RaptorChain, a semi-asynchronous EVM chain!

Why?

As an EVM enthusiast, I rapidly used EVM chains, either as an user or as a builder.
However, something became obvious day by day: latency.
Basically, in order to send a transaction, a blockchain user has to broadcast it, then wait for a miner to include it into a block. The delay between broadcast and inclusion is called latency.
It also poses a censorship problem if a significant number of miners decide to block transactions from a specific address (the problem is even bigger with PoS chains).
And, even though the best EVM chains were already able to handle about a hundred transactions per second, this problem remained unsolved.

So why is RaptorChain different?

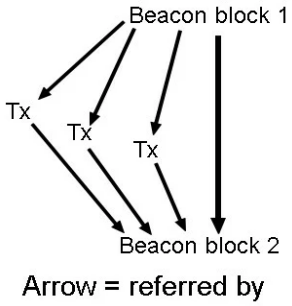
While Ethereum and its forks assume transactions have to execute sequentially in order to be valid, it turned out not to be necessary.
Basically, a transaction only needs the involved accounts' states to be accurate, while other accounts' state virtually doesn't matter.
It allows transactions to come in a different order AS LONG AS they don't affect the same accounts.
Implementing this results in a DAG-like data structure, using former states of involved accounts to determine whether a transaction is valid or not.

Summary

1 - The Consensus Engine

1.1 - The Beacon Chain

In early stages of development, it turned out that synchronizing a DAG-like structure was very messy. In fact, it required so many data transfers that production implementation was nearly impossible. And there comes the Beacon Chain.
Basically, when a transaction is sent, it has to include the last beacon block's hash. Then, after ensuring it was the last block, the transaction's hash will be added to the block's transaction list. This transaction history is then sorted to avoid messing with transaction order, hashed using keccak256, and the hash is referred to in the next block.



1.2 - Cross-chain implementation

Since Beacon blocks are lightweight, it enables very light clients to load them without loading full transaction history. For example, there can be a solidity light client on another EVM chain, loading and indexing RaptorChain beacon blocks.
This was leveraged by including cross-chain messages inside RaptorChain blocks, these cross-chain messages being fired by precompiled contracts on RaptorChain, and executed by light clients on receiving chains, allowing to send data from RaptorChain to other chains.
The other way (from other chains to RaptorChain), however, is processed differently, by getting RaptorChain nodes to directly query data from other chains' nodes.
This cross-chain implementation currently allows to bridge RPTR token across a variety of EVM chains including BNB Smart Chain (RPTR's origin chain), Polygon, RaptorChain (of course) and Fantom.

2 - The Execution Engine

2.1 - The EVM implementation

The EVM implementation allows running solidity smart contracts on RaptorChain. It uses the same opcodes as Ethereum, but with slightly different implementations to fit RaptorChain's specifications or bring additional features to the EVM.

2.2 - RaptorChain-specific precompiled contracts

In order to bring additional features, such as cross-chain implementation, some precompiled contracts (which are written using native code) are implemented.
Here's a list of the ones that are specific to RaptorChain:

- **0x0000000000000000000000000000000000000097**: RaptorChain-BSC Bridge
  Sending RPTR to this contract will result in a cross-chain message being fired, allowing to bridge RPTR to BNB Smart Chain
  Fun fact: this was defined because 97 is BSC Testnet's ChainID, and it wasn't changed when mainnet went live.
- **0x000000000000000000000000000000000000FEeD**: Cross-Chain Data Feed
  This allows contracts on RaptorChain to pull data from supported chains, and to initiate cross-chain calls.

2.3 - Note: partially differing opcode behavior

Due to the whole underlying logic being different from Ethereum's forks, it involves some changes to the way EVM opcodes work, which will be listed here

- **TIMESTAMP (0x42):** TIMESTAMP opcode currently returns timestamp of latest beacon
  block, which could be embarrassing for yield farming contracts.
  However, there's plans to rework the TIMESTAMP opcode in a further upgrade
- **GASLIMIT (0x45):** Since beacon blocks don't physically limit blocks's gas limit, GASLIMIT opcode returns evm's max unsigned integer (2**256-1)

3 - The RPTR token

3.1 - The troubled origins of RPTR token

RPTR token started as an average BSC shitcoin, which was originally named RAPTR. It aimed to fund reforestation projects, and used the Philosoraptor meme as a mascot. However, tensions emerged in the project, and one dev (Miguel) got greedy and rugged it, leading it to collapse in late 2021. However, after that, a new dev (Yanis) took over, and migrated RPTR token to a new smart contract with a supply figures reduction (1 million to 1). It also aimed to give RPTR a new start, even though it didn't work as expected (Miguel removed his last share of liquidity after the new contract was released, on november 1st, 2021, leading it to fall down badly).
RPTR token's supply is currently about 536 millions.

3.2 - Tokenomics

While RPTR token's troubled origins keep us from getting a proper supply allocation chart (but some whales bought a lot after the collapse), there's a lot to say about current token's emission.
RPTR has a linear emission of 1 RPTR per BSC block (or approximately 28800 RPTR/day since BSC has an average block time of 3 seconds), which funds staking/farming rewards on BSC.
Since sending transactions on RaptorChain requires users to pay gas fees, half of these fees are being burned, aiming to make RPTR deflationary once network usage rises.

3.3 - Token Contract

The RPTR token contract is now **0x44c99ca267c2b2646ceec72e898273085ab87ca5** on BSC. RPTR is also present on other blockchain networks thanks to cross-chain bridges (either native thanks to RaptorChain's cross-chain implementation or through third party bridges such as Wormhole).
You can find the full list of contracts on CoinMarketCap.