CHARON BRIDGE

Description

This is a whitepaper for the Inter-Blockchain Bridge (IIB) between Ethereum/Polkadot and FreeTON that will allow a fast and economically efficient transfer of ETH, ERC20 tokens and Polkadot crypto assets.

Abstract

Charon is a service that includes two related components: **Charon Bridge**, which is responsible for wrapping assets from other blockchains and **Charon Swap** as Constant Product Market Maker (CPMM), which is a decentralized exchange for trading crypto assets.

This whitepaper focuses on architecture and economical aspects of **Charon Bridge**.

There are describing architecture and working workflow of transferring assets at **Bridge Architecture** and **Workflow** chapters.

Chapter **Interblockchain Oracle and Consensus Mechanism** contains description of decentralized consensus, award and slashing principles, protection methods from common known attacks, use of {t,n} - threshold signatures and review of validator's software.

Chapter **Transfer data between blockchains** includes information about transferring custom user's data between blockchains.

There are detailed descriptions about economical aspects and decentralized self-governance at chapters **Staking**, **Governance** and **Economics of bridge**. **Charon Token** chapter includes description of the purpose of its and main distribution principles.

These chapters are suitable for almost every blockains that support running smart-contracts, including **Ethereum**. Separate chapter **Implementation details of Polkadot bridge** contains reasons why approach was chosen.

Solidity interfaces chapter contains technical description of Smart Contract's interfaces.

This whitepaper was made for contests: **Contest: Ethereum↔FreeTON Bridge Design and Architecture** [16] and **Contest: PolkaDot↔FreeTON Bridge Design and Architecture** [17].

Research

In common case IIB have to provide execution of several actions:

1. Lock assets on blockchain A
2. Verify and mint tokens on blockchain B
3. Burn tokens on blockchain B
4. Verify and unlock assets on blockchain A

Third-party is required to verify locking and burning assets. There are several approaches:

1. Centralized approach

Third-party in this case can be Escrow organization. As a rule, it's a legal entity in a specific country with local laws. It requires passing KYC/AML procedures for users to enforce local laws. It charges fee for the operating of the bridge and is responsible for safety of locked assets.

Cons: problem of trust to Escrow, no censorship resilience, unfair comissions.

Examples: wBTC on Ethereum, wrapped tokens by FTX on Solana and etc.

2. Separate blockchain

Third-party is a separate blockchain with decentralised consensus. Nodes of this blockchain act as an intermediate part between two other blockchains. Creating a separate blockchain can be too expensive. Existing projects still work in progress.

Examples: AION, Anyswap [1]

3. Proof Verification

In this case, there is no trusted third-party. Blockchains can verify each other using smart contracts. Oracle nodes only require transferring blocks from one blockchain to another. Smart contract validates the block and stores it. After that, anyone can approve lock/burn/etc action from another blockchain and execute it.

This process is divided into two parts:

1. Validation an external block
2. Verification transaction or state transition in a specific block

The validation block process depends on the consensus protocol of the external blockchain. For example, in PoW blockchain you need to check the completed work. Anyone can provide a block to smart contract. And the contract should choose the longest branch as true and pay rewards to nodes. Validation of the next block in PoS systems is to verify validator's signatures of all blocks starting from zero or specific block.

After block validation smart contract can verificate specific transitions using Merke root and Merke path. It allows you to store not all transactions, but only the hash of the tree. This is a fully trustless process without a third-party participation.

There are several common optimizations to reduce gas consumption and simplify verification process. You can batch several blocks and pass them to the

smart contract. The smart contract should cache results of the verification. You can validate only key blocks where there is a change in the list of validators.

Using zkSnark for verification is a prospective approach in building IBB. In this case, validation occurs off-chain. To submit a new block to a smart contract the validator has to provide proof of this. Complexity of proof is O(n*log(n)). Verification of proof occurs on-chain with O(1) and requires a minimum amount of gas. Supporting zero knowledge cryptography by blockchain is very prospective.

Unfortunately, many of these require interoperability of blockchain's cryptography. For example, FreeTON uses ed25519 for signing blocks, but in Ethereum this method is still in draft since 2008 [2]. Also hash functions differ in Merke Proof and PoW consensus. One of these solutions is supporting cryptography from external blockchains by virtual machine.

Example of this: ETH-NEAR Rainbow Bridge [3].

According to the authors of this whitepaper, this method is the most promising solution.

4. Consensus based approach

There are validators which are responsible for verifying actions from blockchains. They use consensus mechanism to reach agreement and create the next block of transactions.

Consensus can be Proof of Authority(PoA) or Proof of Stake(PoS). In PoA case, when creating a bridge, need to define preset nodes which should be interested in the properly worked bridge and predefined fees. They can vote for new nodes or fees. To create a new block required ⅔ + 1 signs from the count of all validators. In PoS consensus each node has a stake which is used for voting for a new block. Due to this, anyone can become a validator node who has enough stake. Validators can vote for slashing unfair nodes. PoS and PoA can be used together, where each node

has the same amount of stake. Examples of these bridges: Parity bridge [4], Poa network <-> Ethereum bridge [5], RSK <-> ETH Token Bridge [6], Solana Wormhole [7].

Consensus can use on-chain or off-chain mechanisms. On-chain occurs on smart contract and offers transparency and reliability. It simplifies synchronisation between nodes, helps to avoid freeloading attacks and enforce following on consensus protocol. More information will be provided in the following sections. Example of off-chain is the Wormhole project [7]. It uses gossip protocol for propagating new blocks and messages and offers less fees and higher speed of consensus. But implementation is more complex and vulnerable to availability attacks. Off-chain can replace on-chain as a more prospective solution after ensuring in reliability and stability.

