https://github.com/Music-Protocol/record-token
22/03/2024

RECORD-TOKEN

Music protocol documentation

INTRODUCTION

Music protocol is an application in solidity that provides direct interactions between users and music artists.

The system defines its own custom internal currency, *MusicProtocolRECORDTokens*, and manages their wallets, funds, and investments by developing the ERC20 standards of the *Ethereum blockchain*. Users, through their RECORD tokens, can invest, support and interact with their favorite artists and the system will manage the recognition of the support provided through a weighted voting system based on the investments made.

The application makes contracts to oversee the controlled release of tokens, manage users and their funds, control, manage and remunerate artists and define the operation of the DAO. This ensures that creators receive fair rewards based on the influence they have on their followers and the platform ecosystem.

CONTRACTS

Application has 4 smart contracts carefully implemented to cover all the required functionality.

- MusicProtocolRECORDToken
- MusicProtocolDAO
- MusicProtocolRECORDTokenManagement
- ArtistStaking

The contracts are implemented in different branches used for proper application development.
ArtistStaking can be read on gas-saving-v2-checkpoints.
MusicProtocolRECORDToken can be read on 14-logic-for-token-generation & 21-hardcap-a-1bln branches.

MusicProtocolRECORDToken

MusicProtocolRECORDToken is the smart contract dedicated to the initialization of tokens and their management. The following features are developed in branch ’14-logic-for-token-generation’.

Initialization

The **`constructor`** receives as a single parameter the address associated with the ArtistStaking contract, to securely connect the two contracts and coordinate them. It is needed to implement stake transfer checks and to verify that only usable tokens can be transferred.

Ownership features

The contract inherits the features implemented by the Pausable and Ownable2Step standards provided by the OpenZeppelin library. The **`transferOwnership`** and **`renounceOwnership`** functions can be used only by the owner to modify the ownership of the contract. The owner can also lock and unlock the smart contract and their functionalities by the **`pause`** and **`unpause`** functions.

Erc20 features

All the standard features of ERC20 tokens are available, plus some specific functions have been overridden to realize the required behaviors. The **`mint`**, **`burn`** e **`burnFrom`** functions provide the interface for implementing basic functionality, they can only be called by the owner.

Releasable Payment

The functionality of undermining/transferring tokens and blocking them at the time of transaction has been realized. The locked tokens will be released linearly throughout the entire lock-up period.
The functions enabling the gradual release of tokens are **`mint_and_lock`** and **`transfer_and_lock`** both of which receive the beneficiary's address, the amount of tokens to transfer, the start of the release period and its duration as input. Before sending the request, they map the transfer data through a specifically created data structure called **`releasablePayment`** using the beneficiary's address as key. Gradual payment can only be made once for each address. Tokens generated in this way and still blocked can be used to invest in artists' stakes.

The actual release occurs each time an address attempts to transfer tokens. This is made possible by overriding the standard function **`_beforeTokenTransfer`**, where the control logic is implemented. It ensures tracking of actual balances and implements functionalities allowing the staking of locked resources, it also updates the payment duration and the amount of blocked tokens to guarantee the release flow.

However, **`balanceOf`**, the standard function of ERC20 tokens, always considers the total quantity of tokens owned, whether locked or unlocked.

Release Logic

The release logic is implemented by 3 interacting functions: **release**, **released** and **releasable**. Underlying these functions is the **_vestingSchedule** function, which calculates the amount of total tokens released through an analytic function that linearly associates the elapsed time, the effective duration and the total amount of tokens locked.

_vestingSchedule and **released** are called together by **releasable** which makes the difference between the tokens returned by **_vestingSchedule** and the amount of tokens already released provided by **released**.

Finally, the **release** function, called by **_beforeTokenTransfer**, saves the release in the data structure and making the transaction effective.

Staking functions

The functions that realize the basic mechanics to make stakes are **lock** and **pay**.

The **lock** function allows users to transfer their tokens to the stake address, it takes as input the user's address and the amount of tokens to be staked. This function can only be called from the fanToArtistStaking address.

The **pay** function allows artists to be paid by minting the amount of tokens to the artist's address. The logic by which **pay** is called is implemented in ArtistStaking. It takes as input the same parameters of the **lock** function and has equal access control.

