NNS Documentation

Yongxin Liu; Jianying Li

May 28, 2018

Contents:

1 NNS Background **1**
- 1.1 Why do We Need NNS? . . . 1
- 1.2 Relationship Between NNS and ENS . . . 2

2 NNS System Design Overview **3**
- 2.1 NNS System Functions . . . 3
- 2.2 NNS Architecture . . . 3
- 2.3 NNS Economic Model . . . 6
- 2.4 Domain Name Browser . . . 7
- 2.5 Reverse Resolution . . . 7
- 2.6 Roadmap . . . 7

3 Design Details **9**
- 3.1 NNS Protocol Specifications . . . 9
- 3.2 Detailed Explanation NameHash Algorithm . . . 10
- 3.3 Detailed Explanation of Top-level Domain Name . . . 11
- 3.4 Detailed Explanation of Owner Contract . . . 15
- 3.5 Detailed Explanation of Registrar . . . 16
- 3.6 Detailed Explanation of the Resolver . . . 16
- 3.7 NNS Domain Name Registration Mechanism . . . 17
- 3.8 Cyclically redistributed SGAS token Technology . . . 20

4 Summary **23**

5 Developers **25**

6 Indices and tables **27**

CHAPTER 1

NNS Background

NNS is the NEO Name Service, a distributed, open, and extensible naming system based on the NEO blockchain.

Our primary goal is to replace irregular string such as wallet address and smart contract Hash which are hard to memorize for humans with words and phrases. We will offer ending in ".neo" name service first.

Through name service, people don't need to remember address and Hash they don't understand anymore. You could make a transfer or use contracts by just knowing a word or phrase.

NNS can be used to resolve a wide variety of resources. The initial standard for NNS defines resolution for NEO addresses or Smart contracts(ScriptHash), but the system is extensible, allowing more resource types to be resolved in future without NNS upgrades.

1.1 Why do We Need NNS?

When Satoshi Nakamoto designed Bitcoin address, he created base58 encode by himself rather than adopting base 64 encode commonly used in coding community. In the base58 encode, he deleted some ambiguous characters: 0(zero), O(capital letter o), I(capital letter i) and l(lower case letter L).

This reflects Satoshi Nakamoto's consideration for the usability of blockchain address. However, blockchain address is still not human-friendly enough, because it's too long, hard to memorize and not easy to compare whether it's right or wrong. As blockchain's popularity increases, the shortcomings of its address will be more obvious.

As we won't use a 32-byte string as an E-mail account today, alias service could provide a huge help for the usability of blockchain system. As IPFS has its alias service IPNS, and Ethereum ENS, We argue that NEO system should have its own alias service. We call it as NEO Name Service(NNS), NEL community will increase the usability of NEO blockchain by providing NNS service.

The primary usage scenario of alias service is transfer of tokens via alias, especially for those accounts who need to make public their wallet addresses and do not need to change their addresses frequently. For example, when an ICO is underway, the project initiator need to make public its official wallet address on its official website.

But even the official wallet address is modified by hackers, it is difficult for investors to notice that. So if the project initiator could make public a short and easy-to-remember address alias, then even it's modified, it could be easily found, thus preventing wallet address from being modified by hackers. What types of resource an alias points to is extensible, as long as corresponding resolver is achieved.

Besides pointing to an account address, an alias could also point to a contract address, so smart contracts can be invoked via alias. There will probably be many smart contract templates, thus mistakes could be avoided if the alias service is used to invoke smart contract templates.

As blockchain is the infrastructure of next generation Internet, an increasing number of services will be based on blockchain. For example, the decentralized cloud storage service. File addressing is done through the file's hash value-the only identifier.

We could give a hash value an alias such as a file name that could be understood easily, then we map the alias to the file's hash value to achieve the file addressing. So alias service could be used together with NEOFS-the decentralized file storage based on NEO in future. NNS could also provide alias service for decentralized messaging, decentralized email service and so forth as more and more services are being built on NEO.

1.2 Relationship Between NNS and ENS

NNS and ENS share the same goal of increasing the usability of blockchain. But they are based on different blockchain platforms and serve different blockchain platforms.

We would like to extend our thanks to ENS, because we drew on experience from ENS when we designed NNS system, meanwhile we also made lots of innovative designs. For example, we split the owner contract from registry module to achieve more flexible ownership control.

We have two types of resolutions: quick resolution and complete resolution. We have introduced a smart token in our economic model to achieve redistribution of system costs.

CHAPTER 2

NNS System Design Overview

2.1 NNS System Functions

NNS system has two functions: first, to resolve a human-readable name like beautiful.neo into machine-readable identifiers like NEO's address; second, to provide descriptive data for domain names, such as whois, contract interface description and so forth.

NNS has similar goals to DNS. But based on blockchain architecture design, NNS is decentralized and serves blockchain network. NNS operates on a system of dot-separated hierarchical names called domains, with the owner of a domain having full control over the distribution of subdomains.

Top-level domains, like '.neo' and '.gas', are owned by smart contracts called registrars. One registrar manages one root domain name and specifies rules governing the allocation of their subdomains. Anyone may, by following the rules imposed by these registrar contracts, obtain ownership of a second-level domain for their own use.