Requirements

- External Gate Contract (EGC) securely accepting and releasing external blockchain tokens;
- FreeTON Wrapper Contract (FWC) minting and burning wrapped tokens;
- Interblockchain Oracle (IBO) ensuring communication between External blockchain and FreeTON;
- Decentralized Security Layer (DSL) ensuring protection from money stealing.

Additionals

- Fully decentralised approach.
- Universal approach compatible with most blockchains.
- Transferring user’s data between blockchains.
- Staking mechanism, which increases decentralisation level and adds ability making profit.
- Protection from common known attacks.
- Validator reputation system, slashing.

- Self-governance for managing the bridge.
- A clear and predictable economy, governance-token without pre-sale or fund raising.
- Low fees due to consensus at FreeTon and using {t,n} - threshold signature at external blockchain.
- Modular software.

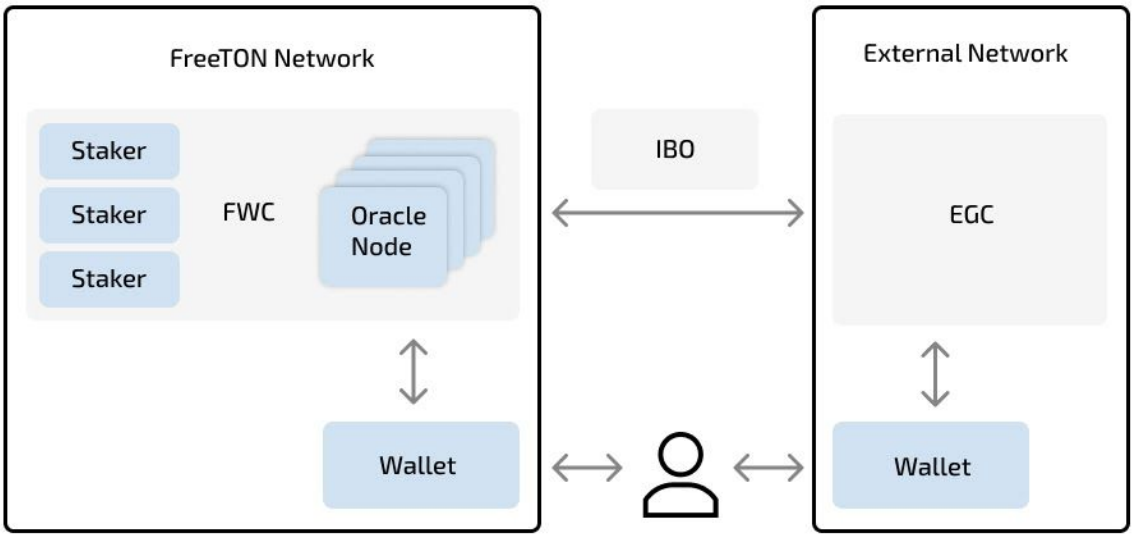
Bridge architecture

1. **External Gate Contract (EGC)** - single or several smart contracts in the External network. Which are responsible for transferring assets to FreeTON network by locking them and unlocking assets on burning them in FreeTON network.
2. **FreeTON wrapper contract (FWC)** - single or several smart contracts in the FreeTON network. Which are responsible for minting new assets on locking them in EGC and backward transfering to EGC on burning them.

   Also, FWC is used for reaching consensus on-chain. It allows reduced fees and higher speed for consensus communication.

3. **Interblockchain Oracle (IBO)** - decentralised oracle, which is responsible for transferring actions between EGC and FWC. It's an Oracle Node that votes on correctness messages using locked stake for reaching consensus. Anyone can launch node, lock state and participate in this process. Proof of Stake is used.
4. **Staker** - it's a User who wants to delegate their stake and get a part of the Oracle node's profit.

Notice: EGC and FWC can be used to transfer custom user-defined messages without locking assets.



Pic. 1. Architecture of bridge

WorkFlow

There are 2 main cases which should be considered: **Wrap** and **Unwrap** assets.

Wrap assets workflow:

1. User sends assets to the EGC contract with the address of destination in the FreeTON network and a small amount of fees to cover the validator's costs on gas.
2. EGC locks assets and creates the event with order for transferring to FWC.
3. User has to pay a fee for wrapping assets on FWC. For that he needs to calculate the fee from FWC and pass it with order id. The fee depends on several factors of the bride.
4. Next, Oracle nodes collect paid orders into a single block and transfer them to FWC. There is the *block_period_time* of wrapping assets during which Oracle nodes can collect orders.

5. Next, FWC checks signatures of new block and mint assets. Oracle nodes have to mark the order as *done* on EGC.
6. If User will not pay a fee, there is a way to return assets. Order contains two parameters *expire_time* and *lock_time*. Expire time is the time after which Oracle nodes can't process order. Lock time is the time after which User can unlock his assets. *Lock_time* has to be more *expire_time* at least two *block_period_time*.

Unwrap assets workflow has the same steps as wrapping:

1. User locks assets to the FWC contract with the address of destination in the External network and pays a fee for transferring.
2. FWC locks assets and creates the event with order for transferring to EGC.
3. User has to pay a small amount of ethers to EGC as a fee to cover the validator's costs on gas.
4. Next, Oracle nodes collect paid orders into a single block and transfer them to EGC.
5. Next, EGC checks signatures of new block and unlock assets.
6. Next, on removing the order from FWC, burning of assets occurred.

For creating new blocks there is a special algorithm that describes a deterministic approach on how to collect new block, order transactions. So validators don't need to reach consensus about the contents of the block, only for it's hash. More information about that in chapter about *Interblockchain Oracle*.

On creating each block validators can vote about pause of all bridge or specific operations. It needed to resolve disputes if there are inconsistencies in the following protocol by validators. Pausing and resuming also can be activated by initiating voting.

