Solid over the Interplanetary File System

Fabrizio Parrillo
Dept of Mathematics and Computer Science
University of Basel, Switzerland
<fabrizio.parrillo@unibas.ch>

Christian Tschudin
Dept of Mathematics and Computer Science
University of Basel, Switzerland
<christian.tschudin@unibas.ch>

***Abstract*—Solid is a moderate re-decentralization architecture for the Web. In Solid, applications fetch data from storage providers called "pods" which implement discretionary access control, permitting users to grant and revoke access rights and thus keeping control over their data. In this paper, we propose to apply an additional de-verticalization by introducing IPFS as a common backend for Solid pods, the goal being to prevent pod provider lock-in and permitting users to consider "self-poding" where the heavy persistency task is out-sourced to IPFS. We have implemented this Solid-over-IPFS architecture using the open-source "Community Solid Server" and successfully ran several Solid applications over IPFS.**

***Index Terms*—IPFS, Solid, Dweb**

I. Introduction

The Solid project [1] is Tim Berners Lee's answer to the massive centralization that has occurred in the Web: a few companies dominate the Web ecosystem and are safeguarding and mediating most of the world's social media content. For example, privacy violations [2][3] have been reported over and over, yet giants like Facebook, Twitter or Google continue to grow. Solid aims to get the end-users back in control by separating service from data and identity providers (which in the above cases are coinciding each time). In Solid, data is stored in personal online data stores "pods" operated by independent pod providers from which service providers have to fetch them if the user grants such access.

Although Solid shifts the power balance towards the data owner, it puts pod providers in a quasi-central role. This position raises centralization risks insofar as barriers could be created to make the provider change prohibitively expensive (e.g. uploading data to Amazon's cloud is cheap while data export is expensive), as well as a risk of an oligopoly dominated by the incumbents or –if real competition would ensue– "market consolidation" working in that direction.

IPFS [4], on the other hand, has a more radical vision of a peer-to-peer style Web where the uniting element is a flat content-addressable storage space hosted in a decentralized way. However, IPFS is currently not user-friendly as the data access protocol is only supported directly by the Brave browser and lacks compelling end-user applications.

In this paper, we argue in favor of extending Solid "at the bottom of the protocol stack" and to separate the tasks of pod operations (i.e. implementing access control and data query support) from data persistence which IPFS would handle

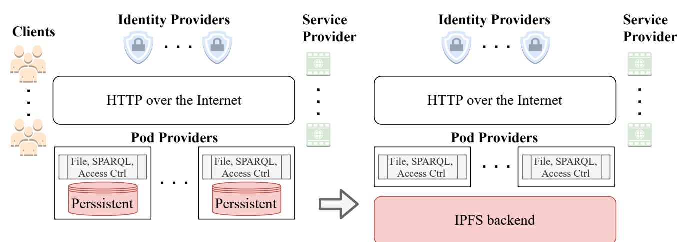

Fig. 1. Replacing Solid's persistency layer with IPFS

(see Fig. 1). In this way, IPFS becomes the uniting element among all Solid pod operators, effectively preventing them from holding the data hostage. Moreover, end-users and their browser do not have to (but can) be IPFS-aware because this burden now shifts to the tech-savvy pod operators, offering a smooth transition to a more IPFS-centric world.

We believe that our approach benefits and strengthens both projects. Solid reduces its vulnerability to storage centralization risks and for IPFS, it provides compatibility with the current Web without having to wait for all browsers and service providers to adopt IPFS.

This paper is structured as follows: First, we briefly introduce Solid and IPFS. For Solid, we describe how Solid structures data, how clients and servers communicate and how an exemplary application benefits from Solid. For IPFS, on the other hand, we provide a deep dive into its block storage model up to file system abstractions. In Section III we present our rationale to layer Solid over IPFS. Section IV describes our "IPFS data accessor" implementation and its integration into the storage module of a Solid server. We then report in Section V on our encouraging evaluation results before concluding.