2.2 NNS Architecture

NNS has four components: 1. Top-level domain name contract.(domain name root is the script that manages root domain name.)

2. The owner (the owner could be a personal account's address or a smart contract)
3. Registrar (A registrar is simply a smart contract that issues subdomains of that domain to users. The root domain name specifies a root domain name's registry.)
4. Resolver (responsible for resolving a domain name or its subdomain names.)

2.2.1 Top-level Domain Name Contract

Domain name root is the manager of all the information of a root domain name such as .test. No matter it is a second-level domain name like aa.test or third-level domain name like bbb.aa.test, both of their owners are kept in the domain name root which keeps the following data in the form of a dictionary.

1. The owner of the domain name
2. The registrar of the domain name
3. The resolver of the domain name
4. The TTL(time to live) of the domain name

2.2.2 Owner

The owner of the domain name could be either an account address or a smart contract. (ENS's design is a smart contract that owns a domain name is called registrar. Actually, registrar is the owner's exception. We separate the owner of the domain name from the registrar, making the system clearer.) Owners of domain names may:

1. Transfer the ownership of the domain name to another address.
2. Change registrar. The most common domain registrar is "administrators allocate subdomains manually."
3. Change the resolver.

A smart contract is allowed to be the owner, which provides a variety of ownership models.

1. The domain name co-owned by two persons. The transfer of domain names or changing registrars is only possible with two people's signatures.
2. The domain co-owned by multi-person. The transfer of domain names or changing registrars is only possible with more than 50% of people's signatures.

If the owner of the domain name is an account address, the user could invoke registrar's interface to manage second-level domain names.

2.2.3 Registrar

(ENS's design is a smart contract which owns top-level domains is called registrar. Actually, the registrar is the owner's special case. Our separation of the owner of the domain name from the registrar makes the system clearer. Most users don't sell their second-level domain names, so most users just need to configure a resolver rather than a registrar.)

A registrar is responsible for specifying subdomain names of a domain name to other owners. The registrar operates by invoking the domain root's script. The domain name root checks whether the registrar has the authority to operate this domain name. The registrar has two functions:

1. To re-specify subdomain names of a domain name to other owners.
2. To check whether the owner of a subdomain name is legal or not because second-level domain names could be transferred to others after third-level domain names are sold.

In the complete resolution, the registrar will be asked whether its subdomain names are allocated to specified owners. If not, the resolution is invalid. A registrar is a smart contract. There could be many types of registrars.

1. "First come, first served" registrar. Users are free to grab domain names. Our testnet's .test domain names will be registered in a "first come, first served" way.
2. Administers allocate registrars manually. An administrator sets up how the ownership of subdomain names is processed. Individuals with second-level domains allocate subdomain names manually.
3. Registrar auction. Testnet's .neo domains and mainnet's .neo domain names will be registered in the way of auction with collaterals.

2.2.4 Resolver

NNS's main function is to finish the mapping from the domain name to the resolver. The resolver is a smart contract which interprets names into addresses.

Any smart contracts which follow NNS resolver rules could be configured to resolvers. NNS will provide general-purpose resolvers. If new protocol types need to be added, the direct configuration can be done without changing NNS system if current NNS rules are not disrupted.

2.2.5 Resolution Rules

Domain Name Storage

The domain name NNS stores are 32-byte hashes, rather than the plain text of the domain name. Here are reasons for this design.

1. A unified process allows any length of the domain name.
2. It preserves the privacy of the domain name to some extent.
3. The algorithm for converting a domain name to a hash is called NameHash, and we'll explain it in other documentation.

The definition of NameHash is recursive.

1. for example aaa.neo corresponding

```
hashA  =  hash256(hash256(".neo") + "aaa")
```

2. then bbb.aaa.neo corresponding

```
hashB  =  hash256(hashA+"bbb")
```

3. then ccc.bbb.aaa.neo corresponding

```
HashC  =  hash256(hashB+"ccc")
```

This definition allows us to store all levels of domain names, level 1, level 2, to countless levels, in a data structure: Map <hash256, parser> in a flat way. This is exactly how the registrar saves the resolution of domain names.

This recursive calculation of NameHash can be expressed as a function:

```
Hash = NameHash ("xxx.xxx.xxx ...");
```

for the realization of NameHash, please refer to *Detailed Explanation NameHash Algorithm*.

Resolution Process

The user invokes the resolution function of the root domain name for resolution, and the root domain name provides both complete and quick resolution. You can invoke it as need. You can also query the resolver and invoke it by yourself.

Quick resolution

Quick resolution root domain name directly searches the resolver of a complete domain name. if not, search the parent domain name's resolver and then invokes the resolver for resolution. There are fewer operations for a quick resolution, but there's a flaw: the third-level domain name is sold to someone else and the resolver exists, but the second-level domain name has been transferred. At this point, the domain name can still be resolved.

Complete resolution

In a complete manner, the root of the domain name will start with the root domain name and queries ownership and TTL layer by layer. It will fail if they don't comply with.

More operations are needed in the complete resolution and operations has a linear growth with the layer number of domain names.

