Algorithm for Cross-shard Cross-EE Atomic User-level ETH Transfer in Ethereum 2

Raghavendra Ramesh
Consensys Software R & D
raghavendra.ramesh@consensys.net

Abstract

We address the problem of atomic cross shard value transfer in Ethereum 2. We leverage on Ethereum 2 architecture, more specifically on Beacon chain and crosslinks, and propose a solution on top of the netted-balance approach that was proposed for EE-level atomic ETH transfers. We split a cross-shard transfer into two transactions: a debit and a credit. First, the debit transaction is processed at the source shard. The corresponding credit transaction is processed at the destination shard in a subsequent block. We discuss various scenarios of debit failures and credit failures, and show our approach ensures atomicity.

1   Introduction

Ethereum 2 is planned to have 64 shards. The blocks on the shard chains synchronise with the Beacon chain using crosslinks. The crosslinks are the stateroots (root of Merkle tree of the shard state), and provide a means to validitate the integrity of a shard state on another shard.

The execution in Ethereum 1 happens in a fixed fashion using EVM and fixed notion of account id and balances. The plan is to generalise this notion of execution environment (EE) in Ethereum 2. For instance, an environment similar to Bitcoin can be hosted on Ethereum 2 as one EE. Some EEs could use another virtual machine instead of EVM. The EEs and users of them are distributed across multiple shards. In such a scenario, the cross-shard cross EE ETH transfers become necessary. This paper presents an algorithm for atomic user-level ETH transfers between two EEs hosted on two different shards.

Because one execution environment is complete in its own means, we need to maintain ETH balances of each of these execution environments separately. An EE could provide another currency on top of ETH . Users are associated with EEs in the sense that an EE serves as a home for multiple users. The problem of transferring value from a user a of EE E_1 to another user b of another EE E_2 involves two aspects:

1. the transfer of ETH from EE E_1 to EE E_2, and

2. the transfer of values from the user a to user b.

Vitalik Buterin proposed a *netted balance approach* ([1]) for atomic cross-shard transfer of ETH between EEs. This paper proposes user-level atomic cross-shard transfer on top of this netted balance approach of EE-level transfers.

2   Netted balance approach

Vitalik Buterin proposed a netted balance approach ([1]) for atomic cross-shard transfer of ETH between EEs. The naïve idea is for every shard to maintain its balance of every EE. In the netted-balance approach, every shard maintains the part-balances of every EE on every other shard.

Suppose we have three shards A, B and C. Consider an EE e. Then, the balance information of e on shard A is distributed on all shards A, B and C. So, each shard now stores an ordered triple of part-balances. For example the e's balance information on A is stored as an ordered triple of the form $(10, 20, 30)$, meaning e's part-balance on shard A is 10, e's part-balance on shard B is 20, and e's part-balance on shard C is 30. Continuing the example, suppose we have triples $(-5, 10, 20)$ on B and $(1, -2, 3)$ on C. To obtain e's full balance on shard A, we need to sum up e's part balances on shards A, B, and C, i.e., $10 + (-5) + 1 = 6$ ETH .

The benefit of this arrangement is that the transfer of x ETH from shard A to shard B can be affected by an intra-shard operation only on the source shard (operation completely on A here) by subtracting x ETH from A's part-balance and adding x to B's part-balance. The A's triple changes from $(10, 20, 30)$ to $(10 - x, 20 + x, 30)$. The triples on other shards are not touched. Thus, the cross-shard EE-level ETH transfer is accomplished by a single atomic operation on a single shard.

However, the downside is that querying of an EE's balance on a shard is not a single operation because all the part balances on all other shards need to be fetched and summed.

This idea naturally extends to cross EE transfers too.

3   Atomic User-level Transfer

The core idea of this proposal is to extend and leverage the netted-balance approach of distribution of EE-balances to outstanding user-level credits and outstanding user-level reverts. The netted state (extends the netted balance) is used as a channel to communicate outstanding credits to recipient shard and outstanding reverts to the sender shard. We now describe the approach in full detail now.

