tl;dr

- this doc covers an approach to gas sponsorship (apps paying gas for users).
- the approach resembles block building on Eth by pushing simulation load to gateway actors (kettles) before propagating to network (suave chain/eth)
- pro:

saves money for users + better UX

- pro:

saves network resources

- con:

makes being a kettle harder

- con:

brewers have to do more work to tell if they should run an app

- tradeoff:

can avoid some of the downsides by introducing a small cost for users for local computation

- Requirement: kettles must be building full blocks

- pro:

saves money for users + better UX

- pro:

saves network resources

- con:

makes being a kettle harder

- con:

brewers have to do more work to tell if they should run an app

- tradeoff:

can avoid some of the downsides by introducing a small cost for users for local computation

- Requirement: kettles must be building full blocks

We should think about gas for a few reasons:

- network/validator/kettle revenue
- scarce resource allocation
- DoS protection
- gas ux leaves much to be desired throughout industry (e.g. user required to pay gas even when app governance/other party would gladly bear those costs and eventually refund)
- several use cases don't require users to bridge funds to suave, but paying gas would

Gas sponsorship addresses the last two points.

While someone

has to pay for computation, we are trying to make the user do as little as possible of that by offloading to the brewer or a 3rd party like SUAPP governance.

In Ethereum, clients usually do some work for every transaction to check that its valid. Any work done on messages that are not valid transactions is uncompensated work for the node operator, thus the system is designed to be able to very cheaply tell if a received message is a valid transaction. Once a message is considered valid, the user can be charged according to the rules of the network. After this, the app-specific contract logic kicks in for which users are made to pay.

[

image

855×385 78.4 KB

](https://collective.flashbots.net/uploads/default/original/2X/9/980996878447e43054eead296691a72d9a880c9d.png)

A naive

way of adapting this model is to change the model from having the infra operator pay for early execution and then users for the rest to a model in which infra operators pay first and then apps pay for business logic

.

[

image

787×330 72.2 KB

](https://collective.flashbots.net/uploads/default/original/2X/9/922866e2bda01cc915d4516c3865bf92f23aeea9.png)

Unfortunately, this leads apps open to being drained by reverting executions

. Presumably app devs are willing to pay gas costs because the user is doing something valuable in their execution. However, reverting requests are not supposed to have any effects (state transitions or outputs) and thus do not present value to the application. We could move away from this model but this would be very inconsistent with the EVM execution people are used to.

Thus, the brewer is open to an attack of spammed reverted execution.

[

image

787×330 75.8 KB

](https://collective.flashbots.net/uploads/default/original/2X/b/bb9979260b0cd58a050640afa496c68839f04afa.png)

We could address this by having users pay for reverted messages and apps pay for successful messages. The problem with this is that we pessimistically need to require users to have full funds for a reverting transaction, undoing any UX gain.

Another way of dealing with this attack is to give applications some fixed amount of "free" app logic, which the brewer must pay for along with the standard spam protection overhead. This "free" logic would be used to ascertain if the execution is valid in an application-specific way.

[

image

787×330 69.3 KB

](https://collective.flashbots.net/uploads/default/original/2X/6/6827bde63b7aeda87e007c683fd37ab8d96a8104.png)

Let's call this "free" logic, "filter gas"

. The viability of this approach depends on balancing the incentives of actors. We need to prevent:

- apps from getting DoS'd because they don't have enough filter gas.
- brewers from being DoS'd because its too expensive for them to check if a message is valid.

One may reasonably wonder if interesting apps could tell if executing a request is valuable without basically having to execute the whole thing anyway.

This is a substantial concern, but even if an execution has to be basically fully simulated, this may still be valuable. In fact, this is what we have with PBS on Ethereum today. The full "execution" of a valid transaction involves propagating it to thousands of Ethereum nodes that re-execute transactions. Block builders prevent senders of reverting transactions from having to pay for this expensive final step, by using exactly the described technique - free simulations. (paymasters in 4337 work the same way I think).

[

image

856×396 43.7 KB

](https://collective.flashbots.net/uploads/default/original/2X/2/2b087a5b53d6436425dc46cc9e2aa1f179e0e993.png)

The downside of this technique is that it introduces complexity to the role of infra operation. Kettle operators now have to have a DevOps person deal with spam vectors and do the math so that executing the valid messages still pays for all the spam that had to be processed.

This complexity narrows who can be a kettle operator and should make it much more challenging for kettle operators to start running apps (they need to do due diligence to assess whether it would be profitable).

We could address this by introducing a small fixed cost for users which covers local execution and then have apps pay the costs of network-wide execution. This would amount to having some small gas payment for execution in kettles and then having transaction execution (on suave chain or elsewhere) paid by applications.

[

image

856×396 47.2 KB

](https://collective.flashbots.net/uploads/default/original/2X/0/03260f74e677552d7c22e307c6c92dcaedea50f7.png)

The final boss is app logic which changes depending on where in the block a transaction executes.

There is still an attack in which a malicious user sends a request which eventually leads to an Ethereum transaction that succeeds at the top of the block, but fails if a piece of state has been altered. If the malicious user manages to alter the state with a second transaction that lands higher in the block, they could still drain the application with reverting transactions.

If SUAVE is producing full blocks then this isn't a concern. If it isn't, some applications can avoid this attack by restricting users' logic like in 4337 but this is quite limiting.

Analysis of this gas sponsorship approach

- pro:

saves money for users + better UX

- pro:

saves network resources

- con:

makes being a kettle harder

- con:

brewers have to do more work to tell if they should run an app

- tradeoff:

can avoid some of the downsides by introducing a small cost for users for local computation

Open Questions:

Naturally, there are many open questions, but the one that seems top of mind is how best to implement the sponsorship on chain. If a user makes a call to one function and then that function calls out to another contract, does the first contract account pay for everything or does each contract pay for its respective gas? If we do the latter, the offchain simulation checks would be wildly complicated.