Solid: A Platform for Decentralized Social Applications Based on Linked Data

Andrei Vlad Sambra*, Essam Mansour*, Sandro Hawke*, Maged Zereba*, Nicola Greco*, Abdurrahman Ghanem*, Dmitri Zagidulin*, Ashraf Aboulnaga*, and Tim Berners-Lee*

*Decentralized Information Group, MIT CSAIL
*Qatar Computing Research Institute
{asambra@mit.edu,emansour@qf.org.qa,sandro@w3.org,mzereba@qf.org.qa, ngreco@mit.edu,abghanem@qf.org.qa,dmitriz@mit.edu,aaboulnaga@qf.org.qa, timbl@w3.org}

Abstract. This paper presents Solid, a decentralized platform for social Web applications. In Solid, users' data is managed independently of the applications that create and consume this data. The user's data is stored in a Web-accessible personal online datastore (or pod). Solid allows users to have one or more pods from different pod providers, while at the same time enabling users to easily switch between providers. Developers can use Solid protocols, which is based on existing W3C recommendations, for reading, writing and access control of the contents of users' pods. In Solid architecture, applications can operate over data owned by the user or the user has access to regardless the location of this data on the Web. Users can also control access to their data, and have the option to switch between applications at any time. This is paradigm shift in integrating social features into Web applications. Our new paradigm produces a novel line of social applications. We have used Semantic Web technologies to build Solid prototypes, allowing us to demonstrate its utility through a set of applications for common day-to-day tasks. We also conduct experiments to show the scalability of the Solid prototypes.

Keywords: Decentralization, Social Web, Linked Data Platform (LDP)

1 Introduction

Today there are many Web applications developed to foster social interaction. In this paper, we refer to these applications as *social Web applications*. There are many successful social Web applications, such as Facebook, Twitter, Wikipedia, Craigslist, Doodle, and many more. Developers of social Web applications can either build their own platform for storing and sharing data, or rely on existing social network platforms. Unfortunately, these platforms have their own protocols and APIs for storing and accessing data, as well as for access control.

Centralized social Web platforms can also cause numerous problems for users and application developers. For example, users cannot easily move data between

platforms or switch between similar applications that would reuse their data. Moreover, existing social network platforms control the data access APIs and often change its features[1], thus freedom from these platforms is important. This is particularly the case for applications that access valuable data – e.g., from the financial or medical domain. Developers are restricted to the data access APIs provided by a specific platform, and cannot easily develop applications that can run on multiple platforms. These and other problems of centralization have been recognized for a long time, and there have been many proposals for decentralizing the social Web such as Diaspora[2], Musubi [3], and WebBox [18], among others. However, none of these proposals has been widely adopted yet, and the technical details of the decentralization platform are still very much a topic of discussion among researchers and practitioners (see, for example, the recent Redecentralize conference[3]).

This paper presents a decentralized platform for social Web called *Solid* (for Social Linked Data). Solid is based on RDF and Semantic Web technologies, and it achieves the goals of providing data independence and simple yet powerful data management mechanisms. In Solid, each user stores their data in an online storage space that we call a *personal online datastore* (pod). Pods are Web-accessible storage services, which can either be deployed on personal servers by the users themselves, or on public servers by pod providers similar to current cloud storage providers (e.g., Dropbox). It is possible for a user to have more than one pod. A user can choose among different pod providers, since Solid applications can work with any pod server regardless of its location or service provider. Different pod providers can offer different degrees of privacy, reliability (e.g., availability or latency guarantees), or legal protection (e.g., the legal frame specific to the country where the pod is hosted).

Solid applications are implemented as client-side Web or mobile applications that read and write data directly from the pods. Solid makes it easy to develop and use social features, because application data is always accessible on the Web, under the governance of an access control mechanism. Applications can aggregate data from different sources on the Web by accessing both the pod of the user running the application and pods belonging to other users. Solid allows multiple applications to reuse the same data on a pod. Users can choose among different applications that provide similar functionality. When a user switches to a new application, this application can access all the user's existing data, since applications are *by design* decoupled from the data that they use.

