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 specificially 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 use *netted* shard states as channels to communicate pending credits and pending reverts. We discuss various scenarios of debit failures and credit failures, and show our approach ensures atomicity even in the presence of a Byzantine Block proposer.

The benefits of our approach are that we do not use any locks nor impose any constraints on the Block Proposer to select specific transactions. However we inherit the limitation of an expensive operation from the netted-balance approach of querying partial states from all other shards.

Keywords: cross-shard, Ethereum 2, atomicity, netted-balance

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. We have published this approach at [2].

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 [3]. 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).

We use **ToCredit**(a, x, b) to denote a system event with sender details of user a, the transfer amount x, and the receipient details of the user b, and elide the block number and the index number when they are obvious from the context.

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.

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:

- *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.

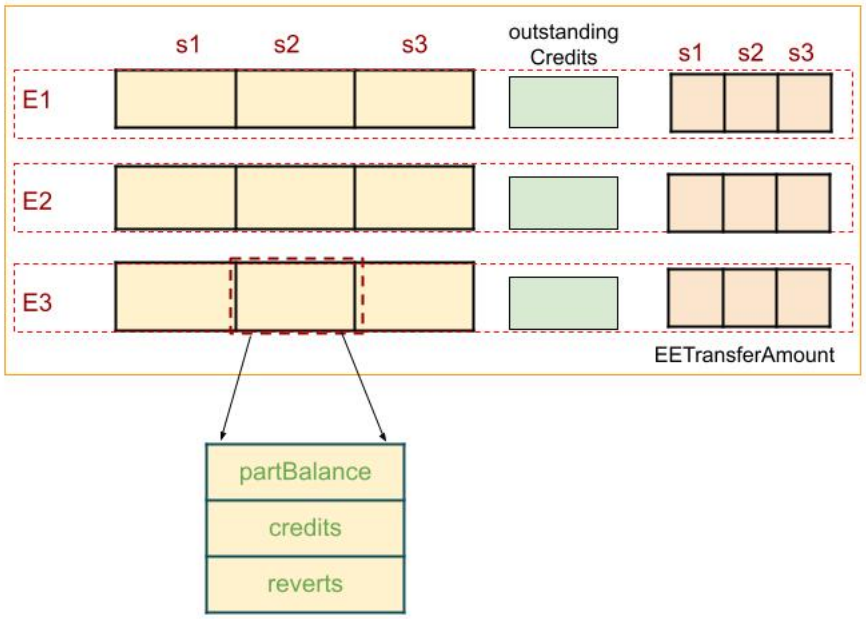

Figure 1: Shard State

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:

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

The $k-1$ in the equation indicates that the corresponding debit transactions were processed at $k-1$ block.

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 contains the $EETransferAmount$ that bookkeeps the net amount to be transferred at EE-level.

For every user of every EE, the shard state also stores the user's balance. We use $s.userBalance[E, a]$ to denote the balance of user a of EE E on shard s. Because the primary purpose of the shards is to divide the content, the users' balances are not distributed. Users are typically assigned a shard.

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. As previously mentioned, 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 separate transactions but as enshrined execution processing.

Without loss of generality, 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 $s_i.partState$s, $1 \leq i \leq n$, are correct using Merkle Proofs and crosslinks. Compute the real balance of E_i on s_1 using

   $$realBalance[s_1, E_i] = \sum_n s_n.partState[s_1, E_i].balance.$$

   For every shard s and every EE E, set $s_1.EETransferAmount[s, E]$ to 0.

2. **Preprocess pending credits**

   - Add entries to $s_1.outstandingCredits$ $[s', E', (k-1)] \mapsto \bigcup_n s_n.partState[s_1, E_i].credits$

- Kick out expired credits from $s_1.outstandingCredits$. If there is an entry with $[s', E', k']$ such that $k' + timeOut == k$ then do:
  - $s_1.partState[s', E'].reverts := s.outstandingCredits[s', E', k']$.
  - Delete the entry with $[s', E', k']$.
  - $s_1.EETransferAmount[s', E'] := \sum_r x_r$ where $r \in s_1.partState[s', E'].reverts$ and x_r denotes r's transfer amount.

3. **Process user-level reverts**. For every $r \in s_1.partState[s_1, E_i].reverts$, $s_1.userBalance[E_i, sender(r)] += x_r$, where x_r denotes r's transfer amount. After this operation, $s_1.partState[s_1, E_i].reverts$ is set to $\emptyset$.

4. **Process transactions.** For every pair (s_2, E_j)

   (a) Select cross-shard transactions $t_1, \ldots, t_m$ between (s_1, E_i) and (s_2, E_j) to be included in the block. It can be a new transaction from the transaction pool, or a credit transaction from $s_1.outstandingCredits[s_2, E_j]$.