II. Background

A. Solid

Solid [1] stands for **So**cial **Li**ked **D**ata and is a decentralized platform for the social Web. The Solid Community Group[1] openly develops the project. Furthermore, the startup Inrupt[2] is a major contributor to the open-source Solid-Ecosystem and provides Solid enterprise solutions. Solid's decentralization

[1] https://www.w3.org/community/solid/
[2] https://inrupt.com/

ISBN 978-3-903176-39-3 © 2021 IFIP

strategy builds based on separating storage, identity, and service providers. In such an environment, the storage providers equip users with a personal online data store (pods). Pods are Web-accessible storage nodes that implement a fine-grained authorization mechanism and can also be self-hosted. The identity providers allow users to create a global identifier to authenticate with pods. The service provider supplies the user with apps and services that operate on the pods' data they were granted access. Ideally, service providers' apps use interoperable data formats and data objects to prevent vendor lock-in users. In short, Solid aims to create a system where individuals can maintain their autonomy, control their data and privacy and can choose applications and services to fulfill their needs [5]. The following two sections introduce the relevant bits for this paper:

1) Linked Data Platform: Solid bases the concept of *resources* on the definition of the Linked Data Platform (LDP) [6] where *resources* are the most general class. All other classes, namely the *RDF*, *non-RDF* and *container resources*, are subclasses of the former. In other words, the LDP defines files as *RDF* and *non-RDF resources* and folders as *containers*.

2) Application Protocol: Solid allows a client to communicate with pods over the HTTP protocol. All Solid complaint servers must implement an HTTP interface and the required methods that allow clients to create, read, update and delete resources. Servers can also implement and advertise that they accept SPARQL Protocol and RDF Query Language (SPARQL) queries to manipulate resources or perform more complex data retrieval tasks [1].

3) Community Solid Server: Together with the Node Solid Server[3] (NSS) and the Enterprise Solid Server[4] (ESS), the Community Solid Server[5] (CSS) is an open-source implementation of the Solid specification [7].

The modular architecture allows for experiments with new ideas on the server-side. Currently, the server code is in beta stage. Nevertheless, a reasonable portion of the specification seems to be implemented, as some Solid apps can already be run against it.

The CSS server architecture reflects the Solid specification and comprises five main modules: server, linked data platform, authentication, authorization and storage. The storage module connects the pod to the underlying storage system. Currently, the storage module implements an in-memory, a file system and a SPARQL data accessor.

4) Solid Applications: Solid apps ideally follow a user-centric data model that allows other apps to reuse the data. For example, any app (which was granted access to) could reuse a user's contact list by following the links starting from the user's profile document. MediaKraken[6], which allows users to curate lists of their favorite movies, is well designed in this regard as it shows how an application can combine data from multiple sources: The movie list is stored on a pod and the movies' information comes from other services like IMDb. This concept is highly relevant for today's user experience: With Solid, instead of having a locked up list for each service, a favorites list could be stored on the user's pod and be used independently of the streaming service.

[3]https://github.com/solid/node-solid-server

[4]https://inrupt.com/products/enterprise-solid-server/

[5]https://github.com/solid/community-server

[6]https://noeldemartin.github.io/media-kraken/

B. IPFS

The Interplanetary File System (IPFS) is a distributed file system that combines ideas from established systems and concepts such as Distributed Hash Tables (DHTs), BitTorrent [8], Git [9] and Self-Certified Filesystem (SFS) [10] to create a new decentralized content infrastructure upon which other application can be built. The IPFS protocol can be divided into several sub-protocols with different functionalities. The following sections introduce three aspects in greater detail:

1) Objects data structure: The *block* is the elementary unit of the IPFS data model. Blocks contain the raw binary data and the Content Identifier (CID), representing the data's cryptographic hash. The latest CID format (CIDv1) is a concatenation of 5 binary segments (<v><c-t><h-alg><h-len><h>) where the first segment defines the CID format version (v), the second identifies the block's content type (c-t), the next segment defines the used hash algorithm (h-alg), followed by the hash's length (h-len) and finally the hash (h) itself. In addition, the string representation requires a prefix that defines the base encoding. Protocol Lab[7] created the Multiformat[8] project to define self-describing protocols which can be used independently from IPFS. In the above CID's definition we implicitly introduced multi**hash** (a self-describing hash protocol), multi**codec** (a self-describing serialization protocol) and multi**base** (a self-describing base encoding protocol). This scheme makes the CIDv1 future-proof and crypto-agile from the bottom up and results in an immutable storage model with a self-certifying namespace. The IPFS peers use a distributed hash table and the BitSwap protocol to distribute and retrieve content.

