Meta Transactions

NEP-366 introduced the concept of meta transactions to Near Protocol. This feature allows users to execute transactions on NEAR without owning any gas or tokens. In order to enable this, users construct and sign transactions off-chain. A third party (the relayer) is used to cover the fees of submitting and executing the transaction.

Overview

Credits for the diagram go to the NEP authors Alexander Fadeev and Egor Uleyskiy.

The graphic shows an example use case for meta transactions. Alice owns an amount of the fungible tokenFT . She wants to transfer some to John. To do that, she needs to callft_transfer("john", 10) on an account namedFT .

The problem is, Alice has no NEAR tokens. She only has a NEAR account that someone else funded for her and she owns the private keys. She could create a signed transaction that would make theft_transfer("john", 10) call. But validator nodes will not accept it, because she does not have the necessary Near token balance to purchase the gas.

With meta transactions, Alice can create aDelegateAction , which is very similar to a transaction. It also contains a list of actions to execute and a single receiver for those actions. She signs theDelegateAction and forwards it (off-chain) to a relayer. The relayer wraps it in a transaction, of which the relayer is the signer and therefore pays the gas costs. If the inner actions have an attached token balance, this is also paid for by the relayer.

On chain, theSignedDelegateAction inside the transaction is converted to an action receipt with the sameSignedDelegateAction on the relayer's shard. The receipt is forwarded to the account fromAlice , which will unpacked theSignedDelegateAction and verify that it is signed by Alice with a valid Nonce, etc. If all checks are successful, a new action receipt with the inner actions as body is sent toFT . There, theft_transfer call finally executes.

Relayer

Meta transactions only work with arelayer . This is an application layer concept, implemented off-chain. Think of it as a server that accepts aSignedDelegateAction , does some checks on them and eventually forwards it inside a transaction to the blockchain network.

A relayer may choose to offer their service for free but that's not going to be financially viable long-term. But they could easily have the user pay using other means, outside of Near blockchain. And with some tricks, it can even be paid using fungible tokens on Near.

In the example visualized above, the payment is done using FT. Together with the transfer to John, Alice also adds an action to pay 0.1 FT to the relayer. The relayer checks the content of theSignedDelegateAction and only processes it if this payment is included as the first action. In this way, the relayer will be paid in the same transaction as John.

Keep in mind The payment to the relayer is still not guaranteed. It could be that Alice does not have sufficientFT and the transfer fails. To mitigate, the relayer should check theFT balance of Alice first. Unfortunately, this still does not guarantee that the balance will be high enough once the meta transaction executes. The relayer could waste NEAR gas without compensation if Alice somehow reduces her FT balance in just the right moment. Some level of trust between the relayer and its user is therefore required.

Limitations

Single receiver

A meta transaction, like a normal transaction, can only have one receiver. It's possible to chain additional receipts afterwards. But crucially, there is no atomicity guarantee and no roll-back mechanism.

Accounts must be initialized

Any transaction, including meta transactions, must use NONCEs to avoid replay attacks. The NONCE must be chosen by Alice and compared to a NONCE stored on chain. This NONCE is stored on the access key information that gets initialized when creating an account.

Constraints on the actions inside a meta transaction

A transaction is only allowed to contain one single delegate action. Nested delegate actions are disallowed and so are delegate actions next to each other in the same receipt.

Gas costs for meta transactions

Meta transactions challenge the traditional ways of charging gas for actions. Let's assume Alice uses a relayer to execute actions with Bob as the receiver.

1. The relayer purchases the gas for all inner actions, plus the gas for the
2. delegate action wrapping them.
3. The cost of sending the inner actions and the delegate action from the
4. relayer to Alice's shard will be burned immediately. The conditionrelayer == Alice
5. determines which actionSEND
6. cost is taken (sir
7. ornot_sir
8.).
9. Let's call thisSEND(1)
10. .
11. On Alice's shard, the delegate action is executed, thus theEXEC
12. gas cost
13. for it is burned. Alice sends the inner actions to Bob's shard. Therefore, we
14. burn theSEND
15. fee again. This time based onAlice == Bob
16. to figure outsir
17. ornot_sir
18. . Let's call thisSEND(2)
19. .
20. On Bob's shard, we execute all inner actions and burn theirEXEC
21. cost.

Each of these steps should make sense and not be too surprising. But the consequence is that the implicit costs paid at the relayer's shard areSEND(1) +SEND(2) +EXEC for all inner actions plusSEND(1) +EXEC for the delegate action. This might be surprising but hopefully with this explanation it makes sense now!

Gas refunds in meta transactions

Gas refund receipts work exactly like for normal transaction. At every step, the difference between the pessimistic gas price and the actual gas price at that height is computed and refunded. At the end of the last step, additionally all remaining gas is also refunded at the original purchasing price. The gas refunds go to the signer of the original transaction, in this case the relayer. This is only fair, since the relayer also paid for it.

Balance refunds in meta transactions

Unlike gas refunds, the protocol sends balance refunds to the predecessor (a.k.a. sender) of the receipt. This makes sense, as we deposit the attached balance to the receiver, who has to explicitly reattach a new balance to new receipts they might spawn.

In the world of meta transactions, this assumption is also challenged. If an inner action requires an attached balance (for example a transfer action) then this balance is taken from the relayer.

The relayer can see what the cost will be before submitting the meta transaction and agrees to pay for it, so nothing wrong so far. But what if the transaction fails execution on Bob's shard? At this point, the predecessor isAlice and therefore she receives the token balance refunded, not the relayer. This is something relayer implementations must be aware of since there is a financial incentive for Alice to submit meta transactions that have high balances attached but will fail on Bob's shard.

Function access keys in meta transactions

Function access keys can limit the allowance, the receiving contract, and the contract methods. The allowance limitation acts slightly strange with meta transactions.

But first, both the methods and the receiver will be checked as expected. That is, when the delegate action is unwrapped on Alice's shard, the access key is loaded from the DB and compared to the function call. If the receiver or method is not allowed, the function call action fails.

For allowance, however, there is no check. All costs have been covered by the relayer. Hence, even if the allowance of the key is insufficient to make the call directly, indirectly through meta transaction it will still work.

This behavior is in the spirit of allowance limiting how much financial resources the user can use from a given account. But if someone were to limit a function access key to one trivial action by setting a very small allowance, that is circumventable by going through a relayer. An interesting twist that comes with the addition of meta transactions.