Pod providers also benefit from Solid because users can switch providers easily, so there is a lower barrier to entry for new providers. Moreover, since users have a high degree of control over their data, these users are more likely to store data with larger volume and variety in their pods. A larger volume and variety of data per user is good for pod providers. For example, a pod

[1] http://techcrunch.com/2015/04/11/twitter-cuts-off-datasift-to-step-up-its-own-b2b-big-data-analytics-business/
[2] https://diasporafoundation.org
[3] http://redecentralize.org/conference/

provider that is supported by advertising can run analytics on the wide variety of data stored in each pod (with the user's consent, of course) to decide the best advertisement to serve to the user.

In summary, this paper presents the design of decentralized platform for social Web applications. Our platform supports collaborative data management that remains under the control of its owners. Our contributions are as follows:

- We developed the Solid platform[4], which allows Web and mobile developers to easily develop social applications with high degree of flexibility and interoperability in accessing data in users' pods. Solid combines several Web standards, such as WebID, Linked Data Platform, RDF, and SPARQL, in a coherent manner. By doing so, it offers a standards-based incentive to the developers to adopt this technology.
- We developed several prototypes for pod servers, which can be used directly by a user or a service provider to manage user's data and enable access to it by Solid applications.
- We developed several Solid applications for common day-to-day tasks, such as a contact manager, microblogging, scheduler and calendar applications, and conducted experiments to show that Solid is capable of accommodating a variety of application use cases, and to study the behavior of our pod servers when working with multiple pods.

The rest of this paper is organized as follows. Section 2 presents an overview of the Solid platform. In Section 3, we highlight Solid's decentralized identity and authentication mechanism. Section 4 presents the data access protocol in Solid. Section 5 describes the requirements for pod management systems and overviews our different prototypes. In Section 6, we present a set of Solid applications and evaluate the performance of our pod systems. Section 7 presents related work and Section 8 concludes.

2 The Solid Platform

The Solid platform relies as much as possible on existing W3C standards to realize the architecture shown in Figure 1. The platform specifies all the protocols required in the figure, such as authentication, application-to-pod and pod-to-pod communications. In this section, we present an overview of how applications access data in Solid, with more details in the following sections. Our Solid organization in gitHub has the details of the Solid specification[5].

In Solid, applications use authentication protocols as means to discover the user's identity and profile data, as well as relevant links that point to the user's pod and application data. Decentralized authentication, a global ID space, and global single sign-on are a critical part of the Solid ecosystem. Solid uses WebID [13] to provide these features, although other solutions exist and can potentially interoperate with Solid. In Solid, a user has to register with an identity

[4] https://github.com/solid/
[5] https://github.com/solid/solid-spec

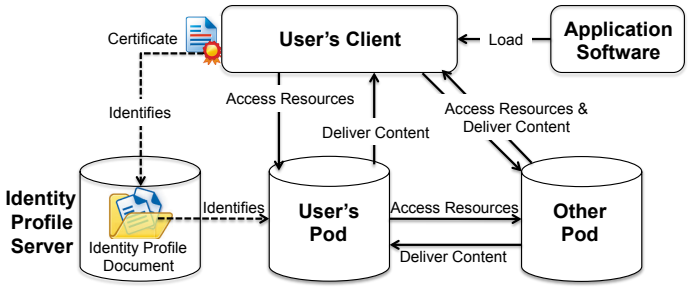


Fig. 1. Solid Architecture. The user controls his/her identity using an RDF profile document, often stored on a pod server. The user loads a Solid application from an application provider. The application obtains the user's pod from the identity profile. It then follows links from the profile to discover data on the user's pod, as well as on other pods, performing authentication when needed.

provider, most likely being his/her pod provider. The identity provider stores the user's WebID profile document associated with a cryptographic key.

In Solid, data is managed in a RESTful way, as defined by the Linked Data Platform (LDP) recommendation [8]. New data items are created in a container (which could be called a collection or directory) by sending them to the container URL with an HTTP POST or issuing an HTTP PUT within its URL space. Items are updated with HTTP PUT or HTTP PATCH. Items are removed with HTTP DELETE. Items are found using HTTP GET and following links. A GET on a container returns an enumeration of the items in the container.

