Co-written by Alex Beregszaszi (@axic) and Sina Mahmoodi (@sinamahmoodi), with the involvement of other Ewasm team members. Thanks to Guillaume Ballet (@gballet) and Danny Ryan (@djrtwo) for valuable feedback.

This post will provide a high-level overview. For a more detailed semi-specified version refer to therepository, which additionally includes example contracts.

This post goes into one variant of theEth1x64 experiment. To recap, Eth1x64 builds on Phase 1.5 and explores the "put Eth1 on every shard" idea. The main motivations of this thought experiment are to narrow down the Phase 2 design space initially and focus on cross-shard mechanics, encourage Eth1 developers to experiment with shard-aware EVM contracts and take the learnings into the final WebAssembly-based Phase 2 design.

This proposal tries to be minimally invasive to assess the feasibility of actually including these changes into Eth1. Partly because in any Phase 2 roadmap where the Eth1 mainnet becomes shard 0 and communication between shard 0 and other shards is desired, changes similar to the following will be required. These changes might also turn out to be useful for transfering Ether between the validator accounts on the beacon chain and shard accounts.

ToC

Bird's-eye view

The main requirement for any phase 2 proposal is to support the secure transfer of Ether between shards. In this variant, we adopt the receipt model in which the receipts include a payload field. The result is a set of changes intended for EVM to support async messages (which can have Ether value attached) from contracts to contracts (or EoAs) on other shards. The general flow looks like the following:

- A contract on the sending shard creates a receipt via the XMSG

opcode. This receipt specifies the recipient, value and payload.

- A contract on the receiving shard receives a message via the XCLAIM

opcode, providing a proof to the receipt. The receipt is then marked as claimed to prevent double-spend.

Async messages are unidirectional, i.e. the outcome from processing the message at the receiver's end will not be automatically relayed to the sender (unlike the CALL

opcode). However the receiving contract can optionally initiate a new async message directed at the sender (seeRejection and callback). The payload field further makes it easier to reward relayers for executing the second leg of the message if the user doesn't have funds on the receiving shard (see Gas payer). Additionally, contract level experimentation with alternative cross-shard mechanisms is assisted by two opcodes, namely BEACONBLOCKHASH

and SHARDID

.

Eth1 block changes

Initiating a message requires a new receipt to be created. Message receipts created during the execution of each block are accumulated and commited to in the block header. Note this will be different from the existing receipt trie, which is filled by LOG

, to protect against contracts crafting similar looking receipts with the goal of minting Ether.

Moreover, shard states keep track of receipts claimed from every other shard in a binary trie. The trie key is computed by hash(concat(sourceShardId, slot))

. Leafs in the trie encode which receipt indices produced in that shard slot have already been claimed (e.g. bitmask 0b0010

if only receipt index 2 out of 4 receipts has been claimed). In future work we will extend this trie to incorporate pagination with the goal of keeping the trie from growing indefinitely. We're assuming the tree implements the following interface:

interface ClaimedReceipts { set(sourceShardId: number, slot: number, index: number): void has(sourceShardId: number, slot: number, index: number): boolean }

In summary, each Eth1 block header will include these two additional fields:

- xmsgReceipts

: similar to but distinct from the existing receiptsTrie

. This tree commits to the receipts generated within each block separately, i.e. it doesn't grow indefinitely.

- claimedReceipts

: root of the claimed receipts tree is committed to in the header to enable Eth1 miners to generate stateless witnesses

Sending a message

To initiate a message, users (EOAs) will have to invoke a (proxy) contract that executes the XMSG

opcode (an alternative to proxy contracts is discussed here). After making sure the sender has enough balance and reducing their balance by the specified value, EVM will generate a receipt with the following structure and push it to the list of xmsgReceipts

:

MessageReceipt { // Sender fromShardId: number // [0, 63] fromAddress: Address // Receiver toShardId: number // [0, 63] toAddress: Address // Can be zero value: Wei // Can be empty payload: Bytes // Slot at which receipt was created slot: number // Index for sourceShard -> destShard messages in slot // First message in each slot has index 0. index: number }

Note that during the execution of a block, the client must keep track how many cross-shard messages it has issued to every other shard to be able to fill in receipt.index

.

The aforementioned proxy contract would resemble the following snippet (Yul syntax):

// Treating the input to the proxy as shard_id, destination, payload. let shard_id := calldataload(0) let address := calldataload(1)

// Pass on any Ether sent with the transaction. let value := callvalue()

// Copy the payload from the calldata to memory offset 0. let payload_offset := 0 let payload_size := sub(calldatasize(), 64) calldatacopy(payload_offset, 64, payload_size)

// XMSG returns index, ignoring it here pop(xmsg(shard_id, address, value, payload_offset, payload_size))

Gas cost analysis

Apart from the typical condition checks, XMSG performs two operations. Creating the receipt, which can be priced similar to the LOG

opcode. It also has to increment the in-memory value indices[receipt.toShardId]

. We therefore expect the cost to be similar to but higher than LOG

.

