AdaDomains White Paper

Website: adadomains.io

Twitter: @adadomains_io

15th August 2021

1 Introduction

Blockchain based naming systems evolved from the traditional Domain Name System (DNS). The main purpose of DNS is to translate human-readable domain names to corresponding IP addresses which identify resources on the Internet. This resolution is done by using domain name servers which are centralized and typically owned by some large organization that provides authority.

The goal of `AdaDomains` is to build a completely open and decentralized naming system on the Cardano blockchain. Our ecosystem will support resolution of domains, for example `hello.ada`, to different types of resources depending on the desired use case. If you want to make a crypto payment to the domain, it will be resolved to the owner's Cardano wallet address, but when searching for the domain in your browser it will instead be resolved to the website's content hash (using other distributed storage systems, such as IPFS). Domains can also be used to store other personal information to be publicly available on the blockchain, like contact information (email, Twitter handle, etc.) or other cryptocurrency addresses.

Since every domain is represented as an NFT (non-fungible token) it can be freely traded with other market participants. We intend to provide a marketplace to facilitate easier exchanges - for owners to list their domains for sale and for buyers to buy them.

2 Preliminaries

Before we dive into our project, we would first like to provide some necessary technical background.

2.1 Domains on Ethereum

Several domain based projects already exist on the Ethereum blockchain, most notably `Ethereum Name Service` [2] (ENS) and `Unstoppable Domains` [5]. Both of them use a very similar architecture for their blockchain naming systems. The main smart contract is called the *Registry*, which contains a mapping from domain names to their *Resolvers*. Each Resolver is another smart contract, which maps domain names to their actual records (for example Ethereum addresses or content hashes). Therefore, to resolve a domain, one first has to look up the correct Resolver in the Registry and then perform the actual resolution.

Since Ethereum often has periods of very high transaction traffic, which leads to extreme transactions fees, Ethereum based projects inevitably inherit these drawbacks. This was one of the main reasons why we chose Cardano as the platform for our project, because we can benefit from its low transaction fees and higher throughput.

2.2 Cardano's EUTXO Model

The most notable difference between the Cardano and the Ethereum blockchains is the accounting model used.

Ethereum uses account-based ledger model, which represents the state of the system as a collection of accounts (controlled by private keys or smart contracts), which hold balances of different assets. When a transaction happens, one (or more) of the sender's account balances are decreased and the corresponding receiver's account balances are increased. If the account is controlled by a smart contract, it can also store additional

data and implement functions to interact with it. This global state is constantly updated by every validator in the network.

Cardano instead uses UTXO (unspent transaction output) model, which can be thought of as a directed acyclic graph where nodes are transactions and edges are transaction outputs. Each transaction therefore consumes some UTXOs and produces new ones. A crucial fact is that every UTXO can only be spent once and it must be consumed entirely, i.e. it cannot be partially used by one transaction and the remainder later by another transaction. Global state is not stored, UTXOs are only representing local balances. The global balance can be calculated as a sum of all UTXOs controlled by a given address.

A common analogy is to think of an account-based model as a bank where you can transfer funds from one bank account to another. All your assets are pooled together into your total account balance and you can use this balance partially. For example, if you have 10 ETH, you can send 3 ETH to someone else and your new balance will be reduced to 7 ETH. On the other hand, a UTXO model is similar to cash where you have to spend a bill entirely and you possibly get some change in return. With our previous example, you would spend a 10 ADA bill and get back two bills, one for 2 ADA and one for 5 ADA (or just one bill for 7 ADA if it exists), thus creating new UTXOs.

Cardano extends the traditional UTXO model with the Extended UTXO model [1] (EUTXO). Each UTXO can have a *datum* field attached, which can be used to store data. Besides regular wallet addresses, UTXOs can now also reside at script addresses, which are controlled by *validator scripts*. These are Cardano's smart contracts that are run when a UTXO is being consumed to validate if it can be spent or not. A validator script has access to the entire spending transaction with all its inputs and outputs as well as additional user inputs called *redeemers* (one for each UTXO being spent). The script address can be determined from the hash of the validator script.

2.3 Tokens on Cardano

Every native token on Cardano is uniquely identified by two pieces of information – the *currency symbol* and the *token name*. Currency symbol is determined from the hash of the *minting policy script*, which is executed every time a transaction tries to mint new tokens. If the minting policy script runs successfully, new tokens are allowed to be minted, otherwise the transaction fails. Token name is used to differentiate between tokens coming from the same minting policy. If two tokens have the same minting policy and the same name, they are considered equal and completely interchangeable.

2.3.1 State Keeping

Tokens can be used to keep state which is needed in almost every decentralized application. The only way to store state on Cardano is to embed it in the datum field of a UTXO. This UTXO needs to be marked somehow so we know how to find it when we want to use it. We can do this by adding a unique token (an NFT) to the value of the UTXO. The NFT now acts as a proof of provenance for the state, i.e. the only correct state is the marked one.