2) Complex Objects – Files: More complex objects can be built by reusing the described blocks as a foundation. To this end, IPFS uses "Merkle directed acyclic graphs" (Merkle DAG), a data structure that permits to build several important applications on top of it such as versioned file systems, blockchains, or a permanent Web. The Merkle DAG is defined by `IPFSObject` and `IPFSLink` protobuf objects. Where a `IPFSObject` represents a node and the `IPFSLink` represents a link of the Merkle DAG. A `IPFSLink` has three properties: a CID, the size of the target block and a freely chosen name property. A `IPFSObject` has two properties: a collection of `IPFSLink` objects and a data property that can store arbitrary data. For example, IPFS UnixFS stores the Unix file system's information with a Merkle DAG as shown in Figure 2. A file is a `IPFSObject` without links and a folder is a `IPFSObject` with zero or more `IPFSLink` to other folders or files. In reality, UnixFS does some block size

[7]https://protocol.ai/

[8]https://multiformats.io/

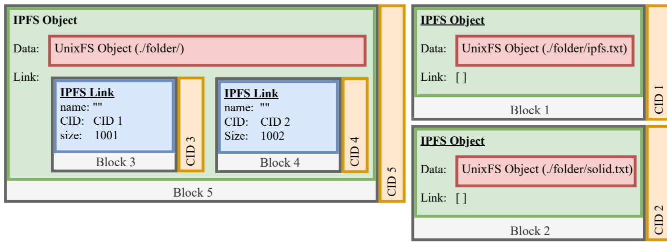

Fig. 2. Schematic representation of the IPFS UnixFS Merkle DAG

TABLE I
SOLID AND IPFS PROJECT COMPARISON

Property	Solid	IPFS
Decentralization type	moderate	radical
Decentralization strategy	social	technical
Decentralization target	service provider	infrastructure
Decentralization methods	service-separation, interoperability and access control	immutable and peer-to-peer file system
Addressing Scheme	location	content
Compatible with Web2.0	Yes	No (currently)

optimization by splitting big files into multiple blocks. Nevertheless, in both cases, the resulting file system is immutable and self-certifying.

3) Naming and Mutable State: Working directly with IPFS' UnixFS is not practical as the file system is immutable; moreover, CIDs are not human-friendly identifiers. Addressing the first issue, a change to a file does not result in a mutation but rather in creating a new file containing the mutated content: in other words, the Mutable File System (MFS) creates the illusion to the users that they can mutate files. On top of this, the MFS translates a path from a location-addressed scheme to a content-addressed scheme by mapping the human-friendly paths to CIDs. It is worth mentioning that whenever a file changes in a subdirectory, the MFS must not only re-reference the CID of the changed file but all parent directories and files up to the root because of the underlying immutable block storage model.

III. RATIONALE

Table I shortly summarizes the fundamental decentralization properties for Solid and IPFS. The Solid project is a moderate approach trying to impose decentralization by defining a specification targeting the service provider's application model that separates the services and pushes them to invest in interoperability. Together with a fine-grained access control mechanism, Solid aims to give users back control over their data. For example, a user might have multiple Solid identities (government, cooperate and private) and multiple pods. A Solid application interacts with multiple pods simultaneously by consuming and presenting information it was granted access to depending on the configured access control for the user identity and the service provider.

The IPFS project, on the other hand, is more radical from a technical viewpoint. The protocol is designed in a full peer-to-peer fashion, using concepts like DHTs and SFS to build an immutable file system. Moreover, the IPFS project successfully introduced Filecoin [11] as an incentive layer on top of IPFS in order to sustain a storage infrastructure capable of running distributed applications and implement smart contracts. Ultimately, this system does not target the service providers but tries to replace their current infrastructure – the cloud.

In short, IPFS could provide the storage infrastructure for Solid pods while the Solid specification could govern how apps should be built on top such that users keep control over their data. The following two sections will shortly introduce further opportunities for each project.

A. Opportunities for Solid

An IPFS-capable Solid server opens several future research directions, such as offline-first adoption, pod provider-independence, and data persistency with minimal centralization risks. These options ultimately boil down to a pod's content being stored on the IPFS network as a series of snapshots.

Having file-system snapshots means that switching pod providers only require the root CID of the Merkel DAG to be known such that a new pod service can be instantiated by a different pod operator with a new location. This location-independence may have some implication regarding Solid's location-based addressing scheme (referencing a pod's service instance instead of a pod's data), which needs to be researched in greater detail.