For transferring data between blockchains can be used the same process as for assets. Instead of value, the user needs to pass 256 bits of other information. More information about this in chapter about *Transfer data between blockchains*.

Interblockchain Oracle and Consensus Mechanism

Election step

The bridge has to work correctly and be stable. For that the following conditions has to be met for each Oracle Node:

1. The node has to receive payment for the work.
2. The node has to be penalized for misbehaving.

The process of Oracle node selection has requirements for participants:

1. The participant has enough locked stake in Charon Token, more than minimal stake.
2. The participant provides the public key for signing.
3. The participant provides the required amount of keys for multi-party {t,n} - threshold signature.
4. The participant has enough level of trustworthiness, more than minimal level.
5. The spread of participant's stakes has to not exceed the max factor.
6. The total stake has to not exceed the max stake.
7. Amount of elected participants has to not exceed the max amount.

Consensus mechanism is on the FreeTON network to reduce gas cost and fast communication. Also, on the change list of Oracle Nodes, need to generate a new public key from node's keys. Example of this [8]. It's a multi-party {t,n} - threshold scheme which will be used on EGC to validate blocks. FWC will unlock stakes of old Oracle nodes only after committing the first block by the new set of Oracle nodes. So they can be sure that EGC will accept their new key.

Parameters of the elector mechanism can be modified by using voting. Among them: time of one cycle electing, protection time of locking stake, minimal stake, maximal stake, maximal spread of stakes, max amount of nodes, minimal level of trustworthiness and etc.

Level of trustworthiness can't be considered as the separation of trusted and non-trusted nodes. As the node can change the key at any time. But it's usable as automatic mechanism safety of funds for Stakers.

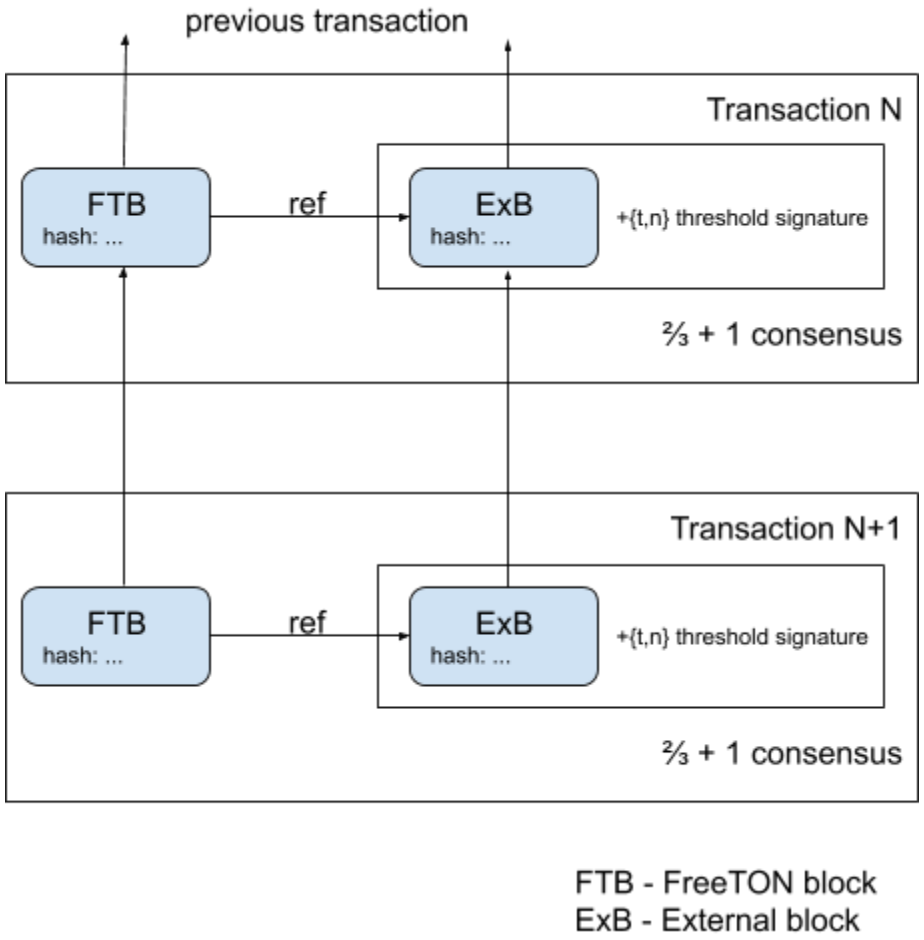
For a higher level of decentralisation there is Stakers. Every user can participate as a Staker. Oracle node can provide information about staker conditions to FWC. For example, the amount of commission. But Oracle nodes are required to pass a minimal amount of stake to pay fines for inactivity, mistakes and etc. So Staker funds will be in safety while node's stake is more than 0.

Block creating step

Main algorithm of creating new blocks has to be deterministic. Every Oracle node has to get the same result. There is only one source of truth. This is not a difficult task because a new block depends on a specific smart contract in external blockchain. The process of creating new blocks by Oracle:

1. FWC has the last signed block and Oracle should wait when this block will be accepted by EGC.
2. Watch changes on FWC and EGC and wait when fully paid order will appear.
3. Wait blockchain's specific time/amount of blocks for finality.
4. Collect order and go to step 2, wait for the new order.
5. Meanwhile Oracle node has to check next conditions:
    a. Reaching max time since the last block.
    b. Reaching max amount of orders in one block.
    c. Reaching max amount of assets in monetary terms.
6. If so, Oracle node has to collect all user's orders, bridge's system messages into two blocks, one for FWC and one for EGC. Order messages in a deterministic way. And timestamp each block with time of the last action *Tb*.
7. The EGC's block has to include the hash of the previous EGC's block with the last action for randomization purpose (Pic. 2).

