Prototype of subdomains implementation for Aeternity node

Aeternity Crypto Foundation
by
Karol Skočik
October 2019

Abstract In this paper we explain how an unusual, “Once and only” means to define a hierarchy in naming systems could be designed and implemented in the context of Aeternity node and summarize the results of this approach.

Motivation

Hierarchies are genuinely formed in nature and organizations, and ability to represent them straightforwardly increases the usefulness of any Naming System.

In our Naming System, we wanted to explore usability and implementation difficulty while respecting the limitations outlining the boundaries of the design space.

The limitations were either imposed by us or already present due to the way certain parts of the Naming System functionality is implemented. We were looking for a short implementation time, minimal impact or interference with already implemented preclaim/claim/update/revoke life cycle of top level names, and ability to represent hierarchy of names directly in Merkle-Patricia Tries only.

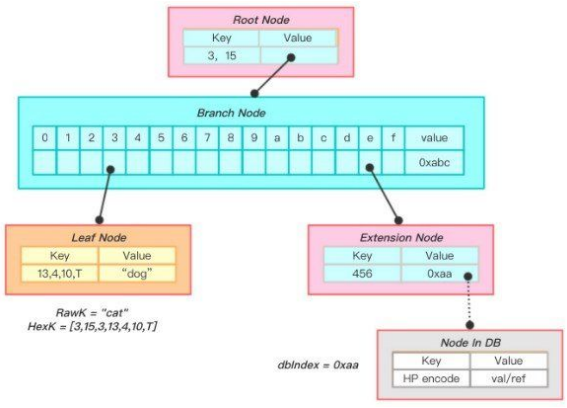
In the following text, we use the term name and domain interchangeably.

Design

The requirement to not interfere with already implemented functionality of top names created distinction between top names and subsequently defined subdomains.

While top name can be altered by end user incrementally issuing transactions at each phase of the top name lifecycle, subdomain can not be addressed on its own as a separate unit - it is always a part of a subdomain trie of a top name. Absence of means how to alter attributes of subdomains stems from inability to represent arbitrary parent-child relation with Merkle-Patricia Trie.

The (Patricia - prefix - part of the) trie itself does form a hierarchy amongst hashes of the identifiers (shown on Image 1.). To work with, and not against this feature of the trie, the solution was to project the membership of subdomains to a top name by constructing the hash of subname in a shape, which would also represent a hierarchy of hashes.



(1) Hierarchy of hashes

Lets label hashing function which can accept byte array (binary) of any length on input and outputs 32 bytes length output as *H*, and ++ as a concatenation operator joining two binaries together.

By *top* we mean top-name, *sub* labels subdomain.

Top name, when hashed, forms a 32 bytes binary:

Hash(top) = H(top)

Subdomain hash, forms 64 bytes:

Hash(sub) = H(top) ++ H(sub)

Since Merkle-Patricia Trie is also a prefix tree, it supports iteration over child nodes which share the same prefix of the hash. This iteration can be generalized to a fast mapping over a set of subdomains belonging to the same top name.

The elements of the set don't form a hierarchy amongst themselves, only towards the top name. If we wanted to introduce hierarchy amongst subdomains, we would have to split each subdomain to a reversed list of parts and construct hash recursively, e.g. for subdomain = "a.b.top" (omitting the registrar):

Hash(a.b.top) = H("top") ++ H("b") ++ H("a")

While this would open up the possibility to change attributes of subdomain incrementally, the downside is variable length of the hash. For a 5 level deep subdomain, the hash would take (5 + 1) * 32 = 192 bytes, and a malicious user could a lot of deeply nested subdomains causing inflation of the trie on purpose, occupying a lot of space.

Once and only vs incremental approach to define hierarchy

Incremental approach is common and widely used in naming systems. Without strong restrictions on the backing store, with ability to represent hierarchy directly, it is the best and most versatile approach. It allows end user to address subdomain directly when changing its properties.

Once and only approach is useful when there are no means to represent hierarchy, as it's sufficient to base it on *set membership relation*.

As the name suggests, the end user has to supply all subdomains with their pointers (objects subdomain resolves to) for a top name at once in a subname transaction.

All operations common for naming system - addition, removal or modification of any subdomain pointer, is done exclusively via subname transaction by supplying new subdomains.