When we want to access the state in a smart contract (to read or to change it), we need to spend the belonging UTXO. In the same transaction, we must also create a new UTXO with the same marking NFT containing the new state (which should be the same as the old state in case of read-only access). A system which constantly keeps an internal

state and transitions between states based on some logic (encoded in smart contracts) is called a *state machine*. State token UTXOs are therefore located at the respective script addresses which validate state transitions.

3 Project Architecture

We will first describe a simplified version of our project. In the next section, we will discuss additional mechanisms for scaling, which are crucial to the actual protocol implementation.

3.1 Simplified Version

The two main types of components of the protocol are the *registry* and individual *domains*. Both types are represented by tokens, each type has a different minting policy (and therefore also a different currency symbol). The registry is minted by a *one-shot policy*, meaning there will always exist exactly one token with this currency symbol. Its state carries a list of all already minted domains. Domain tokens keep state associated with each individual domain, for example, the owner, payment address, records (stored as key-value bytestring pairs), subdomains list, etc. The domain minting policy allows minting of new domains based on the registry's state. If a domain is not present in the registry's list of domains, the user is allowed to mint it. The registry and domains are located at their respective script addresses, which determine how their states can be changed, for example, the domain owner is allowed to change domain records but other users are not.

There is also a third component to the protocol, called the *configuration* (also represented by its own token). Its state stores administrative parameters and allows `AdaDomains` team to change them, like domain pricing in case of special offers or to offset ADA price volatility.

3.2 Example Transactions

To better illustrate how valid transactions are constructed from the protocol's components, we have outlined two examples. When we specify a token as an input or output UTXO, we actually mean a UTXO marked with that token (since all tokens are NFTs there should be exactly one such UTXO).

- Minting of a new domain `hello.ada`
  - **Input UTXOs**:
    - Registry token, spent with redeemer `Mint hello <user's wallet address>`
    - Configuration token, spent with redeemer `Mint`
    - User's UTXO with enough ADA for fees and domain price
  - **Output UTXOs**:
    - Registry token, sent to registry's validator script address
    - Configuration token, sent to configuration's validator script address
    - `hello` domain token, sent to domain's validator script address
    - UTXO with domain price in ADA, sent to `AdaDomains` wallet address

    - Possibly additional UTXOs to return excess ADA, sent to user's wallet address
  - **Requirements for successful transaction**:
    - `hello` is not included in input registry's domain list
    - Exactly one new domain token is minted and it has `hello` as its token name
    - Output registry's domain list is the same as input registry's domain list with an extra `hello` element added
    - Configuration's state must not be changed
    - `hello` domain's state has owner set to user's wallet address, its records and subdomains are empty
    - Domain price paid to `AdaDomains` wallet address is the same as specified in the configuration's state

- Changing `hello.ada` domain's payment address
  - **Input UTXOs**:
    - `hello` domain token, spent with redeemer `ChangePaymentAddress <payment wallet's address>`
    - User's UTXO with enough ADA for fees
  - **Output UTXOs**:
    - `hello` domain token, sent to domain's validator script address
    - Possibly additional UTXOs to return excess ADA, sent to user's wallet address
  - **Requirements for successful transaction**:
    - Transaction must be signed by input `hello` domain's owner
    - `hello` domain's state has payment address set to payment wallet's address, other parts of the state must not be changed

Other types of transactions, for example, updating domain records (like website's IPFS content hash or custom personal information), are similar to the second example above except that we are changing a different part of domain's state.

3.3 Problems

There are two main problems with minting domains as described in the first example of the previous section – concurrency and transaction size.

Since there is only one registry token and each UTXO can be spent at most once, only the first domain-minting transaction included in the current block can be valid. The following transactions in the same block which also try to spend the same registry token's UTXO will fail. This means that only one user can mint a new domain in each block. Blocks on Cardano are currently created roughly once every 20 seconds.

Another limitation is the transaction size – the current maximal transaction size limit is 16kB. Since registry's domain list is stored in the UTXO's datum field it has to be included in the transaction. Even if we assume that each domain (stored as a bytestring) uses only 10 bytes of space (each allowed character uses 1 byte, so we assume domains are at most 10 characters long), we could still only store approximately 1600 domains in total, which is obviously not enough.

4 Scaling

We propose two scaling solutions to help mitigate the shortcomings of the simplified protocol architecture.

4.1 Registry Splitting

To deal with limited transaction size, we will implement a simple idea – instead of having a single registry we will have multiple registries, each storing only a small subset of already minted domains. The main question then is which registry a certain domain belongs to?

Domains are stored on-chain as bytestrings. One way to assign a registry to a domain would be to look at the first byte (storing the first character of the domain name) and based on its value separate domains into registries. For example, the `hello` domain's first byte is 104 (corresponding to character `h` in UTF-8 encoding) so it should belong to registry 104. However, there are some drawbacks to this approach. The total number of registries is limited to only 256. Moreover, the distribution of domains into registries is highly skewed. Most registries will be empty because they represent bytes which do not correspond to allowed characters (lower-case letters, numbers and the hyphen) and even among allowed ones, some are more likely than others to appear at the start of a domain name (`q`, `x` and `z` are the least frequent letters in the English language [4]).