Using IPFS as a storage backend potentially enables offline-first operations of Solid apps. However, diverging pod snapshots need to be handled, either by designating which of the forks should become the new (global) root for a given user or by merging cherry-picked changes into the primary IPFS substrate, which may require app-dependent actions. More insights from other re-decentralization projects like Hypercore Protocol [12] and Secure Scuttlebutt [13] might be helpful here.

B. Opportunities for IPFS

The IPFS project can profit from Solid regarding two IPFS roadmap[9] goals, namely the Decentralized Web and the Personal Web.

Regarding the Protocol Lab's Personal Web goal, we observe that Solid is fully aligned as it is all about putting users in control over their data and access to it. Although Protocols Labs' definition of Decentralized Web does not fully align with the Solid specification (as IPFS envisions the Decentralized Web to function in fully peer-to-peer fashion), HTTP-based client-server interaction allows organic growth and reuse of existing end-user browsers, in parallel with the classic Web. This fluid transition is a compelling short-term strategy where end-users only have to learn about the pod concept but do not have to wait for all browsers to support the IPFS protocol.

[9] https://github.com/ipfs/roadmap/blob/a0ef950be691d74680050e77a89a12d1ca000fc6/README.md

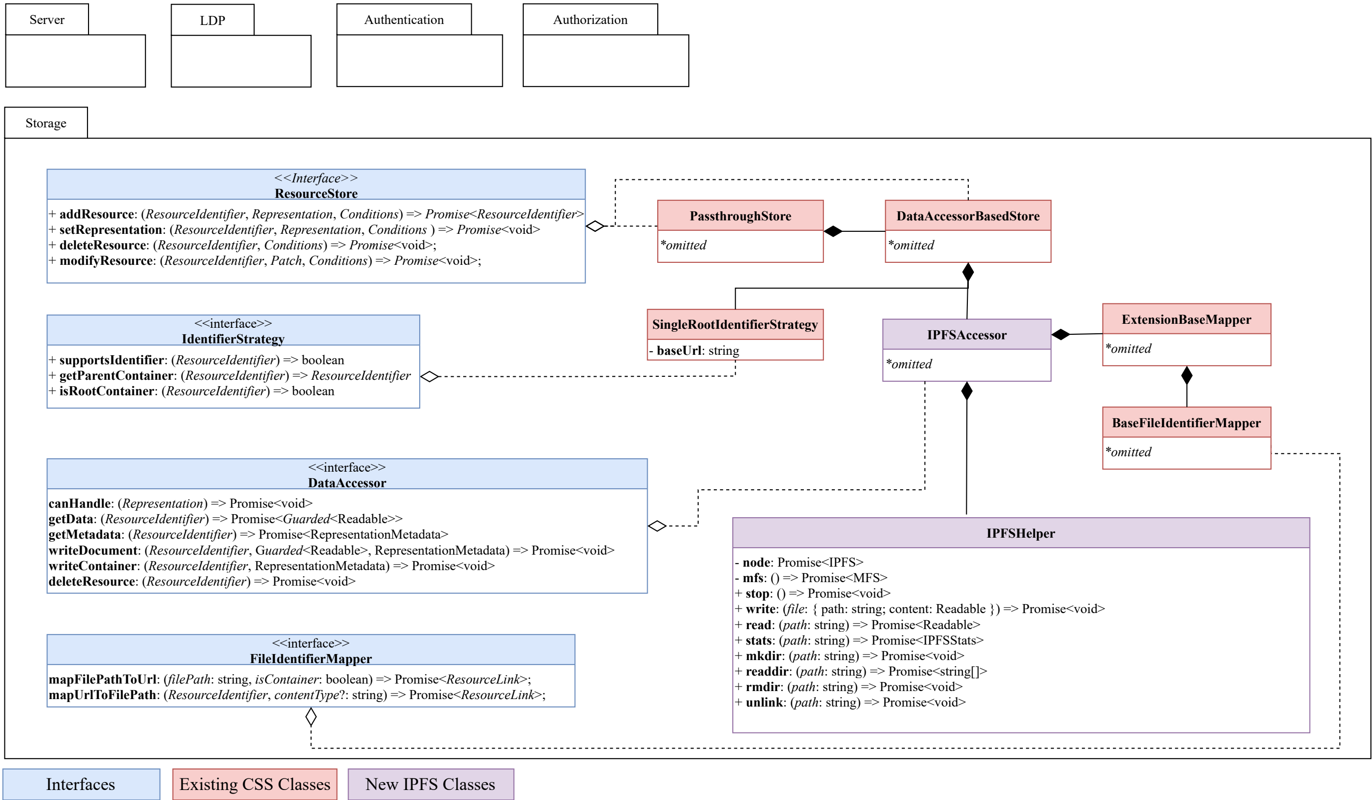

Fig. 4. Detailed Solid Storage Module Class Diagram

IV. IPFS Data Accessor for Solid

We successfully extended the CSS server with an additional data accessor. Figure 3 shows the existing `FileDataAccessor` which we replaced with the new `IpfsAccessor` that heavily builds on the existing `FileDataAccessor`. The `IpfsAccessor` connects the Solid pod to the IPFS network as the persistence layer. Our implementation adds the ipfs.js library as a dependency to the CSS source code. As a result, if we run the CSS with a configuration that instantiates an `IPFSAccessor`, a full-fledged IPFS node will be started. We tested our extension with unit tests and ran some experiments, yielding positive results with existing Solid apps. As the CSS server is bleeding-edge technology, we use this opportunity to give an in-depth technical level overview of the classes relevant to our implementation and facilitate future developments.

A. The IPFS Accessor Class for CSS