Hard cap

In the branch `21-hardcap-a-1bln`, a hard cap value is established to restrict the quantity of tokens that the owner can generate. Two variables, **MAX_MINT** and **minted**, are employed for this restriction.

The validation occurs within the **mint** and **mint_and_lock** functions by referencing the **MAX_MINT** value. After the execution of the **_mint** function, the **minted** value is updated accordingly.

There is no validation within the **pay** function. The artist’s reward is not added to **minted** value.

Logic of tokens release

ReleasablePayment Structure

This structure has 6 parameters inside:

- **releasableBalance:** it is the total amount of tokens that is conferred when the release is finished. This parameter does not change.
- **duration**: it is the total duration of the release. This parameter does not change. tokens: it is the total amount of tokens minus the amount of tokens staked before their release.
- **updateDuration:** it is the stake duration proportionate to the tokens parameter, it is changed when tokens are staked before they are released.
- **start:** indicates the time of the start of the release.
- **released:** indicates how many tokens have already been released.

This structure is needed to track tokens that have been staked before their release and to control the release proportionally.

When a gradual release is assigned to an address, the amount saved in **releasableBalance** is added immediately to the balance of that address. Therefore, if the address has received a gradual release, there is a need for control logic that properly handles transactions. This logic is accomplished through the use of the **_beforeTokensTransfer** function.

_beforeTokenTransfer

This function aims to properly handle both the release of tokens and the staking of tokens not yet released by modifying parameters inside the releasablePayment structure associated with the address (debt and excess). Before applying the release and debit logics it checks that the balance of the sender address is sufficient for the transaction, if it is not so it will let the transaction continue without making any changes and it will be reversed from the base function _transfer of the ERC20 standard.

Simple TRANSACTION from an address that has received a phased release of tokens

If the transaction we need to handle is a simple transaction between addresses, by an address that has received a phased release of tokens, first we call the **release** function that allows the sender address to update how many tokens released it has. Next, we test that the tokens owned (**ownedTokens**) are greater than the amount of tokens to be transferred. If this is not true, the function is reversed otherwise it continues as a simple transaction through ERC20 standards.

Transaction to stake tokens: STAKE

If the transaction tries to stake tokens, using the fanToArtistStaking address as recipient, first releases the tokens by the **release** function as in the previous case and then check whether the quantity of tokens sent for staking is greater than those owned in that moment. If this is true this means that the address globally has

enough tokens to make this stake, because it has passed the first check based on its balance, but has not yet accrued enough. We want the address to be able to use tokens on the platform that have not yet been accrued, so we calculate a debt in relation to the quantity of requests minus the quantity of tokens owned and we update the tokens variable of the releasablePayment to keep track of it.

TRANSACTION TO REGAIN TOKENS PUT ON STAKE: REDEEM

If an address, which has received a phased release of tokens, tries to reacquire tokens staked it must be checked that it has not incurred a debt through staking. First it is checked if there is a difference between the relasableBalance value and the tokens parameter of the relasablePayment, then, if there is a debt, it is checked if the amount of tokens it is withdrawing is less than the debt: in this case the whole amount of tokens is put inside the tokens variable. Otherwise, if the amount of tokens it is withdrawing is greater than the debt, the amount of the debt is entered within the tokens variable

DEBT AND UPDATEDDURATION

When the tokens variable is changed, the updatedDuration variable is changed proportionally. This is necessary to make sure that the amount of tokens accessed during a given period considers the staking of tokens not yet released. This variable will block the gradual release of the release function until this debt is reacquired from the stake, but the release will continue to consider the elapsed time and consequently release the correct amount of tokens when the debt is settled during the next transaction or stake. To fully understand this process, it is necessary to understand how the release of tokens happens.

RELEASE

Release is done through the release function, which uses three underlying functions: **releasable**, **released**, and **_vestingSchedule**.

The only thing the release function does is acquire the amount of releasable tokens at the time of the call by the **releasable** function and adds it to the released variable of the address releasablePayment. The recipient address is takes as input.

The releasable function also takes as input the address at which tokens are to be released and acquires the actual amount of releasable tokens by making the difference between the value returned by **_vestingSchedule** and the value of returned by the **released** function.

The **released** function returns the value saved within the released variable of releasablePayment, also captures the address of the release recipient as input.