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

   - If t_n is a cross-shard debit transaction of the form $a_n \xRightarrow{x_n} b_n$ and $realBalance(s_1, E_i) > s_1.EETransferAmount(s_2, E_j) + x_n$
     - include t_n to the block,
     - if t_n executes successfully
       - $s_1.userBalance(E_i, a_n)$ -= x_n (implied with successful execution of t_n)
       - $s_1.EETransferAmount[s_2, E_j] += x_n$
       - emit **ToCredit**(a_n, x_n, b_n) System Event
       - $s_1.partState[s_2, E_j].credits \cup = \{a_n \xrightarrow{x_n} b_n\}$
   - Else if t_i is a cross-shard credit transaction of the form $b_n \xrightarrow{x_n} a_n$ AND Merkle Proof check of the included **ToCredit** System Event passes AND $realBalance(s_1, E_i) > s1.EETransferAmount(s_2, E_j) + x_n$
     - include t_n to the block
     - Remove t_n from $s_1.outstandingCredits[s_2, E_j, k']$ where the block number k' is derived from the included **ToCredit** System Event.
     - if t_n executes successfully then $s_1.usersBalance[E_i, a_n] += x_n$ (implied by the successful execution of t_n)
     - if it fails
       $s_1.partState[s_2, E_j].reverts \cup = \{(b_n, x_n, a_n)\}$
       $s_1.EETransferAmount[s_2, E_j] += x_n$.

   (c) Process EE-level transfer
   $s_1.partState[s_1, E_i].balance- = s1.EETransferAmount[s_2, E_j]$,
   $s_1.partState[s_2, E_j].balance+ = s1.EETransferAmount[s_2, E_j]$.

3.5 Features of the algorithm

Let us visit some of the notable features of the above algorithm.

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 an expired credit transfer. The EE-level revert happens in the same slot as the failed / expired credit transfer. 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.
3. Transaction identifiers need to be unique inside the time-out blocks window.
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. We feel this is the best state to be in 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.
6. A time-out is required to kick out long pending credit transfers. The second bullet of step 3 describes this procedure. The idea is to move out the expired user-level credit transfers, convert them into user-level reverts on the sender shard, thus achieving a fixed size for $outstandingCredits$ datastructure.
7. No locking / blocking.
8. No constraint on the block proposer to pick specific transactions or to order them.
9. **Main goal.** Atomicity of a cross-shard transfer.
10. **Demerit.** 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, in the EE-level netted-balance approach the querying 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 need to query from all EE's, even for the EE's not touched in this block.

4 Scenarios

In this section, we apply the algorithm from Section 3.4 in different kinds of representative scenarios and show how atomicity is preserved.

Assume that a_1, a_2, a_3 are users on (s_1, E_1) and b_1, b_2, b_3 are users on (s_2, E_2).

4.1 Optimistic case

Let us look at the optimistic case, where everything happens as expected. Suppose we include three cross-shard transactions as shown in Figure 2 in a block on the shard s_1. In the very next block on s_2, the $s_2.outstandingCredits$ is updated with 3 pending credit transactions. In the same block or subsequently in some block on s_2 these credit transactions are processed.

Note that the EE-level transfers are complete at the s_1's block itself. The s_2's block could process the credit transactions all in the same block or in different blocks. The blocks processing the credit transfers could include new cross-shard transactions as shown in Figrue 2 or other intra-shard transactions.

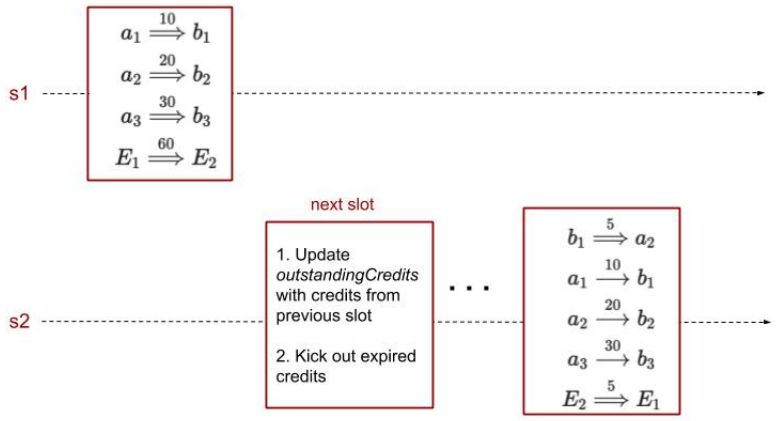

Figure 2: Optimistic case

4.2 When credit fails

Let us look at the case when the processing of a credit transaction fails. Consider a slight variation from the above scenario, where the credit transaction $a_3 \xrightarrow{30} b_3$ fails or is expired. Then in the same block the EE-level revert happens (with E_1 getting its 30 ETH back), and finally in the very next block on s_1 the user-level reverts are processed.

