Decentralized Name Service (DeNS)

Repo — https://github.com/RSquad/DeNS
(contest branch: contest)
Tg — @inyellowbus

Key Concepts

As stated in the terms of the contest, everything should be based on TIP-2. Without going into details, the idea behind this improvement proposal is to exclude possible infinite ledgers from contracts.

Why is this needed? Hypothetically, registries in contracts, such as a list of all rates or a list of all domain names, can be extremely large. In addition to the fact that there is a store price, contracts with unlimited registries may sooner or later cease to function correctly because their size is somehow limited.

How to solve this? Free TON has an elegant solution to solve this problem. Since all addresses are calculated from the contract code, public key and initial data, we can, with the same public key (usually zero), deploy the same contract code, changing only its initial data. This allows, by setting some unique identifier in the initial data, to calculate the address of the child contract from anywhere. The principle of operation remains - we, as well as from the register, can get the address of the contract in which the data is located by the unique identifier without creating the register as such, while completely eliminating the possibility of replacing the child contract, since when its address changes, the code will also change, which will lead to change the address.

But how to solve the problem of “fake” contracts that can be deployed with the same parameters but not the parent contract? To do this, in all child contracts in the initial data, in addition to the identifier, the address of the parent contract is written. This allows you to check when deploying a contract (in the constructor) on who is deploying this contract and whether it is the parent contract, otherwise reject the message. Moreover, if this check is removed, the contract code will change, which will lead to an incorrect address from the point of view of the parent contract.

In the application, you will often find references to the static variables _name and _deployer, which are used as the identifier and address of the parent contract to implement the concept described above.

Smart Contract System

The Main Smart Contracts of the System

Base

The contract that contains constants, modifiers, and functions used by all or most of the system's other contracts.

Getters

```
function hashAmountWithSalt(uint128 amount, uint salt) public pure returns (uint
hash)
```

The function that is used to calculate the hash amount & salt. Available in all contracts implementing this contract.

It's really important to have an on-chain function to calculate the hash from multiple chunks of data. If you calculate the hash on the client, and check on the blockchain, the system may fail, for example, in concatenation. As you can see from the function, it takes the amount and the salt, somehow connects them, and returns a hash, which guarantees the same result with the same parameters passed. The key problem with such calculations is often related, for example, to data types in JScript. This is a big task to somehow connect two abstract variables, take a hash from them, send it to the blockchain, and when reviling, either transfer the original data, or already connected, and hope that the result of calculation the blockchain will be the same as on the client. This approach eliminates similar problems with the transfer format.

DensRoot

This is the main contract of the system. The purpose of this contract is to deploy auctions and NICs. It provides functions for calculating Auction addresses and NICs by domain names.

Key Public Interfaces

```
enum ContractType { NIC, Auction, Bid }
function updateImage(ContractType kind, TvmCell image) external override
```

The function is a callback for the Store. Required for obtaining and updating images of the smart contracts of the System.

```
struct RegBid {
  string name;
  uint hashAmount;
  uint16 duration;
  uint owner;
}
```

RegBid - a format for submitting data for registering a new name. It contains the name, sha256 hash from amount & salt, the desired registration duration in years, and the public key of the future certificate owner.

```
function registerNic(RegBid regBid) public
```

The function is used to apply for the registration of a new name. Resolves the NIC address and sends value to it to determine the existence of the NIC. Called only by internal messages with a value greater than 1 ton.

Getters

```
function resolve(string name) public view returns (address addrNic)
```

The function is used to calculate the NIC address based on the submitted name. It accepts all utf-8 characters except the dot.

The function can accept both first-level domains and domains of any nesting level as input parameters. If a subdomain is passed, such as google / translate/helloworld, the result of executing the function (addrNic) will be the address of the third-level NIC, and no matter what level of nesting you pass, the function will calculate the correct address of the NIC.

```
function resolveAuction(string name) public view returns (address addrAuction)
```

The function is used to calculate the Auction address based on the submitted name for that the auction was created. It accepts all utf-8 characters except the dot and slash. The function calculates only the first-level domains.

Functions for Testing

```
function forceRegisterNic(
       string name,
       uint owner,
       uint expiresAt
) public view returns (address addrNic)
```

Instantly registers the NIC bypassing the auction. By default, it is commented out, used for tests.

