Here is a natural way for staging user-level ETH transfers on top ofnetted balance approach for EE-level transfers. Adopts ideas from non-sequential receipts (meta-ee) approach.

As is typical, the user-level transfers are split into a debit transaction on the sender shard, and a credit transaction on the recipient shard in a subsequent slot. However, utilising the netted balance approach, the EE-level transfers happen atomically and synchronously. Can we call this approach quasi-synchronous!

Preliminaries

- s_1, s_2

shards.

- E_1, E_2

execution environments.

- a_i

users on (s_1,E_1)

, b_j

be users in (s_2, E_2)

for some index sets that i

and j

range on.

Transaction Types

- Cross-shard debit transfer
- a_i \stackrel{x_i}{\Longrightarrow} b_i

, cross-shard transfer of x_i

ETH from the user a_i

on (s_1,E_1)

to the user b_i

on (s_2,E_2)

.

- Submitted on sender shard.

- Emits a Credit

System event on success.

- a_i \stackrel{x_i}{\Longrightarrow} b_i

, cross-shard transfer of x_i

ETH from the user a_i

on (s_1,E_1)

to the user b_i

on (s_2,E_2)

.

- Submitted on sender shard.

- Emits a Credit

System event on success.

- Cross-shard credit transfer
- a_i \stackrel{x_i}{\longrightarrow} b_i

, credit transfer of x_i

ETH to b_i

on (s_2,E_2)

, which is from a_i

on (s_1, E_1)

.

- Submitted on recipient shard.

- Includes the Credit

System Event and the Merkle Proof for it.

- Bitfields are used for replay protection.

- Emits a Revert

System Event on failure.

- a_i \stackrel{x_i}{\longrightarrow} b_i

, credit transfer of x_i

ETH to b_i

on (s_2,E_2)

, which is from a_i

on (s_1, E_1)

.

- Submitted on recipient shard.

- Includes the Credit

System Event and the Merkle Proof for it.

- Bitfields are used for replay protection.

- Emits a Revert

System Event on failure.

- Cross-shard revert transfer
- a_i \stackrel{x_i}{\longleftarrow} b_i

, a revert transfer of x_i

ETH to a_i

when the credit transfer of the form a_i \stackrel{x_i}{\longrightarrow} b_i

fails.

- Includes the Revert

System Event and the Merkle Proof for it.

- Bitfields are used for replay protection.

- a_i \stackrel{x_i}{\longleftarrow} b_i

, a revert transfer of x_i

ETH to a_i

when the credit transfer of the form $a_i \stackrel{x_i}{\longrightarrow} b_i$

fails.

- Includes the Revert

System Event and the Merkle Proof for it.

- Bitfields are used for replay protection.

System Event messages

System Events are similar to application event messages in contract code, but are unforgeable by the application. We have two System Events: Credit

and Revert

. All of them include sender details (shard-id, EE-id, user address), recipient details, transfer amount. They also include the block number of the shard block where this event is emitted, and an index number starting from 0 (for every block).

Algorithm for the Block Proposer

Suppose a block proposer selects some transactions $t_1, ..., t_n$

, where every t_i

, $1 \le i \le n$

, is either a cross-shard debit transfer from a user in E_1

to a user in E_2

, or a credit transfer from a user in E_2

to a user in E_1

.

- net = 0; // tracks the net amount for EE-level transfers
- for i : 1 … n
- if t_i: $b_i \stackrel{x_i}{\longrightarrow} a_i$

AND BitFieldCheck

(t_i

) passes AND Merkle Proof check of Credit

System Event passes AND realBal(s_1,E_1

) > net + x_i

- include t_i

to the block

- if t_i

executes successfully * bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- if failure
- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- include t_i

to the block

- if t_i

executes successfully * bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- if failure
- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i
- if t_i : a_i \stackrel{x_i}{\Longrightarrow} b_i

AND realBal(s_1,E_1

) > net + x_i

- include t_i

to the block

- if t_i

executes successfully

- bal(a_i

) -= x_i

(implied)

- net += x_i
- emit Credit

(a_i, x_i, b_i

) System Event

- end for
- if t_i: b_i \stackrel{x_i}{\longrightarrow} a_i

AND BitFieldCheck

(t_i

) passes AND Merkle Proof check of Credit

System Event passes AND realBal(s_1,E_1

) > net + x_i

- include t_i

to the block

- if t_i

executes successfully * bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- if failure
- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- include t_i

to the block

- if t_i

executes successfully * bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- bal(a_i

) += x_i

; (is implied in a successful execution)

- SetBit

(t_i

);

- if failure
- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- emit Revert

(b_i, x_i, a_i

) System Event

- net += x_i

- if $t_i : a_i \stackrel{x_i}{\Longrightarrow} b_i$

AND realBal(s_1,E_1

) > net + x_i

- include t_i

to the block

- if t_i

executes successfully

- bal(a_i

) -= x_i

(implied)

- net += x_i

- emit Credit

(a_i, x_i, b_i

) System Event

- end for

- s_1

.bal[s_1,E_1

] -= net

- s_1

.bal[s_2,E_2

] += net

Note that when processing a pending cross-shard credit transfer transaction b_i \stackrel{x_i}{\longrightarrow} a_i

, the EE-level transfer is already complete.

Discussion

Replay protection

Bitfields are used for replay protection. Recipient EE keeps a map from (\mathit{senderShard, senderEE, senderBlockNumber}) \mapsto b_1...b_n

where b_i

corresponds to $i$th credit transfer transaction created in the shard block numbered \mathit{senderBlockNumber}

of \mathit{senderShard}

.