2.3 NNS Economic Model

Two kinds of tokens were introduced in the economic model of NNS. One is NNC, which is a UTXO asset and has a total supply of 1 billion. The other is SGS, which is a NEP5 token. It's bound with NEO's GAS at the ratio of 1:1 and they can be converted with each other.

The NNS root domain name is initiated by the NNC holder's voting. There are two modes of voting.

One is that the administrator initiates the root domain name voting, and the domain name is activated when less than 30% of the votes are against it in 3 days.

The other one is that any NNC holder starts the root domain name voting. If more than 50% of the votes are in favor of it within 3 days, the root domain name will be activated. Either way, voters or non-voters will be participants in the game. The GAS income from domain name registration will be redistributed to NNC holders.

2.3.1 NNC: an equity proof token

NNC is an equity proof token introduced into the NNS system. In order to sustain the system, NNS has introduced a fee redistribution system. The fees charged for all domain name auctions will be completely redistributed to NNC's holders.

The initial issuance of the NNC is in the form of airdrop(s). NNC will be only airdropped to NEO holders. Specific airdrop rules will be announced in the future.

2.3.2 SGAS-a kind of gas token

In order to facilitate the use of GAS in application contracts, NNS has issued a NEP5-based token with a total supply of 100 million, which is bound with NEO's GAS at the ratio of 1:1 and they can be converted with each other freely..

The GAS used for the redemption of SGAS will be stored in the account of the SGAS contract. NNS will not transfer or use this GAS. Therefore, it is guaranteed that as long as the user holds the SGAS, it can be converted to the same amount of GAS.

In the NNS system, SGAS mainly has the following functions:

It can be converted with GAS and vice-versa - Recharge/withdraw from the registrar. - Participate in domain auctions. - Auction fee payment

In addition to being used within the NNS system, since the SGAS itself is a NEP5 token system that is deployed on the Mainnet, all contract applications can also use this SGAS contract to perform convenient intra-contract GAS operations.

2.3.3 Bonus pools

When a user bids for a domain name, NNS will generate income from SGAS. There are two main sources:

1. The bid winner. If the user wins the bid and obtains the domain name ownership, then the bid winner will be charged all the bid fund as the fee.
2. The bid loser. For users who participate in the auction, but lose the bid, 5% of the bid is charged as a fee.

All fee income will be transferred to bonus pools. In the bonus pools, all NNC holders can receive SGAS in proportion to their NNC holdings.

2.4 Domain Name Browser

NNS domain name browser is the entrance which provides NNS domain name query, auction, transfer and other functions.

2.5 Reverse Resolution

NNS will support reverse resolution which will become an effective way to verify addresses and smart contracts.

2.6 Roadmap

First quarter, 2018

- January 2018, officially released NNS technical white paper
- January 2018, completed the technical principle test and verification
- January 31st, 2018, release the NNS Phase 1 testing service, including registrar and resolver, on the test net, anyone can register unregistered and rules-compliant domain names.
- February 2018, launch testnet-based Domain Name Browser V1

Second quarter, 2018

- March 2018, issue NNC on testnet.
- March 2018, release NNS Stage 2 testing service including bidding service on testnet when anyone can apply to NEL for NDS bidding test domain name
- April 2018, launch testnet-based domain name browser V2.
- May 2018, issue NNC on mainnet.
- June 2018, release NNS service on mainnet. Here comes Neo domain name era.
- June 2018, release mainnet-based domain name browser.

CHAPTER 3

Design Details

3.1 NNS Protocol Specifications

The URL we usually use on the Internet is as follows,

```
http://aaa.bbb.test/xxx
```

1. HTTP is a protocol, the domain name and protocol will be passed on separately when NNS service is requested.
2. aaa.bbb.test is the domain name, NNS service request using the hash of the domain name
3. Xxx is the path, the path is not processed at the DNS level, the same goes with nns, if there is a path, it will be processed in other ways.

Definitions of using strings in NNS protocol The following are tentative definitions.

HTTP

HTTP protocol points to a string, which means it's an Internet address.

addr

addr protocol points to a string, which means it's an NEO address. Like: AdzQq1DmnHq86yyDUkU3jKdHwLUe2MLAVv

script

script protocol points to a byte[], which means a NEO ScriptHash. Like: 0xf3b1c99910babe5c23d0b4fd0104ee84ffeec2a5

One and the same domain name is processed differently by different protocols.

- http://abc.test may point to http://www.163.com
- addr://abc.test may point to AdzQq1DmnHq86yyDUkU3jKdHwLUe2MLAVv
- script://abc.test may point to 0xf3b1c99910babe5c23d0b4fd0104ee84ffeec2a5

3.2 Detailed Explanation NameHash Algorithm

3.2.1 Namehash

The domain name NNS stores are 32byte hashes, rather than the plain text of the original domain name. Here are reasons for this design.