NIC

Important Dataset and Constructor Parameters

```
string static public _name;
address static public _deployer;
address static public _root;
```

_name — the name that the NIC is registered to _root — the DensRoot contract address is required for the refund of funds after the expiration of the certificate.

```
constructor(uint owner, uint expiresAt, TvmCell imageNic)
```

owner — the certificate owner public key
expiresAt — the timestamp to that the certificate is valid
imageNic — the contract code is used for subdomain deployment

Key Public Interfaces

```
function setTarget(address target) checkDestruct public
```

The function is used for setting the address that this domain points to. Can be called by the inbound external message only by the contract's owner.

```
function setOwner(uint owner) checkDestruct public view
```

The function is used for changing the contract owner. It is used if you need to transfer a subdomain to another person or, for example, when selling a domain. Can be called by the inbound external message only by the contract's owner.

```
function registerSub(string name, uint owner) checkDestruct public view returns
(address addrNic)
```

The function is used for registering a subdomain. Can be called by the inbound external message only by the contract's owner.

```
function destruct() public
```

The function is used for destroying a contract after it expires. When destroyed, all unspent funds are transferred to DensRoot and the address becomes writable again.

Getters

```
struct Whois {
  uint owner;
  address target;
  uint expiresAt;
}
function whois() public view returns (Whois _whois)
```

Auction

Important Dataset and Constructor Parameters

```
string static public name;
address static public deployer;
uint8 public _state = 0;
```

```
uint128 public _start;
uint128 public _reveal;
uint128 public _end;

uint128 public _maxAmount;
uint128 public _secondAmount;
address public _leader;

TvmCell _imageBid;
```

name — the domain name for that the auction is being held
_state — auction state (0 — accepts bids, 1 — bid disclosure stage, 2 — the auction is over)
_start, _reveal, _end — timestamps of state changes
_maxAmount — highest bid submitted
_secondAmount — second highest bid submitted
_leader — auction winner
_imageBid — bid contract code

Key Public Interfaces

```
function placeBid(uint hashAmount, uint owner) public view
```

The function is used to submit a bid. It accepts the bid owner (the potential domain owner) and the amount + salt hash. It deploys the Bid. It is called only by on-chain messages.

```
function revealBid(uint hashAmount, uint128 amount) external override
```

The function is used to perform the bid disclosure. The function only accepts messages from existing bids and only discloses the bid if the Auction is in the "1" state. The function calculates the leader, the maximum bid, the second highest bid and remembers it.

Getters

```
function resolveBid(uint hashAmount) public view returns (address addrBid)
```

Calculates the bid address by hashAmount.

Functions for Testing

```
function forceRevealState()
```

Instantly puts the Auction in the "1" state. By default, it is commented out, used for tests.

```
function forceEndState()
```

Instantly puts the Auction in the "2" state. By default, it is commented out, used for tests.

Bid

Important Dataset and Constructor Parameters

```
uint static public _hashAmount;
address static public _deployer;
uint public _owner;
uint128 public _amount;
```

_hashAmount — hash amount & salt. Additionally, it is used as a unique bid identifier
_owner — bid owner
_amount — the disclosed amount of the bid that is placed only after the accepted reveal

Key Public Interfaces

```
function reveal(uint128 amount, uint salt) public view
```

The function is used to disclose the bid. Accepts amount and salt, checks the original hash and the new one obtained by calling Base.hashAmountWithSalt. If the hashes are equal, it sends a message with the data to Auction.revealBid

```
function success(uint128 amount) external override
```

The callback that is called by the deployer to confirm the bid disclosure. The revealSuccess event will be released.

```
function success(uint128 amount) external override
```

A callback that is called by the deployer to notify about the rejection of the bid disclosure. The revealRejected event will be released.

```
event revealResolved(address addrBid);
```

Will be released if the bid is successfully disclosed.

```
event revealRejected(address addrBid);
```

It will be released if the bid is not successfully disclosed.

Store

The contract is used for storing system contract codes. It issues contract codes on request.

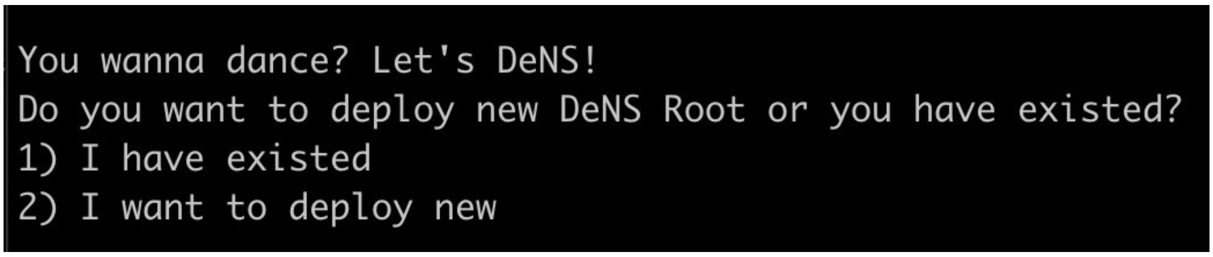
DensDebot

This is a Debot Contract for the System written according to the current specification.