The **_vestingSchedule** function acquires as input the releasablePayment structure of the address, passed by the releasable function and the time when the release was executed. This function can return the value 0 if the current timestamp is less than the start value of the releasablePayment, it can return the value tokens in case the current timestamp is greater than the start time of the releasablePayment (start) and the entire updatedDuration value, and finally it can return a partial value

of tokens proportional to the time passed and inversely proportional to the updatedDuration because the release is linear with respect to time.

As can be seen, the **_vestingSchedule** function uses the updatedDuration and tokens values of the relasablePayment to calculate the amount of releasable tokens, so if these values are changed, the release can be controlled in the appropriate way.

ARTISTSTAKING

ArtistStaking is the smart contract that implements the necessary features for users' investment and artists' remuneration. It also takes on board the voting power granted to users in relation to their investment. The following features are developed in branch ’gas-saving-v2-checkpoints’ and the voting system features are implemented on ‘governance-module’ branch.

DATA STRUCTURES

ArtistStaking contains several data types and data structures to implement its operation.

Data types:

- **`Stake: uint256 amount, uint40 start, uint40 end`**
- **`ArtistReward: uint256 rate, uint40 start, uint40 end`**
- **`ArtistCheckpoint: uint256 tokenAmount, uint40 lastRedeem, uint256 amountAcc`**

These *data types* are used in the following ***structures***:

- **`_stake`**: a nested mapping **`address`** → **`address`** → **`Stake`**, the first address is the artist, the second address is the user.
- **`_artistReward`**: a dynamic **`ArtistReward`** array that tracks changes in artists' reward rate.
- **`_artistCheckpoints`**: a mapping **`address`** → **`ArtistCheckpoint`** for the operations of the **`_calcSinceLastPosition`** function, a function required to calculate the artist's reward.

Other important structures:

- **`_verifiedArtists`**: a mapping **`address`** → **`bool`** to keep track of verified artists.
- **`_votingPower`**: a mapping **`address`** → **`uint256`** to keep track of each investor's voting power.

INITIALIZATION

The function **`initialize`** takes 6 input parameters:

- **`address MusicProtocolRECORDToken_`**: defines the address of the Web3NativeToken smart contract and checks that must be different from zero address.
- **`uint256 artistMusicProtocolRECORDTokenRewardRate`**: defines the artist reward rate.
- **`uint40 min`**: defines the minimum stake time.
- **`uint40 max`**: defines the maximum stake time.
- **`uint256 limit_`**: defines the limit of rewards.
- **`uint256 changeRewardLimit_`**: defines the time interval before the **`artistReward`** can be changed.

In this function the MusicProtocolRECORDToken contract interface **`_MusicProtocolRECORDToken`** is initialized, the first data of the **`_artistReward`** array is assigned with **`rate`** equal to reward rate parameter and the minimum and maximum stake periods are assigned to **`_minStakePeriod`** and **`_maxStakePeriod`**.

The variables **_rewardLimit** and **_changeRewardLimit** are set to the value of the corresponding parameters.

This initializer function can only be called once.

OWNERSHIP FEATURES

The contract inherits the features implemented by the Ownable2Step standard provided by the OpenZeppelin library. The **transferOwnership** functions can be used only by the owner to modify the ownership of the contract.

ARTIST VERIFICATION

The **addArtist** function takes two addresses, artist and sender, as parameters and can only be called by the owner. It verifies that the artist's address is different from address zero, and in case that artist is not already verified it inserts the address into the **_verifiedArtists** mapping by assigning the boolean value to true. Finally, it creates the first **ArtistCheckpoint** of the artist by assigning it to the apposite mapping.

VOTING POWER

This contract implements functionality inherited from OpenZeppelin's VotesUpgradeable contract. When a user invests in an artist through the functions inside this contract, the system assigns to user the voting power calculated by the quantity of tokens invested. The assignment is done by updating the dedicated **_votingPower** mapping and calling the **_transferVotingUnits** function. This voting system will be the basis of the DAO functionality of this application.

_CALCSINCELASTPOSITION FUNCTION

This function is used by the three main features of the staking system. It is necessary to calculate the amount of tokens transferable to the artist, update the amount of tokens staked on him, and update the redemption timestamp.