Figure 4 shows a detailed class diagram of the CSS storage module to which we add the new data accessor. From a top-down perspective, the implementation uses a `PassthroughStore`, which invokes the corresponding methods on the `DataAccessorBaseStore`. Our setup composes the `DataAccessorBaseStore` with a `SingleRootIdentifierStrategy` and the `IpfsAccessor`. The `SingleRootIdentifierStrategy` resolves all resource identifier from a single root on which all identifiers

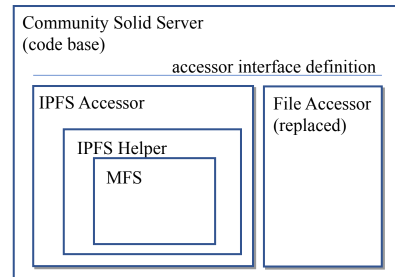

Fig. 3. Our IPFS Accessor is a full replacement for the CSS file accessor module

build on. As shown in Figure 4 the `IpfsAccessor` implements the `DataAccessor` interface. The `IpfsAccessor` can only handle binary data. Therefore the `canHandle` method throws a `BadReuqstHttpError` for all representation other than binary. The `getData` method returns a byte stream of the corresponding resource. If the resource's identifier is a container resource, the method throws a `NotFoundHttpError`. The `writeDocument` method writes data to a file identified by the resource identifier. The methods use the `FileIdentifierMapper` to map the resource identifier from an URL to a file path. As an example, *https://example.org/helloIpfs.txt* is mapped to */helloIpfs.txt*. It is important to note that the MFS class requires the file path to start with *slash*. Those paths are later passed to the

```
@prefix dc: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.
@prefix posix: <http://www.w3.org/ns/posix/stat#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix ipfs: <http://ipfs.io/ns/ipfs#>.

<>
 a
  ldp:Container, ldp:BasicContainer, ldp:Resource;
 posix:size  0;
 posix:mtime 1615461843;
 dc:modified
  "2021-03-11T11:24:03.643Z"^^xsd:dateTime;
 ipfs:cid
  <QmQPvWCPym2ZpUun3a5fK8XjZuHvf3GTS6xAPgbf4QTKRn>;
 ipfs:cid
  <QmQPvWCPym2ZpUun3a5fK8XjZuHvf3GTS6xAPgbf4QTKRn>;
 ldp:contains <.acl>.

<.acl> a ldp:Resource;
 posix:size 390;
 dc:modified
  "2021-03-11T11:23:57.442Z"^^xsd:dateTime;
 posix:mtime 1615461837;
 ipfs:cid
  <QmXHv61YkhVXazoq8Qa55HyiyP7VXhxDu7kMbTnTHqRxx5>.
```

Listing 1. A Solid pods' root container that exposes the CID

`IpfsHelper` while invoking basic file system primitives.

Listing 1 shows the resulting Turtle representation of the root directory after starting the server for the first time. This Turtle file results from a client executing an HTTP `GET` request against the pod's root container. The server resolves the request to the `IpfsAccessor`, which ultimately retrieves the data that describes the root container (the information about the root container and the .acl file). Listing 1 also shows the exemplary IPFS vocabulary with the prefix *http://ipfs.io/ns/ipfs#* which we defined to expose the `cid` property.

