Economic Protocol In Dusk Smart Contracts

Emanuele Francioni* and Matteo Ferretti†
Dusk Network

INTRODUCTION

The economic model of Dusk encompasses the mechanisms enabling smart contract owners to create economic value through the services they offer. It consists of levying service fees, offsetting gas costs for users, and optimizing gas payments for improved profitability. Essentially, the economic model allows service providers to be productized and generate revenue.

TERMINOLOGY

Throughout this document, we use the following terms:

- **contract**: smart contract method activated by the user through a transaction.
- **user**: party initiating the **contract** transaction.
- **gas**: unit measure of computational resource.
- **gas limit**: the maximum amount of resources allocated by a user or a contract to perform a computation.
- **gas consumed**: the cost incurred in executing a computation.
- **gas unspent**: the difference between **gas limit** and **gas consumed**. If a computation requires more resources than those allocated, then we incur an **insufficient gas** error.
- **gas price**: amount of DUSK that a sender is willing to spend per unit of gas (the amount is specified in Lux, where 1 Lux equals 10^{-9} DUSK). The higher the **gas price**, the more incentivized the block generators are to include the transaction in the next block. This way, transactions with high gas prices are generally confirmed more quickly. Because of this, **gas price** generally determines the transaction priority.
- **p**: percentage of **gas consumed** paid to the contracts. The remaining percentage is paid to the **block generator**. The optimal value of **p** is estimated to be between 20% and 40%, pending threat model analysis.
- **ICC** (inter-contract call): interaction between multiple contracts. ICCs cannot be called directly. As such, the contract methods should be tagged as **C2C** (contract-to-contract) or **direct**. The former can only be called by another contract, while the latter can be called exclusively by a user (through a transaction).
- **fee**: amount of DUSK charged by a smart contract to a user, when the former requires payment for its services. The **fee** is known by the transfer contract (changeable through a transaction), but needs to be communicated to the user, who approves/signs it. This way the possibility of fee malleability is removed, thus preventing a bait-and-switch attack where a contract could change the fee with a high priority transaction and drain the user's funds.
- **contract account**: in the scenario where the contract pays for gas on behalf of the user, this account is where fees are accrued and gas is paid.
- **provider**: intermediary service entity between a **user** and a **contract**.
- **transfer contract**: the smart contract responsible for handling DUSK and implementing any logic related to the economics of transactions.
- **block generator**: full-node that is eligible to propose a candidate block to the network.

SPECIFICATION

In this section, we describe how gas is handled in different scenarios. We summarize all cases in Table I.

Scenario 1: User pays gas

This case mirrors the conventional gas expenditure method in most blockchains. The **user** specifies a **gas limit** and a **gas price**.

Normal flow

In this case, (100-**p**)% **gas consumed** is transferred to the **block generator**, and **p** % is distributed to all touched contracts, weighted proportionally to the amount of **gas consumed** by each contract.

This is the default scenario, where the **user** pays for gas and every **contract** gets paid for their usage. The

* emanuele@dusk.network
† matteo@dusk.network

Scenario	Normal flow			Insufficient gas	
	gas unspent	gas consumed	fee	gas consumed	fee
1. User pays gas	100% reverts to the **user** (as change)	• (100-**p**)% awarded to **block generator** • **p** % paid to executed contracts distributed according to **gas consumed** by each **ICC**	no fee	100% to **block generator** (paid by **user**)	no fee
2. User pays gas, contract applies fee	100% reverts to the **user** (as change)	• (100-**p**)% awarded to **block generator** • **p** % paid to executed contracts distributed according to **gas consumed** by each **ICC**	paid to charging contract	100% to **block generator** (paid by **user**)	returned to **user**
3. Contract pays gas, contract applies fee	100% reverts to the **contract** (as change)	• (100-**p**)% awarded to **block generator** • **p** % paid to executed contracts distributed according to **gas consumed** by each **ICC**	paid to charging contract	100% locked for a number of epochs	returned to **user**
4. Percentage of Obfuscated Amount	100% reverts to the **user** (as change)	• (100-**p**)% awarded to **block generator** • **p** % paid to executed contracts distributed according to **gas consumed** by each **ICC**	paid to charging contract	100% to **block generator** (paid by **user**)	returned to **user**
5. Autocontracts	100% reverts to the **contract** (as change)	• (100-**p**)% awarded to **block generator** • **p** % paid to executed contracts distributed according to **gas consumed** by each **ICC**	no fee	100% locked for a number of epochs	returned to **user**

TABLE I. Summary of how gas and fees are handled in each scenario.

genesis contracts - the contracts handling DUSK- which are present from the first block - use this strategy.

