Traceability system based on the Ethereum blockchain, IPFS and the Hypercube DHT

Blockchain and Cryptocurrencies

Emmanuele Bollino, Liam James

a.y. 2022-2023
Alma Mater Studiorum - University of Bologna

Contents

1 Overview

This project aims to implement a traceability architecture with a search system based on three main components:

- Ethereum blockchain
- IPFS
- Hypercube DHT [Zichichi *et al.*]

The basic idea behind this architecture is to stray away from a standard client-server model as much as possible by leveraging on the main advantages about decentralization brought about by its components.

We imagine as a use case the one where a user wants to track a series of cars for which various information such as the brand, its colour and an image are stored by the system.

The Ethereum blockchain allows to host executable code in the form of Smart Contracts. Each contract represents a car and stores its brand, its colour and a link to the corresponding image on IPFS.

The Hypercube DHT is used to index the cars based on their details (keywords). This way, each vehicle can then be retrieved on the basis of its brand, its colour or both.

More information on the contracts and the Hypercube will be given in the next sections.

2 Use case

The present study focuses on automobiles as the primary subject of investigation. Specifically, the vehicles under consideration are delineated by their respective brand, color, and accompanying image. Noteworthy examples of feasible brands encompass Ferrari, Lamborghini, and Maserati, while the feasible color options consist of red and yellow. It is posited that each brand manufactures automobiles available in all the enumerated colors. Consequently, six distinct combinations arise from the intersection of brands and colors, with the assumption that these parameters are mandatory components of the entities.

3 Implementation

The actual implementation of the architecture [Section 3.1] is entirely done with the Python language. We exploited Docker in order to have more containerized environments and simulate on a single machine the deployment of multiple independent nodes. There is a container for each node of the Hypercube, one for the Ganache blockchain, one for IPFS, and another one for the interactive client [Section 3.4].

3.1 Architecture

The architectural framework employed in this project centers on a synthesis of cutting-edge technologies, namely the Ethereum blockchain, the IPFS (Inter-Planetary File System) protocol, and the Hypercube DHT (Distributed Hash Table). These components are strategically leveraged to fulfill distinct yet integral functions within the system.

Creation For each car entity a dedicated smart contract is deployed, facilitating the storage of pertinent attributes, encompassing the car's brand, color, and a reference to the image's location within IPFS. The instantiation of each car is achieved through the utilization of a factory contract, effectively managing the systematic generation of these entities. Subsequently, the resulting address of the newly created car entity is judiciously incorporated into the Hypercube DHT through a well-considered encoding mechanism of relevant keywords [Section 3.3]. This integration into Hypercube ensures streamlined and proficient search functionalities, enabling efficient access and retrieval of car entities based on their distinctive attributes [Image 1].

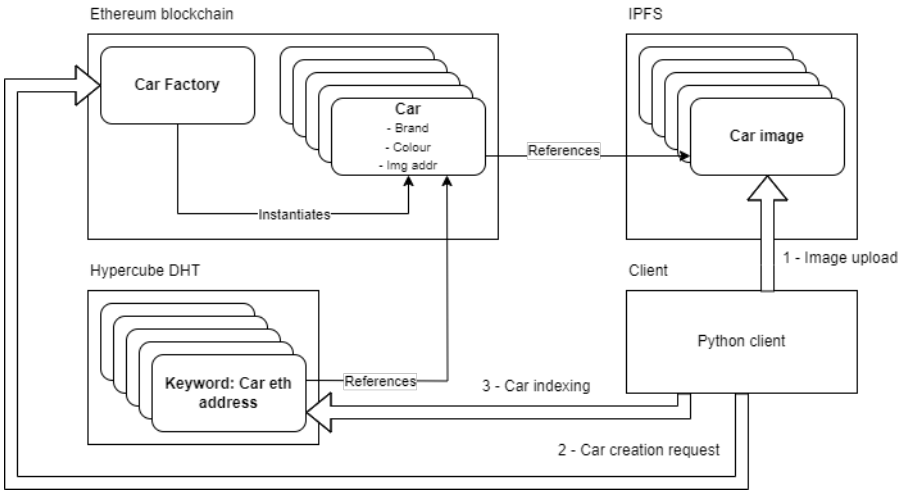

Figure 1: Architecture and creation of a car instance

Retrieval In order to retrieve the data of the cars, we use the search capabilities of Hypercube. The search is based through keywords, brand and colour in our case [Section 3.3]. Once we have the address of the desired car, we call the blockchain to retrieve the colour, brand and the image reference on IPFS. Finally, we download the image from IPFS and we have the whole information.