Application data in Solid is stored in documents that are identified by a Uniform Resource Identifier (URI).Solid distinguishes between structured data represented using the Resource Description Framework (RDF), and unstructured data can be of any type (e.g., videos, images, Web pages, etc.). This allows structured data to be parsed and serialized in various syntaxes like Turtle, JSON-LD (JSON with a "context"), or RDFa (HTML attributes). RDF follows the REST principles in which resources have individual URIs.

Solid applications read and write data stored in users' pods via RESTful HTTP operations. Besides LDP support, Solid servers may offer optional SPARQL support. Servers that support SPARQL allow applications to express complex data retrieval operations, including operations that require server-to-server communication via link-following SPARQL (more details in Section 4.3). This simplifies Solid application development, since it enables a developer to delegate complex, multi-pod data retrieval operations to the server.

In Solid, the pod servers are application-agnostic, so that new applications can be developed without having to modify the servers. For example, even though LDP 1.0 contains nothing specific to "social", many of the Social Web Working Group User Stories[6] can be implemented in Solid, using only LDP and application logic, with no need to change code on the server.

[6] http://www.w3.org/wiki/Socialwg/Social_API/User_stories

3 Decentralized Identity System

Solid applications do not require authentication like most common applications do, since the user does not need to authenticate to the application provider. Instead, Solid applications may force a "fake" authentication request in order to obtain the user's WebID from the client certificate. From then on, authentication is only performed between the client (browser) and the user's pod where data is located. This practice saves the users the trouble of having to type their WebID during the login process.

For Solid to operate in a truly decentralized manner, it requires a global identity space in which users can easily manage and extend their own identity and credentials. While existing identity protocols such as OAuth [5] and OpenID Connect [10] offer a certain degree of decentralization, they do not fit our preference for RDF-based profile data, making it difficult to extend the profile information with additional user attributes. Solid uses WebID [13] to implement a global identity management architecture based on the concept of decentralized identity providers, which in turn when coupled with WebID-TLS [7] allows for Web-scale single sign-on.

3.1 Decentralized Identity based on WebID

A global and decentralized social Web requires that each person be able to control their identity and that this identity be linkable across sites, thus placing each person in a Web of relationships. WebID is a simple and universal identification mechanism that is decentralized and openly extensible. It is currently a work-in-progress open standard within the World Wide Web Consortium (W3C)[7], to which we are actively contributing.

The general idea behind WebID is that *agents* (e.g., a person, an organization, a group, etc.) create their own identities by linking a *unique identifier* in the form of an HTTP(S) URI to a *profile document*, a type of Web page that any Web user is familiar with, and which uses a standardized RDF serialization format. The profile document contains all the necessary information to create a Web of trust which allows people to link together their profiles in a public or private manner. Such a Web of trust may then be used by Web services to make authorization decisions, by allowing access to resources depending on the properties of an agent, such that he/she is known by some relevant people, works at a given company, is a family member, is part of some group, etc.

3.2 WebID-TLS Authentication

The WebID-TLS protocol[8] is a decentralized authentication protocol which enables secure and efficient authentication on the Web. It enables people to authenticate onto any site by simply choosing one of the client certificates proposed

[7] http://www.w3.org/2005/Incubator/webid/spec/identity
[8] http://www.w3.org/2005/Incubator/webid/spec/tls

to them by their browser. These certificates can be created by any Web site for its users. Unlike classic client certificate authentication that relies on Public Key Infrastructure (PKI), WebID-TLS does not require the certificates to be signed by a trusted Certificate Authority. The reason is that WebID-TLS uses client certificates simply as a means to perform public key authentication.

The client certificate contains a field called *Subject Alternative Name* in which the WebID is located, thus linking the certificate (i.e., the key pair) to the user's WebID. During the authentication process, verifiers only need to match the certificate's public key against the public keys listed in the profile document obtained by dereferencing the WebID. From a user's point of view, the complete process of WebID-TLS authentication is simply a one click operation in which he/she chooses the WebID certificate. The user is not required to remember and expose any credentials (e.g., password) in order to authenticate. Work has been done to extend WebID-TLS with access delegation [17], allowing agents to perform tasks on behalf of users.

