the cluster. If the balance was sufficient, the fees will be deducted whether the transaction is processed successfully or not. In fact, if any of the transaction instructions return an error or violate runtime restrictions, all account changesexcept the transaction fee deduction will be rolled back.

Fee Distribution#

Transaction fees are partially burned and the remaining fees are collected by the validator that produced the block that the corresponding transactions were included in. The transaction fee burn rate was initialized as 50% when inflation rewards were enabled at the beginning of 2021 and has not changed so far. These fees incentivize a validator to process as many transactions as possible during its slots in the leader schedule. Collected fees are deposited in the validator's account (listed in the leader schedule for the current slot) after processing all of the transactions included in a block.