The process of data retrieval pertaining to the cars is executed by harnessing the search capabilities of Hypercube. This search mechanism is predicated upon

key parameters such as keywords, brand, and color [Section 3.3]. Upon obtaining the specific address corresponding to the desired car entity, a subsequent invocation is made to the blockchain to access comprehensive information encompassing the color, brand, and the reference to the image stored on IPFS. Subsequently, the image retrieval from IPFS culminates in the acquisition of the entirety of pertinent information related to the identified car entity. This methodical process ensures a systematic and effective means of accessing and assembling all relevant data components associated with the car entities.

3.2 Storage - Blockchain and IPFS

Smart contracts are suitable to store a tiny quantity of information, whereas the purpose of IPFS is to store a larger amount of data. That's why we use both technologies, Ethereum smart contracts to store basic information such as brand and colour, and IPFS to archive larger data like images.

Images The images associated with the cars are securely stored within IPFS, while their corresponding references are incorporated into the smart contracts governing the car entities. This strategic integration allows for the synergistic utilization of the distinct capabilities offered by both the Ethereum smart contracts and IPFS, thereby harnessing their full potential to ensure robust and efficient management of the car-related data.

Factory The deployment of new cars is demanded to a factory contract that is in charge of creating new cars, emitting events about new car creations and storing the plain addresses of the created cars. Notably, the factory is designed to be lightweight, lacking a comprehensive keyword-based retrieval system for car addresses. However, in pursuit of enhanced performance about gas consumption, a clone factory [`https://github.com/optionality/clone-factory`] has been devised to facilitate the efficient deployment of new car clones from an existing base car contract. To do so, we had to deploy a fictitious car to be cloned and add a method to the car contract to instantiate all the attributes. Through this systematic approach, the ability to replicate cars efficiently has been established.

3.3 Search - Hypercube

It is known that complete decentralisation provides important advantages in terms of data sovereignty and the absence of bottlenecks. However, one big downside of this kind of systems is the scarce efficiency of data look-ups.

The Hypercube DHT allows to cope with this issue by giving the possibility to implement complex queries without introducing centralised components.

The idea is to map each object (in our case, each car) to a keyword set depending on its details/keywords. In order to do so, a hash function is used so that each keyword set can be represented by a binary string where the 1s identify the

keywords included in the set. In other words, each keyword has a fixed position within the string. This means that a keyword set is simply a r-bit string where the position of each keyword is activated by a 1 in the corresponding position. These r-bit strings are then used to identify logical nodes in a DHT network.

From a topological point of view, such a DHT is composed by logical nodes connected to one another through edges. It can be denoted as $H_r(V, E)$ where V represents the set of vertices while E is used to refer to the edges. Finally, r identifies the dimensionality of the hypercube and it will be referred to as the hypercube size.

Each of the 2^r vertices, as already stated, represents a logical node while an edge is formed between two nodes when the binary strings that identify them differ by only one bit. Mathematically speaking, the distance between two nodes is computed using the Hamming distance. In summary, it is computed by performing a bit-by-bit exclusive OR operation on the binary strings representing the two nodes and then performing a sum on the resulting string.

Using this DHT, there are two kinds of searches that can be performed:

- the Pin Search: the aim is to find all objects associated exactly with the given keyword set
- the Super Set Search: this procedure also searches for objects that can be described by keyword sets which include the give one

In our case, we decided to adopt a hash function that performs a one hot encoding of the keywords turning them into r-bit strings where each keyword has its own position within the string. For example, with $r = 5$, the brand "Ferrari", one of the possible keywords, could be encoded as 00001 while the colour "red", another of the available keywords, could be encoded as 01000. At this point, the keyword set $\{Ferrari, Red\}$ would result in the combination of the two previous binary string through a bit-by-bit OR operation (i.e. 01001).

This approach has a few advantages:

1. there are no collisions meaning that every possible keyword set is uniquely identified by its r-bit string
2. considering the function *one* that associates to each keyword set the set of positions where a 1 appears within a r-bit string, it always holds that

$$one(K_1) \subset one(K_2) \tag{1}$$

   and that the resulting sets only differ by one element resulting in the corresponding logical nodes being neighbours (K_n refers to a set with cardinality equal to n). This allows the *Superset Search* to always yield the correct results whether only the brand or only the colour of the cars are specified. In fact, once a logical node whose ID corresponds to the keyword set passed to the request is reached, the request is forwarded to

all its neighbours (i.e. the nodes that are connect to this one through an edge).

