Priority Fees: Understanding Solana's Transaction Fee Mechanics

What's this Article About?

Solana is fast

. Yet, even on the fastest blockchain available, users want optimized transaction processing for important transactions. Priority fees are a way to ensure a user's transaction is placed at the front of the execution ordering queue. These are additional, optional fees that a user can add to their transaction.

This article briefly explores the nuances of transaction processing on Solana. It covers transactions, their lifecycle, and how transaction fees work. Then, the article explores priority fees, how to implement them programmatically, and best practices.

Transactions and Their Lifecycle

Transactions are used to invoke Solana programs and enact state changes. They are bundles of instructions (i.e., directives for a single program invocation) that tell the validator what actions to perform, on what accounts, and whether they have the necessary permissions.

The general lifecycle of a transaction on Solana is as follows:

- The user has a clear goal for an action they want to perform. For example, Alice wants to send Bob 10 SOL
- The user generates a transaction for their desired action. For example, Alice creates a transaction with an instruction to transfer 10 SOL from her account to Bob's. Alice also includes a recent blockhash and signs the transaction using her private key
- The user sends the transaction to the network. Then, they receive information on whether the transaction has been successfully added. For example, Alice sends the transaction with a confirmed commitment status. When the transaction is confirmed, she receives a transaction signature. Alice can use this transaction signature on a block explorer, such as X-ray, to see that she sent Bob 10 SOL successfully. Her account has been debited 10 SOL, whereas Bob's was credited 10 SOL

When users send a signed transaction to the network, they use an RPC provider such as Helius. Helius' RPCs receive the transaction and check the current leader schedule. On Solana, only specific validators are responsible for appending entries to the ledger at certain times. The leader is responsible for producing a block for its currentslot and is assigned four consecutive slots. The signed transaction is sent to the current leader and the next two leaders.

The current leader validates the signed transaction and performs other preprocessing steps before scheduling the transaction for execution. Unlike other L1s like Ethereum, Solana has no global transaction queue. Most validators use the scheduler implementation provided by Solana Labs. However, validators running the Jito validator client use a pseudo-mempool (i.e., MempoolStream) to order transactions. The default scheduler is multi-threaded, with each thread maintaining a queue of transactions waiting to be executed. Transactions are ordered into blocks by combining first-in-first-out (FIFO) and priority fees. It's important to note that this ordering is inherently non-deterministic as transactions are assigned to execution threads somewhat randomly.

When a transaction is executed, it is propagated viaTurbine, and its fees are paid accordingly.

How Transaction Fees Work on Solana

Transaction fees are small fees paid to process transactions on Solana. The currentleader processes transactions sent through the network to produce ledger entries. When the transaction is confirmed as a part of state, a fee is paid to support the economic design of Solana. Transaction fees fundamentally benefit Solana. They:

- Provide compensation to validators
- Reduce network space by introducing real-time cost for transactions
- Provide long-term economic stability to the network via protocol-captured minimum fee

Solana relies on inflationary protocol-based rewards to secure the network in the short term. The network has a scheduled global inflation rate to reward validators to achieve this. For the long term, Solana relies on transaction fees to sustain security. A fixed portion (initially set at 50%) of each transaction fee is burned, with the rest sent to the current leader. Solana burns fees to fortify the value of SOL while discouraging malicious validators from censoring transactions.

Transaction fees are calculated based on a statically set base fee per signature, and the computational resources used

during the transaction measured in Compute Units (CU). This base fee can range from 50% to 1000% of the target lamports per signature. The default target per signature is currently set at 10,000. Each transaction is allocated a maximum CU budget known as the Compute Budget. Exceeding this budget leads to the runtime halting the transaction and returning an error. The maximum budget per transaction is 1.4 million CU, and the blockspace limit is 48 million CU.

What are Priority Fees?

Due to these limitations, computationally heavy transactions could fill blockspace, delaying other transactions. Solana introduced an optional

fee to allow transactions to prioritize themselves against other transactions in the leader's queue known as a priority fee. Paying this fee effectively boosts your transaction, resulting in faster execution times. This is useful for time-sensitive or high-value transactions. The fee priority of a transaction is determined by the number of compute units it requests. The more compute units a transaction requests, the higher the fee it'll have to pay to maintain its priority in the transaction queue. Charging more for more compute units prevents computationally heavy transaction spam.

