Aion ERC20 Token New Features

Nuco Engineering Team

August 28th 2017

Following is a documentation of intent of new features, API changes and state changes.

High Level Overview

From a high-level, we have a view of how Tokens will move from Ethereum to the AION-1 network (E2A). This will be handled by a set of specialized nodes listening in to particular events emitted by the ERC20 token contract. To facilitate this, certain features have been added to the contract. We imagine this API to be feature specific for the AION-1 platform, therefore some API changes have occured to reflect that.

Additionally, because we are not sure of the approach taken from the other route (A2E) we have modified the Ledger state to account for the possibilities (proofs, push, pull) etc.

For a visual view of the state changes, refer to the UML diagram here. This diagram is also available (but very small!) in the appendix.

Changes

Following is a number of reasoning behind the changes.

Burning

Our proposed burning mechanism is a one way mechanism (E2A), with respect to the contract, burning is simply a publically available function call where the user can specify the desired address to transfer the currency to, and the value of said transfer. Internally, the currency redirects to a `burnAddress`, a state that is maintained by all three contracts. This address is the address that currency is moved to when it is burned. The value can possibly be held by us, or be a random account outside the reach of all users.

If the transfer is valid, then the event emitted will be picked up by the bridge, and transferred to the other bridge. We imagine this bridge to be fully owned by the foundation.

Additionally, burning can be enabled or disabled, therefore allowing us to disable burning in the event of a crisis or instability in the network.

Receiving

Receiving refers to the act of transferring value back from AION-1 (this can be renamed appropriately). Presently we have not decided upon the mechanism for reception, therefore we have designed our state with several possibilities in mind. Because the controller logic can be swapped out, we're confident the data structures we have made accessible in the Ledger will accomodate for our proposed changes.

Disabling

We previously discussed (under the term dismissal or kill) the concept of dismissing or rendering a contract inactive in case of a hard-fork duplication. We feel this issue is already adequately addressed by the `pause()` functionality.

Code Changes

The following are code changes necessary for implementation of the proposed features. If the reader is part of the confirmation team or unit testing, the following are the important changes.

Added Event Definitions

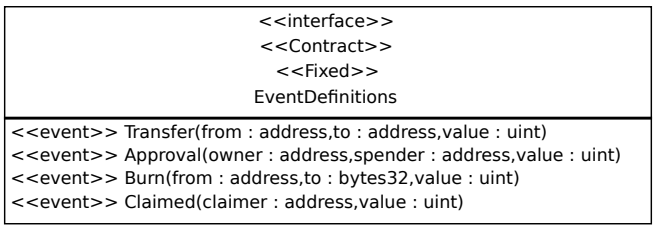

Figure 1: Event Definitions

There are two new events added, one is `Burn` which can be used to track burns on the contract. The second is `Claim` (un-burn...?), which is used when a user claims Tokens into the contract.

Ledger.sol

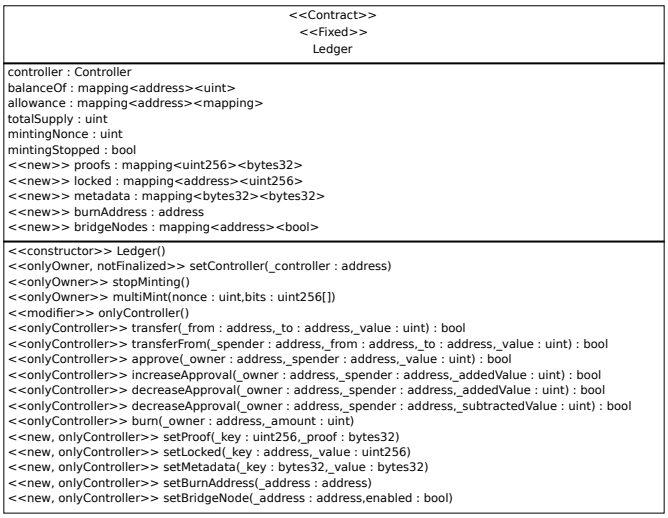

Figure 2: Ledger

Ledger accomodates several new states and the removal of the burn functionality.