3.1 Preliminaries

For i in natural numbers, let s_i's denote shards, E_i's denote execution environments, a_i, b_i's denote users. We use the concept of *System Event messages* from [2]. These messages are similar to application event messages in contract code, but are unforgeable by the application.

In our approach, we use one System Event message called **ToCredit** , which includes:

- sender details (shard-id, EE-id, user address),

- recipient details (shard-id, EE-id, user address),

- transfer amount,

- the block number of the shard block where this event is emitted, and

- an index number starting from 0 (for every block).

3.2 Transactions

A cross-shard transfer of x ETH from an user a_i on (s_1, E_1) to an user b_i on (s_2, E_2) is split by our system into two transactions in a natural way: a cross-shard debit transfer ($\Longrightarrow$) and a cross-shard credit transfer ($\longrightarrow$), corresponding to deducting x on the source side and adding x to the destination side respectively.

Cross-shard debit transfer transaction is signed by the sender a_i, and the signature is stored in the fields v, r and s as in Ethereum 1. It contains a unique transaction identifier. It is submitted on sender shard, and emits a **ToCredit** System event on success.

Cross-shard credit transfer transaction is submitted on recipient shard, and includes the **ToCredit** System Event and the Merkle Proof for it.

[[What does **ToCredit** system event contain]]

3.3 Shard State

In the netted-balance approach, each shard s stores the part-balances of every EE on every shard. It can then be seen as a matrix, say $s.partBalance$ of size number of shards $\times$ number of EE's. Here, $s.partBalance[s_i, E_j]$ gives the part-balance of EE E_j on shard s_i, which is recorded at shard s. The real balance of EE E_j on shard s_i is given by :

$$realBalance[s_i, E_j] = \sum_k s_k.partBalance[s_i, E_j].$$

We introduce more elements on top of part balances in the shard state, and we use the name *partState* matrix in place of *partBalance* matrix. Each cell of this matrix has 3 elements, as shown in Figure 1. They are:

3

- *partBalance* (as in netted-balance approach),

- *credits*, the set of cross-shard credit transactions that needs to be imported on the destination shard,

- *reverts*, the set of cross-shard revert transactions that needs to be effected.
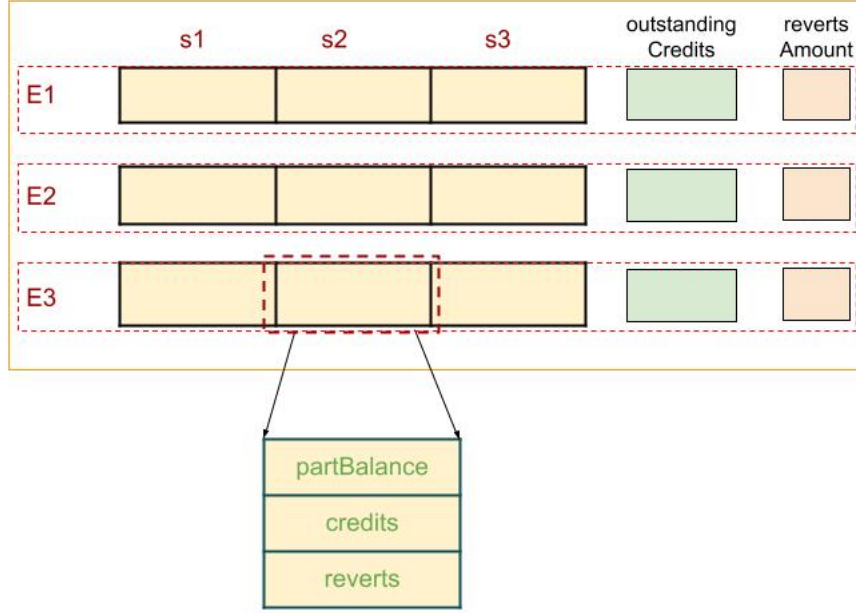

Figure 1: Shard State

The shard state contains a map called *outstandingCredits*. At the time of processing a block k on shard s, for every EE E, *s.outstandingCredits* adds the outstanding credit transactions targeted to (s, E) from a shard s' and an EE E' whose corresponding debit transactions were successfully processed in the immediately previous block at shard s'. So, we have:

$$s.outstandingCredits[E][s', E', k-1] = \{ \ e \mid e \in s'.partState[s, E].credits\}$$

There are two ways to remove an entry from *s.outstandingCredits*. One, is by the successful proccessing of the credit transfer transaction, the other is when we *time-out* processing of the credit transfer. The time-out is required to guarantee a bound on the transfer time. Typically time-outs are specified in terms of number of blocks.