_calcSinceLastPosition takes three parameters as input:

- **address artist**
- **uint256 amount**
- **bool isAdd**

First, it calculates the tokens earned up to that time. If the amount of tokens in stake on the artist is greater than zero, it iterates a for loop to keep track of changes in the vector **_artistReward**. Inside this loop it finds the interval of time elapsed since that moment and adds to **accumulator** variable the amount of calculated tokens. The function that calculates the number of tokens releasable is proportional to the number of tokens on the stake, the time interval and the **_artistReward** rate of the interval. Lastly, it is divided by a constant factor equal to 10e9.

When the for loop is done it increments the quantity of **amountAcc** variable of the **ArtistCheckpoint** inside **_artistCheckpoints** map.

The parameter **isAdd** determine the behaviour of the second operation. If true, the function adds the amount parameter to the **tokenAmount** variable inside

_artistCheckpoints map. Else, it decrements **tokenAmount** by the same value. In the end, it updates the **lastRedeem** value inside the checkpoint for future calls.

STAKE

The investment of tokens is done through their addition to artist-dedicated stakes. The function **stake** takes as parameters the artist's address, the amount of tokens to be invested, and the expiration time. The amount of tokens invested must be greater than or equal to one unit, which is 1e-18 token.

It ensures the accuracy of the parameters and utilizes the **_MusicProtocolRECORDToken** interface to execute the function **lock**, validating the success of the call's outcome.

After that, it initializes a **Stake** struct with the parameters and assigns it to the **_stake** map, increases the user's **votingPower**, updates the voting power by **_transferVotingUnits** function and calls the **_calcSinceLastPosition** with **isAdd** parameter set to true to increase the stake.

A user's voting power **is not affected** by the duration of his stake. The only parameter that is used in the calculation of **votingPower** is the amount of tokens that are staked.

STAKE TOOLS

Users have three tools to modify the stakes already created.

increaseAmountStaked: this function, after verifying the existence of the stake and ensuring it's not expired, executes the **lock** function through the **_MusicProtocolRECORDToken** interface. If the function is successful, it adds the amount parameter to the amount variable of the corresponding stake by the **_stake** mapping. It increases the variables dedicated to the voting system and finally calls **_calcSinceLastPosition** to effectively update the artist's stake.

extendStake: this function extends the duration of a stake. The requirements for its execution are that the stake exists, is not already expired, that the time extension passed as a parameter falls within the values of **_minStakePeriod** and **_maxStakePeriod**, and that the new overall duration is not greater than the current timestamp plus **_maxStakePeriod**. The only change it makes on the data is to update the end parameter of the data inside **_stake**.

changeArtistStaked: The function changes the beneficiary of an already made investment. This function checks for the existence of a stake under the old beneficiary's name, **artist**, and the non-existence of one under the new beneficiary's name, **newArtist**. Afterward, it ensures that the existing stake isn't expired and that **artist** and **newArtist** are not the same addresses. To change the beneficiary, it creates a new **Stake** data with the **start** set at the time of the call and inserts it into the **_stake** map under the new artist's name. Subsequently, it calls **_calSinceLastPosition** twice: first with **isAdd** set to true to update the **ArtistCheckpoint** of the new artist, and then with **isAdd** set to false to decrement the tokens from the **ArtistCheckpoint** of the old artist and potentially record the

accrued reward up to that point. Finally, it deletes the data dedicated to the old artist within the **_stake** structure.

addStakes: this function can take as input a vector of addresses, and two vectors of integers uint256 and uint40, respectively, representing the amount of tokens and the duration of associated with each address. This function can be used to send multiple stake requests at the same time, if one such operation fails all operations are reversed.

REWARD

Remuneration of artists is done through two functions based on the **_calcSinceLastPosition**, **getReward**, the external wrapper, and **_getReward**, the internal function.

They take in input the artist's address and **_getReward** calls **_calcSinceLastPosition** with the amount parameter equal to 0 to get the updated amount of tokens releasable through artist's checkpoint, without changing the stake. Afterwards, it calls the function **pay** by using the **_MusicProtocolRECORDToken** interface and changes the value of **amountAcc** to zero.