4 The Read/Write Protocol

The Solid read/write protocol enables application-to-server and server-to-server communication. Currently, two possible methods of reading and writing data are supported. A RESTful method based on the Linked Data Platform (LDP) [8] (a recent Recommendation from W3C), and a method based on SPARQL queries. The LDP protocol supports basic manipulation and retrieval of resources. SPARQL and link-following SPARQL are used for complex data retrieval.

4.1 Basic Data Manipulation using LDP

The Linked Data Platform (LDP) specification defines a set of rules for HTTP operations on Web resources, some based on RDF, to provide an architecture for reading and writing Linked Data on the Web. The most important feature of LDP is that it provides us with a generic and standard way of RESTfully writing resources (documents) on the Web, without having to rely on less flexible conventions (APIs) based around sending form-encoded data queries using POST. LDP focuses on two important concepts, *resources* and *containers*.

LDP Resources (LDPRs) are HTTP resources that comply to simple patterns and conventions defined by LDP. LDPRs can be RDF or non-RDF resources. LDP defines how Web servers should handle HTTP requests to create, access, modify, or delete LDPRs. In Solid, the LDPR is the minimum data granularity accessed by an application (e.g., an event in a calendar application).

LDP Containers (LDPCs) are collections of LDPRs, similar to how blog posts are grouped into blogs, wiki pages are grouped into wikis, and products are grouped into catalogs. LDPCs are themselves LDPRs, which means that one can create a *hierarchy* of nested LDPCs, similar to a directory hierarchy in a file system. Adopting LDP is important for Solid since developers no longer have to create new APIs every time they create Linked Data applications.

4.2 Extensions to LDP

We have found that in some cases, using the existing LDP features was not enough. For instance, to optimize certain applications we needed to aggregate all resources from a container and retrieve them with a single GET operation. We implemented this feature in a similar way to "globbing" in the UNIX shell. Doing a GET on any URI which ends with a "*" will return an aggregate of all the resources that match the indicated pattern. For example, if one would like to fetch all resources of a container in one request, they could do a GET on *https://example.org/data/**. The aggregation process is not recursive, so it will not apply to child containers.

Another useful feature that is not yet part of LDP deals with using HTTP PUT to create new resources. This feature is useful when the client wants to make sure it has absolute control over the URI namespace (e.g., migrating from one pod to another). Although this feature is defined in the HTTP specification [4], we decided to improve it slightly by having servers create the full path to the resource if it did not exist before. For instance, consider a calendar application that uses a URI path structure based on dates (i.e., /2016/05/01/event1) where *event1* is a new event. A PUT request is to create a new resource called *event1*, as well as the missing month (i.e., 05) and day (i.e., 01) containers under /2016/.

4.3 Complex Data Retrieval

LDP cannot express complex data retrieval operations such as filtering and aggregation. Application development can be significantly simplified if the application can delegate such complex operations to the server. In Solid, all pod servers must support LDP, while some servers may optionally support SPARQL.

Applications that rely on SPARQL support can express complex data processing tasks using SPARQL queries and delegate these tasks to a SPARQL-supporting server. We classify SPARQL queries in Solid into *link-following* and *local* queries. A local query accesses only predicates located on the user's pod, while a link-following query is a query that accesses predicates on multiple pods by following links between the user's pod and other users' pods.

The query does not need to explicitly refer to these remote pods. The pod server of the user issuing the query is to recover the links to be followed in other pods. Therefore, the developer does not need to know the actual distribution of the data. The pod server can tell that this is a link following query if it accesses data in two different pods. The SPARQL endpoint of a user's pod is advertised through Link headers that can be discovered when doing HTTP GET/HEAD on the URI of the user's pod.

The server has to support SPARQL query processing on heterogeneous and logically integrated data stores. There are different systems proposed for such query processing, such as Fedx [14] and ANAPSID [1]. A pod server can use one of these systems or develop its own mechanism.

