Suppose that a user wants to make an ERC20 token transfer, but pay transaction fees in that ERC20, instead of ether. We can do that without modifying the base layer protocol by extending the ERC20 protocol as follows. We add a function submitSignedPayments({to: address, value: uint256, fee: uint256, deadline: uint256, seq: uint256, sig: bytes[65]}[])

, which works as follows. For each submitted payment:

- Let sender = ecrecover(hash(to, value, fee, seq, deadline, self), sig)
- Verify that balances[sender] >= value + fee
- Verify that seqs[sender] == seq
- Verify that block.number <= deadline
- Set balances[sender] -= value + fee
- Set balances[to] += value
- Set balances[msg.sender] += fee
- Set seqs[sender] += 1

Users wishing to send ERC20 tokens could publish offchain messages offering a fee, and then simply wait until someone includes their message in a transaction.

Note that the 21000 gas overhead of a publishing transaction can be shared between multiple ERC20 contracts, by creating a central routing contract that accepts a list of signed payment objects and then calls submitSignedPayments

of each ERC20 contract with its associated signed payment as needed.

We can abstract this much further if we combine it together with an account abstraction scheme. Consider an abstraction scheme where user accounts are contracts, and transactions are calls to accounts, with the msg.sender

set to the block.coinbase

. We assume also that with account abstraction, users can create transactions that queue up multiple operations. However, user accounts can’t just be called by transactions, you can also have another contract call the user account as part of a transaction.

A user can send a transaction with two operations: (i) send the msg.sender

some tokens of some ERC20, (ii) perform some other task. A series of transactions of this type can then be put into a wrapper transaction which pays ETH fees, which can then get published on chain. The publisher of the wrapper transaction would collect the tokens.