8. The FWC's block has to include the hash of EGC's block and hash of previous FWC's block (Pic. 2).
9. Oracle can commit signed blocks in FWC in the period between *Tb* and *Tb + Tsign* which signing time is variable parameter.
10. FWC validates parameters of these blocks. Such as: previous block hashes, current block hashes, node's signatures.
11. FWC has to collect node's commits until time of signing is expired. If so, smart contract move next.
12. FWC has to check that this block is signed with more than ⅔ of stake. If so, smart contract accept blocks.
13. FWC remembers nodes who provide correct blocks.
14. Nodes who were late or sign not valid blocks are fined. Their stake decreases and adds to reward for these blocks.
15. FWC executes actions from FWC block, minting or burning wrapped assets, deleting completed orders.
16. Oracle nodes who provide correct blocks get rewards.
17. FWC saves hashes of FWC and EGC blocks.
18. Next, one of Oracle nodes has to collect a multisig signature for EGC from FWC and send the block with signature to EGC.
19. EGC has to validate hashes of previous block, current block and execute orders from block.



Pic. 2. Block models

There is limitation for the amount of assets that can be transferred. It should not exceed the maximum amount or part of the current stake in monetary terms. First time it will be a fixed amount for safety of funds while transferring. But in the future can be improved by providing exchange rates through oracles.

There is a freeloading attack. One of the Oracle nodes can take hashes and blocks from other participants without spending time and resources on the work. It can lead to decreased protection of the bridge. The node who knows that other participants freeload his blocks can try to make an incorrect block and get more than ⅔ votes. To avoid this attack the consensus can be splitted in two steps. Firstly

the smart contract has to collect only signatures without revealing the blocks, after the signing time *Tsign* has passed nodes have to commit blocks.

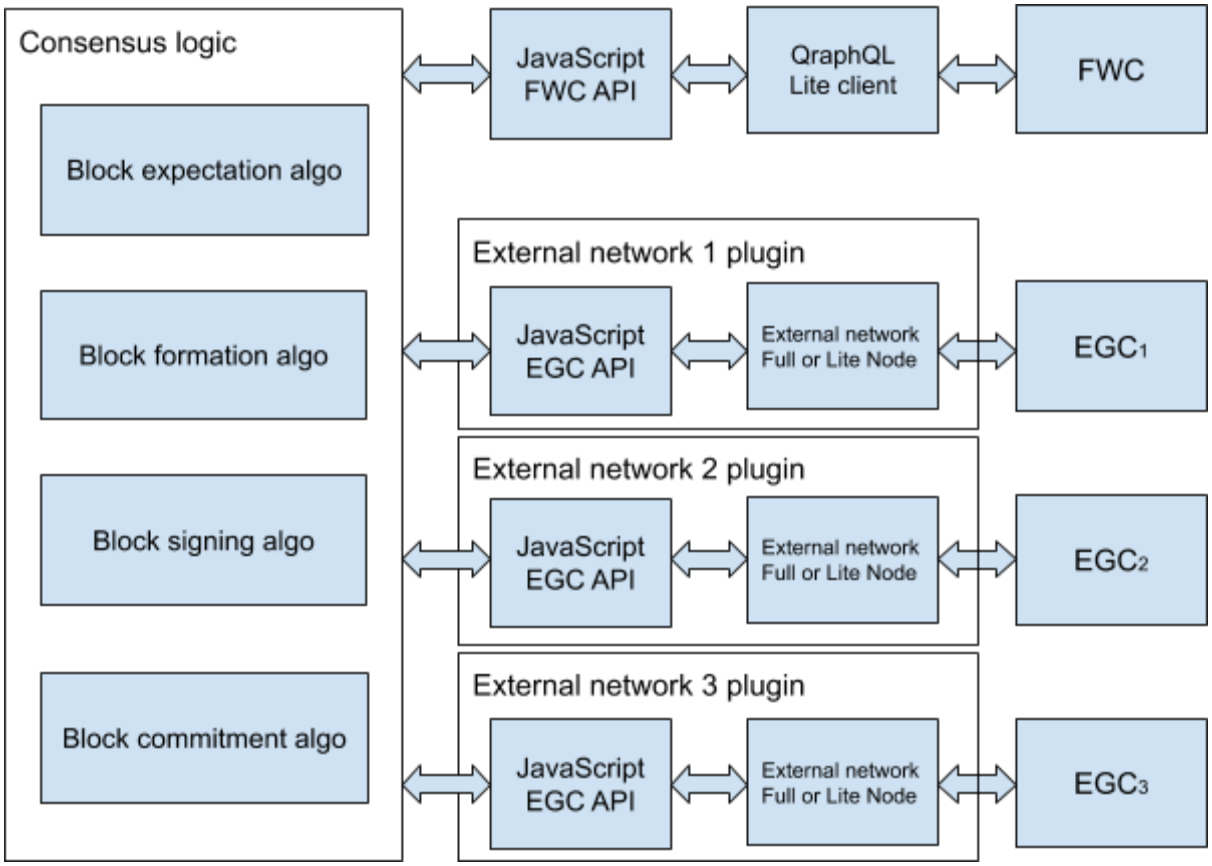
One of the Oracle nodes is required to commit new block to EGC. We should avoid the situation when several nodes will try to commit a new block simultaneously as it requires payment for minner's fee. Oracle nodes need to choose one Oracle node that has to commit the new block to EGC. For this we offer to use pseudo-random selection. Also there is a time limit for committing this block by chosen node. And all costs in the EGC will be paid using fee from users.

During the entire time anyone can join to Oracle nodes, transfer a stake and etc. This doesn't affect the current cycle of validation. On the next cycle of validation a new list of nodes will be selected.