Our method improves on this approach by using two mechanisms – *name hashing* and *dynamic registry splitting*. Instead of working directly with domain names, we first use *SHA256* algorithm to hash them into bytestrings of fixed length 32. Since hashes are very randomized because they were designed to avoid collisions, the distribution of bytes in the resulting 32-length sequences is much more uniform. As an example, the byte sequence of the hash of `hello` starts as $44, 242, 77, 186, 95, \ldots, 152, 36$. Initially, there will be only 8 registries denoted by $0, 1, \ldots, 7$. To calculate the correct registry for a domain, we first apply integer division by 32 to its hash's byte sequence – for `hello` the result would be $1, 7, 2, 5, 2, \ldots, 4, 1$ (the first number is $44 \operatorname{div} 32 = 1$, the second $242 \operatorname{div} 32 = 7$, etc.). We call this the *registry sequence* of a domain because we can read the assigned registry from it. Since the first element is 1, the correct registry is also 1.

If the registry 1 gets too big after some other domains have been minted, we can perform a split. This means that we delete it (burn its token) and create 8 new registries denoted by $10, 11, \ldots, 17$. We also have to reassign all domains which previously belonged to registry 1. Since the first two elements of `hello`'s registry sequence are 1 and 7, we reassign it to registry 17. The splitting procedure ensures (by deleting the old registry) that there always exists exactly one registry which is *coherent* with a given domain's registry sequence, i.e. the sequence starts with the registry's name. For `hello`, all possible coherent registries are $1, 17, 172, 1725, 17252, \ldots, 172525501711657400020522301344 41$.

In a minting transaction, it is easy to verify whether the provided registry is correct for the desired domain. We only have to compute the registry sequence and check for coherence. The maximal number of registries is 8^{32} which is more than enough.

4.2 Pooling Layer

Having multiple registries already provides a part of the solution for dealing with concurrency issues. Since each registry belongs to its own UTXO, multiple users can mint

domains belonging to different registries at the same time (in the same block). The only problem remains if domains belong to the same registry.

To solve this we introduce a *pooling layer* to our protocol. Instead of minting each domain separately, users first have to submit their minting requests to the pool. This is done by locking the domain price value of ADA in a UTXO and sending in to a specific script address. These funds can be recalled by the user at any time before the domain is actually minted.

When a user wants to trigger the minting process, he has to do it for all the requested domains in the pool which belong to the same registry as his desired domain. This means that multiple domains from the same registry can be minted in one transaction and other users do not have to wait for their turn. It is important to note that the user who triggered the minting is not in any way disadvantaged because he will also mint other users' domains besides his own. On the contrary, he also has a financial incentive to do so since a part of every domain's price will be given to the user who performs the minting as a reward.

5 Use Cases

- **Simplified Payments**

  Instead of having to remember someone's Cardano wallet address, you can easily send them a payment just by knowing their domain name. Our platform interface will implement a smart contract which will resolve the chosen domain to the correct address on the blockchain and issue the payment.

- **Decentralized Websites**

  Decentralized websites are those that have their content (or the hash of the content) stored on the blockchain. To enable viewing of decentralized websites from a regular Internet browser (such as Google Chrome), we will provide a browser extension. It will communicate with the Cardano blockchain via a custom gateway and resolve `.ada` domains to their content. We will also build a service similar to *eth.link* [3] which will allow users to view decentralized websites without an extension by simply adding a suffix to their `.ada` domain.

- **Custom Information Storage**

  Users can store arbitrary custom information (in the form of key-value pairs) with their domain on the blockchain. This can be useful, for example, if they wish to provide publicly available contact information, such as their email, Twitter handle, etc.

- **Subdomains**

  Owners of domains can mint subdomains which offer even more ability for customization. For example, businesses can have several payment addresses and a unique subdomain linked to each one of them. For regular orders they might use `pay.business.ada` and for express delivery `express-pay.business.ada`. Another example of subdomain usage would be for displaying different website content, like `blog.ada` for blog posts and `contact.blog.ada` to show additional contact information.

5.1 Marketplace

The NFT collectors community on Cardano has been growing rapidly, so we know that trading (buying and selling) of NFTs is very important to a lot of our customers. That is why we will also build a decentralized marketplace for domains as part of our platform. We also believe that this will bring more engagement to our community and help with spreading the word about our project.

References

[1] Chakravarty, M., James Chapman, K. Mackenzie, Orestis Melkonian, M. P. Jones and P. Wadler. The Extended UTXO Model. Financial Cryptography Workshops, 2020.

[2] Ethereum Name Service, `https://docs.ens.domains`

[3] EthLink, `https://eth.link`

[4] Letter Frequency, `https://en.wikipedia.org/wiki/Letter_frequency`

[5] Unstoppable Domains, `https://docs.unstoppabledomains.com`