Thanks to @villanuevawill for his numerous contributions to this outline.

This was initially inspired by Vitalik's "Generic fee payment" mechanism in hisproposal for a minimal phase 2, and is written under the assumption that the execution layer resembles something similar to what he describes.

We think it is beneficial to think of this as a rough outline of how a single execution script could be supported by clients to handle gas payments for arbitrary ShardTransaction

s (some functions are just black boxes). Please point out any issues you might come across. Also, I'd love to know some of the other potential approaches.

_

Each slot, the block proposer pays for the gas consumed by the block they create. This creates a strong incentive for block proposers to seek methods in which they can recoup the gas consumed and potentially accrue some profit.

The following assumptions are made:

- Clients will implement a mempool construction which allows the block proposer to choose transactions to include by gas_price

described in UserTransaction

. For this reason, it is likely that all transactions on the network will be propagated inside a WrappedTransaction

.

- Assets which expect to be used to pay for gas implement a function that allows a valid collator to transfer gas_limit * gas_price * asset_to_gwei_ratio + fee

from a user's account with only a signed transaction from the user. This is the optimistic portion.

- There is a method (or way to implement a method) getBlockProposer()

which returns the current block proposers BLSPubKey

.

In order to submit a transaction, a user must determine if the collator they wish to interact with accepts the token they plan to use to pay for their transaction. It is up to the collators to verify the validity of the token, it's witness_verification

, and update its fees accordingly. The standard for which that process occurs is out-of-scope for this proposal.

Fee Market Execution Script Specification

The general idea behind the fee market executor is that "collators" will verify and submit transactions. Collators pay the block proposers in gwei, while "users" will send funds to the collators in some other manner.

For definitions of ShardTransaction

, DepositData

, Withdrawal

, WithdrawalReceipt

, FormattedReceiptData

, callExecutionScript

, and check_and_set_bitfield_bit

see Vitalik's proposal.

Objects

UserTransaction

{ # arbitrary shard tx to execute "tx": ShardTransaction(E,T),

tx which will process the fee payment
"payment": ShardTransaction(P,PaymentData),

the shard on which the transactions will be executed
"shard_id": uint64,

generic cryptographic signature of this transaction
"witness": Witness

}

WrappedTransaction

{ # the transaction submitted by the user "tx": UserTransaction,

collator's nonce
"nonce": uint64,

collator's public key
"pubkey": BLSPubKey,

collator's signature
"signature": BLSSignature

}

Account

{ # account nonce "nonce": uint64,

account value
"value": uint64,

account public key
"pubkey": BLSPubKey,

epoch at which a withdraw can be completed
"withdrawable_epoch": uint64

}

PaymentData

{ # fixed collator fee in tokens "fee_amount": uint64,

gwei to token ratio to payout
"ratio": uint64

price in gwei the user is willing to spend
"gas_price": uint64,

max gas to spend on the transaction
"gas_limit": uint64,

generic address from which the collator can receive payment
"collator": bytes32

collator's official public key
"collator_bls": BLSPubKey

some hash of the user's tx so that it and the payment tx
are cryptographically linked
"tx_hash": bytes32,

the user can later prove if the generic transaction was
never executed, using this as a lower bound for the proof
"slot_submitted_at": uint64

}

Interface

depositToStake

Once a withdrawal receipt has been created on the beacon chain, it can be consumed using this executor on a specific shard. This will update the collator’s balance and allow them to begin submitting transactions to the fee market.

def depositToStake(receipt: WithdrawalReceipt, proof: WithdrawalReceiptRootProof): assert

verify_withdrawal_receipt_root_proof(get_recent_beacon_state_root(proof.root_slot), receipt, proof)

```
receipt_data = deserialize(receipt.withdrawal.data, FormattedReceiptData)

assert receipt_data.shard_id == getShard()
assert getStorageValue(hash(receipt_data.pubkey)) == b''

check_and_set_bitfield_bit(0, receipt)

setStorage(hash(receipt_data.pubkey), serialize(Account(
    nonce=0,
    value=receipt.amount,
    pubkey=receipt_data.pubkey,
    withdrawable_epoch=FAR_FUTURE_EPOCH
)))
```

submitTransaction

By performing the requested transaction and optimistically performing the gas accounting, it avoids increasing the transaction count by a multiple of N (where N is the number of transactions that must occur to manage the gas accounting). The following function is oversimplified and will need some way to limit the gas usage per call and to catch reverted calls so that the proper account can still occur.