For every EE E, the shard state also contains the *revertsAmount* that bookkeeps the net amount to be reverted at EE-level.

3.4   Block Proposer Algorithm

We now present the main contribution: the algorithm for the block proposer in handling cross-shard cross-EE transfers with the guarantee of atomicity. The idea is to split a cross-shard transfer into a debit and a credit. First, a debit transaction is processed at the source shard, and the EE-level transfer is fully effected. If successful, the corresponding user-level credit transaction is queued on to the destination shard, which is processed in a subsequent block. If the credit fails, then we do the EE-level revert and user-level reverts NOT as a separate transaction but as an enshrined execution processing.

Without loss of generality we assume that a Block Proposer (BP) is proposing a block numbered k on shard s_1. Then BP executes the following steps for every EE E_i.

1. **Initialisation.** Obtain the part states of E_i from every shard. Ensure that the obtained $s_i.partStates$, $1 \leq i \leq n$, are correct using Merkle Proofs and crosslinks. Compute the real balance of E_i on s_1 using

   $$realBalance(s_1, E_i) = \sum_i s_i.partState[s_1, E_i].balance.$$

   Bookkeep a $revertsAmount$ for every shard s and EE E. Initialise this $revertsAmount$ to 0, i.e., for every shard s and every EE E, we have $revertsAmount(s, E) = 0$;

2. Update $BitFieldMap$

   - Add entries for outstanding credits: $s', E', (k-1)] \mapsto \{e \mapsto 0 | e \in \bigcup_i s_i.partState[s_1, E_i].credits\}$
   - Kick out expired credits: If there is an entry with $[s', E', key]$ such that $key + timeBound == k$ then do:
     - Update $s_1.partState[s', E'].reverts := \{e | (e \mapsto 0) \in BitFieldMap\,(key) \wedge sender(e) \in (s', E')\}$.
     - Delete the entry with key.
     - Update $net(s', E') := \Sigma_e\, x_e$ where $e \in s_1.partState[s', e'].reverts$ and x_e denotes the transfer amount.

3. Process user-level reverts

   - Update $reverts := \bigcup_i s_i.partState[s_1, E_i].reverts$
   - For each $t_i \in reverts$, $sender(t_i).balance+ = x_i$.

4. **Reset.** For every pair (s_2, E_j), reset the following.

   - $s_1.partState[s_2, E_j].credits = \emptyset$
   - $s_1.partState[s_2, E_j].reverts = \emptyset$

5. For every pair (s_2, E_j)

(a) Select cross-shard transactions $t_1, \ldots, t_n$ from (s_1, E_i) to (s_2, E_j) to be included in the block. Can it from the ReservedCredits / BitFieldMap also?

(b) For every $i \in \{1, \ldots, n\}$:

- If $t_i : a_i \xRightarrow{x_i} b_i$ AND realBalance(s_1, E_i) ¿ net$(s_2, E_j) + x_i$
    - include t_i to the block
    - if t_i executes successfully
        * balance(a_i) -= x_i (implied with successful execution of t_i)
        * net(s_2, E_j) += x_i
        * emit **ToCredit** (a_i, x_i, b_i) System Event
        * s_1.partState$[s_2, E_j]$.credits $\cup =$ t_i;
- Else if $t_i : b_i \xrightarrow{x_i} a_i$ AND Merkle Proof check of **ToCredit** System Event passes AND realBal(s_1, E_i) ¿ net$(s_2, E_j) + x_i$
    - if *expired*(t_i) OR *BitCheck(t_i)* fails delete t_i from transaction pool
    - else
        * include t_i to the block
        * *SetBit*(t_i);
        * if t_i executes successfully - bal(a_i) += x_i; (implied with successful execution of t_i)
        * if failure - s_1.partState$[s_2, E_j]$.reverts $\cup =$ t_i; - net(s_2, E_j) += x_i

