TUSO

Platform

Architecture

Explanation of Architecture used

to implement **TUSO** Platform

By Morteza Shahabi

5 March 2022

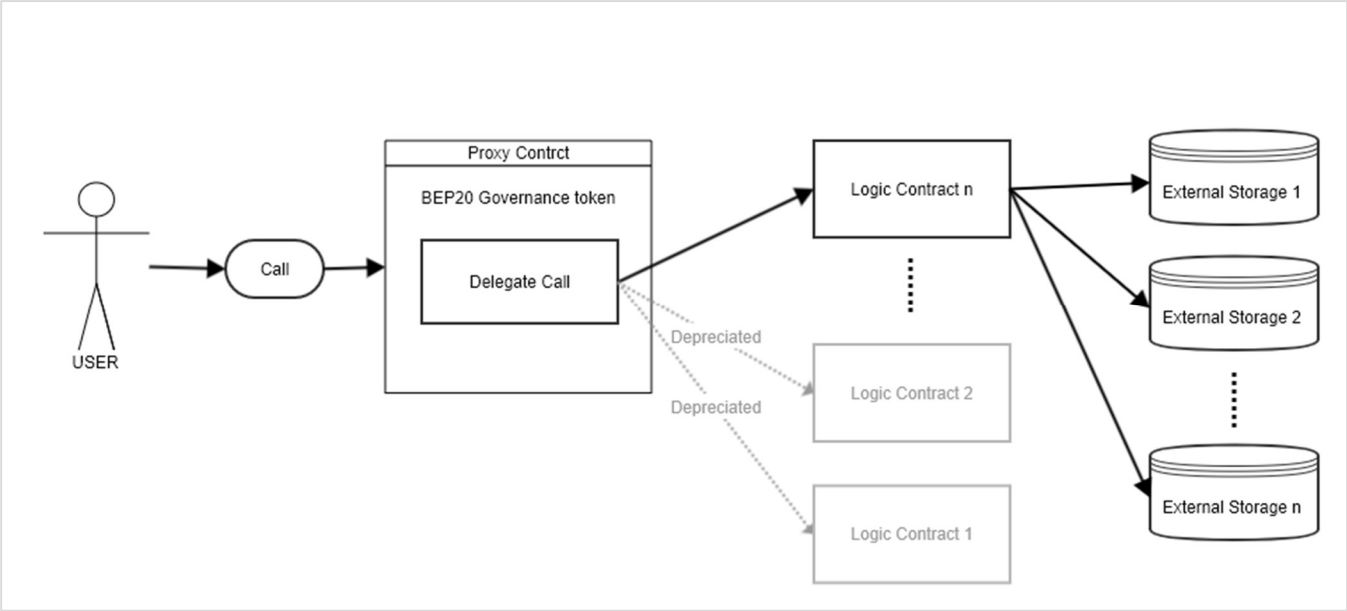
Since blockchain smart contracts are immutable, the development team must have great foresight in order to be prepared for future challenges.

Regarding the project policies, the platform functionality must be extendable. To achieve this goal, structure architecture must provide upgradability which means adding extra features must be attainable. And of course, debugging must be possible too.

In this manner, we'll discuss the aspects and methods of design and implementation of the contract.

Topics under deliberation: Page

General overview:



Let's have a closer look at the figure above.

User calls the Proxy contract to request an operation.

The Governance token interface has been implemented on the proxy contract. If the request is related to the governance token it will be processed inside the proxy contract. And if the request is about other functionalities, it will be forwarded into the logic contract via a delegate call.

The logic contract has the ability to interact with all external storages to get or set variables.

Segments and tasks:

Proxy Contract:

The proxy contract has two roles: (or maybe three)

1- Governance Token Operator:

All the defined tasks for BEP20 tokens plus additional platform tasks.

2- Delegate call intermediator:

Making delegate calls to logic contract to perform the platform operations.

3- Voting organizer:

Performing voting operation.

Logic Contract:

The logic contract is the main regulator of the service flow. All decentralized functionalities of the platform getting occur by this contract.