REDEEM

The redeem function has two parameters that identify the stake, they are the artist's address and the user's address. It checks that the parameters are correct and that the stake duration has elapsed before executing itself.

It calls the **_calcSinceLastPosition** function to update the stake and redeem the user's investment. As a result, the **isAdd** parameter is set to false and the **amount** parameter is equal to the total tokens invested. After this call, the function decrements the **_votingPower** value related to the user and call the Votes functions. Finally, it deletes the related data from the **_stake** mapping.

_REWARDLIMIT

ArtistStaking allows any address to redeem artists' rewards by calling **getReward** function. The artist's address will get the rewards in newly minted tokens. The rates of this mint are stored in **_artistReward** array. When the total value of tokens is calculated, the various rates associated with each time frame are taken into count. To make this mechanic safe from gas issues it is necessary to limit the maximum number of rates that can be used in the reward calculation, this implies that if the getReward is not done periodically **the artists may not receive the accumulated tokens.** The insertion of a new value within the **_artistReward** array can only occur after a larger interval than the interval specified in **_chageRewardLimit** has elapsed. In addition, the **_rewardLimit** can be modified by the governance to be readjusted to the users' timings.

MUSICPROTOCOLRECORDTOKENMANAGEMENT

This contract is responsible for generating the calls that handle the contracts. Administrator can interact with it to manage the entire system, assign roles, manage artists and update the DAO whitelist.

INITIALIZATION

The constructor receives two parameters: the address of the MusicProtocolRECORDToken and fanToArtistStaking contracts. After checking and storing the addresses for internal calls, it grants all the defined roles to the contract owner.

ROLES

Roles are defined to determine responsibilities and functions of administrators.

- Admin: the account with this role is able to grant or revoke other roles.
- Minter: the account with this role is able to mint tokens.
- Burner: the account with this role is able to burn tokens.
- Artist verifier: the account with this role is able to add artist to F2A whitelist.
- Artist remover: the account with this role is able to remove artist from F2A whitelist.

FUNCTIONS

This contract provides all the interface functions needed to manage the system. They can be called by the owner and administrators with the appropriate role. The following is a complete list of functions with a brief description, functions are divided for each role.

Admin functions:

- **`transferMusicProtocolRECORDToken`** transfers ownership of the MusicProtocolRECORDToken contract.
- **`transferArtistStaking`** transfers ownership of the ArtistStaking contract.
- **`pauseMusicProtocolRECORDToken`** blocks any type of transaction of the MusicProtocolRECORDToken contract.
- **`unpauseMusicProtocolRECORDToken`** unlocks any type of transaction of the MusicProtocolRECORDToken contract.
- **`changeMusicProtocolRECORDToken`** modify the MusicProtocolRECORDToken interface by assigning another instance of the MusicProtocolRECORDToken contract.
- **`changeFTAS`** modify the ArtistStaking interface by assigning another instance of the ArtistStaking contract.
- **`changeArtistRewardRate`**: change the inflation rate of ArtistStaking.

Minter functions:

- **`mint`** executes mint Web3NativeToken function.

Burner functions:

- **`burn`** executes mint Web3NativeToken function.
- **`burnFrom`** executes mint Web3NativeToken function.

Artist Verifier functions:

- **`addArtist`** adds the artist address array to ArtistStaking whitelist.

Artist Remover functions:

- **`removeArtist`** removes the artist address array to ArtistStaking whitelist.

Custom

The function **`custom`** can be used to execute others function of Web3NativeToken, ArtistStaking and Web3NativeTokenDAO contract. This function is used to change the ownership of this contracts and to access the **manageWhitelist** and **switchWhitelist** function of the Web3NativeTokenDAO contract. It can only be used by the owner.

AddArtist and RemoveArtist

In the branch `gas-saving-v2-checkpoints`, these functions are modified to operate through the input of address arrays instead of single address. The gas consumption of the for-loop inside this function can be elevated.

Vesting manager

In the branch `14-token-for-generation`, the role of the vesting manager is added. This role is allowed to interface with the Web3NativeToken contract to use the **`mint_and_lock`** and **`transfer_and_lock`** functions. The **`transfer_and_lock`** function first transfers the caller's token to the MGMT. Therefore, to ensure the execution of **`transfer_and_lock`**, an approve function must be executed first.