(c) Update EE-level transfer - s_1.partState$[s_1, E_i]$.balance -= net(s_2, E_j); - s_1.partState$[s_2, E_j]$.balance += net(s_2, E_j);

Remarks 1. Before processing a pending cross-shard credit transfer transaction $b_i \xrightarrow{x_i} a_i$, the EE-level transfer is already complete. 2. User-level reverts happen in the immediate next slot after a failed or expired credit transfer. The EE-level revert happens in the same slot as the failed / expired credit transfer. 3. Transaction identifiers need to be unique inside the *time-bound* 4. There is a corner case where the sender disappears by the time revert happens, then we end up in a state where there is ETH loss at user-level, but not at EE-level. This appears to be a correct state when such a situation happens. 5. A transaction is not included in a block if the EE does not have sufficient balance. Note that EE balance check is done even for credit transfers because of potential reverts.

BitFieldMap The datastructure *BitFieldMap* is stored in the shard state to protect against replays of credit transfers and to time-out credit transfers. It is a map from a block number to a set of elements of the form $(t_i \mapsto b_i)$, where the block number identifies the block on the sender shard where a cross-shard debit transfer t_i happened, and b_i is a bit initially set to 0. - *BitCheck*(t_i) returns true when there is $(t_i \mapsto 0)$ for any block number. Because transaction identifiers are unique, a transaction appears only once in *BitFieldMap*. Zero indicates that the credit is not processed yet. One indicates that the credit is

already processed. If no entry is present, then this is an invalid credit and false is returned. - *SetBit*(t_i) sets the bit for t_i to 1 for the block number that is already existing.

A *timeBound* is required to kick out long pending credit transfers. The second bullet of step 2 describes this procedure. The idea is to move the expired user-level credit transfers as user-level reverts on the sender shard, and achieves a fixed size for *BitFieldMap*.

A *timeBound* is required to kick out long pending credit transfers. The second bullet of step 3 describes this procedure. The idea is to move the expired user-level credit transfers as user-level reverts on the sender shard, and achieves a fixed size for *BitFieldMap*.

Processing Revert Transfers A revert transfer is placed in the shared portion of the EE-level state when either the corresponding credit transfer fails or expires. It is stored in the local shard state, but in the portion that is reserved for the sender shard. The step 4 collects all the portions and processes the user-level reverts.
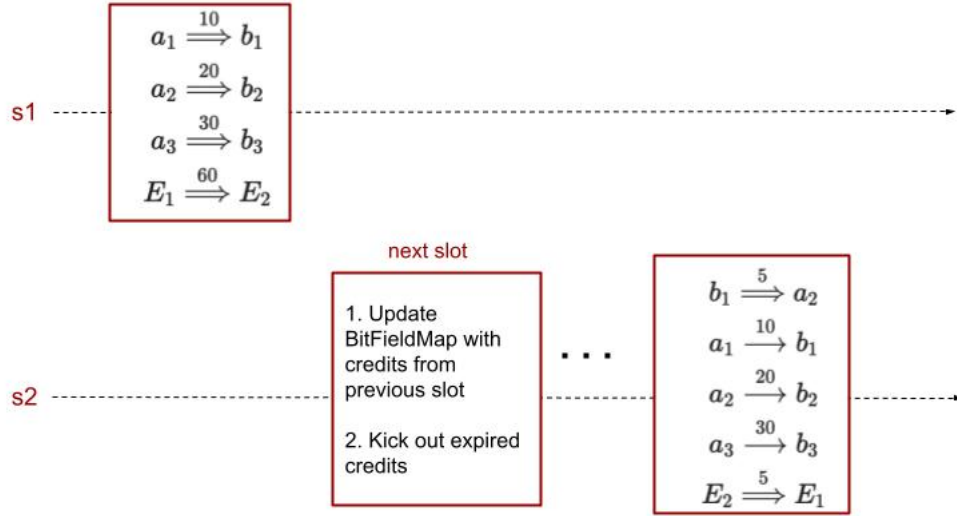
Note that the EE-level revert happens on the recipient shard in the same block where the credit transfer fails or expires. However, the user-level revert happens in the very next slot on the sender shard. This technique pushes the revert to the EE host functions instead of treating them as separate transactions. This avoids complex issues like revert timeouts and revert gas pricing.