1. A unified process allows any length of the domain name.
2. It preserves the privacy of the domain name to some extent.
3. The algorithm for converting a domain name to a hash is called NameHash,

3.2.2 Domain, Domainarray, and Protocol

The URL we usually use on the Internet is as follows,

```
http://aaa.bbb.test/xxx
```

1. HTTP is a protocol, the domain name and protocol will be passed on separately when NNS service is requested.
2. aaa.bbb.test is the domain name, NNS service request using the hash of the domain name
3. Xxx is the path, the path is not processed at the DNS level, the same goes with nns, if there is a path, it will be processed in other ways.

NNS uses domain name's array rather domain name, which is a more direct process. Domain name aaa.bb.test is converted into byte array as ["test","bb","aa"] You could invoke the resolution this way

```
NNS.ResolveFull("http",["test","bb","aa"]);
```

And let the contract to calculate namehash.

3.2.3 NameHash Algorithm

NameHash algorithm is a way to calculate hash step by step after converting domain name into DomainArray. Its code is as follows:

```
// algorithm for converting domain names into a hash
static byte[] nameHash(string domain)
{
    return SmartContract.Sha256(domain.AsByteArray());
}
static byte[] nameHashSub(byte[] roothash, string subdomain)
{
    var domain = SmartContract.Sha256(subdomain.AsByteArray()).Concat(roothash);
    return SmartContract.Sha256(domain);
}
static byte[] nameHashArray(string[] domainarray)
{
    byte[] hash = nameHash(domainarray[0]);
    for (var i = 1; i < domainarray.Length; i++)
    {
        hash = nameHashSub(hash, domainarray[i]);
    }
```

(continues on next page)

(continued from previous page)

```
    return hash;
}
```

3.2.4 Quick Resolution

Complete resolution introduces the whole DomainArray and let smart contracts to check every layer's resolution one by one. Calculating NameHash could also be done on Client, and then is passed into smart contracts. It's invoked this way:

```
// query http://aaa.bbb.test
var hash = nameHashArray(["test","bbb"]);// can be calculated by Client
NNS.Resolve("http",hash,"aaa");// invoke smart contracts
```

or

```
//query http://bbb.test
var hash = nameHashArray(["test","bbb"]);// can be calculated by Client
NNS.Resolve("http",hash,"");// invoke smart contracts
```

You may be thinking why querying aaa.bbb.test is not like this.

```
// query http://aaa.bbb.test
var hash = nameHashArray(["test","bbb","aaa"]);// can be calculated by Client
NNS.Resolve("http",hash,"");// invoke smart contract
```

We have to consider whether aaa.bb.test has a separate resolver. If aaa.bb.test is sold to someone else, it specifies an independent resolver so that it can be queried. If aaa.bb.test does not have a separate resolver, it is resolved by bb.test's resolver. So this cannot be queried.

The first query, regardless of whether aaa.bb.test has an independent resolver, can be found.

3.3 Detailed Explanation of Top-level Domain Name

3.3.1 Function Signature of Top-level Domain Name Contracts

The function signature is as follows:

```
public static object Main(string method, object[] args)
```

Deploying adopts a configuration of parameter 0710, return value 05

3.3.2 Interface of Top-level Domain Name Contract

Top-level domain name's interface is composed of three parts Universal interface. It does not require permission verification and can be invoked by everyone. Owner interface. It is valid only when it's invoked by the owner signature or the owner script. Registrar interface. It's valid only when it's invoked by the registrar script.

3.3.3 Universal Interface

The universal interface doesn't need permission verification. Its code is as follows.

```
if (method == "rootName")
    return rootName();
if (method == "rootNameHash")
    return rootNameHash();
if (method == "getInfo")
    return getInfo((byte[])args[0]);
if (method == "nameHash")
    return nameHash((string)args[0]);
if (method == "nameHashSub")
    return nameHashSub((byte[])args[0], (string)args[1]);
if (method == "nameHashArray")
    return nameHashArray((string[])args[0]);
if (method == "resolve")
    return resolve((string)args[0], (byte[])args[1], (string)args[2]);
if (method == "resolveFull")
    return resolveFull((string)args[0], (string[])args[1]);
```

rootName()

Return the root domain name that the current top-level domain name corresponds to, its return value is a string.

rootNameHash()

Return NameHash the current top-level domain name corresponds to, its return values are byte[]

getInfo(byte[] namehash)

Return a domain name's information, its return value is an array as follows

```
[
    byte[] owner//owner
    byte[] register//registrar
    byte[] resolver//resolver
    BigInteger ttl//TTL
]
```

nameHash(string domain)

Convert a section of the domain name into NameHash. For example:

```
nameHash("test")
nameHash("abc")
```

Its return value is byte[]

nameHashSub(byte[] domainhash,string subdomain)

Calculate subdomain name's NameHash. For example:

```
var hash = nameHash("test");
var hashsub = nameHashSub(hash,"abc")// calculate abc.test's namehash
```

it's return value is byte[]

nameHashArray(string[] nameArray)

Calculate NameArray's NameHashaa.bb.cc.test corresponding nameArray is ["test","cc","bb","aa"]

```
var hash = nameHashArray(["test","cc","bb","aa"]);
```

resolve(string protocol,byte[] hash,string or int(0) subdomain)

resolve a domain name

The first parameter is a protocol

For example, HTTP maps a domain name to an Internet address.