On-chain consensus is clear and verifiable. Misleading nodes can be easily detected and fined by comparison with off-chain solution or {t,n} - threshold signature. Bridge requires only one consensus mechanism at FWC, where all blocks are validated and verified, EGC remains only check signature, accept new block and execute orders.

In our case multi-party {t,n} - threshold has specificity. Each participant has a different weight of vote in staking. In multi-party signatures each participant has the same weight. To solve this problem, we offer the following algorithm:

1. During elections a new list of Oracle nodes the maximum stake divides on 10, so we have defined a stake for a single key.
2. Other Oracle nodes get the corresponding amount of required keys to their stake by integer division.
3. The remainder of integer division doesn't participate in staking and not lock.
4. So each Oracle node gets from 1 to 10 keys for voting and n parameter is defined as the sum of all keys.



Pic. 3. Structure of software for Oracle node

Software can be developed on NodeJs platform with modular architecture to easily extend functionality and support new blockchains. JS EGC API and JS FWC API repeat interfaces BlockInterface for EGC smart-contract and ConsensusInterface and ValidatorInterface for FWC smart-contract. First time software can provide only command line interface.

Transfer data between blockchains

For data transferring between blockchains used the same consensus mechanism with a small difference:

1. Instead of amount assets, the user has to provide 256 bit custom value, receiver address and signature if needed.

2. When transferring have been completed FWC will send message with value and signature to the receiver address using a specific interface.

Also user can transfer state of smart contract from one blockchain to another. To do this, user has to implement the special interface for his contract so Oracle node can create a merkle root of the state suitable for destination blockchain. After that the user can get merkle root on the opposite bridge and verify state using merke path. For example, user may have a database presented as a mapping structure in a smart contract and want to add the possibility for users of this contract to work on the opposite blockchain with small fees and fast messages. After transferring merkle root, his users have to verify their state in the opposite smart contract and after they are ready to use this contract. Fees for Oracle nodes depend on the type and size of data.

Another way to increase the amount of transferring data is using sha256 hash. In this case the user before transferring has to calculate on-chain the hash of the data and pass it to the bridge. When transfer will be completed the user has to pass data to the opposite bridge's contract. Contract has to verify data and hash, and transfer data to the receiver address.

These approaches can work bidirectional, for transferring data from one blockchain to another and vice versa.

Staking

To reach a high level of decentralisation there is the ability for every user to stake his funds. In this work is used staking with a limited amount of nodes. Process of staking has the following steps:

1. Every user can fetch parameters of each Oracle node such as deposit amount, rewards, fee, statistics.
2. Statistics include working time, amount of completed blocks.

3. User has to choose a node with satisfied parameters and enter their own stake.
4. In the next election all stake or part of it can be locked in favor of the chosen validator.
5. When a user's stake is not locked, the user can return funds back. If the user wants to return a locked stake you should request the smart contract that in the next election time the user's stake will not participate in staking.

On the finish election period smart contract estimates profits and losses for every participant in staking and sets the balances accordingly.

Profits are calculated in the following way:

1. Calculates the difference between initial and final stake.
2. Oracles nodes get rewards for their work from this profit difference.
3. Rest of the difference distributes between stakers in proportion in their stakes.

Losses are calculated in the following way:

1. Calculates the difference between initial and final stake.
2. Oracles nodes have to pay a loss difference from their deposit.
3. Rest of the difference distributes between stakers in proportion in their stakes.

Oracle node can finish their work, it has to request it for a smart contract. After the finishing cycle and protection time, deposit and staker’s funds will be returned.

Governance

Governance is an essential part of a decentralised system. Because it must adapt to changeable conditions in an effective way. So the bridge has variable parameters that affect coordinated work. Some of them can be automatically calculated, but others can not be calculated due to complexity and unpredictable external conditions.

For example, a large number of validators increases expenses for transferring blocks and at the same time increases confidence in the system. Increasing confidence can lead to an increasing number of users and decreasing fees. Fees must react to competitors' prices, but decreasing fees can lead to withdrawal funds by stakers...

So, participants (holders of Charon Token) have to manage and regulate these parameters in a decentralised way to reach the best result. Participants have to be economically motivated in an efficient working bridge. We offer a mechanism of self-governance. Voting occurs by using Charon Token, it can be locked in staking or in a pool of liquidity.

Each participant, holder of Charon Token, can create a request to change parameters. For that he must provide unique id and desired value. Other participants can approve or reject the new value by voting Charon Token. Significance of his vote is defined by his stake. The final decision is calculated using soft majority voting (SMV) [9]. If soft supermajority approval is reached the new parameter will be accepted otherwise the old value will remain. Also, there is maximum time for voting, after which request is considered as rejected. We offer 2 cycles of validation.

In addition to parameters, holders of Charon Token can vote for supporting new assets and their parameters.

Also governance has the ability to confiscate funds in case theft of user’s funds by validators during global locking.

Economics of bridge

Actions that stimulate the growing value of the bridge for users must be rewarded, harmful actions must be punished. So earning should include payments from users for useful actions. Also there are costs on miner’s fees for each blockchain network so the user must have to pay for them in FreeTON and External blockchain.

To optimize costs on operation of bridge more as possible actions performed at FreeTON network. Also there is used batching orders and actions into one block to reduce the amount of interactions with blockchains. Many of operations are performed at FreeTON network during on-chain consensus so optimizations have to reduce costs. At external blockchain is used {t,n} - threshold signature so costs on this are defined as O(1).

Primary parameters are changeable variables so the bridge can find the optimal values during the working or while testing it on testnets. These are fee on the action of FreeTON and External blockchain, the cost for the punishment, for invalid block or signature, also amount of fee by validators for staking (defined by validator).

For safety purposes Charon Token will be launched. More details in next chapter.