Benefits - No locking / blocking. - No constraint on the block proposer to pick specific transactions or to order them. - Atomicity

Demerits In every block, a BP has to get the outstanding credits and reverts from every other shard. This is inherited from the netted balance approach, where a BP requires the netted balances from all shards. However, it is restricted to the sender EE's that are derived from the user-level transactions included in the block. The problem is aggravated here, because we would need to request from all EE's, even for the EE's not touched in this block.
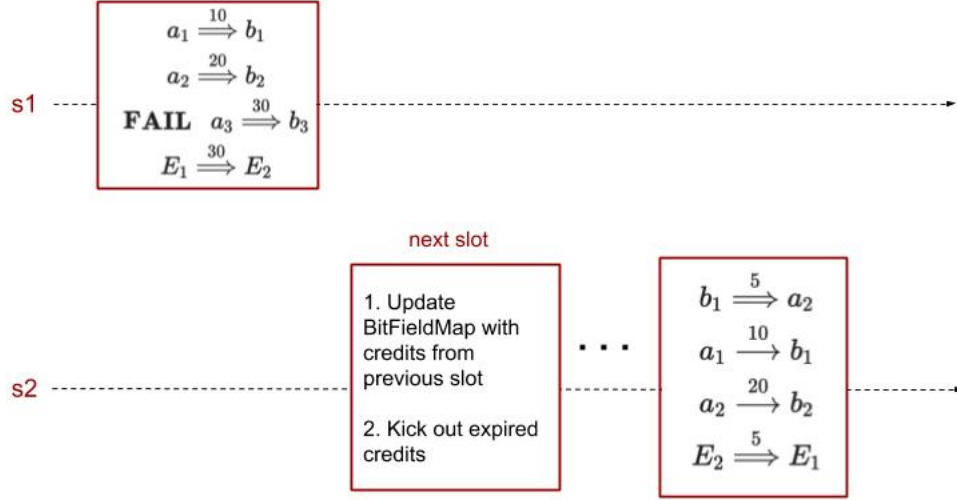
4 Scenarios

4.1 Optimistic case

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

s1

next slot

1. Update BitFieldMap with credits from previous slot

2. Kick out expired credits

. . .

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

s2

4.2 When Debit fails

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

s1

next slot

process reverts

next slot

1. Update BitFieldMap with credits from previous slot

2. Kick out expired credits

. . .

$$b_1 \overset{5}{\Longrightarrow} a_2$$
$$a_1 \overset{10}{\longrightarrow} b_1$$
$$a_2 \overset{20}{\longrightarrow} b_2$$
$$\textbf{FAIL} \quad a_3 \overset{30}{\longrightarrow} b_3$$
$$E_2 \overset{35}{\Longrightarrow} E_1$$