Subname transaction

The execution of subname transaction progresses in several steps.

1. Check if top-name is present, and claimed (not in revoked state)
2. Mapping over present subdomains (if any) by using Hash(top) as prefix, removing all
3. Writing subdomains to MP Trie, using Hash(sub) for each subdomain. Hash(top) is a prefix of Hash(sub)

Example of subname transaction (using pseudo Erlang syntax) could look as follows:

```
% Shortcut of encoder function:
Enc   = fun aeser_api_encoder:encode/2
% Public key of the caller (32 bytes):
PK    = <<3,10,235,201,174,12,...>>
% Account Id from public key:
AccId = Enc(account_pubkey, PK).
% results to ak_2LioM...

% Top domain name, present in NS:
Top    = <<"top.test">>
TopId  = Enc(name, Top) % nm_3CSV...
% Subdomain sub1.top.test
Sub1   = <<"sub1">>
Sub1Id = Enc(subname, Sub1) % sn_c3…
% Subdomain sub2.sub1.top.test
Sub2   = <<"sub2.sub1">>
Sub2Id = Enc(subname, Sub2) %
sn_c3ViM...

% List of subdomains to define,
% both subdomains point to account of
the % caller under key labeled "acc"
Defs   = [
  #{<<"name_id">> => Sub1Id,
    <<"pointers">> =>
       [#{<<"key">> => <<"acc">>,
          <<"id">> => AccId}]},
    #{<<"name_id">> => Sub2Id,
      <<"pointers">> =>
         [#{<<"key">> => <<"acc">>,
            <<"id">> => AccId}]}
]

% Subname transaction
Tx = #{
  account_id => PK,
  Name_id    => TopId,
  definition => Defs,
  Fee        => 1000000000
 }
```

The **Tx** above, Erlang map, could then be JSON encoded and posted to **names/subname** endpoint, resulting in encoded Subname Transaction. After signing and posting of encoded Tx, we receive back a hash of transaction, and wait for its execution and modification of naming system Merkle-Patricia Trie.

Limitations

Slightly inconvenient usage of subname transaction (the need to supply all subnames) can be hidden by middleware or tools, which can provide the illusion of incremental updating of the subdomains tree for end user.

More serious limitation not straightforward to address easily is hitting the microblock limit.

The size of the serialized subname transaction depends on the number and length of each subdomain supplied as parameter to the transaction. This size is a significant contributor to the final *gas* cost of transaction.

With too many and/or too long subdomains defined in the transaction, it is easy to hit the microblock gas limit.

Experimentally, we have found that with top name set to "topname.test" (7 characters and registrar) and 6 characters long subdomains, we can define 68 subdomains in a subname transaction.

Since subdomains in this prototype are not first class entities as top names are, e.g. a single subdomain can not be addressed in isolation for changing its properties and doesn't follow the lifecycle like top names do, their utility is constrained to the main feature of naming systems - resolving a name to a value.

Any other advanced functionality (like transfer or revoke of subdomain) would

require lifting of the limitations we were constrained to work under.

Future work

To provide subdomains functionality which matches the features commonly used in widespread naming systems, we would have to lift the limitations.

Subdomain itself should be a first class entity, directly addressable and changeable in isolation.

Functionality like transfer or revoke affects all the descendants of top name or interim subdomains and as such to work as expected, it would require the ability to represent hierarchy, eg. parent - child relations conveniently.

The data store keeping these relations wouldn't have to be kept under consensus, which would free us to use data structure other than Merkle-Patricia Tries - hierarchy could then be directly reflected by distance from the root node.

Before committing to any particular implementation strategy, first we would need to ask questions like, how are subdomains going to be used? How does transfer work in the presence of subdomains, and others.

Conclusion

We have shown that it is possible to implement reasonably usable subdomains in naming system, without the ability to represent hierarchical relationships, via set membership relation simulated in Merkle-Patricia Tries - ubiquitous backing data structure in blockchains.

References:

Implementation:
https://github.com/aeternity/aeternity/tree/aens-subdomains
Protocol description:
https://github.com/aeternity/protocol/tree/aens-subdomains
Merkle-Patricia Trie:
https://github.com/ethereum/wiki/wiki/Patricia-Tree