Charon Token

Platform will consist of two parts:

- Charon Bridge, which is responsible for wrapping assets from other blockchains
- Charon Swap as Constant Product Market Maker (CPMM) [15], which is a decentralized exchange for trading crypto assets. In more details it will be in separate whitepaper.

Charon Token is utility token, which gives the right for:

- Participate in consensus of the bridge.
- Participate in validator’s staking
- Voting in governance

Also, holders of the Charon Token can provide liquidity for trading pairs with this token at Charon Swap.

Total amount will be 100 million Charon Tokens. And the initial 10 million tokens will be distributed in the following way:

1. 15% for developers team
2. 15% for bounty programs
3. 20% for providers of liquidity
4. 20% for validators and their stakers
5. 20% for traders

Distribution will occur once by day with equal share proportional to the contribution of participants. This distribution of tokens for:

1. Gradual decentralization process in development and management
2. Attracting interest of participants
3. Community engagement in development process

One of the goals is to increase the system's safety by using tokens for staking. In case of performing an attack by using ⅔ stake of the bridge by one interested party, when this attack is discovered it will lead to withdrawals of funds by all participants. And as a result the token's market value will drop so Attackers will lose their funds. It increases the cost of this attack.

Important notice that Charon Token is not an investment instrument, holding of them doesn’t imply making a profit or increasing their amount. Distribution of tokens will occur only for useful actions for the bridge by participants. Making profit from validating and staking implies blocking them in consensus and closely connected with risk of losing them in case of misbehaving by validators.

Implementation details of Polkadot bridge

Polkadot is the new secure and scalable blockchain with hub-and-spoke transport layer. The main chain is Relay chain, which connects together other blockchains - parachains [10]. It allows different parachains to work together and exchange messages in a secure way. There are three main approaches to develop applications for Polkadot: smart contracts on compatible parachain, own parachain or parathread. Each approach has pros and cons and is suitable for concrete cases. Let's review them in detail relative to our bridge.

	Parachain	Smart Contract
Speed of development	−	+
Ease of deployment	−	+
Complexity of logic	+	−
Maintainence overhead	−	+
Level of customization	+	−
Strict resource control	−	+
Native chain features	+	−
Scalability	+	−

Pic 4. Comparing parachain and smart contract [11]

Building your own parachain is considered as the most complex and with a slow development speed approach. Polkadot is still actively developing but main parameters of parachain work are already known:

1. Need validators for parachain.
2. Need collators for building parachain blocks and submitting to validators.
3. Validators check each block by using Proof-of-Validity (PoV) and sign its. When a block will have enough signatures it is included in the Relay chain.

4. Next, the block is checked for availability. And only after this block is marked as Approved.

Besides, validator's protocol includes additional features such as verifiable random function (VRF), verifiable delay function (VDF) and etc. which impose additional costs on the validating process. So parahchain provides best safety and functionality, but completely redundant in our solution for the following reasons:

1. PoS consensus in our Bridge is already operated on the FreeTON network.
2. Decentralised staking and slashing is already organised on FreeTON network.
3. Availability of blocks is not required.
4. Checking the block on the external blockchain is just checking signature on correctness.

Parachain has an advantage over smart contracts by developing complex and expensive logic. In our bridge the calculations close to minimal: validation block by checking signatures, executing orders and removing old orders. Also one of the advantages of our solution is the universal approach. It allows us to apply this bridge on many blockchains which support running smart contracts.

As a result using smart contracts for our Bridge we think is more suitable.

Polkadot's parachains can run smart contracts similar to how it does Ethereum. Currently one of the parachains suitable for running smart contracts is Moonbeam [12]. Moonbeam supports EVM-based smart contracts so this makes it easier to port Ethereum's contracts. Also this project is compatible at address level and RPC api, so can be used with existing libraries such as web3 [13]. Moonbeam launched their testnet in 2020 September [14], at the initial step by using PoA. For gas payment they have native token glimmer. Launch of Moonbeam's parachain in Kesama Canary Network is planned for Q4 2020.

Solidity interfaces

FWC smart-contract has to implement:

1. **CharonTokenInterface** - Interface for interaction with Charon Token. Supported operations: user balance, transfer, getting info.

```
interface CharonTokenInterface {

/// @notice Get Charon Token total supply
/// @return total in nanotokens, total supply
function totalSupply() public view returns (uint256 total);

/// @notice Get balance of arbitrary address
/// @param who The wallet address
/// @return value in nanotokens, address balance
function balanceOf(uint256 who) public view returns (uint256 value);

/// @notice Transfer tokens
/// @param to Destination wallet address
/// @param value in nanotokens, transfer value
/// @return result true on success
function transfer(uint256 to, uint256 value) public returns (bool result);

/// @notice Get next distrubuition time
/// @return tm Unix time of next distribution
function nextDistributeTime() public view returns (uint32 tm);

/// @notice Distribute Charon tokens, anyone can call
/// @return result true on success
function distribute() public returns (bool result);

}
```

2. **ConsensusInterface** - Interface for validators, Oracle nodes. Supported operations: committing new blocks.

```
interface ConsensusInterface {
```

```
/// @notice Get current cycle ID
/// @return id
function getCurrentCycleId() public view returns (uint256 id);

/// @notice Get current cycle Info
/// @return info
function getCurrentCycleInfo() public view returns (CycleInfo info);

/// @notice Sign new cycle
/// @param pk Validator public key
/// @param sign external signature for new validators
/// @return result true on success
function startNewCycle(uint256 pk, uint256 sign) public returns (bool
result);

/// @notice Get current consensus ID
/// @return id consensus ID
function currentConsensusId() public view returns (uint256 id);

/// @notice Get validator signs in the current consensus
/// @return signs validator signs
function getSigns() public view returns (ValidatorSign[] signs);

/// @notice Send block signature
/// @param consensusId consensus ID
/// @param pk Validator public key
/// @param lockBit automatic lock bit
/// @param fwcBlockSign FWC block sign
/// @param egcBlockSign EGC block sign
/// @return result true on success
function signBlock(uint256 consensusId, uint256 pk, bool lockBit, uint256
fwcBlockSign, uint256 egcBlockSign) public returns (bool result);

/// @notice Commit signed blocks
/// @param consensusId consensus ID
/// @param blocks raw blocks data
/// @return result true on success
function commitBlock(uint256 consensusId, bytes[] blocks) public returns
(bool result);
```

```
}
```