s2
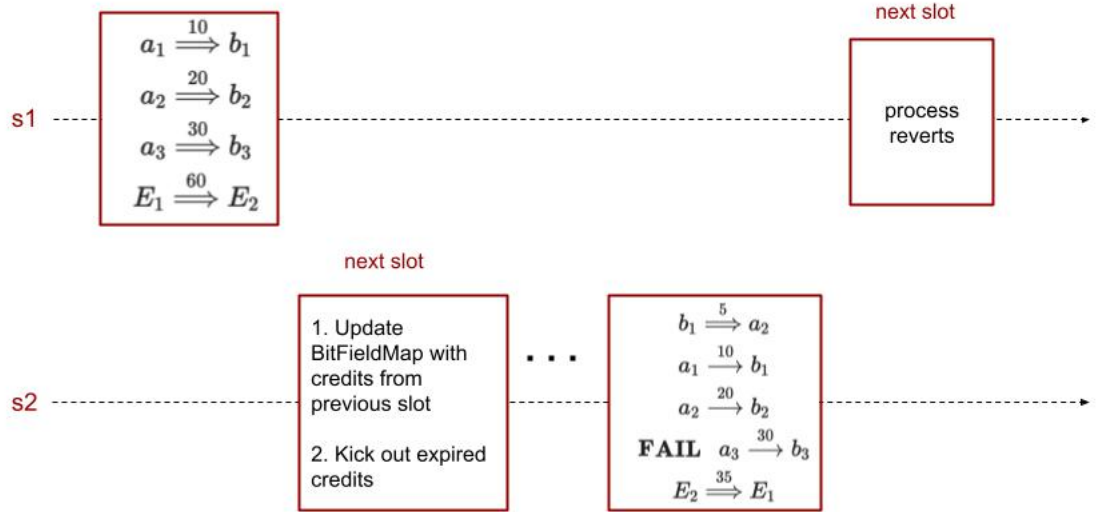
4.3   When Credit fails

Future work * Optimise the space requirement for storing outstanding credits and outstanding reverts. * Explore caching for optimising the reads of part-States of every EE of every other shard in every block (related to the above mentioned demerit).

5   Threat Analysis of a Byzantine Block Proposer

Talking with Roberto Saltini (PegaSys, ConsenSys) @saltiniroberto revealed many scenarios, especially the case when a Block Proposer (BP) is Byzantine. A Byzantine Block Proposer (BBP) might choose to deviate from the above algorithm. It becomes clear from the following that the protocol withstands such a BBP.

Checks done by a Validator / Attester Verify that the received part states from other shards for all EEs are correct. Verify that the data structure Bit-FieldMap is populated with the impending credits for this shard. Verify that the impending reverts are processed, meaning the sender users are credited with the transfer amount. Verify that correct ToCredit System Events are emitted for included and successful cross-shard debit transfer transactions. Verify that correct outgoing credit transfers are written to the appropriate shared state. Verify that the corresponding bit is set when a cross-shard credit transfer happens successfully. Verify that a correct revert transfer is placed for a failed cross-shard credit transfer transaction. Check that the correct amount is transferred at the

EE-level. Because a validator / an attester has access to the current shard state, (s)he can verify points: 2, 3, 5, 6, 7, 8. An attester is also given with the part-States from other shards along with their Merkle Proofs and (s)he has access to crosslinks form the Beacon block. So, (s)he can check points 1 and 8, that is, verify part balances, impending credits and impending reverts. Also because (s)he has access to all the transaction receipts of the transactions included in the block, (s)he can check point 4.

So, if a BBP chooses to

show no or false part EE-balances or set of impending credits or set of reverts, or not update or wrongly update BitFieldMap with impending credits, or not process or wrongly process impending reverts, or not emit or emit with incorrect data the ToCredit System Event not $SetBit(t_i)$, or not include a revert for a failed credit transaction, or not affect appropriate EE-level transfer, his / her block will be invalidated by the attesters, assuming that the two-thirds of the attesters are honest.

Also, in the discussion, @saltiniroberto pointed out that BitFieldMap is more of a set rather than a map to bits. It can be replaced by a set, where BitCheck checks membership and SetBit removes the element from the set.

6 Conclusion

References

[1] Vitalik Buterin. An even simpler meta-execution environment for eth. https://ethresear.ch/t/ an-even-simpler-meta-execution-environment-for-eth/6704, December 2019.

[2] Peter Robinson. Atomic cross shard function calls using system events, live parameter checking, and contract locking. https://ethresear.ch/t/ atomic-cross-shard-function-calls-using-system-events-live-parameter-checking-contract-l 7114, March 2020.