4.3 When debit fails

Consider the case when the initial debit transaction fails. This is a very simple case, as nothing needs to be done. Simply this transaction has to be resubmitted as in the case of Ethereum 1.

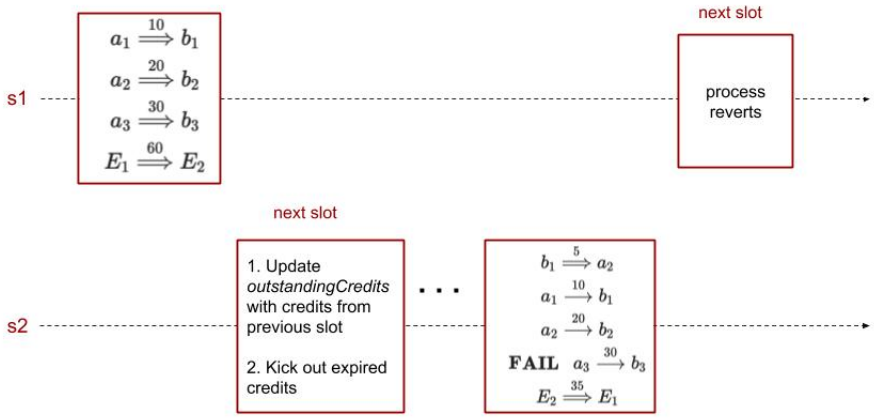

Figure 3: When credit fails

5 Threat Analysis of a Byzantine Block Proposer

Consider the case of a Byzantine Block Proposer (BBP). A BBP might choose to deviate from the above algorithm. It becomes clear from the following that the protocol withstands such a BBP.

Let us recollect the checks done by a validator / attester.

1. Verify that the received part states from other shards for all EEs are correct.
2. Verify that the data structure *outstandingCredits* is populated with the impending credits for this shard.
3. Verify that the impending reverts are processed, meaning the sender users are credited with the transfer amount.
4. Verify that correct **ToCredit**System Events are emitted for included and successful cross-shard debit transfer transactions.
5. Verify that correct outgoing credit transfers are written to the appropriate part state.
6. Verify that the outstanding credit transfer is removed when a cross-shard credit transfer happens successfully.
7. Verify that a correct revert transfer is placed for a failed cross-shard credit transfer transaction.
8. Check that the correct amount is transferred at the EE-level.

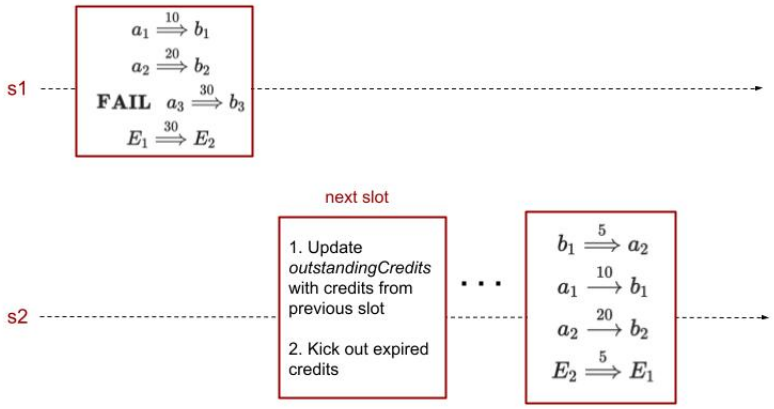

Figure 4: When debit fails

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 *partState*s 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 *outstandingCredits* with impending credits, or
- not process or wrongly process impending reverts, or
- not emit or emit with incorrect data the **ToCredit**System Event
- 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.

6 Conclusion

We presented an atomic cross-shard cross-EE user-level value transfer algorithm for Ethereum 2 in presence of a Byzantine Block Proposer on top of an existing netted-balance approach for EE-level transfers.

As part of the future work, we plan to optimise the space requirements for storing outstanding credits and outstanding reverts, and explore caching for optimising the reads of partStates of every EE of every other shard in every block (related to the above mentioned demerit).

Acknowledgements

We thank Roberto Saltini, Peter Robinson, Nicholas Liochon and David Hyland-Wood for all the insightful discussions.

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] Raghavendra Ramesh. Atomic asynchronous cross-shard user-level eth transfers over netted ee transfers. `https://ethresear.ch/t/atomic-asynchronous-cross-shard-user-level-eth-transfers-over-netted-ee-transfers/7277`, April 2020.

[3] 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-locking/7114`, March 2020.