3. **GovernanceInterface** - Interface for holders of Charon Token. Supported operations: create new voting, vote by using tokens, get current votings.

```
interface GovernanceInterface {

/// @notice Get registered poll ids
/// @return polls list of poll ids
function getPollIds() public view returns (uint256[] polls);

/// @notice Get poll info
/// @param pollId poll ID
/// @return info poll information
function getPollInfo(uint256 pollId) public view returns (PollInfo info);

/// @notice Create new poll
/// @param owner creator address
/// @param stakeValue collateral value
/// @param paramId parameter ID to change
/// @param proposedValue new value for parameter
/// @return pollId poll ID if registered
function createPoll(uint256 owner, uint256 stakeValue, uint32 paramId,
uint256 proposedValue) public returns (uint256 pollId);

/// @notice Vote for poll
/// @param pollId poll ID
/// @param vote true for accept, false for reject
/// @param owner voter address
/// @param stakeValue vote stake value
/// @return result true on success
function votePoll(uint256 pollId, bool vote, uint256 owner, uint256
stakeValue) public returns (bool result);

/// @notice Delete poll when it accepted or expired and return collateral
stake
/// @param pollId poll ID
/// @return result true on success
function deletePoll(uint256 pollId) public returns (bool result);
```

```
}
```

4. **ValidatorsInterface** - Interface for managing validators. Supported operations: create new validator, remove validator, get current validators, detail information\statistics about validator, deposit stake for validator, withdraw stake.

```
interface ValidatorsInterface {

/// @notice Get validators IDs
/// @return validators list of validators IDs
function getValidatorIds() public view returns (uint256[] validators);

/// @notice Get validators info and statistics
/// @param validatorId validator ID to query
/// @return info validator info
function getValidatorInfo(uint256 validatorId) public view returns
(ValidatorInfo info);

/// @notice Add new validator
/// @param pk validator public key
/// @param reservedValue validators own stake
/// @param opts validators options
/// @return result true on success
function addValidator(uint256 pk, uint256 reservedValue, ValidatorOpts
opts) public returns (bool result);

/// @notice Change validator options
/// @param pk validator public key
/// @param opts validators options
/// @return result true on success
function changeValidatorOpts(uint256 pk, ValidatorOpts opts) public returns
(bool result);

/// @notice Add stake to validator
/// @param validatorId validator ID
/// @param addr stake owner address
/// @param value stake to add
/// @param autoRenewal is staking auto renewal enabled
```

```
/// @return result true on success
function addValidatorStake(uint256 validatorId, uint256 addr, uint256
value, bool autoRenewal) public returns (bool result);

/// @notice Remove stake from validator
/// @param validatorId validator ID
/// @param addr stake owner address
/// @param value stake to remove
/// @return result true on success
function removeValidatorStake(uint256 validatorId, uint256 addr, uint256
value) public returns (bool result);

/// @notice Get stake unlock time
/// @param validatorId validator ID
/// @param addr stake owner address
/// @return tm Unix time when stake will be unlocked
function getUnlockTime(uint256 validatorId, uint256 addr) public view
returns (uint32 tm);

/// @notice Withdraw unlocked stake
/// @param validatorId validator ID
/// @param addr stake owner address
/// @param value stake to withdraw
/// @return result true on success
function withdrawValidatorStake(uint256 validatorId, uint256 addr, uint256
value) public returns (bool result);

}
```

5. **WrappedTokenInterface** - Interface for working with wrapped assets. Supported operations: mint tokens, burn tokens.

```
interface WrappedTokenInterface {

/// @notice Get supported tokens list
/// @return tokens list of tokens with info
function getTokenSupport () public view returns (TokenInfo[] tokens);

/// @notice Get payment for wrap
/// @param chainId blockchain transfer from
```

```
/// @param transferredValue value to transfer
/// @return value in nanoCRYSTALS, payment for transfer
function getWrapPayment(uint256 chainId, uint256 transferredValue) public
view returns (uint256 value);

/// @notice Pay for order
/// @param chainId blockchain transfer from
/// @param orderId order ID
/// @param value transfer payment, in nanoCRYSTALS
/// @return result true on success
function payForWrap(uint256 chainId, uint256 orderId, uint256 value)
public returns (bool result);

/// @notice Get payment for unwrap
/// @param chainId blockchain transfer to
/// @param transferredValue value to transfer
/// @return value in nanoCRYSTALS, payment for transfer
function getUnwrapPayment(uint256 chainId, uint256 transferredValue) public
view returns (uint256 value);

/// @notice Burn wrapped tokens and pay for transfer
/// @param chainId blockchain transfer to
/// @param addr owner address
/// @param transferValue tokens value to transfer
/// @param paymentValue transfer payment, in nanoCRYSTALS
/// @return result true on success
function payForUnwrap(uint256 chainId, uint256 addr, uint256 transferValue,
uint256 paymentValue)  public returns (bool result);

}
```