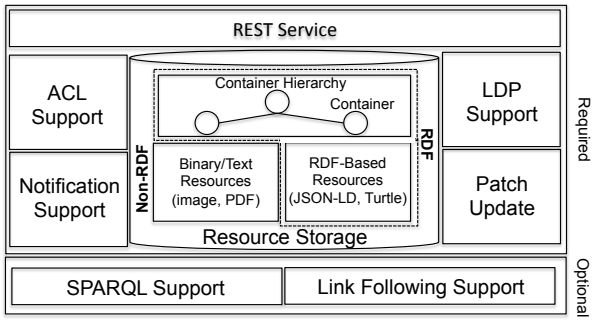


Fig. 2. Pod architecture. A pod stores RDF and non-RDF resources. Pod servers support LDP, patching resources, access control, live updates, and optionally SPARQL.

5 The POD Management System

This section presents an overview of the features that should be implemented by any pod server, illustrated in Figure 2. Pods use LDP to organize data into containers that group together resources, giving each container and resource their own URI. A pod server supports patching resources, access control lists (ACLs), live updates, and optionally SPARQL. We developed several Solid prototype servers, namely **gold**[9], **ldnode**[10], **ldphp**[11], and **meccano**[12], which are currently available as public services on the Web [9].

5.1 Resource Storage

There are several ways in which the underlying storage for RDF data can be implemented in a pod server, e.g., using the file system, a key-value store, a relational database system, or a graph database system (i.e., a triple store). The file system is the option adopted by three of our pod implementations: **ldphp**, **gold**, and **ldnode**. In this case, both RDF and non-RDF resources are stored as files, including the ACL resources and the metadata documents corresponding to non-RDF resources. File systems ensure efficient access to text and binary files, so using a file system suits applications that require only resource-level manipulation, e.g., reading or writing a document.

There are several efficient RDF engines, and it is worth considering using them for storing the RDF data (container metadata and RDF resources). The RDF database system can also be used to store the metadata of non-RDF resources, while the resources themselves are stored elsewhere such as in a file system. Using an RDF database simplifies querying large data sets, efficient data retrieval (i.e, subsets of graphs), as well as patch operations. However, supporting LDP and ACLs requires more work than for a file-based implementation.

[9] https://databox.me
[10] https://databox2.com
[11] https://rww.io
[12] https://meccano.io

Our **meccano** server stores data in the Jena[13] RDF database system and implements all Solid operations via SPARQL queries. **Meccano** also implements complex data retrieval through link-following SPARQL.

5.2 LDP and Patching Support

Pods should support basic LDP [8] operations that apply to resources and containers, namely HTTP GET, POST, DELETE, OPTIONS and HEAD, plus our extensions to LDP discussed in Section 4.1. Supporting LDP operations in a file system is straightforward since they deal with resources that can be mapped directly into the file system namespace. Special adaptors may be needed when using a database system, since each LDP operation has to be mapped to an equivalent query or a set of queries supported by the underlying DB system.

Solid uses basic SPARQL queries to patch resources. Depending on what modifications have to be performed to the resource graph, it uses a succession of DELETE and INSERT statements, separated by a **;** character. For example, renaming the title of a container implies sending a DELETE query followed by an INSERT query. An RDF-based pod server can easily support Solid patches since the patch operations can be directly mapped into SPARQL 1.1 Update.

5.3 Access Control

When a user runs a Solid application, that application can access not only his/her pod, but also the pods of other users. Different users and groups identified by WebIDs can be allowed various forms of access to resources. We adopted the WebAccessControl (WAC)[14] ontology to describe access control at the level of a container or resource. WAC almost follows the access control system used in Linux file systems. The WAC ontology has different types of access modes, namely Read, Write, Control, and Append.

Each resource (containers included) can have an associated ACL resource that can be manipulated just like any regular resource. If a container or resource does not have an ACL, it inherits the authorization of its parent container. The URIs for the ACL resources are advertised through *Link* headers that can be discovered when doing HTTP GET/HEAD on regular resources. For example, a container, whose URI is *https://example.org/storage/*, may have a corresponding ACL resource with a URI ending with *.acl*, i.e. *https://example.org/storage/.acl*. Alternatively, a resource *https://example.org/storage/test* may have a corresponding ACL resource at *https://example.org/storage/test.acl*.

Our **meccano** server maintains ACL resources in the same RDF graph as the data resources. ACLs are enforced transparently by re-writing the LDP or SPARQL queries to check the ACLs against the corresponding triples. This mechanism supports ACLs at the granularity of triples as well as resources.