def submitTransaction(tx: WrappedTransaction): assert bls_verify(message_hash=hash_tree_root({nonce: tx.nonce, data: tx.data}), pubkey=tx.pubkey, signature=tx.signature)

```
# deserialize the transactions into objects
user_tx = deserialize(tx.tx, UserTransaction)
payment_tx = deserialize(user_tx.payment.data, PaymentData)

# verify that the user's tx was not tampered with
assert verify_witness(user_tx.witness)

# no recursive transactions to the fee market
assert user_tx.tx.executor != getExecutor()

# ensure the transaction is on the correct shard
assert user_tx.shard == getShard()

# verify the collator is not in the process of withdrawing
assert is_collator_valid(tx.pubkey)

# run the user's transaction
gas_spent = callExecutionScript(
    user_tx.tx.executor,
    user_tx.tx.data,
    payment_tx.gas_limit,
)

# transfer the fee + gas_spent to the collator using `transferFeeToCollator`
# note: if the collator trusts the execution script the user's transaction is
# coming from, there could be additional logic added to skip this under the
# assumption that the user's transaction will natively forward a gas payment
# from the user's wallet
gas_spent += callExecutionScript(
    user_tx.payment.executor,
    user_tx.payment.data + gas_spent,
    TRANSFER_GAS_LIMIT
)

# transfer the gas_spent + the constant gas used to the block proposer
transfer(
    tx.pubkey,
    getBlockProposer(),
    (CONSTANT_GAS + gas_spent) * user_tx.gas_price)
)
```

slashCollator

By optimistically transferring funds to the collator, we expose an exploit where the collator could send a transaction which transfers the fee to themselves without actually submitting the user’s transaction. In this case, the user (or possibly anyone) could submit a proof of exclusion where between the time the transaction was submitted to the time that the fee was collected, the transaction was not executed.

def slashCollator(tx: UserTransaction, proof: TransactionExclusionProof): # verify that the tx was not included and the fee was redeemed not_included = verify_tx_exclusion_proof(get_recent_beacon_state_root(proof.slot), tx, proof
)

```
if not_included:
    # penalize the collator for redeeming the fee w/o submitting the tx
    transfer(tx.pubkey, tx.collator, proof.gas_spent * tx.gas_price + INVALID_TX_PENALTY)
```

startWithdrawalToBeacon

Once a block producer has accrued some value or a collator is ready to withdraw some collator from slashable action and immediately exiting the executor. The following function is highly unoptimized: the balance could be withdraw depending on a ratio check, the block producer shouldn't need to wait for a lock up, etc.

def startWithdrawalToBeacon(account: bytes32, nonce: uint64): account_data = deserialize(getStorageValue(account), Account) assert account_data.nonce == nonce assert account_data.withdrawable_epoch == FAR_FUTURE_EPOCH assert bls_verify(message_hash=hash_tree_root({nonce: nonce}), pubkey=account_data.pubkey, signature=signature)

```
account_data.nonce += 1
account_data.withdrawable_epoch = getCurrentEpoch() + MIN_WITHDRAW_DELAY

setStorage(account, account_data)
```

finalizeWithdrawalToBeacon

Once an account withdrawable_epoch has arrived, they may submit a transaction to finalize the withdrawal. This will withdraw their entire balance to a receipt which may be redeemed on the beacon chain.

def finalizeWithdrawalToBeacon(account: bytes32, nonce: uint64, deposit_data: DepositData): assert verify_deposit_data(deposit_data)

```
account_data = deserialize(getStorageValue(account), Account)
assert account_data.nonce == nonce
assert account_data.withdrawable_epoch <= getCurrentEpoch
assert account_data.value == deposit_data.amount
assert bls_verify(
    message_hash=hash_tree_root({nonce: nonce, deposit_data: deposit_data}),
    pubkey=account_data.pubkey,
    signature=signature
)

account_data.nonce += 1
account_data.value = 0
account_data.withdrawable_epoch = FAR_FUTURE_EPOCH

setStorage(account, account_data)
saveReceipt(2**256-1, deposit_data)
```

transfer

(private)

Since the transfer function is private and only invoked at the end of submitTransaction

, there is no need to verify a signature or nonce.

def transfer(sender: BLSPubKey, target: BLSPubKey, amount: uint64): sender_account = deserialize(getStorageValue(sender), Account) target_account = deserialize(getStorageValue(target), Account)

```
assert sender_account.value >= amount

setStorage(sender, Account(
    pubkey=sender_account.pubkey,
    nonce=sender_account.nonce,
    value=sender_account.value - amount,
    withdrawable_epoch=sender_account.withdrawable_epoch
))
setStorage(target, Account(
    pubkey=target_account.pubkey,
    nonce=target_account.nonce,
    value=target_account.value + amount,
    withdrawable_epoch=target_account.withdrawable_epoch
))
```

Generic Asset Interface

For an asset to be compatible with the above architecture, it would need to implement the following interface. The gas_used

parameter should not be part of the witness / signed transaction as it will be appended to the call data by the fee market

execution script. This should still be secure since gas_used

is not bias-able by a collator.