Insufficient gas

If there is insufficient gas to execute, the **gas consumed** up is paid to the **block generator**.

Reward to contracts

In this scenario, contracts are rewarded directly by their **gas consumed**. Since the **user** is the one ultimately paying, it can be considered that the user is paying for a contract's execution. This is the default behavior of most blockchains, with the added twist that contracts are rewarded with **gas consumed**.

Scenario 2: User pays gas, contract applies fee

This scenario mirrors the previous one, but with an additional **fee** that the **user** must add to the transaction, specified by the **contract**. The **user** sets the **gas limit**, **gas price**, and the **fee** paid.

Normal flow

As in Section , (100-**p**)% of the **gas consumed** is transferred to the **block generator**, and **p** % is distributed to all touched contracts, weighed proportionally to the amount of **gas consumed**.

The **fee**, on the other hand, is paid directly to the **contract** by the **user**.

Insufficient gas

If there is insufficient gas to execute, the **gas consumed** up is paid to the **block generator**, and the **fee** is refunded to the **user**.

Scenario 3: Contract pays gas, contract applies fee

This scenario is similar to the previous one, in that it allows a contract to specify a **fee** that the **user** must pay. It differs however in that the **contract** pays for the gas consumed during a call, as opposed to the previous scenario where the **user** pays for the gas. This scenario allows a contract set a fixed **fee** paid by the **user**.

This allows contracts to effectively subsidize a **user**'s gas costs, and may be used by the contract to incentivize users to use their services. Contracts using this scenario may wish to set a **fee** higher than the gas costs themselves, such that the contract earns a profit from the transaction. The situation where this is not possible is described below in the *User pays fees lesser than gas paid by contract* section below.

Normal flow

The accounting of **gas** is similar to the previous scenario, with the exception that the **gas consumed** being paid by the **contract** as opposed to the **user**. (100-**p**)%

of the **gas consumed** is transferred to the **block generator**, and **p** % is distributed to all touched contracts, weighed proportionally to the amount of **gas consumed**.

Insufficient gas

In the same way as the other scenarios, if there is **insufficient gas** for the execution to complete, the **user** is refunded the full amount of the **fee**, and the **gas consumed** is paid to the **block generator**.

Setting gas price and limit

In the previous scenarios the **user** sets the **gas price** and **gas limit**, and sends a transaction to the network without any interaction with the **contract**. Given that, in this scenario, the **contract** is paying for the gas, it becomes necessary for the **contract** to be able to set the **gas price** and **gas limit** it is willing to pay for a specific transaction.

This imposes the need for the contract to be able to inform the **block generator** of the **gas price** and **gas limit** it is willing to pay for a given transaction, meaning that the binary interface of the **contract** with the **block generator** must be extended to support this functionality. For any of the previous calculations to be possible, the protocol must make data, such as low, average, and high gas price, available to the **contract**. This data will be served, like any other data, using a host function - a function that is not part of the **contract**, but is made available to it by the node.

User pays fees lesser than gas paid by contract

It is possible for a **user** to pay a **fee** that is less than the **gas consumed**. In this case, the **contract** paying for gas and levying a **fee** will be operating at a loss. Such a situation may be exactly what the **contract** intends, however, to ensure that no abuse is possible, the **contract** must be able to specify whether it is willing to operate at such a loss. If the **contract** is not willing to operate at a loss, and the **user** pays a **fee** that is less than the **gas consumed**, the transaction is treated in the same way as if it ran out of gas.

Service provider contract

This scenario is particularly useful for so-called service **provider** contracts. These are contracts designed to provide a service to **users**, such as generating a zero-knowledge proof, a subscription service, off-chain payments, etc. In particular, a zero-knowledge proof **provider** offers several advantages:

1. **Reduced note proliferation**: fees managed by a contract can be accumulated within a single note, which can be withdrawn later.

2. **Instant notification of new services**: wallets are immediately notified of new services and their fees through a simple event broadcast by registrars.

3. **Support for off-chain services**: the contract allows notifications and incentives for off-chain services, such as community-run proof sequencers and provers.

Scenario 4: Contract charges a percentage of an obfuscated amount

This scenario is similar to the second scenario, in that the contract charges a **fee** to the **user**. The difference lies in the fact that in this scenario the **fee** is a percentage of an obfuscated amount. Transactions of DUSK can be obfuscated, meaning that the amount being transacted is hidden from the network. In this scenario the **contract** accepts a percentage of that amount, and is assured of its validity using a zero-knowledge proof.