Priority fees are the product of a transaction's compute budget and its compute unit price measured in micro-lamports: priorityFees = computeBudget * computeUnitPrice

The computeBudget

is the product of the number of instructions in a transaction and its compute unit limit: computeBudget = numOfInstructions * computeUnitLimit

How to Implement Priority Fees Programmatically

A transaction's prioritization fee is set by setting a SetComputeUnitPrice

instruction and an optional SetComputeUnitLimit

instruction. If a SetComputeUnitPrice

instruction isn't provided, the transaction will default to the lowest priority since no additional fee is provided. If a SetComputeUnitLimit

instruction isn't provided, the limit is calculated as the product of the number of instructions in the transaction and the default compute unit limit. The runtime uses the compute unit price and compute unit limit to calculate the prioritization fee, which is used to prioritize the given transaction.

We must add these instructions to our desired transaction to programmatically add priority fees. In Javascript, it looks like the following:

In this snippet, we:

- Set up a test environment using Localhost
- Create two new wallets (i.e., fromKeypair

and toPubkey

)

- Airdrop both wallets 100 SOL
- Ensure that the airdrops were successful
- Create the priority fee instructions (i.e., computePriceIx

and computeLimitIx

)

- Create a transfer instruction, sending 100, 000 lamports from fromKeypair

to toPubkey

- Create a new transaction and add all of the instructions to it
- Attach the latest blockhash to the transaction and sign it
- Send and confirm the transaction was sent successfully

That’s it

- priority fees are just instructions added to a transaction to pay for faster execution times. Most of this snippet configures our development environment and two wallets to transfer SOL between them. What we’re interested in is creating the instructions to add priority fees and attaching them to a transaction:

Best Practices

The order of these instructions matter as they are executed sequentially. If, for example, you have a transaction that exceeds the default computation limit (i.e., 200k CUs) before

extending the computation limit, the transaction will fail. Say we have the following transaction:

- Instruction 1 uses 100k
- Instruction 2 uses 150k
- Instruction 3 extends the compute limit

This transaction will fail unless instructions 2 and 3 are swapped. It is therefore recommended to add the compute limit instruction before adding other instructions to your transaction. Remember, you do not need to use the

SetComputeLimit

instruction if you want to add priority fees to your transaction

- it is entirely optional. The placement of the SetComputePrice

instruction does not matter.

Using the getRecentPrioritizationFees RPC method is recommended to get a list of recently paid priority fees. This data can be used to estimate an appropriate priority fee for transactions to ensure they are processed by the cluster and minimize the fees paid. Alternatively, Helius offers a new Priority Fee API, which we'll cover in the following section.

Transactions should also request the minimum amount of compute units required for execution to minimize these fees. Note that costs are not adjusted when the number of requested compute units exceeds the total units used by a transaction.

Some wallet providers, such as Phantom, recognize that dApps can set priority transaction fees. However, they discourage doing so, citing that it often creates unnecessary complexity for end-users. Instead, they urge dApp developers to let Phantom apply priority fees on the user’s behalf. Solfare, for example, tackles the issue by automatically detecting whether Solana is under load and slightly increases fees to prioritize your transaction over others.

Helius Priority Fee API

Calculating priority fees using the getRecentPrioritizationFees

RPC method makes intuitive sense to calculate fees on a slot basis. However, this can be tough given the ever-changing network conditions and the nature of getRecentPrioritizationFees'

response (i.e., returning a list of values for the past 150 blocks, which is only helpful for gauging the minimum value to set for fees).

The Helius Priority Fee API introduces a new method, getPriorityFeeEstimate

, that simplifies the response into a single value, considering global and local fee markets. This method uses a predefined set of percentiles to determine the estimate. These percentiles, or levels, range from NONE

(0th percentile) to UNSAFE_MAX

(100th percentile, and labeled unsafe to prevent users from accidentally draining their funds). Users also have the option to specify to receive all priority levels and adjust the range used in this calculation via lookbackSlots

. Thus, similar to getRecentPrioritizationFees,

users can look back at a certain amount of slots in the estimate calculation.

For example, we can calculate all

priority fee levels for Jup v6 with the following script:

This endpoint is in active development. For more information, visit theHelius documentation.

Conclusion

Navigating the world of Solana transactions reveals a sophisticated system that balances network efficiency with economic incentives. Understanding how transactions work, in addition to their fees and priority fees, enables developers and users to make more informed decisions to optimize their interactions on Solana. The ability to implement priority fees programmatically opens up new avenues for high-value and time-sensitive transactions. This allows for greater flexibility and efficiency in operations on Solana.

If you've read this far, thank you, anon! Be sure to enter your email address below so you'll never miss an update about what's new on Solana. Ready to dive deeper? Join our Discord to start building the future on the most performant blockchain, today.

Additional Resources / Further Reading

- Lifecycle of a Solana Transaction
- Phantom on Priority Fees
- Solana Documentation on Transaction Fees
- [The Code for ComputeBudget

](https://github.com/solana-labs/solana/blob/4293f11cf13fc1e83f1baa2ca3bb2f8ea8f9a000/program-runtime/src/compute_budget.rs)