For example, addr maps a domain name to an NEO address(which is probably the most common mapping)

The second parameter is the hash of the domain name that is to be resolved.

The third parameter is the subdomain name that is to be resolved.

The following code is applied.

```
var hash = nameHashArray(["test","cc","bb","aa"]);//calculate by Client
resolve("http",hash,0)//contract resolve http://aa.bb.cc.test
```

or

```
var hash = nameHashArray(["test","cc","bb");// calculate by Client
resolve("http",hash,"aa")//smart resolve http://aa.bb.cc.test
```

The return type is byte[], how to interpret byte[] is defined by different protocols. byte[] saves strings. We will write another document to explore protocols.

Second-level domain name has to be resolved in the way of

```
resolve("http",hash,0).
```

Other domain names are recommended to be resolved in the way of

```
resolve("http",hash,"aa").
```

resolveFull(string protocol,string[] nameArray)

Complete model of domain name resolution

The first parameter is protocol

The second parameter is NameArray

The only difference in this resolution is it verifies step by step whether the ownership is consistent with the registration.

Its return type is the same with resolve.

3.3.4 Owner Interface

All of the owner interfaces are in the form of

```
owner_SetXXX(byte[] srcowner,byte[] nnshash,byte[] xxx).
```

Xxx is scripthash.

The return value is one-byte array[0] means succeed; [1] means fail

The owner interface accepts both direct signature of account address calls and smart contract owner calls. If the owner is a smart contract, the owner should determine their own authority. If it does not meet the conditions, please do not initiate appcall on the top-level domain contract.

owner_SetOwner(byte[] srcowner,byte[] nnshash,byte[] newowner)

Ownership transfer of domain names. The owner of a domain name could be either an account address or a smart contract.

srcowner is only used to verify signature when the owner is an account address. It is the address's scripthash.

nnshash is the namehash of the domain name that is to be operated.

newowner is the scripthash of new owners' address.

owner_SetRegister(byte[] srcowner,byte[] nnshash,byte[] newregister)

Set up Domain Registrar Contract (Domain Registrar is a smart contract) Domain Registrar parameter form must also be 0710, return 05 the following interface must be achieved.

```
public static object Main(string method, object[] args)
{
    if (method == "getSubOwner")
        return getSubOwner((byte[])args[0], (string)args[1]);
    ...

    getSubOwner(byte[] nnshash,string subdomain)
```

Anyone can call the registrar's interface to check the owner of the subdomain.

There is no regulation for other interface forms of the domain name registrar. The official registrar will be explained in the future documentation.

The domain name registrar achieved by the user only need to achieve getSubOwner interface.

owner_SetResolve(byte[] srcowner,byte[] nnshash,byte[] newresolver)

Set up a domain name resolver contract (the domain name resolver is a smart contract)

The domain name resolver's parameter form also has to be 0710 and return 05

the following interface has to be achieved.

```
public static byte[] Main(string method, object[] args)
{
    if (method == "resolve")
        return resolve((string)args[0], (byte[])args[1]);
```

(continues on next page)

(continued from previous page)

```
    ...

resolve(string protocol,byte[] nnshash)
```

Anyone can call the resolver interface for resolution.

There are no regulations for other interface forms of domain name resolves. The official resolver will be explained in the future documentation.

The domain name registrar achieved by the user only need to achieve resolve interface.

3.3.5 Registrar Interface

There is only one registrar interface that's called by registrar smart contract.

register_SetSubdomainOwner(byte[] nnshash,string subdomain,byte[] newowner,BigInteger ttl)

register a subdomain name

nnshash is the namehash of the domain names that are to be operated.

subdomain is the subdomain name that is to be operated.

newowner is the scripthash of the new owner's address.

TTL is the time to live of the domain name(block height)

If succeed, return [1], if fail, return [0]

3.4 Detailed Explanation of Owner Contract

3.4.1 Workings of the Owner Contract

The owner contract calls the owner_SetXXX interface of top-level domain name contract in the form of Appcall.

```
[Appcall("dffbdd534a41dd4c56ba5ccba9dfaaf4f84e1362")]
static extern object rootCall(string method, object[] arr);
```

The top-level domain name contract will check the call stack, comparing contract it's called by and the owner that manages the top-level domain name contract. So only the owner contract of a domain name can manage this domain name.

3.4.2 Significance of the Owner Contract

Users could achieve complex contract ownership through the owner contract.

For example:

Owned by two persons, dual signature

Owned by more than two persons, operate by voting

3.5 Detailed Explanation of Registrar

3.5.1 Workings of Registrar Contract

The registrar contract calls register_SetSubdomainOwner interface of the top-level domain name in the form of Appcall.

```
[Appcall("dffbdd534a41dd4c56ba5ccba9dfaaf4f84e1362")]
static extern object rootCall(string method, object[] arr);
```

Top-level domain name contracts will check the call stack, comparing the contract it's called by and the registrar the top-level domain name contract manages.