For example, by performing a Superset Search using only the keyword "Ferrari" the idea is to gather all the cars whose brand corresponds to Ferrari. Once the request reaches the node who is responsible for the keyword set "Ferrari" (its ID and the binary string associated to "Ferrari" coincide) then the node outputs a list of all the objects it stores and the request to do so is forwarded to all its neighbours. Because of the one-hot-encoding, the nodes responsible for the coloured Ferraris (e.g. the node responsible for the objects whose keyword set is $\{Ferrari, Red\}$ and so on) return the lists of objects they store.

In order to interact with the Hypercube DHT we implemented a Python client [Section 3.4] that interacts with the server code provided by [Zichichi *et al.*] (see `https://github.com/AnaNSi-research/hypfs`).

3.4 CLI client

This client allows any user to interact with the system in a very intuitive way through an interactive command line interface by allowing to perform basic operations such as:

- Deploying a new factory or attaching an existing one by providing its address;
  - if a new factory is deployed the client gives a choice between the standard version or the more efficient clone version (see section 4 for more details).
- Creating a new Car smart contract by allowing to choose its brand and colour as well as its image among predefined options. Once the user makes his choices the contract is deployed to the blockchain and inserted into the Hypercube DHT;
- Removing a car from the Hypercube DHT;
- Performing a Pin Search by keyword: in this case the system makes the user pick both brand and the colour of the desired car;
- Performing a Superset Search by keyword (either brand or colour).

4 Tests

As mentioned in section 3.2 we implemented the factory pattern so that all the information regarding the cars flows through a unique contract (i.e. the factory) which keeps track of all the Car contracts deployed on the blockchain. In order to experiment on gas usages we decided to implement a standard version of this pattern using only the native Solidity and a more efficient one named

Clone Factory which was first proposed by the authors of `https://github.com/optionality/clone-factory`. According to the information provided by them the proposed Clone Factory should be much more efficient than the standard factory. This statement is confirmed by the results presented in table 1. However, to correctly instantiate the clone factory, it is needed to deploy before a base instance of the entity to be cloned.

Table 1: Gas consumption

	Std factory	Clone factory	No factory
Deploy	1642604	777180 *(1426973 with base car)*	-
Car creation	693682	201012	649793

It is clear to see that the Clone Factory clearly improves on its standard counterpart with respect to gas usages, both in the deployment of the factory contract and on the creation of a new car.

5 Further developments

To push this project forward, it is possible to improve some aspects of it. First of all, the keyword encoding strategy can be revised also by making some common patterns on the basis of the use case. Then, it is possible to design a fancy frontend for the interaction with the whole system architecture. Finally, it is possible to improve the Hypercube implementation and design.

5.1 Hypercube improvement proposal

One inherent limitation of the Hypercube Distributed Hash Table (DHT) is its assumption that all nodes are intended to index information within the same context. Consequently, any deviation from this context uniformity can result in data mixing, making it challenging to discern data in a straightforward manner. Additionally, a context employing a keyword encoding strategy featuring inefficiencies or imbalanced utilization may lead to a significant waste of resources.

To mitigate these issues, we propose an enhanced version of the Hypercube DHT, emphasizing the recognition and segmentation of distinct contexts. Currently, each node within the Hypercube is tasked with storing a set of strings representing the data associated with the managed keyword. By transforming this data structure into a hash table, we can organize and store data associated with various contexts in a segmented manner. Here, the hash table's keys serve as identifiers for specific contexts, while the values correspond to the sets of data associated with the respective contexts, linked to the keyword managed by the node.

This enhanced approach can be further extended to a multi-user framework, wherein each user serves as an independent context. By leveraging Ethereum's

signed messages EIP-1271, we facilitate the establishment of a public Hypercube DHT, accommodating multiple users operating independently. In this scenario, the key of the aforementioned hash table becomes the user's address. Each request directed to the Hypercube DHT necessitates a signed message for user validation, allowing the Hypercube to autonomously decode and securely associate the sender's address with the request.

In conclusion, this refined approach empowers the utilization of the Hypercube DHT for diverse purposes, accommodating multiple users without compromising the integrity of data associated with specific keywords or contexts. This advancement represents a promising step towards a more versatile and efficient Hypercube DHT system.

References

[Zichichi *et al.*] Mirko Zichichi, Luca Serena, Stefano Ferretti, Gabriele D'Angelo, *Complex queries over decentralised systems for geodata retrieval*, IET Networks, 2022.