DeBots

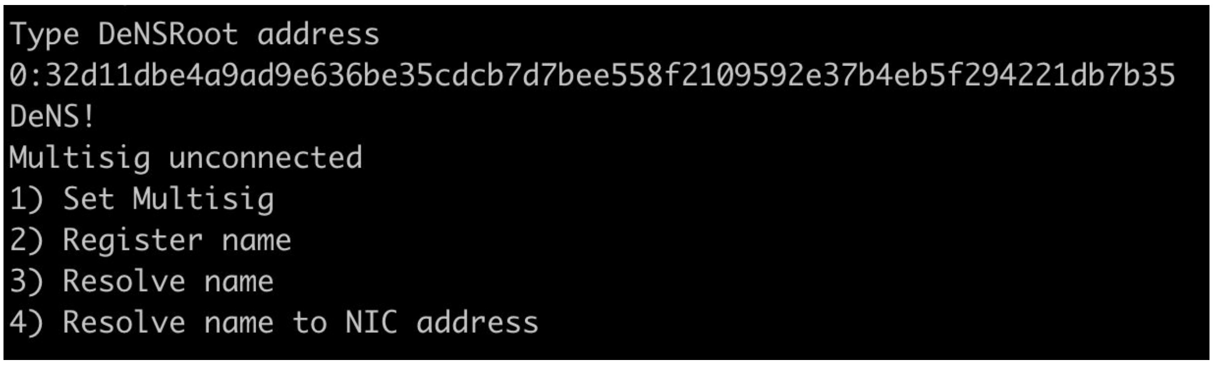
We wrote the DeBot according to the latest specification v0.8.1, that opens up new opportunities for development and significantly speeds up the development process.

1. Use tonos-cli (ver >= 0.8.1)
2. Debot available on devnet
   ./tonos-cli --url https://net.ton.dev debot fetch 0:fc8b51587bb69cd97078d6db2e73409678fffe3dfbc000f04dc433e4055c5fb4
3. After Debot starts you will see the start menu



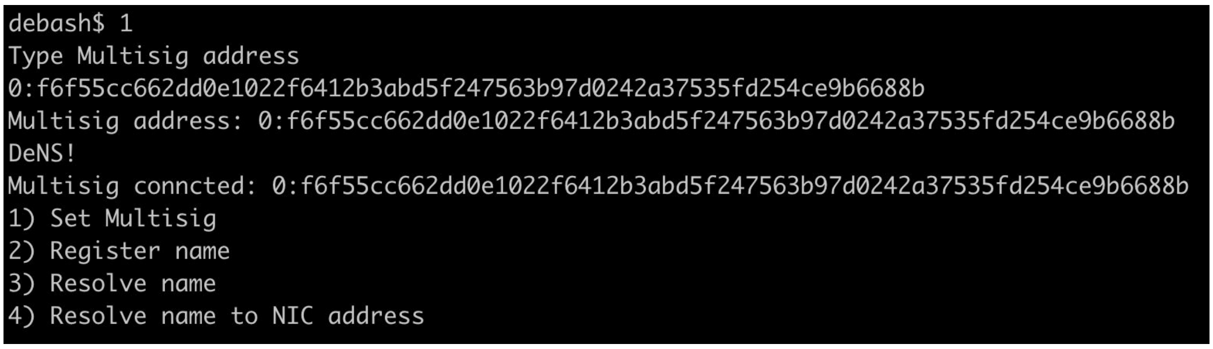
```
You wanna dance? Let's DeNS!
Do you want to deploy new DeNS Root or you have existed?
1) I have existed
2) I want to deploy new
```