6. **UserDataInterface** - Interface for transferring user custom data between blockchains.

```
interface UserDataInterface {

/// @notice Get payment value for EXTERNAL -> TON data transfer
/// @param chainId blockchain transfer from
/// @param dataBytes bytes number to transfer
/// @return value in nanoCRYSTALS, payment for transfer
```

```
function getDirectTransferPayment(uint256 chainId, uint256 dataBytes)
public view returns (uint256 value);

/// @notice Pay for order
/// @param chainId blockchain transfer from
/// @param orderId order ID
/// @param value transfer payment, in nanoCRYSTALS
/// @return result true on success
function payForTransfer(uint256 chainId, uint256 orderId, uint256 value)
public returns (bool result);

/// @notice Get payment value for TON -> EXTERNAL data transfer
/// @param chainId blockchain transfer to
/// @param dataBytes bytes number to transfer
/// @return value in nanoCRYSTALS, payment for transfer
function getIndirectTransferPayment(uint256 chainId, uint256 dataBytes)
public view returns (uint256 value);

/// @notice Transfer data from TON
/// @param chainId blockchain transfer to
/// @param srcAddr SMC source address of data
/// @param dstAddr SMC destination address
/// @param paymentValue transfer payment, in nanoCRYSTALS
/// @return result true on success
function transferData(uint256 chainId, uint256 srcAddr, uint256 dstAddr,
uint256 paymentValue)  public returns (bool result);

}
```

EGC smart-contract has to implement:

1. **TokenInterface** - Interface for managing assets. Supported operations: lock tokens, unlock tokens.

```
interface TokenInterface {

/// @notice Get supported tokens list
/// @return tokens list of tokens with info
function getTokenSupport () public view returns (TokenInfo[] tokens);
```

```
/// @notice Wrap native coin
/// @param value value to wrap
/// @return orderId order ID
function wrap(uint256 value) public returns (uint256 orderId);

/// @notice Get balance of untransferred funds
/// @return value
function getWrapBalance() public view returns (uint256 value);

/// @notice Withdraw untransferred funds
/// @param value value to withdraw
/// @return result true on success
function withdraw(uint256 value) public returns (bool result);

/// @notice Wrap Token
/// @param tokenAddr token address
/// @param value value to wrap
/// @return orderId order ID
function wrapToken(uint256 tokenAddr, uint256 value) public returns
(uint256 orderId);

/// @notice Get balance of untransferred tokens
/// @param tokenAddr token address
/// @return value
function getWrapTokenBalance(uint256 tokenAddr) public view returns
(uint256 value);

/// @notice Withdraw untransferred tokens
/// @param tokenAddr token address
/// @param value value to withdraw
/// @return result true on success
function withdrawToken(uint256 tokenAddr, uint256 value) public returns
(bool result);

/// @notice Transfer data to Free TON
/// @param dataType data type
/// @param srcAddr SMC source address
/// @param sdtAddr SMC destination address
/// @return orderId order ID
function transferData(uint32 dataType, uint256 srcAddr, uint256 dstAddr)
public returns (uint256 orderId);
```

```
/// @notice Get order ID for transfer data
/// @return orderId order ID
function getTransferDataOrderId() public view returns (uint256 orderId);

}
```

2. **BlockInterface** - Interface for validators. Supported operations: commit new blocks, manage validator's access keys.

```
interface BlockInterface {

/// @notice Get current cycle ID
/// @return id
function getCurrentCycleId() public view returns (uint256 id);

/// @notice Get current cycle Info
/// @return info
function getCurrentCycleInfo() public view returns (CycleInfo info);

/// @notice Start new validation cycle
/// @param pk validator public key
/// @param cycleInfo new cycle info (validator list, params, new public
key, etc)
/// @param sign signature for cycleInfo
/// @return id new cycle id
function startCycle(uint256 pk, bytes[] cycleInfo, uint256 sign) public
returns (uint256 id);

/// @notice Commit new block
/// @param pk validator public key
/// @param block new block to commit
/// @param sign signature for block
/// @return result true on success
function commitBlock(uint256 pk, bytes[] block, uint256 sign) public
returns (bool result);

}
```

References

1. Introducing Anyswap — Fully Decentralized Cross Chain Swap Protocol. https://medium.com/@anyswap/introducing-anyswap-fully-decentralized-cross-chain-swap-protocol-82db1155b7a9
2. EIP-665: Add precompiled contract for Ed25519 signature verification. https://eips.ethereum.org/EIPS/eip-665
3. ETH-NEAR Rainbow Bridge. https://near.org/blog/eth-near-rainbow-bridge/
4. Parity Bridges Common. https://github.com/paritytech/parity-bridges-common
5. POA Bridge Smart Contracts. https://github.com/poanetwork/tokenbridge-contracts/
6. RSK <-> ETH Token Bridge. https://github.com/rsksmart/tokenbridge/
7. Introducing the Wormhole Bridge. https://medium.com/certus-one/introducing-the-wormhole-bridge-24911b7335f7
8. Multi-Party Threshold Signature Scheme. https://github.com/binance-chain/tss-lib
9. Developers Contest: Soft Majority Voting system [Finished]. https://forum.FreeTON.org/t/developers-contest-soft-majority-voting-system-finished/65
10. Polkadot network. https://polkadot.network/
11. Polkadot Builders Starter's Guide. https://wiki.polkadot.network/docs/en/build-build-with-polkadot
12. Moonbeam. https://moonbeam.network/
13. Web3. https://github.com/ethereum/web3.js
14. Moonbase Alpha, the Moonbeam TestNet, Now Available. https://medium.com/moonbeam-network/moonbase-alpha-the-moonbeam-testnet-now-available-5a3116fa08dc

15. Formal Specification of Constant Product Market Maker Model. https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf
16. Contest: Ethereum↔FreeTON Bridge Design and Architecture. https://forum.freeton.org/t/contest-ethereum-freeton-bridge-design-and-architecture/2945
17. Contest: PolkaDot↔FreeTON Bridge Design and Architecture. https://forum.freeton.org/t/contest-polkadot-freeton-bridge-design-and-architecture/2974