B. The IPFS Helper Class

The `IpfsHelper` class is a wrapper around the MFS class. The `IpfsHelper` class' primary purpose is to harmonize the MFS class with the Node.js files system interface. The `IpfsHelper` implements the basic file system primitives: `read`, `write`, `unlink`, `lstat`, `readdir`, `mkdir`, `rmdir`. The implementation maps the MFS specific errors to Node.js system errors. The error object mapping is required as the implementations describe errors differently. For example, the IPFS `stat` method returns the string "ERR_NOT_FOUND" while Node.js's `lstat` method returns "ENOENT". For the `lstat` method, we extended the return object to define the CID as an additional property. As a result, the implementation complies with the Node.js file system interface regarding the CSS's upstream modules, which use the `IpfsAccessor` class. Also, the helper provides some utility methods to start and stop the IPFS node.

C. Limitations

Currently, the CSS server can only be used as a resource server as version 0.8.0 does not implement any identity providing functionalities. This missing feature is a minor obstacle for Solid apps that require the user to log in. Nevertheless, we can easily overcome this problem by creating a Solid user with an existing identity provider.

V. CSS Compatibility Tests Results

In this section, we present the compatibility experiments we ran to test our Solid-over-IPFS stack. We try to determine if the existing Solid apps are compatible with the original and our modified CSS server. The compatibility tests are carried out by manually testing the functionality of existing Solid applications. We then rank the applications on the spectrum from 'not functioning' to 'full compatibility', meaning that no irregularities could be spotted. Fixing bugs in existing apps or the CSS server code base was outside the scope of our work.

The test setup includes three components a local CSS server, a Solid remote pod that provides a Solid user identity and Solid apps. We used a Lenovo X1 Carbon Gen8 running Ubuntu 20.04.2 LTS as the operating system and the Firefox 88.0 browser for the testing. The local CSS server, which we set up with two configurations, starts a local instance listening to *http://localhost:3000/*. The first configuration uses the file system, and the second configuration uses the IPFS network to persist data. After each test, we delete the data folder and restart the server.

The second component, the Solid remote server, provides us with a Solid user identity for the compatibility tests since most Solid apps require users to log in before exposing additional functionalities. Therefore we create "*solid-ipfs*", a Solid user at *solidcommunity.net*. At the time of testing, the *solidcommunity.net* server ran NSS 5.6.6. Further, we edited the user's personal profile document[10] by adding the link to *http://localhost:3000/* as the prior location to instruct the apps to persist data on the local CSS pod instance.

The last and third component of the test setup are the Solid *client-side apps* which we ran by navigation to the publicly hosted service instances. Table II shows a consolidated view of the results for the six tested apps:

1) NSS Solid Data Browser: The NSS Solid data browser is the data browser app that comes with NSS. We use the one from *solidcommunity.net*: This data browser detects the localhost storage and displays it in a tab. Unfortunately, it throws a *HTTP 404 Not Found* while trying to open or browse the storage.

2) Inrupt Pod Browser: The Inrupt Pod Browser[11] is the Solid data browser developed by Inrupt. Unfortunately, this data browser did not allow us to browse the local pod since it supports only HTTPS connections.

3) Solid File Browser: So far, the Solid File Browser[12] is the only data browser app that allows us to browse the local pod. The app prompts the user with an option to open a pod at a specific address. We can successfully create folder, files

[10] https://solid-ipfs.solidcommunity.net/profile/card
[11] https://podbrowser.inrupt.com/
[12] https://otto-aa.github.io/solid-filemanager/

TABLE II
CSS AND SOLID APPS COMPATIBILITY TESTS

The test results are classified in four groups: ✓✓✓No misbehavior found even after extensive use; ✓✓App works fairly but not ready for daily usage; ✓App can be launched but bugs show up very early; ✗App can not be tested.

Solid App	CSS with FileDataAccessor	CSS with IpfsAccessor
(1) NSS Data Browser	✓✓	✓✓
(2) Inrupt Pod Browser	✗	✗
(3) Solid File Browser	✓✓	✓✓
(4) Media Kraken	✓	✓
(5) Solid Focus	✓✓	✓✓
(6) Dokieli	✓✓✓	✓✓

and edit them. The browser's only flaw we can detect is that it cannot open files lacking a file extension in their name.