So only the specified registrar contract can manage it. the registrar interface

The registrar's parameter form also has to be 0710 and return 05

```
public static object Main(string method, object[] args)
{
    if (method == "getSubOwner")
        return getSubOwner((byte[])args[0], (string)args[1]);
    if (method == "requestSubDomain")
        return requestSubDomain((byte[])args[0], (byte[])args[1], (string)args[2]);
    ...
```

getSubOwner(byte[] nnshash,string subdomain)

This interface is the norms and requirements of registrars. It has to be achieved because this interface will be invoked to verify rights when a complete resolution is conducted on the domain name.

nnshash is the hash of the domain name

subdomain is the subdomain name

Return byte[] owner's address, or blank

requestSubDomain(byte[] who,byte[] nnshash,string subdomain)

This interface will be used by first come first served registrar. Users call the interface of the registrar to register the domain name.

Who means who applies

nnshash means which domain name is applied

subdomain means subdomain name applied

3.6 Detailed Explanation of the Resolver

The workings of the resolver contract

1. The resolver saves resolution information by itself.
2. The top-level domain name contract calls the resolution interface of the resolver to get resolution information in the way of nep4.

3. When the resolver sets resolution data, it calls the getInfo interface of the top-level domain name contract to verify the ownership of the domain name in the way of Appcall.

```
[Appcall("dffbdd534a41dd4c56ba5ccba9dfaaf4f84e1362")]
static extern object rootCall(string method, object[] arr);
```

Any contract could call the getInfo interface of the top-level domain name contract to verify the ownership of the domain name in the way of Appcall.

3.6.1 Resolver Interface

The resolver's parameter form has be 0710, it returns 05.

```
public static byte[] Main(string method, object[] args)
{
    if (method == "resolve")
        return resolve((string)args[0], (byte[])args[1]);
    if (method == "setResolveData")
        return setResolveData((byte[])args[0], (byte[])args[1], (string)args[2],␣
↪(string)args[3], (byte[])args[4]);
    ...
```

resolve(string protocol,byte[] nnshash)

This interface is the norms and requirements of resolvers. It has to be achieved because this interface will be called for final resolution when a complete resolution is conducted on a domain name.

Protocol is the string of the protocol

Nnshash nnshash is the domains name that's to be resolved.

return byte[] is to resolve the data

setResolveData(byte[] owner,byte[] nnshash,string or int[0] subdomain,string protocol,byte[] data)

This interface is owned by the standard resolver for demo. The owner(currently it only supports the owner of an account address) could call this interface to configure the resolution data.

owner means the owner of a domain name.

nnshash means set up which domain name

subdomain the set-up subdomain name (could pass 0; if the set-up is domain name resolution, non-subdomain name passes 0)

protocol means the string of protocols

data means resolves data

Return [1] means succeed, or [0] means fail.

3.7 NNS Domain Name Registration Mechanism

The domain name is a scarce and unique resource. The core issue that needs to be considered when designing such a system is how to maximize the value of the domain name. This does not mean that the higher the price of the domain name is speculated, the better it is.

But there should be a reasonable process of value discovery and use. The first problem faced by the economic model of the domain name system is how to reasonably realize the initial registration and distribution of domain names?

3.7.1 Current Domain Name Registration

In the registration mechanism of blockchain domain names, there are currently two typical registration methods. One is first-come, first-served, such as a Bitshares; and the other is ENS's auction with sealed bidding registration. Let's first look at the specific steps as well as advantages and disadvantages of these two methods.

First come, first served is the main method of DNS domain registration. Service providers usually set different prices for different domain names. Due to the decentralized nature of blockchain domain names, there will be no complicated pricing in the first come, first served model. Either free or charge a unified registration fee. First come, first served method is relatively simple to implement, and the user operation is relatively simple. However, there is no market pricing process at the time of initial distribution, and there is no multi-participation. The domain name value needs to be discovered completely in the secondary market.

When there are few use cases for blockchain domain names, its value is not high and it is not used a lot, first come, first served method can be used as a convenient implementation, but when the demand for blockchain domain names increases, the first-come-first-served does not reflect the market demand well and does not maximize the value of domain names.

Ethereum's domain name service ENS adopts a sealed auction registration method. Its registration process is composed of bid opening, placing, revealing, and bid winning?. The whole process lasts about 7 days, and the main part is placing a bid and revealing a bid process. The blockchain is an open and transparent ledger system. In order to achieve sealed bidding, the user's bid consists of two parts, one is the real bid, and the other is the confusion money. Others can see your total bid in the system rather than your real bid to avoid the information advantage of later bidding. In the bid revealing stage, the user needs to send his own locally saved ciphertext to the smart contract to reveal the true bid. After the revealing period is over, the final bid winner will be determined.

The disadvantage of auctions with sealed bidding is that they cannot achieve a complete seal, and there will still be information leaks because the actual bid cannot exceed the total price. Second, the user experience is poor. The user needs to save the ciphertext and reveal the bid in the bid revealing period on time. Users need to do a lot of work, or it will lose the qualification and bid fund.

We believe that sealed bids are not completely sealed, and the auction experience is poor, so NNS hopes to find a better auction mechanism.

