

Ethereum Improvement Proposals

All Core Networking Interface ERC Informational Meta

EIP 137: Ethereum Domain Name Service - Specification <>

Author	Nick Johnson
Status	Final
Type	Standards Track
Category	ERC
Created	2016-04-04

Abstract

This draft EIP describes the details of the Ethereum Name Service, a proposed protocol and ABI definition that provides flexible resolution of short, human-readable names to service and resource identifiers. This permits users and developers to refer to human-readable and easy to remember names, and permits those names to be updated as necessary when the underlying resource (contract, content-addressed data, etc) changes.

The goal of domain names is to provide stable, human-readable identifiers that can be used to specify network resources. In this way, users can enter a memorable string, such as 'vitalik.wallet' or 'www.mysite.swarm', and be directed to the appropriate resource. The mapping between names and resources may change over time, so a user may change wallets, a website may change hosts, or a swarm document may be updated to a new version, without the domain name changing. Further, a domain need not specify a single resource; different record types allow the same domain to reference different resources. For instance, a browser may resolve 'mysite.swarm' to the IP address of its server by fetching its A (address) record, while a mail client may resolve the same address to a mail server by fetching its MX (mail exchanger) record.

Motivation

Existing [specifications](#) and [implementations](#) for name resolution in Ethereum provide basic functionality, but suffer several shortcomings that will significantly limit their long-term usefulness:

- A single global namespace for all names with a single 'centralised' resolver.
- Limited or no support for delegation and sub-names/sub-domains.
- Only one record type, and no support for associating multiple copies of a record with a domain.
- Due to a single global implementation, no support for multiple different name allocation systems.
- Conflation of responsibilities: Name resolution, registration, and whois information.

Use-cases that these features would permit include:

- Support for subnames/sub-domains - eg, `live.mysite.tld` and `forum.mysite.tld`.
- Multiple services under a single name, such as a DApp hosted in Swarm, a Whisper address, and a mail server.
- Support for DNS record types, allowing blockchain hosting of 'legacy' names. This would permit an Ethereum client such as Mist to resolve the address of a traditional website, or the mail server for an email address, from a blockchain name.
- DNS gateways, exposing ENS domains via the Domain Name Service, providing easier means for legacy clients to resolve and connect to blockchain services.

The first two use-cases, in particular, can be observed everywhere on the present-day internet under DNS, and we believe them to be fundamental features of a name service that will continue to be useful as the Ethereum platform develops and matures.

The normative parts of this document does not specify an implementation of the proposed system; its purpose is to document a protocol that different resolver implementations can adhere to in order to facilitate consistent name resolution. An appendix provides sample implementations of resolver contracts and libraries, which should be treated as illustrative examples only.

Likewise, this document does not attempt to specify how domains should be registered or updated, or how systems can find the owner responsible for a given domain. Registration is the responsibility of registrars, and is a governance matter that will necessarily vary between top-level domains.

Updating of domain records can also be handled separately from resolution. Some systems, such as swarm, may require a well defined interface for updating domains, in which event we

anticipate the development of a standard for this.

Specification

Overview

The ENS system comprises three main parts:

- The ENS registry
- Resolvers
- Registrars

The registry is a single contract that provides a mapping from any registered name to the resolver responsible for it, and permits the owner of a name to set the resolver address, and to create subdomains, potentially with different owners to the parent domain.

Resolvers are responsible for performing resource lookups for a name - for instance, returning a contract address, a content hash, or IP address(es) as appropriate. The resolver specification, defined here and extended in other EIPs, defines what methods a resolver may implement to support resolving different types of records.

Registrars are responsible for allocating domain names to users of the system, and are the only entities capable of updating the ENS; the owner of a node in the ENS registry is its registrar. Registrars may be contracts or externally owned accounts, though it is expected that the root and top-level registrars, at a minimum, will be implemented as contracts.

Resolving a name in ENS is a two-step process. First, the ENS registry is called with the name to resolve, after hashing it using the procedure described below. If the record exists, the registry returns the address of its resolver. Then, the resolver is called, using the method appropriate to the resource being requested. The resolver then returns the desired result.

For example, suppose you wish to find the address of the token contract associated with 'beercoin.eth'. First, get the resolver:

```
var node = namehash("beercoin.eth");  
var resolver = ens.resolver(node);
```

Then, ask the resolver for the address for the contract:

```
var hash = resolver.addr(node);
```

Because the `namehash` procedure depends only on the name itself, this can be precomputed and inserted into a contract, removing the need for string manipulation, and permitting $O(1)$ lookup of ENS records regardless of the number of components in the raw name.

Name Syntax

ENS names must conform to the following syntax:

```
<domain> ::= <label> | <domain> "." <label>  
<label> ::= any valid string label per [UTS46](http://unicode.org/reports/tr46/)
```

In short, names consist of a series of dot-separated labels. Each label must be a valid normalised label as described in [UTS46](#) with the options `transitional=false` and `useSTD3AsciiRules=true`. For Javascript implementations, a [library](#) is available that normalises and checks names.