4) MediaKraken: As we introduced in the background section MediaKraken[13] is an app that allows users to curate lists of movies. When the app launches, a movie folder is created, but the app stops working after the HTTP `GET` request on *http://localhost:3000/movies/* results in an *HTTP 404 Not Found* response. Without having consulted the app's source code we speculate that the app stops working because of a client-side race condition or a faulty CSS response to the client's request.

5) Solid Focus: Solid Focus[14] is a task management app. This app allows us to freely choose the storage pod on which our data should be persisted. We can successfully create workspaces and to-do lists, but we encountered an error while creating a to-do item. Consequently, the app does not create the to-do item as there are conflicting metadata files indicated by an *HTTP 409 Conflict* response.

6) Dokieli: Dokieli[15] is a client-side editor for decentralized article publishing, annotations and social interactions. We use the app to save a copy of the Dokieli page itself to our local pod. Although we logged in, we can only use the "save as" functionality as the app disables the "save" button. This app is the sole app that we encountered where an error only occurs with the IPFS-configured CSS server: For our IPFS configuration of CSS, the new resource path must end with a file extension, otherwise the server responds with an *HTTP 404 Not Found*. This error is thrown by the server while fetching the stats from the metadata.

A. Discussion of Test Results

Our experiments tested the compatibility between existing Solid apps and the CSS server with two data accessor configurations. Both configurations yield positive results, given the early development stage of the server code. Moreover, we show that the `IPFSAccessor` is usable without further ado and is fully compatible with the IPFS data accessor, except for a minor bug that currently occurs while testing the Dokieli app. The test-driven development of CSS source code mainly drives this positive result. We followed that lead and tested our two classes with unit tests that reach a coverage of 88% for the `IpfsHelper` and 96% for the `IpfsDataAccessor`. This test coverage is slightly lower than the rest of the server's coverage which reaches 100%. Unfortunately, our unit tests are slow as the test initializes an IPFS node for each unit. Future work regarding the CSS implementation could focus on introducing an `Filesystem` interface, which is then used by the `FileDataAccessor`. As a result, the integration tests can test the different filesystems, and the `FileDataAccessor` remains independent of the filesystem.

[13]https://noeldemartin.github.io/media-kraken/
[14]https://noeldemartin.github.io/solid-focus/
[15]https://dokie.li/

VI. CONCLUSIONS

The contributed IPFS data accessor for the CSS server shows that Solid and IPFS are orthogonal projects where Solid classifies as an application layer protocol and IFPS as an infrastructure layer. We believe that both projects can profit from each other. The IPFS project could accelerate its adoption to become the next Web infrastructure. With the help of Solid, the IPFS protocol could be compatible with the current application-layer protocols. This approach could significantly ease IPFS' accessibility for non-technical users.

REFERENCES

[1] A. V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, and T. Berners-Lee, "Solid: A Platform for Decentralized Social Applications Based on Linked Data," 2016.
[2] J. Isaak and M. J. Hanna, "User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection," *Computer*, vol. 51, no. 8, pp. 56–59, 2018.
[3] R. Tony, "France fines Google nearly $57 million for first major violation of new European privacy regime," *The Washington Post*, 2019.
[4] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.
[5] W. S. C. Group, "Solid Technical Reports," 2021.
[6] S. Speicher, IBM-Corporation, J. Arwe, A. Malhotr, and Oracle-Corporation, "Linked Data Platform 1.0," w3c recommendation, W3C, 2014. https://www.w3.org/TR/ldp/.
[7] S. Capadisli, T. Berners-Lee, R. Verborgh, K. Kjernsmo, J. Bingham, and D. Zagidulin, "The Solid Protocol," editor's draft 2021-03-10, Solid Community Group. https://solidproject.org/TR/protocol.
[8] B. Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, Berkeley, CA, USA, 2003.
[9] D. Spinellis, "Git," *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.
[10] D. D. F. Mazières, *Self-Certifying File System*. PhD thesis, Massachusetts Institute of Technology, 2000.
[11] B. Juan, "Filecoin: A decentralized storage network. protocol labs, 2017."
[12] M. Ogden, K. McKelvey, M. B. Madsen, *et al.*, "Dat-Distributed Dataset Synchronization and Versioning," *Open Science Framework*, vol. 10, 2017.
[13] D. Tarr, E. Lavoie, A. Meyer, and C. Tschudin, "Secure Scuttlebutt: An Identity-Centric Protocol for subjective and decentralized applications," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, pp. 1–11, 2019.