THIS IS A WORK IN PROGRESS!

The goal of this post is to provide a rough outline of what phase 2 might look like, to help make the discussion about state and execution more concrete as well as to give us an idea of the level and types of complexity that would be involved in implementing it. This section focuses on withdrawals from the beacon chain and cross-shard calls (which use the same mechanism); rent is unspecified but note that a hibernation can be implemented as a forced cross-shard call to the same shard.

Topics covered:

- Addresses
- Cross-shard receipt creation and inclusion
- Withdrawing from the beacon chain

Addresses

The Address

type is a bytes32, with bytes used as follows:

[1 byte: version number] [2 bytes: shard] [29 bytes: address in shard]

There are many choices to make for address encoding when presented to users; one simple one is: version number - shard ID as number - address as mixed hex

, eg. 0-572-0DF5283B84D83637e3E6AAC675cE922d558b296e8B11c43881b3f91484

, but there are many options. Note that implementations may choose to treat an address as a struct:

{ "version": "uint8", "shard": "uint16", "address_in_shard": "bytes29" }

Because SSZ encoding for basic tuples is just concatenation, this is equivalent to simply treating Address

as a bytes32 with the interpretation given above.

Cross shard receipts

A CrossShardReceipt

object, which contains the following fields:

{ "target": Address, "wei_amount": uint128, "index": uint64, "slot": SlotNumber, "calldata": bytes, "init_data": InitiationData }

InitiationData

is the following:

{ 'salt': bytes32, 'code': bytes, 'storage': bytes, }

Note that in each shard there are a few “special addresses” relevant at this point:

- CROSS_SHARD_MESSAGE_SENDER

: 0x10 - has two functions: * Regular send: accepts as argument (i) target: Address

, (ii) calldata

. Creates a CrossShardReceipt

with arguments: target=target

, wei_amount=msg.value

, index=self.storage.next_indices[target.shard]

(incrementing self.storage.next_indices[target.shard] += 1

after doing this), slot=current_slot

, calldata=calldata

, init_data=None

.

- Yank: accepts as argument target_shard: ShardNumber

. Creates a CrossShardReceipt

with target=Address(0, target_shard, msg.sender)

, wei_amount=get_balance(msg.sender)

, index=self.storage.next_indices[target.shard]

(incrementing self.storage.next_indices[target.shard] += 1

after doing this), slot=current_slot

, calldata=''

,init_data=InitiationData(0, get_code(msg.sender), get_storage(msg.sender))

. Deletes the existing msg.sender

account.

- Regular send: accepts as argument (i) target: Address

, (ii) calldata

. Creates a CrossShardReceipt

with arguments: target=target

, wei_amount=msg.value

, index=self.storage.next_indices[target.shard]

(incrementing self.storage.next_indices[target.shard] += 1

after doing this), slot=current_slot

, calldata=calldata

, init_data=None

.

- Yank: accepts as argument target_shard: ShardNumber

. Creates a CrossShardReceipt

with target=Address(0, target_shard, msg.sender)

, wei_amount=get_balance(msg.sender)

, index=self.storage.next_indices[target.shard]

(incrementing self.storage.next_indices[target.shard] += 1

after doing this), slot=current_slot

, calldata=''

,init_data=InitiationData(0, get_code(msg.sender), get_storage(msg.sender))

. Deletes the existing msg.sender

account.

- CROSS_SHARD_MESSAGE_RECEIVER

: accepts as argument a CrossShardReceipt

, a source_shard

and a Merkle branch. Checks that the Merkle branch is valid and is rooted in a hash that the shard knows is legitimate for the source_shard

, and checks that self.current_used_indices[source_shard][receipt.index] == 0

. If the slot

is too old, requires additional proofs to check that the proof was not already spent (see Cross-shard receipt and hibernation/waking anti-double-spending foe details). If checks pass, then executes the call specified; if init_data

is nonempty and the target does not exist, instantiates it with the given code and storage.

Withdrawal from the beacon chain

A validator that is in the withdrawable

state has the ability to withdraw. The block has a withdrawals

field that contains a list of all withdrawals that happen in that block, and each withdrawal is a standardized CrossShardReceipt

obect.

A CrossShardReceipt

created by the beacon chain shard will always have the following arguments:

{ "address_to": Address(0, dest_shard, hash(salt + hash(init_storage) + hash(code))[3:]), "wei_amount": deposit_value, "index": state.next_indices[dest_shard], "slot": state.slot, "calldata": "", "init_data": InitiationData("salt": 0, "code": init_code, "storage": init_storage) }

Where dest_shard

, salt

, init_storage

, init_code

are all chosen by the withdrawing validator. state.next_indices[dest_shard]

is then incremented by 1. This receipt can then be processed by the CROSS_SHARD_MESSAGE_RECEIVER

contract just like any other cross-shard receipt.

Transactions

A transaction object is as follows:

{ "version": "uint8", "gas": "uint64", "gas_max_basefee": "uint64", "gas_tip": "uint64", "call": CrossShardReceipt }

Executing a transaction is simply processing the call, except a transaction (or generally, any call that it not directly an execution of an actual cross-shard receipt) can only create an account at some address if it has the salt

such that target = hash(salt, hash(code) + hash(storage))

. Note that a transaction simply specifies a call to an account; it's up to the account to implement all account security logic.

Not covered:

- Specific protocol changes to facilitate abstraction
- The exact mechanics of the fee market (see https://github.com/ethereum/EIPs/issues/1559 for a rough idea though)
- The mechanics of the rent mechanism