[13] http://jena.apache.org
[14] https://github.com/solid/web-access-control-spec

5.4 Live Updates

Solid applications can be made more responsive by receiving notifications or live updates from a pod when a resource has been modified. It is much more efficient to have the server notify the clients when changes occur in resources, instead of the client periodically polling a pod for new information. We chose the popular WebSocket protocol [6] to offer live updates by implementing a PubSub system. Clients open a WebSocket connection and *subscribe* to a given resource URI by sending a special command – e.g., *sub https://example.org/data/test*. If any change occurs in that resource (i.e., update or delete), a *publish* event is sent to all the subscribed clients – e.g., *pub https://example.org/data/test* – triggering a refetch of the resource. The WebSocket server URI is the same for any resource located on a given data space (same hostname). To discover the URI of the WebSocket server, clients can use HTTP verbs like GET, HEAD and OPTIONS. The server will include an Updates-Via header in the response – e.g., *Updates-Via: wss://example.org/*.

5.5 SPARQL Support

A pod server can optionally support SPARQL for complex data retrieval. A server that uses an RDF database, such as **meccano**, supports SPARQL naturally. A pod using the file system can also be extended to support SPARQL. An interesting feature of **meccano** is that it supports link-following SPARQL to enable complex data retrieval from multiple servers, thereby simplifying application development. **Meccano** does not assume a priori knowledge about the data or SPARQL endpoints. A preprocessing step is required to analyze the link-following query and identify the endpoints to be contacted to answer the query. An adaptive query execution plan is then generated and executed against these endpoints to answer the query. In the evaluation, we show that **meccano** efficiently supports queries accessing multiple remote pods. Moreover, **meccano** can parallelize queries onto multiple threads on one or more machines, so it can elastically scale to more cores as needed.

6 Evaluation: Applications development and Experiments

To evaluate the developer experience with the Solid platform, we have built multiple social Web applications on Solid [9, 11]. This section gives an overview of the main applications and describes some of the advanced Solid features that they require (e.g., search in a remote pod). The section also conducts experiments to show the scalability of link-following SPARQL using the solid platform.

6.1 Application Development in Solid

The Solid platform includes solid.js[15], a javascript library implementing the Solid protocols. Readers are invited to use the solid.js tutorial[16] [11]. We developed

[15] https://github.com/solid/solid.js
[16] https://github.com/solid/solid-tutorial-intro

Table 1. Solid applications. (*) indicates applications not developed by us.

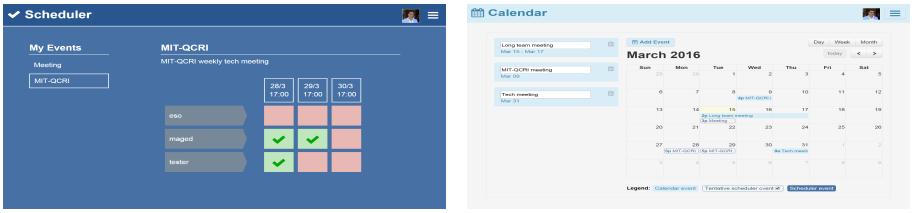
Name	Function	Usable At
contacts	Manage a list of contacts	*http://mzereba.github.io/contacts*
cimba	Microblogging (cf. Twitter)	*http://cimba.co*
calendar	Event manager	*http://mzereba.github.io/calendar*
scheduler	Meeting scheduler (cf. Doodle)	*http://mzereba.github.io/scheduler*
dokieli	Decentralized authoring	*https://dokie.li*
profile-editor	View and update a user's profile	*http://linkeddata.github.io/profile-editor*
warp	Solid file browser	*http://linkeddata.github.io/warp*
zagel	Instant messaging/group chat	*https://solid.github.io/solid-zagel*
inbox	Inbox app to process notifications	*https://solid.github.io/solid-inbox*
***timeline**	decentralized social network	*http://solid-social.github.io/timeline*
***shamblokus**	Strategy game (cf. Blokus)	*http://deiu.github.io/Shamblokus*

several Solid applications for common day-to-day tasks (Table 1). This class of applications accesses the pod of the user running the application and the pods of users that he/she directly connects to (i.e., order of tens or hundreds of pods). For this evaluation, all applications were developed as responsive Web applications, tested in recent versions of Chrome or Firefox.