The logic contract processes the received request from the proxy contract and replies to them. The logic contract may read data from external storages or write data to them. also, may emit some events into the blockchain.

External storage:

The external storage contracts are in charge of saving and loading data. They'll be operated by logic contract.

Inter-segment connections:

Proxy Contract:

Proxy Contract is in the role of User Interface. It would be the access layer of the platform's DAPP. It's in the service of the user. The proxy contract is connected to the user and on another side, it's connected to the logic contract.

Logic Contract:

The logic contract would be the business layer of the platform's DAPP. The logic contract on one side is contracted to the proxy contract and on another side, is connected to all external storages.

External storage Contracts:

External storage contracts are like database tables to the platform's DAPP.

External storage contracts are connected only to the logic contract.

Preparation to change:

Proxy Contract:

The Goal of placing a Proxy Contract in the middle of the user and logic contract is to avoid changing the address in case of deploying a new contract. In this manner, the proxy contract would be permanent and will not change.

Logic Contract:

The logic contract can be changed. but in any deployment, its address in the proxy contract must be updated. The proxy contract only calls the final version of the logic contract.

External storage Contracts:

Since all the data has been stored on external storage contracts, changing them causes either a large cost of data migration or losing data at all. So, in the case of additional data storing requirements, a new external storage contract would be deployed and placed in the service of the logic contract. all the external storage contracts are at the disposal of the logic contract simultaneously.

Functionality structure blueprint:

BEP20 interface:

`function name()` - Returns the name of the token
`function symbol()` - Returns the symbol of the token
`function decimals()` - Returns the number of decimals the token uses
`function totalSupply()` - Returns the total token supply.

`function balanceOf(address _owner)` - Returns the account balance of another account with address `_owner`.

`function getOwner()` - Returns the bep20 token owner which is necessary for binding with bep2 token. NOTE - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

`function transfer(address _to, uint256 _value)` - Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the Transfer event.

`function approve(address _spender, uint256 _value)` - Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

`function allowance(address _owner, address _spender)` - Returns the amount which `_spender` is still allowed to withdraw from `_owner`.

`event Transfer(address indexed _from, address indexed _to, uint256 _value)` - MUST trigger when tokens are transferred, including zero value transfers.

`event Approval(address indexed _owner, address indexed _spender, uint256 _value)` - MUST trigger on any successful call to `approve(address _spender, uint256 _value)`.

Proxy Intermediatory:

Proxy contract construction.

Forwarding calls from user to logic contract using appropriate functions.

Data structure blueprint:

BEP20 interface:

```
string private _name;
string private _symbol;
uint256 private _decimals;
uint256 private _totalSupply;
uint256 private _maxSupply;



// Mapping from token ID to owner address
mapping(uint256 => address) private _owners;

// Mapping from owner to token ID address   => user NFT tokenIDs
mapping(address =>  uint256[]) private _addresTokens;



// Mapping from Id to account balances
mapping(string => mapping(address => uint256)) private _balances;

// Mapping from Id to account freezed balances
mapping(string => mapping(address => uint256)) private _freezedBalances;

// Mapping from token ID to approved address
mapping(uint256 => address) private _tokenApprovals;

// Mapping from owner to operator approvals
mapping(address => mapping(address => bool)) private _operatorApprovals;
```

Proxy Intermediatory:

```
address logicContract;

address logicUpdateApprovalTo;
```

Approximation of Vulnerability:

Challenge 1:

The most important challenge ahead of development is technical issues in the three-layer development model. The complexity of implementation increases the possibility of malfunction which has a huge potential of financial loss.

Solution:

Passing examinations performed by high-level auditors and professional testers will extremely decrease the chance of vulnerability. Surely, because of the large costs, all outsourcing tests would be after alpha tests.

Challenge 2:

Another important sensitive point to be worried about is the large cost of contract upgrades.

Solution:

Although the architecture pattern's greatest achievement is upgradability, the maximum amount of prediction and optimization aside the facilitating strong infrastructure to provide maximum operability is the key to reducing the upgrade necessity.