KDNA and its application in Karma Distribution

Technical Paper

Version 1.0

April, 2019

www.kambria.io **info@kambria.io**

Abstract

Tracking dependencies of a codebase is popular in a package management system, but tracking contribution of a source code to another source code has not been explored. In this paper, we introduce to you the KDNA concept. You will learn how we utilize RDF & IPFS to implement KDNA, how we use it to track contribution of a codebase to another codebase, and how we apply it to solve the Karma distribution problem that was given to you in the Kambria Karma Model Paper.

1. Introduction

Kambria is a collaborative ecosystem with the goal of dramatically accelerating the development and adoption of the world's most advanced robotic technologies. Kambria's mission is to solve the challenges that hinders the speed of robotics development. Challenges include:

- Many engineering domains involved in robot design (mechanical, electrical, etc) lack good tools for collaboration and sharing.
- Missing semantic linkages across systems, including software design to electrical engineering, and electrical engineering to mechanical engineering.
- Currently, the development materials are a chaotic tangle of repos, directories, sheets of BOM parts, text assembly instructions, slicer settings, and supplier names in scattered local servers and cloud storages.

Kambria offers a platform so that participants can collaborate easier, and provides a mechanism called KDNA to help describe the codebases better, while linking them together. People are encouraged to register their codebases on the platform to create a KDNA file for their codebases. The role of KDNA file is to store codebase information like authors, description, BOM, and to store the link between codebases, so that when someone wants to build a product from that codebase, they will know what other components they need, and how to construct all the elements to make a product. More importantly, KDNA helps record the contribution of one codebase into another codebase so that we can distribute Kambria Karma to all relevant contributors.

KDNA files will be implemented in RDF format. We then publish theses files on IPFS so that it can be seen by everyone, and every change on this KDNA file will be tracked and verified so that no one can tamper with it.

2. Background

2.1. RDF

RDF was born in 1997 as a standard model for the Semantic Web. As described by the W3C, “RDF was designed to provide a common way to describe information so it can be read and understood by computer applications.” With HTML or normal text format, computers require a way to understand the content of a blog post or an article on Wikipedia. So, the idea of RDF is that it breaks a sentence into “triples” (called Subject, Predicate, Object), and each component of the “triples” has a link to its meaning.

Many formats for RDF has been invented: RDF/XML, RDF/JSON, N-triples, and Turtle. RDF is now used to create knowledge based models. From time to time, each field has developed its own terminology to encourage interchanging data.

RDF now is used as a standard to store semantic, knowledge base data. It has its own query language like SPARQL to communicate with RDF data. So if we use RDF as KDNA format, it may become standard for codebase sharing. We can allow other organizations to utilize it to present their data, while simultaneously linking all data together. We can easily query the KDNA data, build out graphs, and much more!

2.2. IPFS

IPFS began as an effort by Juan Benet to build a system that can rapidly move versioned scientific data. IPFS creates a joined network of computer resources that store files. Any computer can take part in this network, and any file that is added to this network can be accessed by any computer inside the network in a peer-to-peer manner. When you add a file to IPFS network, it will hash the content of the file and give the hash back to you. This hash is the key for you to access the file. Files with different content will have different hashes. Files with the same content will have the same hash. It means each file content will have a unique link to it (de-duplicating property). So when you add a KDNA file to IPFS network, it can be stored publicly. Anyone who has the hash can access the file but no one can tamper with it because every change you make to the content will generate a different hash. When you access the file with the previous hash, it will still give you the previous content. IPFS synthesizes many of the best ideas from Git and Bittorrent. It also has a version control system in its design, but the “commit” feature is still in development. Currently, we can still track the versions of the file by storing a list of hashes corresponding to each file change ourselves.

3. Implementation

In this section, you will learn how we use RDF and IPFS to implement KDNA concept.

3.1. KDNA schema

The first step to utilize RDF is to create a schema to represent the information and relationships of the codebase. This is the information that should be included in KDNA:

- Basic information
- Links to related codebases it utilizes
- Links to previous KDNA versions of this codebase
- Address of codebase smart contract
- Information describing the assets included in the codebase (the BOM, 3D Model…)

Then we specify the vocabulary needed to convey the information (the properties of KDNA):

- *name*: the name of codebase
- *description*: the description of codebase
- *creator*: the creator of codebase
- *repository*: the link to the git repository of codebase
- *relation***:** a list of codebases relating to this codebase
- *parts*: a list of the links to other codebases that are used by this codebase, and the measure of how much each of those codebases contributed to this codebase
- *ethAddress***:** address of codebase smart contract on Ethereum network
- *previous_version***:** the link to the previous KDNA version of this codebase
- *contents*: a list of assets included in the codebase

To describe the assets in *contents* property; we use those properties:

- *semantic_type***:** semantic type of the content
- *extension***:** the file extension of the content
- *path***:** path to the folder that stores the content

The ‘link’ in *repository* property is just a URL. The ‘link’ in *parts***,** *previous_version* is in fact the IPFS hashes of KDNA files of other codebases. For the *parts* property, it is a list of participants that contribute to the codebase along with the weight numbers, so each participant will have two properties: *hash* & *weight***.**

We also follow some rules when designing terms for our KDNA schema to make sure we do not waste time “reinventing the wheel,” and our schema will be processed easier by many machines. Each term must meet these requirements:

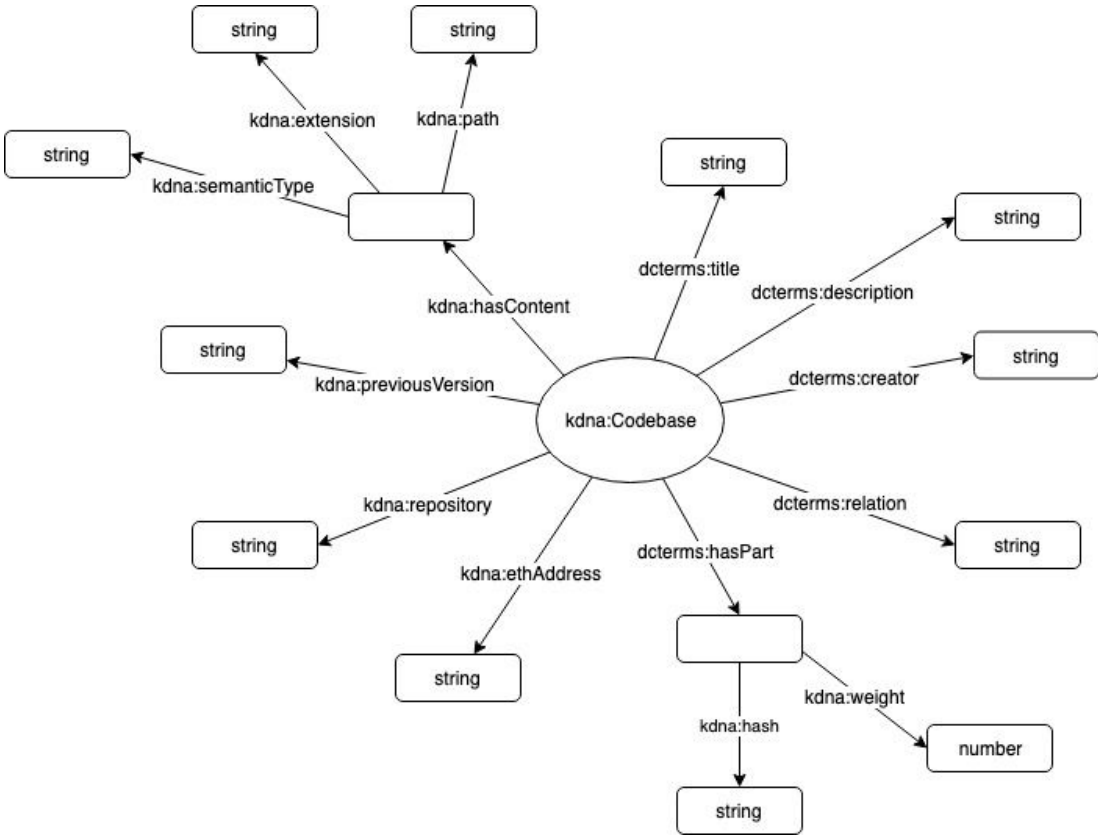
- Does not replicate existing, widely used terms
- Creates sub-classes, sub-properties, and super-classes where appropriate
- Does not accidentally add new semantics to existing terms
- Offers well-defined terms with well-designed, persistent URIs
- Is published in multiple formats for consumption by humans and machines
- Is likely to remain stable for the long-term
- Is discoverable

To meet the requirements, we need to research existing terms, and their usage, in order to maximise the re-use of those terms. We researched many popular vocabularies like “Friend Of A Friend”, “Dublin Core“, Schema.org, and we found Dublin Core (by Dublin Core Metadata Initiative) has defined many terms that we can reuse:

- Property *name* replaced with *dcterms:title*
- Property *description* replaced with *dcterms:description*
- Property *creator* replaced with *dcterms:creator*
- Property *relation* replaced with *dcterms:relation*
- Property *parts* replaced with *dcterms:hasPart*

With the properties that we cannot find popular terms for, we define it ourselves. Those new terms are: *kdna:repository, kdna:previousVersion, kdna:ethAddress, kdna:hasContent, kdna:semanticType, kdna:extension, kdna:path, kdna:hash, kdna:weight***.** They are properties of *kdna***:***Codebase* class.

All of the terms in our schema can be demonstrated in this graph.



These terms are officially defined in http://kdna.kambria.io/kdna-schema

Now we can use this schema to create KDNA for every codebase. We choose the Turtle format to represent the data because of its terse and triple-like style. This is a sample KDNA file:

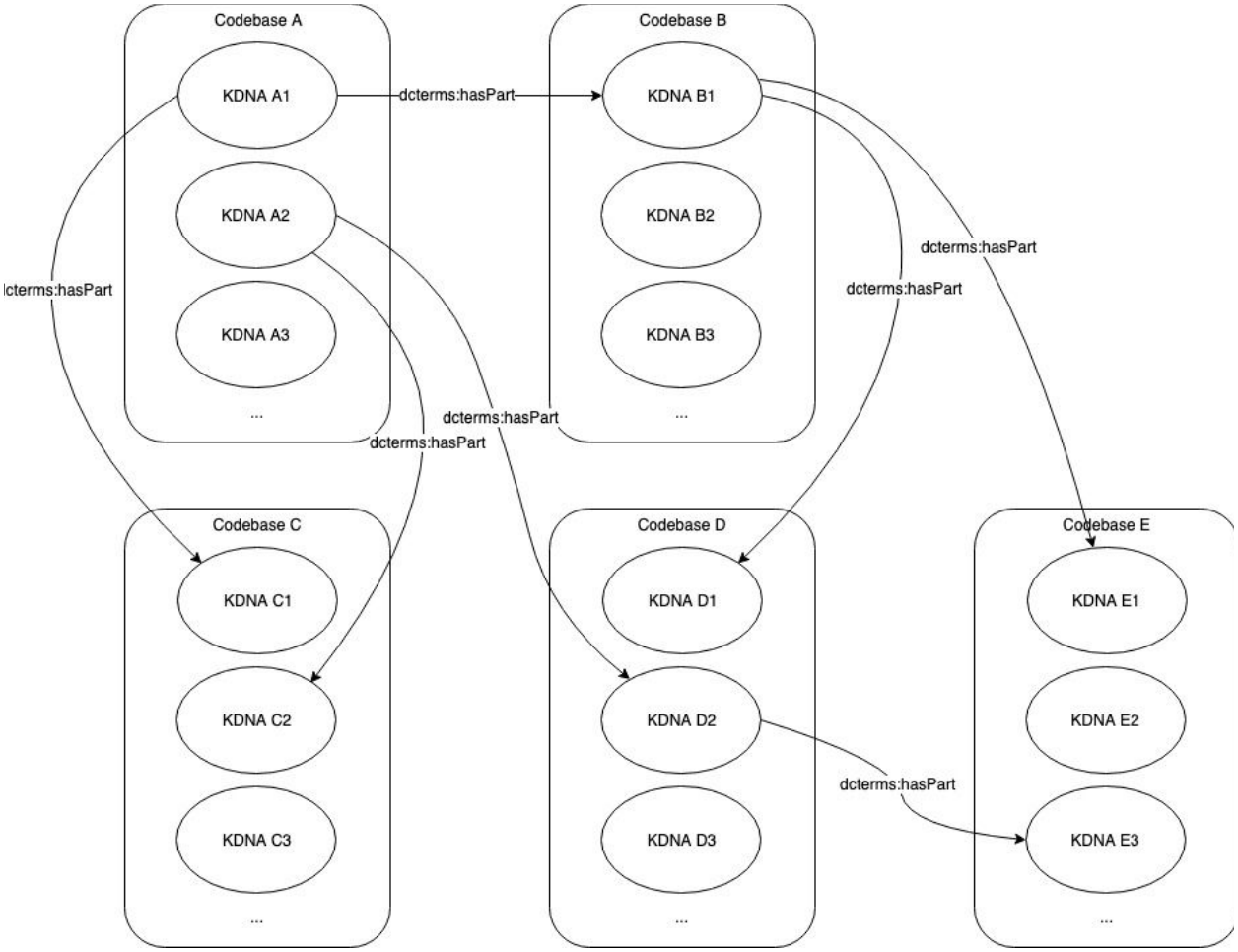
```
@prefix codebase: <https://codebase.kambria.io/codebase/> .
@prefix kdna: <http://kdna.kambria.io/kdna-schema#> .
@prefix dcterms: <http://purl.org/dc/terms/>.

codebase:5b31f3a8033d6613283f97f1 a kdna:Codebase ;
  dcterms:name "Ohmnilabs Arc Reactor" ;
  dcterms:description "OhmniLabs Arc Reactor, 4 Gigawatt output" ;
  dcterms:creator "OhmniLabs" ;
  kdna:repository "https://github.com/kambria-platform/ohmnilabs-arc-reactor" ;
  dcterms:relation "https://github.com/kambria-platform/tb-light-hardware" ;
  dcterms:hasPart [
    kdna:hash "QmUFZxPUoGpeSf5TLwFq1XTzbTwfsL4QNA7G8pCfLnGcr1" ;
    kdna:weight 0.2 ;
  ], [
    kdna:hash "QmahKrogko2ghY8PvZdafewAeFHuQ49YW4yHiCGyFTHfEs" ;
    kdna:weight 0.4 ;
  ] ;
  kdna:previousVersion "QmcDaNG9ytC1m5e1gCbaRFG2cgqX2mUmohPrLoSN5FuZg3" ;
  kdna:ethAddress "0x4155544263f862db7df6c84deb5e5c8cb2122c25" ;
  kdna:hasContent [
    kdna:semanticType "mechanical/stl" ;
    kdna:extension ".stl" ;
    kdna:path "mech" ;
  ], [
    kdna:semanticType "electrical/bom" ;
    kdna:extension ".csv" ;
    kdna:path "bom" ;
  ] .
```

After uploading this file to IPFS, we can access it on the IPFS network:
http://ipfs.kambria.io:8080/ipfs/QmZAW4CDkcB3EZ7FHr4bhRszK5m6NAw9jX2DrkzkF9wEwZ

3.2. The relationships between codebases

The relationships between codebases is represented by the term *dcterms:hasPart***.** In this section, we will cover how *dcterms:hasPart* works.

dcterms:hasPart is to show the paths to the relevant codebases that are used by a codebase. It is very similar to the “dependencies” info in a package.json file of an NPM package. This graph shows how this relationship changes over the lifetime of a codebase.



From the previous section, we showed that if we change the content of KDNA file and add it to IPFS network, it will generate a different hash. So in the same manner, each change we make to KDNA file will create a new KDNA version. The change can be as simple as changing the description, or changing the libraries it uses. For example, in the newer version of codebase A, instead of using codebase B and C as libraries, it changes to use D and newer version of C. So, we have to change the *dcterms:hasPart* info in KDNA file since the *dcterms:hasPart* will store the hashes of KDNA files of the codebases it uses. In turn, codebase C and D can have their own libraries. So with A as a root codebase, we can build a tree of codebase dependencies of codebase A (similar to the tree depicted in dependencies info of package-lock.json file).

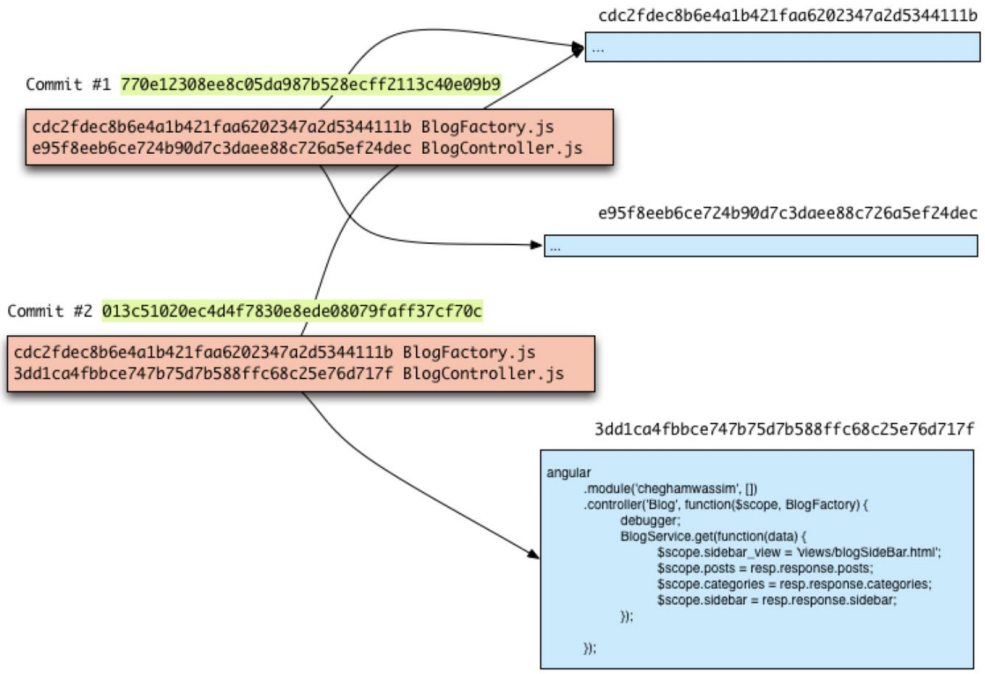
4. DAG in KDNA

4.1. DAG in Git

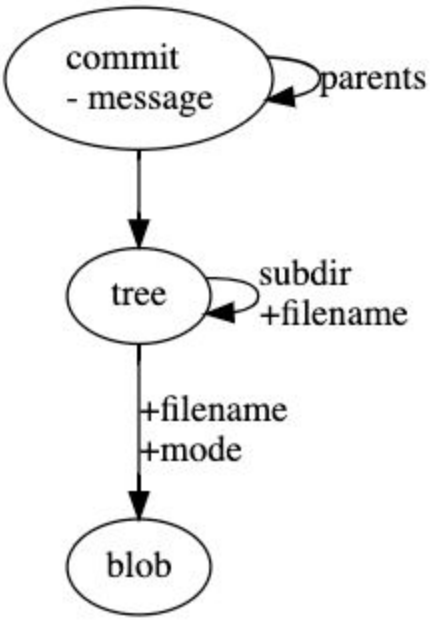
Git is a very powerful and popular version control system. Git offers a Merkle DAG object-model that captures changes to a filesystem tree in a distributed-friendly way. Git object model includes: Files (*blob*), Folders (*tree*), and Changes (*commit*).

All versions of a file will be stored in a special .git/objects folder, but the names of all the files are changed to the hashes of the contents of the files. So, if we have many files with the same content, it will only be stored once in .git/objects folder. A folder (*tree*) in git repo is also represented by a file that stores the structure of that folder. This file is also named by the hash of its content. Every time the structure of the folder or the content inside the folder is changed, it will generate a new folder-represented file. A commit is also a manifest file that points to the folder-represented file of root folder of the repo.

Let us take an example of a project that contains two files: BlogFactory.js and BlogController.js. This figure demonstrates what will happen after you change the BlogController.js file, and do a git commit.



We can display the model in a graph, such as the one below, with each commit having pointer “parents” pointing to the previous commit.



4.2. DAG in IPFS

IPFS also defines a set of objects for modeling a versioned file system on top of the Merkle DAG. This object model is similar to Git's:

1. *block*: a variable-size block of data.
2. *list*: a collection of blocks or other lists.
3. *tree*: a collection of blocks, lists, or other trees.
4. *commit*: a snapshot in the version history of a tree.

IPFS actually learned from the Git object model, but it also introduced some new features: fast size lookups, large file deduplication (adding a *list* object), and embedding of commits into trees.

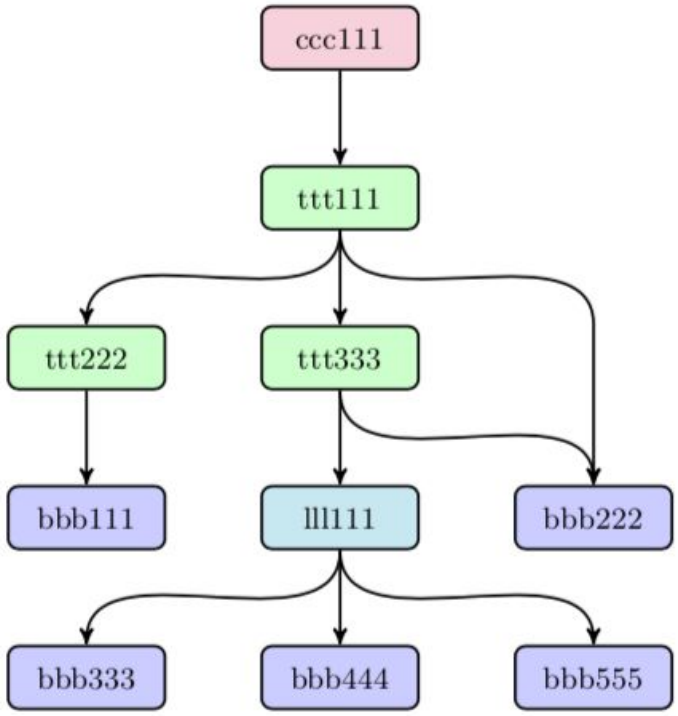
When adding to IPFS, a large file is chunked into smaller addressable units called *blobs*, several IPFS *blobs* concatenated together can make a file, and we model a file with *list* object. The *tree* object represents a directory, a map of names to hashes. The *commit* object will be similar to git, but its implementation in IPFS is still in progress.

IPFS object model is implemented into code like this:

```
type IPFSLink struct {
     Name string // name or alias of this link
     Hash Multihash // cryptographic hash of target
     Size int // total size of target
}
```

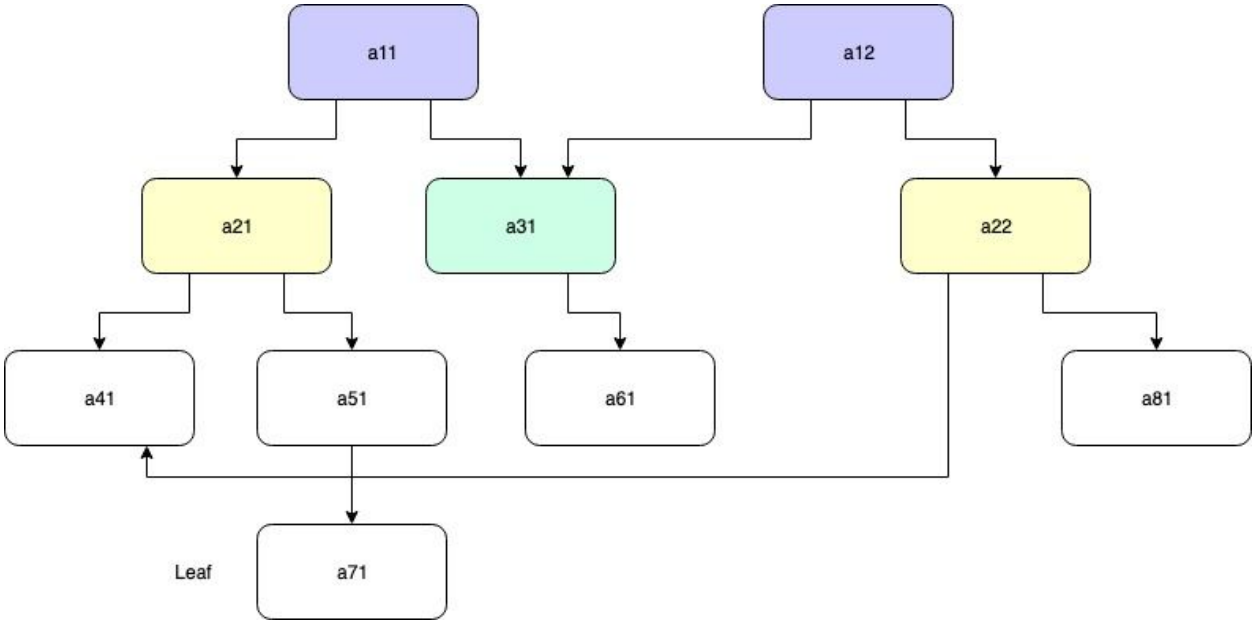
```
type IPFSObject struct {
      links []IPFSLink // array of links
      data []byte // opaque content data
}
```

Below is a sample of IPFS object graph with the top pink node as a commit, green nodes are lists, the cyan node is a tree, and purple nodes are blocks.



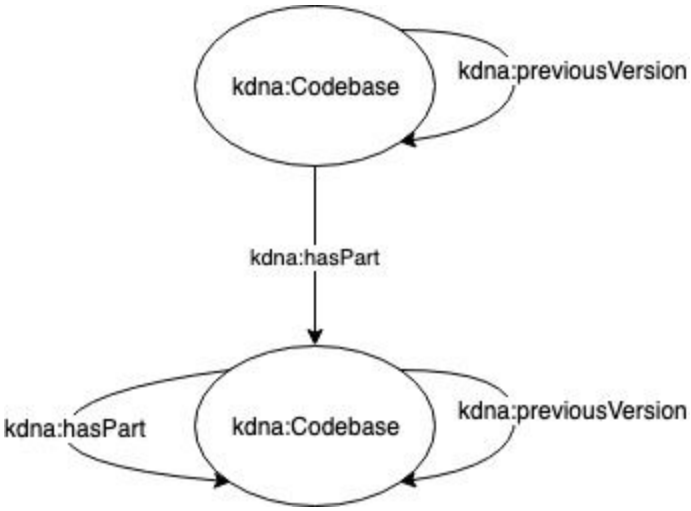
4.3. DAG in KDNA

The KDNA model is built on top of IPFS, and it only contains KDNA files which are very small in size. In the KDNA model, we don't have trees, lists, or commits. The links between KDNA files are the work of *dcterms*:*part*. The KDNA file itself contains the links to other KDNA files.



We can imagine that each KDNA version is a commit. At first, codebase a21 uses two other codebases: a41 and a51. After that, codebase a51 is replaced with a81, so the *dcterms:hasPart* info in KDNA of a21 will change. That creates a new KDNA version a22 of that codebase. a21 is also used by another codebase a11, and when a11 updates to use new version a22, it will also create a new KNDA a12, but it does not update codebase a31, so it still points to the same KDNA of codebase a31, even when a31 may have a newer version a32. Moreover, each KDNA has the *dcterms:previousVersion* property to point to the previous version of the KDNA, so as long as we know the latest KDNA version of a codebase, we can always trace back to the previous versions and their dependencies from time to time.

From that we can have a generalized graph like below



4.4. Apply KDNA DAG in Karma Distribution

Karma is a non-tradeable ledger entry created by Kambria, aiming to encourage participants to contribute to the Kambria platform through incentives based on their contributions. Karma is introduced in detail in Kambria Karma Model Paper.

From the KDNA DAG, we can easily track the Weighted Contribution (W_X) of a project via the *kdna:weight* property of its parent.

For example, with this data from KDNA file of Codebase A, we can determine that it uses two other codebases: B and C. Based on the data, we can easily specify $W_B = 0.2$, $W_C = 0.4$, and we can calculate the contribution of Codebase A itself by using this formula: $W_A = 1 - (W_B + W_C) = 1 - (0.2 + 0.4) = 0.4$.

```
  dcterms:hasPart [
    kdna:hash "QmUFZxPUoGpeSf5TLwFq1XTzbTwfsL4QNA7G8pCfLnGcr1" ;
    kdna:weight 0.2 ;
  ], [
    kdna:hash "QmahKrogko2ghY8PvZdafewAeFHuQ49YW4yHiCGyFTHfEs" ;
    kdna:weight 0.4 ;
  ] ;
```

5. Conclusion

The application of KDNA is not only limited to distributing Karma; we can use it in various other aspects. One example is that KDNA can be used to track and verify the ownership of a codebase because when the information is added to IPFS it can not be tampered. Furthermore, it is very helpful for developers in terms of utilizing codebases. With KDNA, they can find all the related source codes, designs, and instructions they need to build their own products from shared codebases. Finally, the advantages are not limited to inside the Kambria platform; people from outside Kambria can also use KDNA schema to describe codebases themselves and link the codebases, or even link to codebases inside the Kambria platform.

References

1. https://kambria.io/Kambria_White_Paper_v2_20180615.pdf
2. https://docs.npmjs.com/files/package-lock.json#dependencies
3. https://www.slideshare.net/OpenDataSupport/model-your-data-metadata
4. https://joinup.ec.europa.eu/sites/default/files/document/2013-03/Cookbook%20for%20translating%20relational%20domain%20models%20to%20RDF-S.pdf
5. https://www.youtube.com/watch?v=jONZtXMu03w
6. https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf

7. https://www.w3.org/RDF/
8. https://www.w3.org/TR/turtle/
9. https://www.w3.org/TR/sparql11-overview/