# Transactions

# Transaction Finality

Block finality refers to the concept that a proposed block is finalized by the blockchain and can not be reverted, or the cost of reversion is prohibitively high. Block finality is usually measured by how long it takes for the newly proposed blocks to be finalized. For example, in Ethereum, it takes more than six blocks or 1 minute to have reasonable confidence that the block is finalized and can not be reverted.

Thanks to the nature of Harmony's FBFT consensus, blocks may finalize as long as the 2/3 majority quorum is reached on the block. **On Harmony mainnet, it now takes 2 seconds to finalize a newly proposed block and the transactions inside.** Read more about our 2-second finality here:

[2 Seconds Block Time on Harmony Mainnet](https://medium.com/harmony-one/2-seconds-block-time-on-harmony-mainnet-aba78ee5643f)

[In the age of digital business and fast-paced economy, online applications need to provide users with the best experience to retain them…](https://medium.com/harmony-one/2-seconds-block-time-on-harmony-mainnet-aba78ee5643f)

[medium.com](https://medium.com/harmony-one/2-seconds-block-time-on-harmony-mainnet-aba78ee5643f)

# Transaction Fees

Harmony follows the same transaction fee model as Ethereum, where users pay a certain amount of tokens to get their transactions processed and included in the blockchain. Since Harmony is fully EVM-compatible, users can translate the fee model directly from Ethereum and apply it to Harmony. For example, a typical token transfer transaction costs 21000 gas. The gas price can be as low as 0.000000001 ONE (or 10 Gwei as in Ether) since Harmony have high TPS and the network is highly efficient and rarely clogged. This efficiency means a regular transfer cost of only about **0.000021 ONE**.In General, transactions in the Harmony network cost around $0.000001 gas fee.

The reason for Harmony to be able to afford such a low fee is twofold. First, Harmony is a Proof-of-Stake chain where the cost of running a node is much cheaper than PoW chains as no wasteful computation is needed. Second, Harmony is a highly scalable blockchain that currently provides thousands of transactions per second as throughput. This benefit means users don't need to bid with a high fee to get their transactions processed in time. We expect the low fee situation will stay as long as the Harmony network reaches full utilization. Even if that happens, we can solve the problem by extending the network with more shards to provide more transaction processing power.

# Cross-Shard Smart Contract Transaction

We will adopt the asynchronous cross-shard communication model just like we did for cross-shard ONE token transfer. Meaning any smart contract which needs to call another smart contract from another shard will have the sender contract transaction finalized first at initiating shard. Then a receipt will be created, including the call data, and finally, the receipt will be sent and processed at the destination shard.

A new Opcode "CALL2" will be added to the virtual machine used for a cross-shard smart contract call.

CALL2's input is similar to CALL but with another shardID parameter:

(caller ContractRef, addr common.Address, input []byte, gas uint64, value \*big.Int, shardID uint32)

Unlike CALL, which executes the contract code within the same shard, CALL2 won't run any code but only leads to creating a receipt after the transaction confirms successfully in this block. The receipt structure can be like this:

// CXReceipt represents a receipt for cross-shard transaction

type CXReceipt struct {

TxHash common.Hash // hash of the cross-shard transaction in source shard

Sender common.Address

From common.Address

To common.Address

ShardID uint32

ToShardID uint32

Input []byte

Amount \*big.Int

Gas uint64

}

Specifically:

* TxHash: the transaction hash of the original transaction sent from the actual user
* Sender: the address of the transaction sender
* From: the address of the caller smart contract which initiated the cross-shard call
* To: the address of the callee smart contract in another shard
* ShardID: the shardID of the caller smart contract
* ToShardID: the shardID of the caller smart contract
* Input: the input data for the smart contract call, exactly like the payload of a typical intra-shard transaction that calls a smart contract
* Amount: is the amount of ONE that is sent along with the call to the callee smart contract.
* Gas: the gas that's going to be paid to the destination shard.

Once the corresponding block of the receipt is finalized, proof data for the receipt will also be generated. This process is the same as cross-shard ONE token transfer:

// CXReceiptsProof carries the cross-shard receipts and merkle proof

type CXReceiptsProof struct {

Receipts CXReceipts

MerkleProof \*CXMerkleProof

Header \*block.Header

CommitSig []byte

CommitBitmap []byte

}

Once the receipt and proof are obtained, they will be sent from source shard to destination shard (CXReceipt.ToShardID) for processing.

The destination shard process the cross-shard smart contract receipt by:

1. Verify the receipt is valid using the proof data
2. Use the CXReceipt.To as the callee smart contract address and CXReceipt.Input data as the contract call data for contract execution. CXReceipt.Amount and CXReceipt.Gas is also used the same way as regular contract calls.
3. If this contract call produces additional cross-shard contract calls. Additional receipts will be generated, and the same cross-shard call process will happen.

### Design Questions:

1. What if the callee contract doesn't exist in the destination shard?
   1. We could simply abort the cross-shard call but credit the gas to the destination shard. This way, we don't need another cross-shard receipt to go back to the source shard to credit the gas.
   2. Or we could create a brand new contract. This requires the receipts to include the contract bytecode, which can be user-specified or the same as the caller contract's code (in the case of HRC20 contract replication across shards)
2. What if the gas sent in the receipt is not enough to pay for the transaction?
   1. Simply abort the contract call as a typical transaction "Out of Gas" situation.
3. In any failure cases of the cross-shard contract call, the sent token in CXReceipt.Amount should be credited to who?
   1. The sender's address in the destination shard will avoid sending another cross-shard receipt to credit the token back to the sender shard.