- BitFieldCheck

(t_i

) checks to see if the given credit transfer t_i

has already been processed. This is done by checking the bit corresponding to the index specified in the SystemEvent in the above map. If zero, returns true

. If no entry is present, then a new entry is created, and the bits are initialised to zeros, and returns true

. If the corresponding bit is set to 1, returns false

.

- SetBit

(t_i

) sets the bit for this credit transfer to 1.

Similar mechanism is used for revert transfers protecting replays.

We need a sliding-window time-bound

for processing credit transfer transactions. If this time-bound is two epochs, then all the bitfields related to block numbers prior to two epochs are discarded, and revert transactions are placed instead, after affecting the EE-level revert.

Problem 1

Note that the bit is first initialised to zero, when BP first tries to include the credit transfer in a block. If the client never creates and submits a credit-transfer, then the recipient (user) loses the amount, however, there is no ETH loss at the EE-level. The time-out applies to credit transfers that are submitted to the transaction pool. So, a submission of credit transfers need to be incentivised.

Revert Transfers

Note that the revert transactions don't appear in the above algorithm. They are processed as regular transactions, except that we check the Bitfields similar to Credit bitfields (explained above) to ensure double spending.

We do not want revert transfers to fail. So, there are two design choices:

1. Zero-gas

By making the revert transfers not consume any gas, we can expect it to succeed always.

1. ETH loss at user-level

If the revert transfers run out of gas and fail, then we can emit a RevertFail

System Event. The sender (user) loses ETH, however, the sender EE does not lose ETH. One solution is to invoke a EE host function periodically (say, at every checkpoint), and process the accumulated RevertFail

System Events and push the ETH back to the sender (user).

There is still one remote case where the sender

account itself disappears before a revert is processed. Then the revert transfer fails. We end up in a state where the EE balance does not match up with its user-level balances.

Problem 2

: @liochon pointed out that if revert transfers are cheap or zero-gas, then a malicious BP can deliberately (maliciously) fail credit transfers and load up the system with many cheap revert transfers. Because a successful credit transfer does not depend on anything but the presence of the recipient account, a malicious BP cannot fail the credit transfer on demand. However, he has the choice not to include it in the block. This defers the transaction inclusion in the block, and increases the probability of a credit time-out.

Gas billing EE-level transfers

It is easy to bill the EE-level transfers to the transactions that contribute to it, by a small modification in the above algorithm, which book-keeps the participating transactions.

Failing EE-level balance checks

Because we have deferred transfers at user-level, we need to check real balances at EE-level (as shown in the above algorithm). The flexibility of this approach is that depending on the transfer amounts

and the EE balance

some transactions get included and some not.

Benefits

- No locking / blocking.
- No constraint on the block proposer to pick specific transactions or to order them.
- EE-level transfers can be billed uniformly to the contributing transactions. Client's responsibility to put enough gas limit on debit and credit transfers to accomodate EE-level transfers as well.

Drawbacks

- Problem 1
- Problem 2

Examples

Optimistic case

Block on shard s_1

$a_1 \stackrel{10}{\Longrightarrow} b_1\ a_2 \stackrel{20}{\Longrightarrow} b_2\ a_3 \stackrel{30}{\Longrightarrow} b_3\ E_1 \stackrel{60}{\Longrightarrow} E_2$

Subsequent block on shard s_2

$b_1 \stackrel{5}{\Longrightarrow} a_2\ a_1 \stackrel{10}{\longrightarrow} b_1\ a_2 \stackrel{20}{\longrightarrow} b_2\ a_3 \stackrel{30}{\longrightarrow} b_3\ E_2 \stackrel{5}{\Longrightarrow} E_1$

A credit transfer of the form $b_1 \stackrel{5}{\longrightarrow} a_2$

gets included in a subsequent block of shard s_1

.

When debit fails

Block on shard s_1

$a_1 \stackrel{10}{\Longrightarrow} b_1\ a_2 \stackrel{20}{\Longrightarrow} b_2\ \mathbf{FAIL}~~~a_3 \stackrel{30}{\Longrightarrow} b_3 \ E_1 \stackrel{30}{\Longrightarrow} E_2$

Subsequent block on shard s_2

$b_1 \stackrel{5}{\Longrightarrow} a_2\ a_1 \stackrel{10}{\longrightarrow} b_1\ a_2 \stackrel{20}{\longrightarrow} b_2\ E_2 \stackrel{5}{\Longrightarrow} E_1$

…

When credit fails

Block on shard s_1

$a_1 \stackrel{10}{\Longrightarrow} b_1\ a_2 \stackrel{20}{\Longrightarrow} b_2\ a_3 \stackrel{30}{\Longrightarrow} b_3\ E_1 \stackrel{60}{\Longrightarrow} E_2$

Subsequent block on shard s_2

$b_1 \stackrel{5}{\Longrightarrow} a_2\ a_1 \stackrel{10}{\longrightarrow} b_1\ a_2 \stackrel{20}{\longrightarrow} b_2\ \mathbf{FAIL}~~~a_3 \stackrel{30}{\longrightarrow} b_3\ E_2 \stackrel{35}{\Longrightarrow} E_1$

Subsequent blocks on shard s_1

include a revert transfer: $a_3 \stackrel{30}{\longleftarrow} b_3$

, and a credit transfer: $b_1 \stackrel{5}{\longrightarrow} a_2$

. Note that we should not allow revert to fail.

Thanks @drinkcoffee, @hmijail and @SamWilsn for your inputs.