Note that while upper and lower case letters are allowed in names, the UTS46 normalisation process case-folds labels before hashing them, so two names with different case but identical spelling will produce the same namehash.

Labels and domains may be of any length, but for compatibility with legacy DNS, it is recommended that labels be restricted to no more than 64 characters each, and complete ENS names to no more than 255 characters. For the same reason, it is recommended that labels do not start or end with hyphens, or start with digits.

namehash algorithm

Before being used in ENS, names are hashed using the 'namehash' algorithm. This algorithm recursively hashes components of the name, producing a unique, fixed-length string for any valid input domain. The output of namehash is referred to as a 'node'.

Pseudocode for the namehash algorithm is as follows:

```
def namehash(name):  
    if name == '':  
        return '\0' * 32  
    else:  
        label, _, remainder = name.partition('.')  
        return sha3(namehash(remainder) + sha3(label))
```

Informally, the name is split into labels, each label is hashed. Then, starting with the last component, the previous output is concatenated with the label hash and hashed again. The first component is concatenated with 32 '0' bytes. Thus, 'mysite.swarm' is processed as follows:

```
node = '\0' * 32
node = sha3(node + sha3('swarm'))
node = sha3(node + sha3('mysite'))
```

Implementations should conform to the following test vectors for namehash:

```
namehash('') = 0x0000000000000000000000000000000000000000000000000000000000000000
namehash('eth') = 0x93cdeb708b7545dc668eb9280176169d1c33cfd8ed6f04690a0bcc88a93fc
namehash('foo.eth') = 0xde9b09fd7c5f901e23a3f19fecc54828e9c848539801e86591bd9801b
```

Registry specification

The ENS registry contract exposes the following functions:

```
function owner(bytes32 node) constant returns (address);
```

Returns the owner (registrar) of the specified node.

```
function resolver(bytes32 node) constant returns (address);
```

Returns the resolver for the specified node.

```
function ttl(bytes32 node) constant returns (uint64);
```

Returns the time-to-live (TTL) of the node; that is, the maximum duration for which a node's information may be cached.

```
function setOwner(bytes32 node, address owner);
```

Transfers ownership of a node to another registrar. This function may only be called by the current owner of `node`. A successful call to this function logs the event `Transfer(bytes32 indexed, address)`.

```
function setSubnodeOwner(bytes32 node, bytes32 label, address owner);
```

Creates a new node, `sha3(node, label)` and sets its owner to `owner`, or updates the node with a new owner if it already exists. This function may only be called by the current owner of `node`. A successful call to this function logs the event `NewOwner(bytes32 indexed, bytes32 indexed, address)`.

```
function setResolver(bytes32 node, address resolver);
```

Sets the resolver address for `node`. This function may only be called by the owner of `node`. A successful call to this function logs the event `NewResolver(bytes32 indexed, address)`.

```
function setTTL(bytes32 node, uint64 ttl);
```

Sets the TTL for a node. A node's TTL applies to the 'owner' and 'resolver' records in the registry, as well as to any information returned by the associated resolver.

Resolver specification

Resolvers may implement any subset of the record types specified here. Where a record types specification requires a resolver to provide multiple functions, the resolver **MUST** implement either all or none of them. Resolvers **MUST** specify a fallback function that throws.

Resolvers have one mandatory function:

```
function supportsInterface(bytes4 interfaceID) constant returns (bool)
```

The `supportsInterface` function is documented in [EIP 165](#), and returns true if the resolver implements the interface specified by the provided 4 byte identifier. An interface identifier consists of the XOR of the function signature hashes of the functions provided by that interface; in the degenerate case of single-function interfaces, it is simply equal to the signature hash of that function. If a resolver returns `true` for `supportsInterface()`, it must implement the functions specified in that interface.

`supportsInterface` must always return true for `0x01ffc9a7`, which is the interface ID of `supportsInterface` itself.

Currently standardised resolver interfaces are specified in the table below.

The following interfaces are defined:

Interface name	Interface hash	Specification
<code>addr</code>	0x3b3b57de	Contract address

Interface name	Interface hash	Specification
<code>name</code>	0x691f3431	#181
<code>ABI</code>	0x2203ab56	#205
<code>pubkey</code>	0xc8690233	#619

EIPs may define new interfaces to be added to this registry.

Contract Address Interface

Resolvers wishing to support contract address resources must provide the following function:

```
function addr(bytes32 node) constant returns (address);
```

If the resolver supports `addr` lookups but the requested node does not have a record, the resolver **MUST** return the zero address.

Clients resolving the `addr` record **MUST** check for a zero return value, and treat this in the same manner as a name that does not have a resolver specified - that is, refuse to send funds to or interact with the address. Failure to do this can result in users accidentally sending funds to the 0 address.