- `proofs` is a mapping from a uint256 value to a byte array. We imagine that bridges could have a mechanism to `publish` proofs through the controller.
- `locked` refers to locked value, in case we would like bridges to pushed `locked` value directly into the contract, and let users `pull` (acquire it) somehow, this can work in conjunction with the proof mechanism or independant.
- `metadata` is our final layer of extensibility, its a generic mapping between two `bytes32` that can accomodate for any potential future features.
- `burnAddress` (as discussed previously) is an address set by the controller indicate where the transferred funds go to.
- `bridgeNodes` is a mapping from an address to a boolean indicating whether a certain address is part of a bridge. This may or may not be used in the future as part of the `publish` mechanism for `proofs`.

We have added setters for each data structure, and removed the `burn` function (since it now reduces down to a transfer). The logic behind that functionality is

now moved into the controller.

Controller.sol

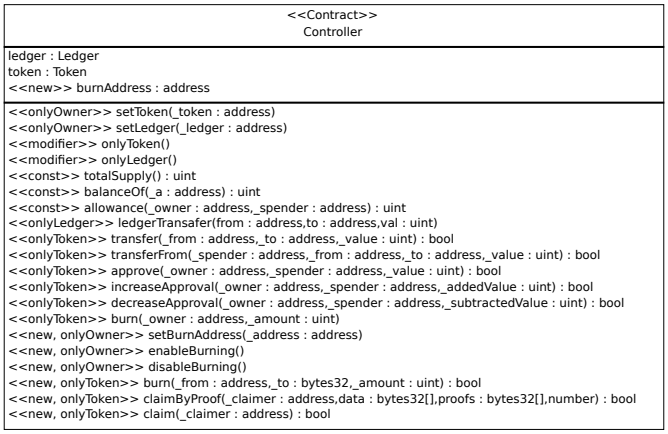

Figure 3: Controller

Controller functionality has remained similar with one changed function and and two added functions.

- `burnAddress` provides the same functionality as was defined earlier and in ledger.
- `burn(_from: address, _to: address, _amount: uint)` has been modified to be AION specific, only accessible by the token contract, it reuquires a from, to and amount. On successful transfer from `_from` to `burnAddress`, an event is emit `ControllerBurn` which the bridges are listening to.
- `enableBurning()` and `disableBurning()` are two functions for enabling or disabling burning functionality, triggered by the owner. These trigger changes to the token contract.

Token.sol

Token has remained similar with some additional functionality.

- `burnAddress` same as discussed before
- `enableBurning()` and `disableBurning()` modifies a boolean `burnable` correspond to a modifier `burnEnabled()`. Allows burning only when `true`.
- `burn(_to: bytes32, _amount: uint)` is the public interface to the user, and calls down to the controller and ledger when a burn is needed.

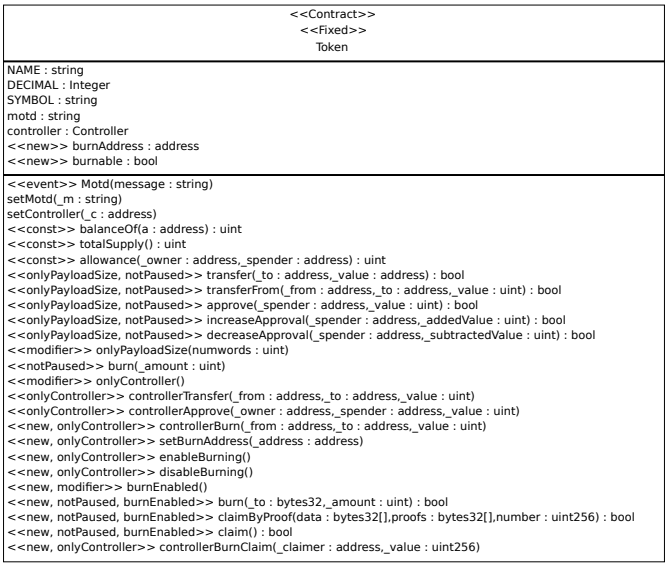

Figure 4: Token (View)

- `claimByProof(bytes32[] data, bytes32[] proofs uint256 number)` related functionality has been added in case the user wants to receive tokens from the other network. As before, because the implementation details are still fuzzy, we have included both the proof mechanism: which operates similar to what New Alchemy described; with the addition of a block number field that the user can define to specify which block the proof was committed in.
- `claim()` places more trust onto the bridge, in that the value is automatically pushed into a locked map, and can be released by calling the contract.

Additional Notes

Some additional notes here:

- multiMint functionality should be thoroughly tested
- Token API needs to be verified by confirmation team and agreed upon that the design is correct
- Ledger contract currently contains conflicting design choices (getters setters vs transfer). Should we switch to a particular design or keep it as is (less risk of errors).