3.7.2 Transparent Bidding Mechanism

NNS will still use the bidding method to achieve the initial registration and distribution of domain names. However, unlike ENS, we adopt a transparent bidding mechanism. The advantage of this is that the user does not need to remember the ciphertext and there will be no bid revealing period. As long as the bidding is over, the final result can be known at once. However, there will be a problem with the transparent bidding. If the bidding period is certain, no one is willing to bid for the domain name at the beginning, because others can get a little more money to outbid you at the end of the auction. In order to solve the problem of the early bidding disadvantage, we introduce `randomness` at the end time of the auction. The auction is composed of two phases.

Fixed Period

The first phase is a fixed period, for example, 3 days. All bids during this period are valid.

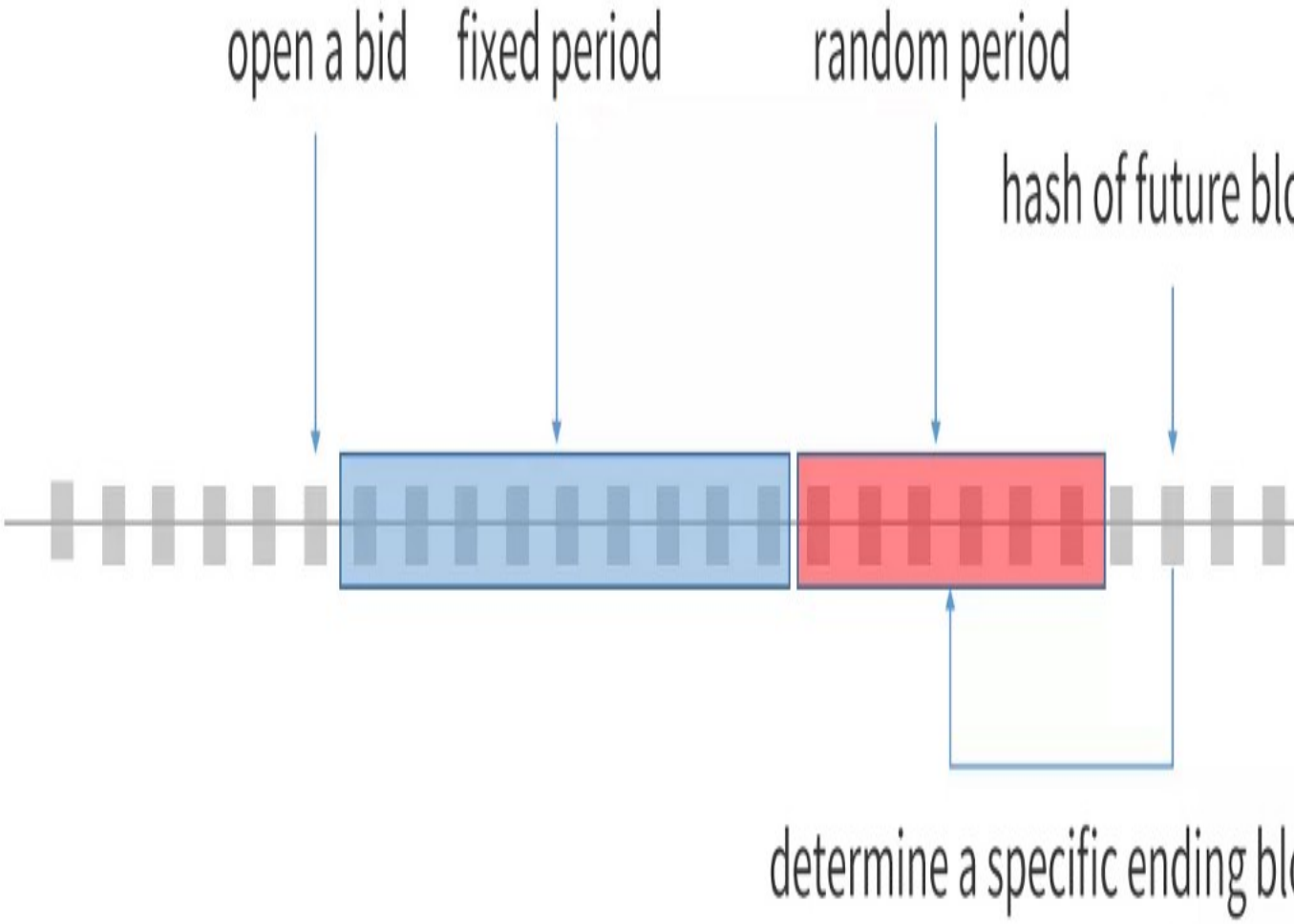
Random Period

If someone bids on the last day of the fixed period, then here comes an additional two-day random period, otherwise the auction ends on the third day of the fixed period. During the random period, the end time of the auction is uncertain.

It is necessary to wait until the hash value of the futures blocks of two additional days is determined. According to the size of the interval, the latter someone bids, the more likely he or she is to fall out of the end time of the auction and thus the bidding will be invalid, so it's better to bid as early as possible.

Ends of auction

After a random period is finished, the bid ending block will be determined according to the hash of future blocks. The bid winner can be determined after all the bids from the bid opening and the ending block are collected.



The end result of this auction method is that if you think that nobody bids against you for a domain name, then you only need to bid within two days after opening the bid, then you can win a domain name after the fixed period is

finished on the third day.