The Solid protocols guarantees efficient performance for social applications regardless these applications use solid.js or not. However, solid.js aims at accelerating the development lifecycle of Solid applications by writing less code. In order to verify that, on the one hand we developed some of the applications using normal javascript frameworks and libraries, such as AngularJS and rdflib.js[17], to access the users' pods. Examples for these applications are our **cimba**, **contact**, **scheduler**, and **calendar** applications. On the other hand, we developed applications using solid.js such as **zagel** and **inbox**. The readers are invited to review the code and use both categories of applications. To use Solid applications, a user needs to sign up with a pod provider and obtain a WebID (if needed) together with data storage. For example, the pod of user Alice on our **gold** server running at *databox.me* would be *https://alice.databox.me/*.

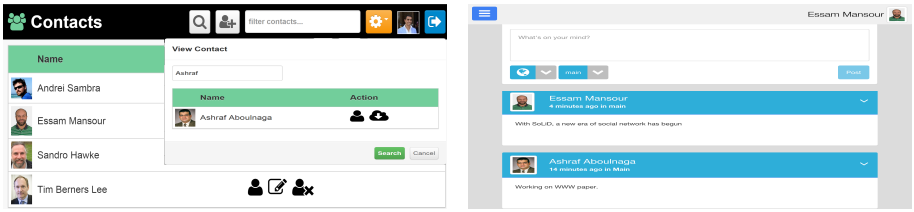
Applications store data in users' pods and access it via LDP. It is possible for all data in a pod to be organized into one container, with each application using a single resource for all of its data. However, it is recommended that applications developed by Solid take advantage of LDP container hierarchies, especially if application data is expected to be large. For example, a contact list application may be designed to hold vCard[18] data in one file, which would make it difficult to share individual contacts, or to store just each address line in one resource, which would be too fine-grained. It is recommended that applications divide data into resources at a reasonable granularity, which affects tasks such as inserting, updating, deleting, or caching data. Note also that ACL policies apply at the level of the resource, which impacts the granularity at which data

[17] https://github.com/linkeddata/rdflib.js
[18] http://www.w3.org/2006/vCard



(a) **scheduler** (b) **calendar**

Fig. 3. Examples of solid applications access shared containers.



(a) **contacts** manager (b) **cimba** for microblogging

Fig. 4. Examples of solid applications need link following queries.

can be shared. For example, one vCard or Event[19] per file allows a user to share individual cards or calendar events with other users, see **contact** and **calendar**.

A Solid application can access containers or resources generated by another Solid application as shown in Figure 3. Hence, Solid supports interoperability at the data level rather than the API level. For example, the **scheduler** application accesses the vCard resources generated by **contacts** to maintain a broad list of candidate participants. Furthermore, **calendar** accesses the schedule events generated by **scheduler** to optionally show these events as tentative scheduler events. Intuitively, users move easily between different applications.

The **calendar**, **scheduler**, **zagel**, and **inbox** are examples of many use cases that can work efficiently using LDP only. In **scheduler**, an event organizer write via HTTP PUT an event invitation in the inbox container of a list of users' pods. The **scheduler** application shows these invitations to the user to accept or reject. In case of acceptance, the application copy only the details of the event via HTTP GET but all responses of the invited users are written in the pod of the event organizer via HTTP PUT. This workflow suits this use case where all invited users fetch responses from one pod using a single HTTP GET. Solid is flexible in supporting different workflows. For example chatting applications like **zagel**, the workflow is to replicate a user's post in both the user's pod and pods of individuals with whom the user is chatting. Thus, each user can access the full chat with an individual/group with a single HTTP GET to the user's pod.