def transferFeeToCollator(fee: uint64, ratio: uint64, gas_price: uint64, gas_limit: u64, collator: bytes32, collator_bls: BLSPubKey witness: bytes, gas_used: uint64): assert verify_witness(witness) assert gas_used <= gas_limit

sender = get_address(witness)

check that the collator exists & has not begun a withdraw
assert is_collator_valid(collator)

# transfer the used gas + the fee without approval
total_payout = (gas_used * gas_price * ratio) + fee
unsafe_transfer(sender, collator, total_payout)

Off-chain Work

In order to minimize the footprint of this construction on chain, there needs to be some work do off-chain in order for transactions and payments to be processed safely and efficiently.

Client

- organizes transactions by gas_price

to ensure the best payout

- verifies the submitting collator has at least MIN_STAKE + gas_price * gas_limit

funds available

Collator

- verifies the user has at least (gas_used * gas_price * ratio) + fee

funds available

- checks the user’s signature on the payment transaction
- the payment transaction matches the agreed upon terms
- the payment transfer is addressed to the collator and the generic address is an address the collator controls
- that the nonce for the user’s payment transaction is valid
- probably more things

Example Usage Flow

Alice: user

Bob: collator

1. Alice generates some transaction she would like submitted to a shard chain. She wishes to pay for the transaction in DAI.
2. Alice determines a gas_limit

large enough to execute her transaction and asks what is the current gas_price

for the shard, what the conversion ratio

is between DAI and gwei, and what the collator’s fee is in DAI.

1. Bob responds with the current gas_price

, ratio

, and his fee

.

1. Alice, agreeing with the Bob's rate, signs three transactions: her original transaction, a transaction which would transfer the payment to the collator, and a UserTransaction

which is a combination of the two. She then sends her UserTransaction

to Bob.

1. Bob receives the transaction and verifies that the transactions are signed by Alice, that the payment details are correct, and that Alice's DAI balance will cover the transaction. It is important to note that if Bob does not do this verification he could be on the hook for the gas consumed by the transactions without being able to receive payment from Alice.

2. Once Bob is confident that the payment transaction will go through, he can submit a WrappedTransaction

to a shard node.

1. The shard node will order its incoming transactions in descending order of gas_price

. Assuming Bob gave Alice a reasonable gas_price

, the transaction should chosen to be included relatively quickly.

1. During execution three main things happen: Alice's transaction is executed, Alice's payment transaction is executed, the collator transfers gas_spent * gas_price

to the block's proposer.

1. After the block is generated, Alice will see that her transaction was correctly executed and Bob will be able to move his newly accrued DAI to any token / shard of his preference.

In the example above, it would be trivial for a wallet to support multiple collators which would create a competitive fee market for transaction execution.

Questions

What if X

is invalid?

Most questions of this form mean the collator should perform some validation before submitting the user's transaction. For example, if the user doesn't have enough balance to cover the transaction they submitted to the collator. The collator is the one that will be liable when the transaction fails, so they should ensure that won't happen.

Why do collators need to stake in this system?

Staking allows the system to operate in an optimistic manner, where the collator is able to retrieve the gas payment from the user without needing transactions split across multiple blocks. This is possible by processing the transaction and the payment in the fee market, synchronously. The stake will deter a collator from redeeming a reward without actually submitting the user's transaction. If this occurs, the user (or their wallet), may submit a transaction proving that the collator acted maliciously and slash a portion of their stake.

Would collators need to stake on every shard?

In the example provided, yes—they would need to stake on every shard they plan to submit transactions on. This could possibly be mitigated using a sort of credit system and by specifying a particular shard on which slashing / withdrawing will occur.

What if I don't want to submit through a collator?

You can easily become your own collator. It would be relatively easy to expand this proposal to support some concept of a whitelist so that transactions from a collator can skip the transaction with the fee and gas payment from the user. This would avoid the need for staking when submitting transactions for the whitelisted addresses since it is assumed that some off-chain agreement has occurred.

What if the collator swaps out the user's transaction for a transaction their own to maximize the gas payment for themselves?

The collator won't actually profit in this case, as they'll need to pay the block producer that gas anyways. The proposal could potentially be expanded to make this behavior slashable.

How could this be optimized?

There are certainly many different optimizations that this proposal could use. Here are a few:

- The fee

could decay the farther from the slot at the time of execution is from slot_submitted_at

. This could incentivize collators to submit transaction quickly.

- Some sort of defi product could help collators raise funds to stake and earn a portion of the fee payout.
- Payment could be collator agnostic by implementing a map on the fee market execution script between the collator and their addresses in various different execution scripts.
- Support “self-collators”