This proposal follows three previous proposals for cross-shard transfers:

1. Receipts with accumulators for “already processed” receipts
2. Nonce-based queues
3. Netted balance maps

The scheme here is based on “EthTransfer” state objects, and is most similar Netted balance maps. Netted balance maps is the only one of the three that tracks transfers using state proofs (both nonce-based queues and receipts with accumulators are based on receipt proofs; an explanation of the difference between state proofs and receipt proofs may follow later).

To better compare this scheme to netted balance maps, diagrams of netted balance maps are included.

This scheme, like the others, completes a transfer in two steps: one to initiate the transfer on the sending shard, and a second to process the transfer on the receiving shard. The second step must wait one slot, because the transfer on the sending shard must be cross-linked into a beacon block for the proof to be valid on the receiving shard.

[

](https://storage.googleapis.com/ethereum-hackmd/upload_24495c44f64780042b0830cebb1b8238.png)

The key thing about all the mechanisms is how they do replay protection. This one achieves replay protection by requiring that the transfer object on Alice’s account be deleted before Bob can delete the object on his account. So the object on Alice’s account is deleted first (by passing a proof that the object is on Bob’s account, which shows that the transfer was completed). And the object on Bob’s account is deleted second (by passing a proof of deletion / exclusion on Alice’s account). This prevents Bob from “replaying” the transfer and crediting his account with the same ETH multiple times.

These two steps to “delete on Alice’s first, delete on Bob’s second” are garbage collection steps (note that if the sender goes into a state where she waits for a success or failure, then the third step is not strictly garbage collection).

Arguably these garbage collection steps shouldn’t be counted as necessary to “complete” a transfer, since the ETH can be used by the recipient immediately after step 2. And the garbage collection can be batched with other sends that come later, or other txn’s, so it doesn’t seem like significant overhead.

[

](https://storage.googleapis.com/ethereum-hackmd/upload_ee9ee70ea2bc66e50a671adb29d3df8e.png)

Compared to Netting

How do those EthTransfer objects compare to Netted balance maps? Looking at the “columns” in a balance map, we can see that they’re in a contract’s state. Already there is some similarity to a contract with a set EthTransfer objects.

[

](https://storage.googleapis.com/ethereum-hackmd/upload_9dde7db0efc770c42f728e60d4031abe.png)

Where the schemes differ is that Netted balance maps have two types of transfers: “same-EE/cross-shard” versus “cross-EE/same-shard”.

In the netted balance scheme, the protocol requires “column proofs” for both types, so the distinction seems a bit artificial.

[

](https://storage.googleapis.com/ethereum-hackmd/upload_17b8de966bfc0da8c0ddccf8808a1392.png)

[

](https://storage.googleapis.com/ethereum-hackmd/upload_cb6ebbe25fcaa6938db5408e0615615c.png)

What the two types emphasize is that with “cross-EE/same-shard” transfers, the receiving EE is on the same shard and so its prestate is known regardless of whether other txn’s have been executed or not (if it was on a different shard, it would only have the same prestate as was cross-linked in the previous beacon block at the very beginning of the shard block, before any txn’s are executed). This means it would be straightforward to execute the code of the receiving EE and check for errors (or check if the EE wants to reject a transfer, for example) when processing an ETH transfer, in the same way that when ETH value is transferred using CALL

on eth1, the code of the called contract is executed.

In contrast, the “same-EE/cross-shard” transfer type does not support any such code execution or error-handling. Instead, it is expected that the EE somehow anticipate any potential cross-shard errors or otherwise implement its own cross-shard

error-handling without the support of the protocol.

From EthTransfer objects to Netted balance maps

Let's better understand the two schemes by tweaking each until it becomes the other.

Starting from EthTransfer objects, first we add a requirement where a cross-shard transfer must go to the same EE (only same-EE/cross-shard, no cross-EE/cross-shard transfers). And we add a requirement where multiple transfers to the same shard get summed in an existing entry in the "cross-shard balance map". Then we get the Netting scheme.

Now starting from the Netting scheme. To the existing two types of transfers (same-EE/cross-shard and cross-EE/same-shard) we add a third type: cross-EE/cross-shard. This third type isn't really that big of a change. Gven that both types already require the "column proofs" that this new type would require as well, it isn't much different mechanically. This third type would add another dimension to the balance map. Instead of a 2d balance map that forms a [from_shard, to_shard]

NxN matrix where N is the number of shards (i.e. one row of [shard_A, shard_B, shard_C]

for each shard) and the EE is fixed, we'd get 3 dimensions [from_shard, to_shard, to_EE]

(from_EE

is fixed, but not to_EE

). But it's easier to just imagine a list of "outgoing transfers" (or EthTransfer

objects) to different shards and different EE's, rather than a 3-dimensional balance map.

The main concern would be if we assume that these transient "transfer objects" in the state are very cheap to create, e.g. if they could be created with as little as 1 wei (and no incentive for garbage collection), then they might bloat the state. The netting scheme enforces a limit on the number of "transfer objects" by netting them into balance maps and by simply not allowing them to be created for cross-shard transfers. But this also makes it impossible to do cross-shard transfers with error-handling (or rather, it kicks cross-shard error-handling up to the EE, which would have put its own transient objects in the state in order to handle errors).

Note that the EthTransferObj scheme does have some extra garbage collection steps, but the Netting scheme would require something similar in order to "zero out" the negative entries (the Netting post leaves that issue open).

Cross-shard vs Same-shard vs Transparent shards

An important consequence of going with a scheme that has two distinct transfer mechanisms for cross-shard versus same-shard is that this would arise in the protocol as two different functions: e.g. cross_ee_call

and cross_shard_ee_call

. If eth2/phase2 only has one EE this won't be of any consequence (every protocol transfer will be cross_shard_ee_call

, and there would be no way to use cross_ee_call

; In this case, we should be concerned what functions the one EE will expose to "child contracts", or "user-deployed contracts", or whatever we call them).

But suppose there are multiple EE's (or top-level contracts, or top-level dapp developer deployed contracts, or whatever we call them). Then every contract would need logic to check whether a destination contract's address is on the same shard or on a different shard, in order to determine which method to use. Otherwise, if a contract ever relocates to a different shard (as long as the protocol is still being designed, we don't know how or when such an event might happen), then contracts that call it would break because they'd be calling it with the wrong method.

On the other hand, if the protocol provides one unified function for doing ETH transfers (regardless of whether the destination contract is on the same shard or on a different shard), not only is it an improvement in usability and developer experience (transparent shards!), but it also leaves open the possibility for the protocol to move contracts around (automatic load-balancing?) without breaking deployed contract code.