4. Select existed DensRoot option and provide address. You will see main menu



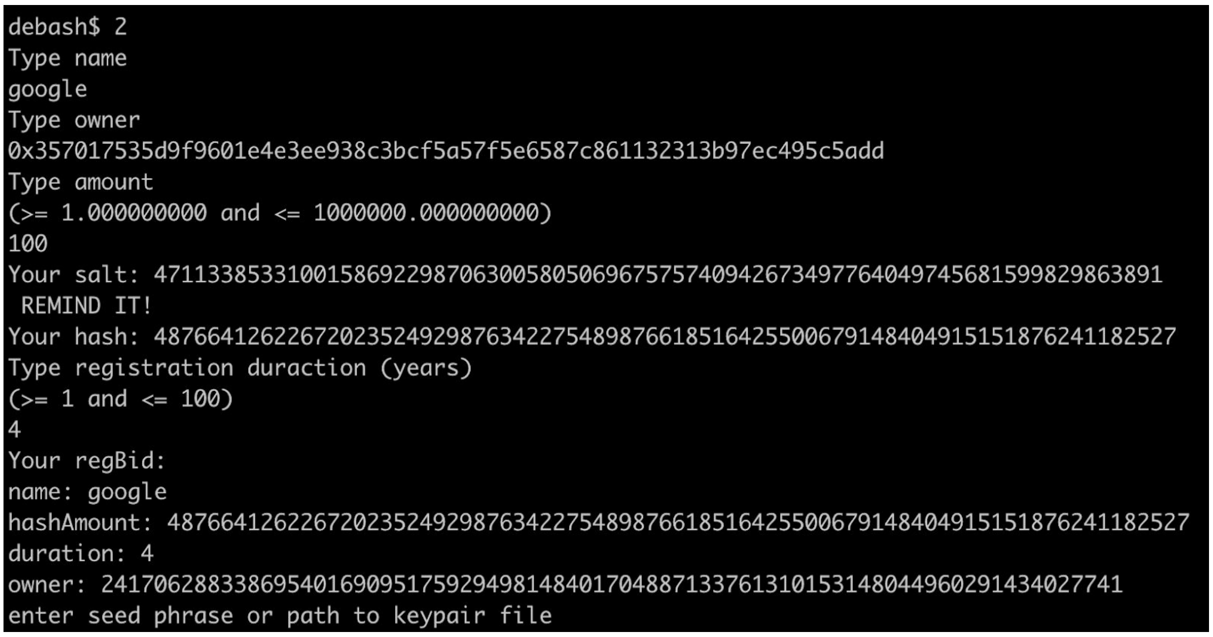
```
Type DeNSRoot address
0:32d11dbe4a9ad9e636be35cdcb7d7bee558f2109592e37b4eb5f294221db7b35
DeNS!
Multisig unconnected
1) Set Multisig
2) Register name
3) Resolve name
4) Resolve name to NIC address
```

5. To work with onchain calls, set up multisig. Now you can see that Multisig connected



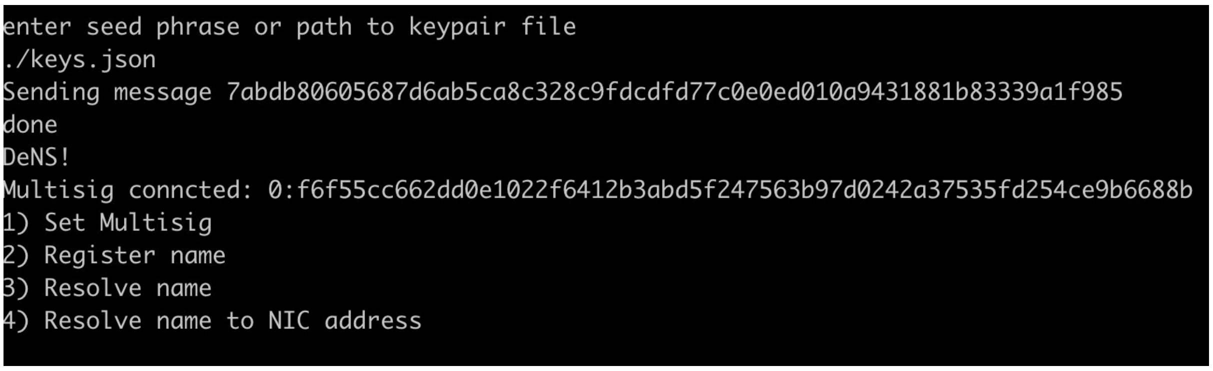
```
debash$ 1
Type Multisig address
0:f6f55cc662dd0e1022f6412b3abd5f247563b97d0242a37535fd254ce9b6688b
Multisig address: 0:f6f55cc662dd0e1022f6412b3abd5f247563b97d0242a37535fd254ce9b6688b
DeNS!
Multisig conncted: 0:f6f55cc662dd0e1022f6412b3abd5f247563b97d0242a37535fd254ce9b6688b
1) Set Multisig
2) Register name
3) Resolve name
4) Resolve name to NIC address
```

