Non-sequential receipt transactions is a cross-shard strategy proposed here https://github.com/ethereum/wiki/wiki/sharding-faq#how-can-we-facilitate-cross-shard-communication. This write-up is not a new proposal it is an in-depth overview and appraisal of non-sequential receipt cross-shard transactions.

A non-sequential receipt cross-shard transaction strategy is an asynchronous execution strategy with a shards serially processing segments of a transaction that mutate state within the shard and publishing a receipt of the processed transaction segment. The transaction of a receipt published is then continued on a foreign shard in a repeating process until the transaction is executed or orphaned.

Non-sequential receipt sequence diagram

[

1608×768 48.1 KB

](https://ethresear.ch/uploads/default/original/2X/7/7731b1cf22c937eabec22fc35f481419fb9079d7.jpeg)

1. User composes a transaction
2. User submits the transaction to mempool
3. Shard A retrieves the transaction from mempool
4. Shard A processes the transaction until it halts due to a cross-shard call
5. Shard A produces a shard block containing a receipt, the shard state root of the shard block is sent for cross-linking in the beacon block
6. A beacon block is published cross-linking the shard state root of Shard A
7. Shard B retrieves the cross-linked receipt
8. Shard B processes the transaction until it executes completely
9. Shard B produces a shard block containing a receipt, the shard state root of the shard block is sent for cross-linking in the beacon block
10. A beacon block is published cross-linking the shard state root of Shard B

Problems

The asynchrony of non-sequential receipts introduces user experience problems.

Gas price variance across shards

Gas prices across shards will vary to load balance the throughput of transactions. Variance in these gas prices can be estimated at the time of transaction creation by the user. During the asynchronous execution of the transaction can result an unanticipated gas price rise that will block the execution and cause a transaction to hang partially executed.

Naively designed cross-shard calls

Poorly designed cross-shard calls can result in a long duration of execution. Using a simple exchange contract as an example. In this example the contracts TokenA.sol

, TokenB.sol

, and Exchange.sol

are all on separate shards.

[

1696×936 60.5 KB

](https://ethresear.ch/uploads/default/original/2X/5/59d989152302be24f52606af1334a56cd894a425.jpeg)

1. Maker calls approve on Token A
2. Taker calls approve on Token B
3. Maker calls the Exchange contract to swap the assets
4. Exchange contract calls transfer method on Token A (cross-shard call)
5. Token A transfers the asset (cross-shard call)
6. Exchange contract calls transfer method on Token B (cross-shard call)
7. Token B transfers the asset (cross-shard call)
8. Exchange contract ends execution

The exchange contract example demonstrates the problems of naively designed cross-shard calls. The transaction results in 4 cross-shard calls resulting in a transaction delay. Assuming 12 seconds beacon block times, as single atomic transaction 3 -> 8 results in an optimistic case 48

second transaction.

Execution collisions

Concurrent mutation of read/write data during execution. Suppose the same exchange contract described above. During step 6

, a second transaction transfers the Taker

balance in Token B

resulting in three scenarios a hung transaction, a reversion, or a non-atomic swap.

1. Hung transaction
2. a transaction where the execution is incomplete and unresolvable
3. Reverted transaction
4. an executed transaction that generates an exception disallowing state mutation
5. Non-atomic swap
6. a partially executed swap transaction where a single counter-party benefits (seetrain and hotel problem)

Trains and hotel problem

Non-sequential receipts transactions can be reverted mid-execution. A transaction that has been partially executed cannot be committed to the state tree until all subsequent receipts have been executed.

Prioritization of receipts vs. transactions

As described in Vitalik's write upImplementing cross-shard transactions A block producer must choose between executing a new transaction from mempool or executing a receipt transaction.

Receipt processing can be done in two separate ways

1. Receipt Queue

A receipt queue creates a reserved class after the initial execution (forced inclusion). This strategy results in enforcing a limit on the inclusion of new transactions in the system as receipts may occupy the finite reserved space in a block.

1. Passive Execution Receipt

A receipt and witness are represented at each execution segment. A block producer will process the initial transaction segment. Each subsequent executing shard is represented with the receipt and appropriate transaction fee by the user. This strategy allows transactions and receipts to be treated valued equally to a block producer.