MUSICPROTOCOLRECORDTOKENDAO

The logic for managing, voting and executing proposals is realized in this contract.

DATA STRUCTURES

The contract generates the **proposal** data structure which has as internal variables:

- **uint256 blockNumber**
- **uitn256 maxProposalMembers**
- **uint256 proposalVoters**
- **uint256 votesFor**
- **uint256 votesAgainst**
- **uint128 timeStart**

proposal data are stored in a mapping named **_proposals** and keep track of accepted proposals. To access this mapping data, a value called **porposalId** is used, which is computed using the **hashProposal** function. It executes the keccak256 function that operates on the encoding of the targets, calldatas and description parameters that define a proposal.

Another important structure is **_votes**. It is a double mapping **uint256** → **address** → **bool** that records who have voted on a specific proposal. The associated address is the user's address, while the first uint256 key is the result of the hashing function keccak256 that takes as parameters the encode of the **proposalId** variables of the proposal and its **timeStart**.

Finally, there is a structure called **whitelistedAddresses** that implements the whitelist of the governance. This whitelist can be disabled by the **whitelistEnabled** boolean that is assigned during the initialization.

INITIALIZATION

The constructor of this contract acquires and assigns 5 initial variables after verifying their correctness. These variables are:

- **address ftas_**: the address for the interface to the fanToArtistStaking. The interface is contained in ftas.
- **uint128 quorum_**: assigned to **_quorum** and determines the quorum percentage to be reached to execute a proposal.
- **uint128 majority_**: assigned to **_majority** and determines the majority percentage to be reached to execute a proposal.
- **uint128 time**: assigned to the variable timeVotes and is the time interval in which the proposal can be voted on.
- **bool whitelist**: assigned to the variable whitelistEnabled.

For these parameters to be accepted, **ftas** must be different from the zero address, and the quorum and majority must be less or equal than 10e8.

PROPOSE

The function **propose** has as input an array of address targets, an array of call data to perform the operation decided on the respective targets in case the proposal is executed, and a string description of the proposal. The function checks that the arrays have the same length and that a proposal with the same **proposalId** does not already exists. If these checks are successful, the proposal, by **proposalId**, is assigned to the **_proposals** mapping. The **timeStart** of the proposal is set to the current timestamp. Also saved within the structure is the **blockNumber** to refer to the votes held by addresses at the time of the proposal and the **maxProposalMembers** parameter that saves the current state of the **_members** variable, the last of which is used if the whitelist is active. The parameters **votesFor** and **votesAgainst** are set to 0.

VOTE

The function **vote** has as input an array of address targets, an array of call data and the string description. These parameters are used to take the **proposalId** of the propose. It also takes a boolean flag to determine whether the vote is for or against the proposal. It successively checks that the sender of that vote is within the whitelist or that the whitelist is disabled before proceeding. If the verification is valid, it computes the **proposalId** and verifies that the user has not already voted using the **_votes** mapping. It acquires the user's voting power by **getVotes** or **getPastVotes** functions of the fanToArtistStaking and adds this quantity to the **votesFor** or to the **votesAgainst** in relation to the flag. Finally, update the **_votes** mapping to keep track of the vote.

EXECUTE

This function can be called, with the same parameters as propose, to execute a proposal accepted by the organization. After calculating the proposalId it checks if the proposal is present and that it is finished. It checks that the proposal has been approved through appropriate **internal functions** and finally deletes the proposal and executes the functions associated with it.

INTERNAL FUNCTIONS

Verification of proposal approvals is done through two internal functions, **_reachedQuorum** and **_votePassed**. The first verifies that the percentage of the quorum is reached, and Its behaviour varies in relation to the status of the whitelist. If the whitelist is active only addresses whitelisted will be considered and the calculation will be made in relation to the number of members present at the time of the proposal. The second verifies that the majority, calculated based on total votes by using **_majority**, is less than the votes in favour.

GETPROPOSAL

getProposal is a function that takes the same input parameters as **propose**, checks whether the proposal associated with those parameters exists, and returns it to the caller.

WHITELIST

The `function` **manageWhitelist** takes a target and a flag as parameters, replaces the value pointed to by **whitelistedAddress[target]** with the flag. It is used to insert or remove a user from the whitelist. It can only be used by the owner.