Scenario 5: Autocontracts

Leveraging the economic model, defined by the previously described scenarios, we are in a position to introduce a new type of contract: autocontracts. Autocontracts are contracts that are executed automatically when a specific event occurs, leveraging the economic model to pay for their own gas, using Scenario 3.

The main benefit of smart contracts is that they can be leveraged to implement complex logic that can automatically executed. This is particularly useful in the context of decentralized finance, where functionality such as limit orders, stop-loss orders, etc. can be implemented.

In its essence, reactive applications can now be implemented on-chain, without a need for a centralized service to monitor the state of the blockchain and react to events.

Considerations

Execution Priority: It is possible to envision autocontracts being executed in a few different orders. The first could be a simple "contract ID order", where the contracts are executed in order of their contract ID. This would be the simplest approach, but would not allow for any contract to be prioritized over another. The second approach could be a "contract fee order", where the contracts are executed in the order of which pays the most fees. This approach would be is more complex, but it would allow for contracts to be prioritized over others.

Any approach taken would need to be carefully considered, as it would become a part of the protocol.

Moment of Execution: There are multiple possible moments in which an autocontract could be executed. The first possibility would be immediately after the triggering event. This would be the simplest approach, but it could lead to some significant drawbacks, such as adding to the block gas limit, and potentially preventing the execution of other transactions. The second approach would be to execute the autocontracts at the end of the block, after all other transactions have been executed. This would be a more complex approach, but might allow the **block generator** to game the system by including transactions that would cause the autocontract to execute in a way that would be beneficial to the **block generator**.

Any approach taken would need to be carefully considered, as it would become a part of the protocol.

MODEL DISCUSSION

Dusk differs from traditional blockchain models by allowing gas costs to be paid by contracts rather than users. This approach has several advantages, including:

- It fundamentally improves the user experience.
- It solidifies long-term developer commitment through a sustainable revenue stream.
- It departs from conventional models where network congestion sets the price, opting instead for a cost-effective utilization-based approach.
- It shifts the focus from speculative tokens to genuine service utility, minimizing scams, and conferring special advantages to financial institutions by aligning with regulatory compliance requirements.

Adopting the economic protocol at Dusk's base layer rather than at the application level yields unique strategic benefits. It promotes a unified approach to the UX of wallets and clients, and incentivizes novel feature creation through a standardized base layer. In fact, the very concept of autocontracts, introduced in Scenario 5 emerges from this setup and would not be possible at the application level, offering a truly unique mechanism to create a scalable market for optimized smart contracts.

Dusk's economic model stands out from traditional approaches by allowing users to pay smart contracts directly and specify their preferred payment method. This model permits the transfer of gas costs to contracts rather than users, aligning more closely with conventional scenarios where service providers bear infrastructural costs.

Albeit other blockchains are trying to create incentives for smart contract developers (owners), they all tend to keep the current gas philosophy quite unchanged, that is users pay for gas, i.e. part of gas spent by the user is thus paid to contracts' owner.

Configuration	**Contract owner incentives**
Traditional	No incentive (owner's margin is **absent**, or depends on the smart-contract's specific logic).
Gas-subsidized contracts	Owner is subsidized by users' gas: a percentile of the gas fees no longer goes to the **block generator**, but to the invoked contracts.
Gas-and-mint subsidized contracts	Owner is subsidized by users' gas and also part of the block reward.

On Fee Denomination

We have chosen to write this document under the assumption that fees are denominated in DUSK. Given a specification of other denominations known to the **transfer contract**, it is possible to extend this economic model to support other denominations for fees. It would then become viable for a **user** pay for a contract's **fee** in any denomination supported by the **transfer contract**, with particular emphasis on EMT (Electronic Money Token) or ART (Asset Reference Token) as established by the MiCA regulatory framework.

Paired with the receiving **contract** being able to specify the denomination(s) that it is willing to accept, this would allow for some interesting use cases, such as frictionless digital currencies and tokenized loyalty programs.

Prior Art

Some blockchains are also trying to create incentives for smart contract developers/owners. Fantom[1] and NEAR[2] propose similar approaches to the one presented in this document, albeit with slightly different implementations.

[1] https://docs.fantom.foundation/funding/gas-monetization
[2] https://docs.near.org/concepts/basics/transactions/gas