Receiving a message

After the block from the sending shard has been crosslinked into the beacon chain, the user can invoke a proxy contract on the receiving shard which executes the XCLAIM

opcode:

/ ** Simplest proxy contract with no error handling logic, etc.* /

// Treating the entire input to the proxy as the proof. calldatacopy(0, 0, calldatasize()) xclaim(0, calldatasize())

The opcode calls the target of the receipt with the given payload and value. It requires a receipt proof as the only argument and has the following flow:

- Check well-formedness of the proof (including the beacon block hash is valid, there is a single receipt in the proof, the receipt.toShardId

points to this shard), revert otherwise

- If receipt is already claimed (claimedReceipts.has(receipt.fromShardId, receipt.slot, receipt.index)

) return 2

- Mark receipt as claimed (claimedReceipts.set(receipt.fromShardId, receipt.slot, receipt.index)

)

- Call receipt.toAddress

with receipt.payload

and receipt.value

and send remainder of gas (following semantics of CALL

here), and return the call’s return value (0 on success and 1 on failure)

Since we introduce CALL

semantics here, we also must be clear about certain properties of the call frame:

- ORIGIN

and CALLER

retain their original meaning, that is the transaction originator and the caller (which in our example is the “proxy contract”)

- CALLVALUE

returns the value of receipt.value

- CALLDATASIZE

/CALLDATACOPY

/CALLDATALOAD

operate on receipt.payload

- We introduce two new opcodes, XSHARDID

and XSENDER

, which return the values of receipt.fromShardId and receipt.fromAddress

, respectively

Gas cost analysis

Processing the message is more expensive than sending. It involves and should account for the following operations:

- Receipt proof verification: (receiptProofSize / 32) * keccakCost

.

- Double-spend check: Look-up value in claimedReceipts

and modify the same value. Similar to SLOAD

followed by a SSTORE

. We're assuming in Eth1x costs for SLOAD

and SSTORE

depend on trie depth and include bandwidth overhead caused by the witness.

- Setting up a CALL

with values from the receipt.

The proofs for the claimed receipts accumulator are batched within the block, similar to how accounts and storage slots are batched in Eth1x. The proof for the receipt itself is submitted by each user in the txdata and is not batched. If we extend XCLAIM

to support multiple proofs simultaneously, then proofs can be batched within the transaction. There are multiple ways to extend XCLAIM

to achieve this.

Gas payer

We assume the end user has enough balance on both the sending and receiving shard to pay for gas. This assumption might not be realistic. This can be worked around at the contract level by allowing third-party entities to relay transactions in expectation of a reward. To achieve this, the sender targets the message to a meta-transaction-like contract on the receiving shard and encodes a reward amount plus the address of the intended recipient in the payload. When XCLAIM

is executed, the contract transfers the reward amount to the tx sender, and the rest to the actual recipient.

Rejection and callback

There are 3 possibilities when claiming a message (assuming the name "outer frame" for the call frame where XCLAIM

is executed and "inner frame" for the child call frame):

- If the outer frame (or any of its parent frames) reverts, the message will not be marked as claimed and won't have any effect
- If the outer frame and the inner frame succeed, the message has had the intended effect and will be marked as claimed
- If the inner frame fails but the outer frame succeeds, the message will be marked as claimed but it won't have any effect, i.e. the Ether value will be lost.

We assume that the contracts expecting cross-shard messages handle failure gracefully and instead of reverting, send a message (with the Ether value attached) back to the receipt sender via XMSG

. The message sender's address is available for this purpose in the inner frame via XSENDER

. If nevertheless the inner frame fails (e.g. due to OOG), the outer frame has the option of reverting the transaction and giving the submitter another chance to submit (e.g. with a higher gas limit). Alternatively the proposal can be extended to allow the outer frame to read the message sender's address for an extra layer of error handling.

Receipt proof

The receipt proof (diagram) starts with the hash of a recent beacon block, which in turn includes a list of all the latest cross-linked shard transitions. It then expands the transition of the sender's shard into its state root and receipt tree's root. It finally expands the receipt tree root to the leaf. The exact structure of the proof depends on two factors. How many beacon block headers we can assume everybody to store and how old is the beacon block against which the proof was created.

Atomic operations

As Vitalik explains:

In general, workflows of the form "do something here that will soon have an effect over there" will be easy; workflows of the form "do something here, then do something over there, then do more things here based on the results of things over there, all atomically within a single transaction" will not be supported.

He then goes on to bring examples of DeFi contracts and how they can operate in a multi-sharded system. As an example, to exchange tokens, if we assume the token contract lives on all relevant shards the user can:

1. transfer to-be-exchanged tokens to dex's shard,
2. perform exchange,
3. optionally move new tokens to "home" shard.

To solve the train-and-hotel problem, contracts that wish to interact could for example implement yanking, or implement a compatible locking mechanism.

Contract-level yanking can be built on top of the built-in messaging primitive. The yankee encodes its storage in a bytearray, sends it via XMSG

to a factory contract on the destination shard which re-constructs a new instance with the same storage. At this point the atomic operation can take place on the same shard, and if desired the result can be moved back to yankee's origin shard.

Examples

For a token example in Solidity see here.