If there is competition for a domain name, then the competition mainly occurs at both ends of the alternating period of the fixed period and the random period. By introducing a random end time of the auction, the late bidding is the less likely to fall within the valid period, avoiding the problem of the late bidding advantage caused by increasing transparency in the auction.

3.7.3 Rent mechanism

The registration mechanism only achieves the value discovery at the time of initial distribution. If the domain name is acquired by someone but never used, it is a waste of value. Later, we will analyze how to use the rent mechanism to promote the circulation in the secondary domain name market.

3.8 Cyclically redistributed SGAS token Technology

In order to achieve the sound development of the NNS system, NNS introduces a cyclical token system, and introduces the equity tokens that provides redistribution proof of system income. The system is composed by three parts:

1. NNC Equity proof token.
2. SGAS NEP5 Token Contract.
3. Coinpool Bonus pool contract.

The main role of NNC is to provide the equity proof when the income from domain auctions is redistributed.

SGAS is a NEP5 asset that is bound to GAS based on a ratio of1:1 to facilitate the use of GAS in application contracts. CoinPool is a contract that receives domain name auction fees and performs redistribution management.

3.8.1 Redistribution mechanism

NNS charges a fee when the user participates in a domain auction. This fee income will be redistributed based on the NNS users' NNC holdings and will be completely redistributed to NNC holders.

3.8.2 Bonus claiming mechanism

The fee income of the NNS system will be transferred to the Bonus pool contract, which records the fee income. Users receive the bonus based on their NNC holdings via the bonus pool contract

3.8.3 Bonus pool details

All the SGAS that the domain name auction receives will be redistributed. The proportion of re-allocation is based on the users' NNC holdings. For cost considerations, we use 10,000 blocks as the unit to explain this and re-distribute the charged SGAS. When a user has an NNC utxo asset and uses it after at least two units, he will be able to receive the dividend from the system income that occurs within blocks when he is in possession of that NNC UTXO asset. Each NNC utxo asset can receive SGAS from the system income that occurs within no more than 15 blocks. If he holds the asset for more than 15 units, the dividend will be claimed in installments. That is to say, if the user has been holding this UTXO asset and hasn't sent it to another address, then he will always receive the SGAS dividend , but cannot claim it. He can claim the dividend only after the user sends this UTXO asset to another address.

SGAS interface (Only more interface than NEP5)

SGAS first meets NEP5 specifications, but the NEP specification interface is not described in details here.

mintTokens

Exchange GAS for an equal amount of SGAS, then transfer it to the SGAS contract account, and then invoke contract's mintTokens to transfer equal amount of SGAS to your account. It needs signing.

refund

The SGAS contract is used to convert the SGAS to the equivalent GAS. The SGAS contract checks the identity of the user and then constructs a contract transaction to transfer the GAS of the SGAS contract account to the user's account and deducts the SGAS in the user's account. This needs signing.

Migrate

Used for contract upgrades. It's used when there is a problem with the contract that requires an updated version. Only super administrators have permission to call it.

CoinPool interface

setSGASIn(byte[] txid)

It's called by the NNS registrar contract to transfer the SGAS income to the bonus pool, which needs to verify the contract address. It can only be called via the NNS registrar. The parameter txid is the transaction ID of the transaction.

countSGASOnBlock(BigInteger blockid)

Used to view the amount of bonuses saved in the specified block range. The parameter blockid is the block height of the query.

Claim

Used by users to claim the SGAS in the bonus pool.

CHAPTER 4

Summary

The NNS project is a smart contract protocol layer application built entirely on the NEO blockchain and is a true blockchain application. The combination of many projects and blockchain is just to issue a kind of token, the service of the token is provided by a central organization, while all services of the NNS are provided by the smart contract, which is distributed and flexible and extensible, without centralization risks.

NNS is a large-scale application of NEO smart contract system. In order to achieve resolver flexibility and scalability, we apply the latest NEP4 dynamic call. In the economic model, we will design the Vickrey auction contract and the Dutch auction contract. In order to achieve an easy redistribution of system costs, we've extended the NEP5 tokens standard and added the concept of coin days to allow system revenue redistribution without locking tokens, blending application tokens and equity into one kind of token.

As domain name services can improve the usability of the blockchain and have rich usage scenarios, there will form an ecosystem around domain names. In the future, we will work with NEO Eco-system C-lients to allow all NEO wallets to support the transfer of tokens via alias. We will also explore new usage scenarios such as working with pet games and giving pets nicknames via NNS. In the future, as the NEO ecosystem is used more and more, the domain name of NNS will become more and more valuable.

CHAPTER 5

Developers

Liu YongxinJason Liu

Founder Co-founder of NEL(a Chinese NEO developer community)

Li Jianying

CTO Co-founder & CTO NEO core developer, co-initiator of NEL(a Chinese NEO developer community)

Liu Qianming

Core developer, full-stack engineer

Zhao Ben

Web developer

Wang Xiangjian (Robbie Wang)

Overseas communities operation manager

CHAPTER 6

Indices and tables

- genindex
- modindex
- search