6. Try to register a new name. Provide some data and remember salt



```
debash$ 2
Type name
google
Type owner
0x357017535d9f9601e4e3ee938c3bcf5a57f5e6587c861132313b97ec495c5add
Type amount
(>= 1.000000000 and <= 1000000.000000000)
100
Your salt: 47113385331001586922987063005805069675757409426734977640497456815998298 63891
 REMIND IT!
Your hash: 48766412622672023524929876342275489876618516425500679148404915151876241182527
Type registration duraction (years)
(>= 1 and <= 100)
4
Your regBid:
name: google
hashAmount: 48766412622672023524929876342275489876618516425500679148404915151876241182527
duration: 4
owner: 24170628833869540169095175929498148401704887133761310153148044960291434027741
enter seed phrase or path to keypair file
```

7. Enter seedphrase of your Multisig to send register message throw your wallet



```
enter seed phrase or path to keypair file
./keys.json
Sending message 7abdb80605687d6ab5ca8c328c9fdcdfd77c0e0ed010a9431881b83339a1f985
done
DeNS!
Multisig conncted: 0:f6f55cc662dd0e1022f6412b3abd5f247563b97d0242a37535fd254ce9b6688b
1) Set Multisig
2) Register name
3) Resolve name
4) Resolve name to NIC address
```

Deploy and initialize the System

To start the system manually, you need to prepare:

- DensRoot contract
- Store contract
- image of NIC contract
- image of Auction contract
- image of Bid contract

System deployment script:

- Deploy Store
- Call Store.setAuctionImage and provide image of Auction contract
- Call Store.setBidImage and provide image of Bid contract
- Call Store.setNicImage and provide image of Nic contract
- Deploy DensRoot and provide the address of deployed Store
- The System is ready to use

A more detailed example can be found at ./tests/parts/deploy-store.ts ./tests/parts/deploy-dens-root.ts

Main use case

1. After the System is deployed and initialized, we can apply for registration of the name
2. Call DensRoot.registerNic, pass regBid with data
3. DensRoot calls Nic.checkExists c {bounce: true} and records the registration request
    - If Nic is already registered, then Nic will respond with DensRoot.existsCallback. DensRoot will issue the DensRoot.nicExists event and transfer information about the registered domain
    - If Nic is not registered then Dns Root message will return. Next, we will consider this particular scenario
4. DensRoot deploit Auction by passing regBid
5. Auction initializes and accepts the first Auction bid
6. Users can place their own closed bids by calling Auction.placeBid
7. Upon the expiration of the Auction, it goes into status "1"
8. Users can disclose their bid during the day by calling Bid.reveal with the initial bid data and attaching the number of ton specified in the bid to the message
9. When a bid is revealed, Auction records the highest bid, leader, and second highest bid
    - *This mechanism is necessary to ensure the operation of rates without registries.*
10. Upon expiration of the disclosure period, the Auction goes into state "2"
    - The auction takes the second highest registered bid from the winner's bid
    - The auction calls IDensRoot.auctionCompleted, passing information about the winning bid
    - Now unplayed bets can be returned
11. DensRoot NIC deployment
12. The winner can set target in the domain contract
13. The winner can change the domain owner
14. The winner can register a subdomain
    - Narpimer {name} / {subname}
    - When DensRoot.resolve ({name} / {subname}) DensRoot
        - Checks domain for valid characters
        - Checks domain for reserved names
        - Splits domains into chunks
        - For the first domain, consider the address where _deployer == DensRoot
        - For the second and subsequent subdomains, calculates the address where _deployer == the address of the previous domain / subdomain
        - Thus, it receives and returns the address of the final subdomain

Testing

All tests of the System are located in the *./tests* directory.

To run tests, it is needed to install Node and tondev package

1. Install node.js
2. Install docker
3. Install tondev. npm install -g tondev. If you encounter problems during installation, read the instructions in the official repository
4. Go to the project folder and install the dependencies with npm install
5. Create .env file at the root of the project and fill it in.
    - Available variables (this example is used to work with Node SE)
        - NSE_GIVER_ADDRESS=0:841288ed3b55d9cdafa806807f02a0ae0c169aa5edfe88a789a6482429756a94
        - NETWORK=LOCAL
6. Create ./bin folder
7. Create ./sbin folder
8. Create ./ton-packages folder
9. Place tonos-cli to ./bin folder if you want to play with debot (recommended)
10. Run Node SE tondev se start
11. Run npm run test:compile — it will compile all smartcontract and run all available tests