Changes to an address **MUST** trigger the following event:

```
event AddrChanged(bytes32 indexed node, address a);
```

Appendix A: Registry Implementation

```
contract ENS {
    struct Record {
        address owner;
        address resolver;
        uint64 ttl;
    }

    mapping(bytes32=>Record) records;

    event NewOwner(bytes32 indexed node, bytes32 indexed label, address owner);
    event Transfer(bytes32 indexed node, address owner);
```

```
event NewResolver(bytes32 indexed node, address resolver);

modifier only_owner(bytes32 node) {
    if(records[node].owner != msg.sender) throw;
    _
}

function ENS(address owner) {
    records[0].owner = owner;
}

function owner(bytes32 node) constant returns (address) {
    return records[node].owner;
}

function resolver(bytes32 node) constant returns (address) {
    return records[node].resolver;
}

function ttl(bytes32 node) constant returns (uint64) {
    return records[node].ttl;
}

function setOwner(bytes32 node, address owner) only_owner(node) {
    Transfer(node, owner);
    records[node].owner = owner;
}

function setSubnodeOwner(bytes32 node, bytes32 label, address owner) only_owner(node) {
    var subnode = sha3(node, label);
    NewOwner(node, label, owner);
    records[subnode].owner = owner;
}

function setResolver(bytes32 node, address resolver) only_owner(node) {
    NewResolver(node, resolver);
    records[node].resolver = resolver;
}

function setTTL(bytes32 node, uint64 ttl) only_owner(node) {
    NewTTL(node, ttl);
    records[node].ttl = ttl;
}
```

```
}  
}
```

Appendix B: Sample Resolver Implementations

Built-in resolver

The simplest possible resolver is a contract that acts as its own name resolver by implementing the contract address resource profile:

```
contract DoSomethingUseful {  
    // Other code  
  
    function addr(bytes32 node) constant returns (address) {  
        return this;  
    }  
  
    function supportsInterface(bytes4 interfaceID) constant returns (bool) {  
        return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;  
    }  
  
    function() {  
        throw;  
    }  
}
```

Such a contract can be inserted directly into the ENS registry, eliminating the need for a separate resolver contract in simple use-cases. However, the requirement to 'throw' on unknown function calls may interfere with normal operation of some types of contract.

Standalone resolver

A basic resolver that implements the contract address profile, and allows only its owner to update records:

```
contract Resolver {  
    event AddrChanged(bytes32 indexed node, address a);  
  
    address owner;  
    mapping(bytes32=>address) addresses;
```

```

modifier only_owner() {
    if(msg.sender != owner) throw;
    -
}

function Resolver() {
    owner = msg.sender;
}

function addr(bytes32 node) constant returns(address) {
    return addresses[node];
}

function setAddr(bytes32 node, address addr) only_owner {
    addresses[node] = addr;
    AddrChanged(node, addr);
}

function supportsInterface(bytes4 interfaceID) constant returns (bool) {
    return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;
}

function() {
    throw;
}
}

```

After deploying this contract, use it by updating the ENS registry to reference this contract for a name, then calling `setAddr()` with the same node to set the contract address it will resolve to.

Public resolver

Similar to the resolver above, this contract only supports the contract address profile, but uses the ENS registry to determine who should be allowed to update entries:

```

contract PublicResolver {
    event AddrChanged(bytes32 indexed node, address a);
    event ContentChanged(bytes32 indexed node, bytes32 hash);

    ENS ens;
    mapping(bytes32=>address) addresses;

    modifier only_owner(bytes32 node) {
        if(ens.owner(node) != msg.sender) throw;
    }
}

```

```

    -
}

function PublicResolver(address ensAddr) {
    ens = ENS(ensAddr);
}

function addr(bytes32 node) constant returns (address ret) {
    ret = addresses[node];
}

function setAddr(bytes32 node, address addr) only_owner(node) {
    addresses[node] = addr;
    AddrChanged(node, addr);
}

function supportsInterface(bytes4 interfaceID) constant returns (bool) {
    return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;
}

function() {
    throw;
}
}

```

Appendix C: Sample Registrar Implementation

This registrar allows users to register names at no cost if they are the first to request them.

```

contract FIFSRegistrar {
    ENS ens;
    bytes32 rootNode;

    function FIFSRegistrar(address ensAddr, bytes32 node) {
        ens = ENS(ensAddr);
        rootNode = node;
    }

    function register(bytes32 subnode, address owner) {
        var node = sha3(rootNode, subnode);
        var currentOwner = ens.owner(node);
        if(currentOwner != 0 && currentOwner != msg.sender)
            throw;
    }
}

```

```
    ens.setSubnodeOwner(rootNode, subnode, owner);  
  }  
}
```

Ethereum Improvement Proposals

Ethereum Improvement Proposals



[ethereum/EIPs](https://github.com/ethereum/EIPs)



[ethereum](https://twitter.com/ethereum)

Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards.