

INTERPLANETARY FILE SYSTEM MEETS CUSTOM DATABASE FOR ENHANCED DATA GOVERNANCE

Elif Tasdemir*, Frank Dressel*

* Deutsches Zentrum für Luft- und Raumfahrt e.V., Institut für Softwaremethoden zur
Produkt-Virtualisierung, Dresden, Germany

Abstract

Peer-to-Peer (p2p) network architecture allows decentralized data sharing, where each node can be both server and client. In contrast to the centralized network, nodes in the p2p network store data in a distributed and collaborative way. P2p network protocol empowers data holders with ownership, privacy and resilience. A prominent example is the InterPlanetary File System (IPFS) which is an open source and content-addressable distributed p2p storage system. IPFS works based on the content. Every data file has a unique hash, similar to the fingerprint. When a file is requested, a query based on the hash is done and nodes having the file will provide it to the user. Large files are split into multiple blocks and distributed to the other nodes. IPFS nodes stores data in local storage, e.g., flat file system (flatfs), or data bases (badgerDB, LevelDB). By default, IPFS uses flatfs for blocks of raw data and LevelDB for everything else such as pins, providers, etc. In this work, we aim to replace default storage of IPFS nodes for data with custom data bases for each node to enable integration of existing solutions without destroying decentralized structure. Particularly, enterprises can integrate their existing databases to IPFS. As a result, collection of richer metadata for provenance and audibility, and queries over the content will be possible. Moreover, it will help to control better governance per node. For instance, one node can be public and other can be private. Finally, automatic deletion of data after retention period or archiving old data, and data encryption will be possible. On the other hand, it is needed to deal with compatibility and integration issues as well as to asses the performance of nodes with databases. In this work, we present a way to the optimum architecture, advantages and limitations of this custom database integration.

Keywords

IPFS; Custom database; Data governance

1. INTRODUCTION

Decentralized data storage system is formed from many independent computers different than a central server or cloud storage. This system provides less dependency on a provider, high probability to access data and grant users ownership with private keys. Studies regarding to decentralized data storage has gained growing interest among academia [1, 2] as well as a number of open source projects are available to test and develop ideas.

InterPlanetary File System (IPFS) [3] is an example of open source project for peer-to-peer decentralized network and is based on content addressing. Public and private network setting are available. It has already thousand of public nodes and published files [3]. IPFS has written in modular way making data storage layer and network layer are separable so that storage layer can be swapped

freely. There are already several data storage back ends [4] integrated to the IPFS. Once data is added to the IPFS, big files are chunked in pieces. Then, the raw chunks and their Content Identifiers (CID)s are stored as value-key pairs.

In this work, we explain methods to swap storage option in IPFS for a custom Data Base (DB). Our work will guide and encourage users who willing to adopt IPFS, but do not want to migrate their data to a new separate storage system compatible with IPFS. Moreover, integrating a DB which is already in production will make DB users to have better data management. Additionally, the integration will not destroy the decentralized structure of IPFS since every node will have a separate storage. Because of pluggable interface of IPFS storage, even hybrid approaches would be possible, such as each node having a different storage platform.

Rest of the paper organized as follow. Section 2 gives a brief overview of the background of IPFS and the selected custom DB for integration. Section 3 explains the strategy for the integration, while Section 4 concluding with discussion and future work.

2. BACKGROUND

In this section, we explain main building block of our custom database and ready-to-use storage systems in the IPFS. We selected the Storage for Heterogeneous Product and Research Data (Shepard) due to being able to scale [5], link heterogeneous data and related metadata as a custom DB.

2.1. Custom DB: Shepard (A Storage for Heterogeneous Product and Research Data)

Shepard [5] was designed for heterogeneous data. These data types are time series, structured, files, spatial data and semantic repositories. It handles incoming requests and directs them to the relevant data bases, TimescaleDB, MongoDB, PostGIS. Neo4j is also used to store metadata and corresponding references between data and data container. Apart from the Web interface, there are also APIs to interact with shepard. For instance, the Python version of the API is documented with code snippets in [6]. These data upload workflow of shepard has some straightforward sequence of steps. As described in Figure 1, a collection is first resource to be created. As a root container, it manages data, sub components and permissions. Next thing to create is a data object to separate different work under same collection. Under the data object, multiple containers can be created to store the actual data. Once data containers are created for each data type, there are reference APIs for each container type to bind each container to its corresponding data object and collection. Container APIs manage then loading data into containers. So in summary, collections, data objects, and containers are similar to tables, rows and columns in traditional DBs.

2.2. IPFS and Default Storage Capabilities

The main building blocks of IPFS are libp2p, bitswap, Kademlia Distributed Hash Table (DHT), CIDs, Merkle DAGs and blockstore. libp2p is a networking framework and consisting of protocols, specifications and libraries to enable communication between peers. Bitswap is a block exchange module in IPFS requesting and sending data block to peers. DHT is a catalog of mapping data and CIDs and answering which peer has what block. CIDs are content identifiers. While

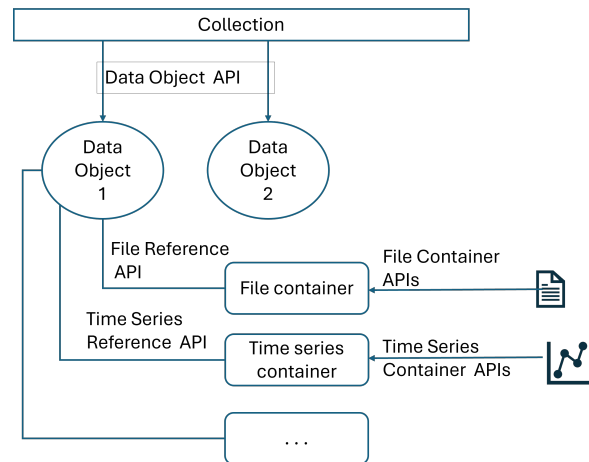


FIG 1. Building blocks of shepard.

the main file has a parent CID and parent CID is mapped to its blocks namely child CIDs such a tree branch. Merkle DAGs are graphs of content addressed blocks to map parent-child relation when a file or a folder is added.

There is a module in IPFS referred as datastore [7] which is an API. It allows storage type to be swapped seamlessly without changing application code. This interface is placed between raw key-value pairs and rest of the IPFS's modules. Datastore module has predefined interfaces i.e., Read (Get, Has, Getsize, Query), Write (Put, Delete), Durability and Lifecycle (close), Batching and many. In conclusion, any new storage should provide these interfaces.

Default IPFS Kubo repository based nodes use two separate storage; one is flatfs storing raw data blocks on disk. The other one is a DB and storing metadata i.e., LevelDB mapping string key to string values. BadgerDB is another key-value store and it is ready-to-use. These optional configurations can be seen in configuration directory having initialization module in IPFS Kubo repository. Flatfs datastore was designed exclusively for content-addressed storage. Its architecture stores data as flat files by utilizing two-level directory structure in the local file system (sharding). FlatFS implements "first-successful-writer-wins" (FSWW) semantics to avoid race between multiple targets writing same key. Moreover disk usage cached frequently to find a balance between accuracy of total disk usage and performance. On the otherhand, levelDB is used as default datastore backend option to store metadata information. It is lightweight and can manage small clusters. However for large number of items, IPFS community recommends to use Pebble or Badger3 options [4]. In conclusion, these are the ready-to-used data storage system in IPFS. Since they are already integrated, selecting one

of them is a madder of configuration work. However, integration of a new custom DB requires developing new interfaces.

3. METHODOLOGY

In this section, we first explain our working environment. Then, we explain how to set a private network formed with IPFS nodes. These private nodes do not connect to the public IPFS DTH or swarm. Peers in private network only discover and exchange data with each other. Afterwards, we explain the module for shepard to swap storage backend of IPFS Kubo. Particularly, how to develop steps of Read and Write functions of shepard in Go programming language. Finally, the logic of plugging this module to the Kubo is explained.

3.1. Environment Set Up

We cloned IPFS Kubo [8] repository which is the first implementation and the most widely used one today. This repository contains core implementation and dependencies. We used containerization technologies to configure experimental environment to ensure reproducibility. Specifically, a `Dockerfile` was created to define base image, install required dependencies. To orchestrate the containerized services, a `docker-compose.yaml` file was implemented. An bash script `entry.sh` was developed and integrated to container image via `Dockerfile` to automate initialization requirements of IPFS private network configurations.

3.2. Private Network Configurations

Private network configurations must be set before creating a node. In this section we explain some configuration settings. These settings has been tested in local environment for development purpose. A slightly different settings may be needed for production phase. Following subsections explains some tested configurations for setting up a private network.

3.2.1. Swarm key generation

Nodes having the same swarm key can talk to each other. There can be multiple options to create a swarm key. However, the steps followed in this work are explained. First, a 64 hex character string with 32 bytes is created. The commands in Listing 1, either with SHA-256 running first two lines or with openssl running fourth line are used. Commands output a string, then this string is given to our `docker-compose.yaml` file as environment variable as shown in Listing 1 in line six. There are additional steps to be followed to integrate the swarm key in IPFS. For instance, the path for swarm key file is created in `entry.sh` as

indicated in line eight and ten in Listing 1. After creating a directory for swarm key, a function was created to convert *hex* format to *base64* for IPFS compatibility. Another function will take the key string from `docker-compose.yaml` and assign to a file content.

Listing 1. Swarm Key Generation

```
1 echo -e "change_there_to_make_key"
2 | sha256sum | awk '{print_$1}'
3
4 openssl rand -hex 32 > swarm.key
5
6 IPFS_SWARM_KEY: "7674cfe4..."
7
8 SWARM_KEY_PATH="$repo/key/swarm/psk/1.0.0"
9
10 SWARM_KEY_FILE="$SWARM_KEY_PATH/psk.json"
```

3.2.2. Profile setting

There are several option to set a profile when nodes are initialized. Code snippets used for initialization are shown in Listing 2 and written in the `entry.sh` file. One of them is `AutoConf`. The `AutoConf` setting enables Kubo nodes to automatically fetch and apply network configuration from a remote JSON endpoint. Setting the profile to *autoconf-off*, as shown in Listing 2 code line 1, makes networking fields empty to allow manual configuration. These networking fields are Bootstrap peers, DNS resolver, delegated routers, and IPNS delegated publishers [9].

Bootstrap peers are the static and public peers of IPFS. They connect new nodes to others. To set a private IPFS network, default bootstrap nodes should be removed. The code line number 2 in Listing 2 is run to remove the IPFS bootstrap nodes. **DNS resolver** in IPFS configuration refers to the mechanism used in IPFS to convert domain names e.g., `example.com` into IPFS content identifiers (CIDs) e.g., `/ipfs/QmHash...`. **Delegated routing**: Delegated routing allows IPFS implementations to offload tasks like content routing, peer routing, and naming to a separate process or server while also benefiting from HTTP caching [10]. It is a method to offload content routing to another process/server using HTTP API. For instance, if a node does not has computing resources to implement DHT, a delegated router can search the DHT for this peer on its behalf. **IPNS Delegated Publishers**: IPNS (InterPlanetary Name System) works similarly to DNS, but it maps a human-readable name or a public-key-based name into a permanent address. Particularly, if a file is changed frequently, the CID of this file will also change each time. However, IPNS will create a public permanent key e.g., `k51qzi5ztr84h...` for this file running the third command in Listing 2. If we check the link <https://ipfs.io/ipns/k51qzi5..>, we will see

updated file content [11]. A Delegated Publisher allows the original owner of an IPNS name to grant permission to another entity (a person, server, or process) to create new IPNS entries on their behalf, without ever exposing the owner's private key.

Listing 2. Additional Network Configurations

```
1 ipfs init --profile=autoconf-off
2 ipfs bootstrap rm --all
3 ipfs name publish /ipfs/<file-CID>
4 ipfs config --json Routing.Type "none"
5 ipfs config --bool Discovery.MDNS.Enabled false
6 ipfs config AutoNAT.ServiceMode disabled
7 ipfs config --bool AutoTLS.Enabled false
8 ipfs swarm connect <node-ID>
```

3.2.3. Configuration settings

Additional configurations applied in this work are as following: 1) Discovering the other IPFS nodes in locale are networks is managed by `Discovery.MDNS.Enabled` configuration option. mDNS enables quick and efficient discovery of IPFS nodes in local networks without any coordination, e.g., without internet connectivity or access to bootstrap nodes. It can be set false for private network. 2) `AutoNAT.ServiceMode`: NAT stands for Network Address Translation. It is a technique allowing multiple devices to share a single public IP address, by translating the devices private (home) IP addresses to the public IP address. It behaves like a firewall by hiding the network details. NAT blocks direct connection of peers behind different NATs. AutoNAT detects NAT type and status. Default is enabled, even it is unset. In this work, it has been disabled. 3) Another disabled configuration is `AutoTLS.Enabled` configuration. AutoTLS is a service allowing websites (browser-nodes) to connect to libp2p nodes, regardless of the network topology. Therefore, these browser-nodes act as a real IPFS node and connect to the rest of nodes. We set it to false, since a swarm key already is provided for our private nodes. Adding another Transport Layer Security may not be needed.

3.2.4. Daemon Routing Options

There are many routing options both for production and experimental purposes. `auto`, `autoclient`, `dht`, `dhtclient`, `dhtserver`, `none` are routing options for production. Default setting is `auto`. In this work, we set routing type `none` as shown in Listing 2 line fourth. The `none` option disables all routing and peer discovery. The discovery via DHT and CID is not possible. Peers need to be connected to each other manually with command in line eight in Listing 2. Detail of other routing options can be found in [11].

3.3. Custom Database Integration

In order to bind custom DB with datastore of IPFS, we need an API of custom DB written in Go programming language. The shepard database has already APIs in several languages, e.g., java, python. However, we generated a new API in Go language to import shepard client in IPFS modules. Shepard follows the OpenAPI specifications. Therefore, it is possible to automatically generate clients for shepard in multi-languages. Afterwards, a datastore module in Go for shepard needs to be created and assigned to the IPFS's datastore. A pseudo code of datastore is given in Algorithm 1. As seen in the Step 1, shepardAPI, datastore and query modules of IPFS are imported to the new package `shepDataStore`. Afterwards, Options and Datastore structs are created to store important parameters of shepard. In Step 3, the function `Create()` receives API key and backend URL of shepard DB to create a new client, a collection, a data object and a file container, which are bound hierarchically. In step 4, declaration of some methods of datastore are listed. The logic of other ready-to-use storage systems are followed. For instance, integration of pebble DB in [12] is an example to get insight of additional methods that need to be developed. For sake of simplicity, we have listed only three methods in the Algorithm 1. In particular, the `Put` function, has the key-value pair as input and place this pair to the shepard DB. Key is the CID of the data block and value is the raw binary data. Since standard shepard API, forces to receive data as file, we need to create a temporally file to store and upload data to shepard. Then, this temporal file is deleted. The file name is the key (CID) of the block. This binary to file conversion is also needs to be followed in `Get` function. Moreover, in `Get` function, we we get all files in the file reference and in a loop we check if the queried key is matching with any file name. In case of the match, `Get` returns the binary data. In final Step 5, the datastore for shepard is assigned to the datastore of IPFS.

3.3.1. Plugging Stage

After developing `shepDataStore` package, a plugging file or module should be developed and should be placed in a directory e.g., `shepard` under `kubo/plugin/plugins/` directory. There are also directories of the other ready-to-use storage systems under `/plugins` directory. The steps to develop plugin file is shown in Algorithm 2. Plugin file should import at least following packages. 1) The datastore that developed for shepards DB. As explained in previous section, we referred it as

Algorithm 1 Shepard-Datastore Interface

```

1: Step 1: import
2: shepclient "shepardAPI",
   dsq "ipfs/go-datastore/query",
   ds "ipfs/go-datastore", etc.
3: package shepDataStore
4: Step 2: Define data structures
5: type Option struct {
6:   Host string
7:   APIKey string
8:   CollectionName string
9:   DataObjectName string
10:  ContainerName string
11: }
12: type Datastore struct {
13:   Client *shepclient.APIClient
14:   auth context.Context
15:   CollectionID int64
16:   DataObjectID int64
17:   ContainerID int64
18: }
19: Step 3: Initialize
20: func (op *Options) Create()
   (*Datastore, error) {
21:   cfg:= shepclient.NewConfiguration()
22:   cfg.Host= op.Host
23:   cfg.Scheme = "https"
24:   client:= shepclient.NewClient(cfg)
25:   auth:= context.WithValue(..., ..,
   "apiKey")
26:   collID:= createCollection(client,
   auth)
27:   dataObjID:= createDataObject(client,
   auth, collID)
28:   fileContainerID:=
   createFileContainer(client, auth)
29:   return &Datastore...
30: }
31: Step 4: Write Methods
32: func (d *Datastore) Put(ctx, key
   ds.Key, value []byte) (error)
33: func (d *Datastore) Query(ctx, q
   query.Query) (query.Results, error)
34: func (d *Datastore) Get(ctx, key
   ds.Key) ([]byte, error)
35: Step 5: Bind ipfs and shepard datastores
36: var _ ds.Datastore = (*Datastore)(nil)

```

shepDataStore. 2) The IPFS packages for plugin, repo and fsrepo.

Another important thing is shown in Step 3. The method `DatastoreTypeName` the the name of this data storage, which will be used in initialization file in configuration folder of IPFS Kubo repository, i.e., `kubo/config/init.go`. Last but not least, the `shepardDBPlugin` is assigned to the `PluginDatastore` of IPFS in Step 4.

This newly created directory should be listed in the `preload_list` which is located under `plugin/loader` directory in Kubo repository. The `preload.sh` file will parse this list.

Addition to adjustment in plugin folder, there are also some lines need to be added in the `kubo/config/init.go` file. A function should be created to receive or define specifications for shepard during the initialization of Kubo nodes. The structure of the function of ready-to-use storage options can be followed. In this function, it is important to use the name that the method `DatastoreTypeName` returns in plugin package. Finally, we can select different plugin option for each created node in `docker-compose.yaml` file by setting `IPFS_PLUGINS:<name>` as a build argument.

Algorithm 2 Plugging the Shepard datastore

```

1: Step 1: import
2: shep "shepDataStore",
   "ipfs/kubo/plugin",
   "ipfs/kubo/repo",
   "ipfs/kubo/repo/fsrepo"
3: package shepDS
4: //Plugins is exported list of plugins
   that will be loaded.
5: var Plugins = []plugin.Plugin
   &shepardDBPlugin,
6: Step 2: Define data structures
7: type shepardDBPlugin struct
8: }
9: type datastoreConfig struct
10:   BackendURL string
11:   APIKey string
12: }
13: Step 3: Write Methods
14: func (*shepardDBPlugin) Name()
   (string)
15: func (*shepardDBPlugin) Version()
   (string)
16: func (*shepardDBPlugin)
   DatastoreTypeName() (string) return
   "shepards"
17: Step 4: Bind ipfs and shepard datastores
18: var _ plugin.PluginDatastore =
   (*shepardDBPlugin)(nil)

```

4. RESULTS

This paper guides interested reader to replace ready-to-use IPFS storage options with a custom database to allow greater control, scalability and customization of decentralized storage for applications. Key technical considerations are implementing an API for custom data base to read and write data, developing a datastore module as an interface between storage and other IPFS building blocks, finally a plugin module to load and list the integrated storage systems in IPFS. The integration empowers developers for an adaptable, efficient and context aware decentralize peer to peer system.

This implementation can be further developed for other data types of shepard, e.g., time series, structured, spatial data types. Moreover, the node performance in terms of the speed and bandwidth can be evaluated for different storage options.

Contact address:

elif.tasdemir@dlr.de

References

- [1] Huayi Duan, Yuefeng Du, Leqian Zheng, Cong Wang, Man Ho Au, and Qian Wang. Towards practical auditing of dynamic data in decentralized storage. *IEEE Transactions on Dependable and Secure Computing*, 20(1):708–723, 2022.
- [2] Leonhard Balduf, Sebastian Henningsen, Martin Florian, Sebastian Rust, and Björn Scheuermann. Monitoring data requests in decentralized data storage systems: A case study of ipfs. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 658–668. IEEE, 2022.
- [3] IPFS. Building blocks for better web. <https://ipfs.tech/> [(accessed: 03.09.2026)].
- [4] IPFS. Datastore backend. <https://ipfsccluster.io/documentation/guides/datastore/> [(accessed: 28.07.2026)].
- [5] Dr Roland Glück, Patrick Kaufmann, Florian Krebs, Felix Jakob Lettowsky, and Michael Vistein. shepard (version 5.1. 2). 2025.
- [6] Institute for Structures and Design of the German Aerospace Center (DLR). shepard architectural documentation. <https://gitlab.com/dlr-shepard/shepyter> [(accessed: 03.09.2026)].
- [7] IPFS. go-datastore. <https://github.com/ipfs/go-datastore> [(accessed: 27.08.2026)].
- [8] IPFS. Kubo: Ipfs implementation in go. <https://github.com/ipfs/kubo> [(accessed: 27.08.2026)].
- [9] IPFS. Kubo command-line. <https://docs.ipfs.tech/reference/kubo/cli/> [(accessed: 28.07.2026)].
- [10] IPFS. The kubo config file. <https://github.com/ipfs/kubo/blob/master/docs/config.md> [(accessed: 29.07.2026)].
- [11] IPFS. Publishing ipns names. <https://docs.ipfs.tech/how-to/publish-ipns/#publishing-ipns-names-with-kubo> [(accessed: 29.07.2026)].
- [12] IPFS. go-ds-pebble. <https://github.com/ipfs/go-ds-pebble/blob/master/datastore.go> [(accessed: 27.08.2026)].