There are also many use cases that need link following queries in order to reduce the number of HTTP requests and offload computations from the client to the pod server, as shown in Figure 4. Examples for this category of use cases

[19] http://motools.sourceforge.net/event/event.html

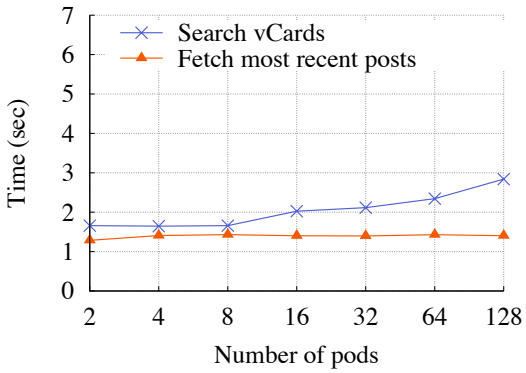


Fig. 5. Scalability of link-following SPARQL for searching vCards to find vCards of "Alice", and fetching the 10 most recent posts from followed channels. Both queries access up to 128 pod servers in the Azure public cloud (in USA and Europe).

are a microblogging, social netwrok, and contact management applications, such as **cimba** [12], **timeline**, and **contacts** respectively. The **cimba** application fetches posts from the channels referenced in the user's subscriptions. A link following SPARQL can be processed by the user's pod server to fetch the top recent posts in each followed channels and return to **cimba** the overall top recent posts. **contacts** is a user interface to social graph information on the user's pod. This application provides an example of the innovative social features supported by Solid, which enable powerful ways for finding and sharing resources based on the social graph of the user. A link following SPARQL query can be processed by the pod server to look for a specific vCard, i.e. a vCard with name Alice, in all pods connected with the user's pod via the WebID in his/her vCards.

Our experience with Solid application development confirms that Solid provides a feature-rich platform that supports portability and interoperability. Applications can work with multiple pod server implementations, and can be easily changed without changing the data (e.g., by forking and adding features).

6.2 Scalability of Link-Following SPARQL

This section evaluates the performance of LF SPARQL on the **meccano** pod server as we scale up the number of remote pods accessed by a Solid application. For this experiment we use the **contacts** and **cimba** applications. The data used in this experiment is synthetically generated RDF data, including user profiles, vCards, and microblogging posts. We generated data for 128 users, where the data for each user contains 1000 vCards and 5000 posts. The vCards and posts are all public, and hence accessible by the SPARQL queries. These users are assigned to different pods servers managed by **meccano**.

We ran our experiments on 19 virtual machines (VMs) on the Microsoft Azure public cloud (D4 instance). Each virtual machine has 28GB RAM and 8 CPU cores. The VMs are distributed among 7 different geographic regions in

USA and Europe. The 128 user pods are distributed evenly among **meccano** instances on the 19 VMs. Each pod is given one core and 4GB of RAM.

Figure 5 shows how **meccano** scales for the two queries as we increase the number of pods accessed by both queries from 2 to 128. **Meccano** analyzes the input query and decomposes it into subqueries. During this analysis, **meccano** also follows links to identify the corresponding pods (including the user's pod) to which the subqueries will be sent. The details of this query decomposition are omitted due to lack of space. All remote subqueries are submitted simultaneously to the corresponding pods using HTTP requests. The results are collected by **meccano** and joined together if needed. Our decomposition mechanism parallelizes access to remote pods and retrieves only relevant results from these pods, which enables **meccano** to scale well for a broad class of Solid applications. Figure 5 shows that the time for fetching recent posts is almost flat as the number of pods increases. This is because the LIMIT clause (i.e. top recent posts) eliminates unnecessary results. The vCard search query is not as flat as the recent posts query, although it still scales sublinearly (doubling the number of pods less-than-doubles the execution time).

7 Related Work

Decentralizing social Web applications has been a long-standing goal for research projects as well as the developer community. However, many of these projects focus primarily on storing data in a decentralized way and sharing it within the social networks of users, without a strong emphasis on how applications will use this data. In contrast, Solid has a strong focus on decoupling data from applications and in addition ensuring that applications have a simple, generic and well defined way to access the data stored in the users' pods.

Diaspora[20] is a decentralized social network platform that enables users to choose the server where their data is hosted and even run their own data hosting server. In that sense, it is similar to Solid. However, the main focus in Diaspora is to act as a social network, where social data is shared between users using specific APIs, and not running diverse applications on stored data. Unfortunately, it does not offer a well defined way to use the same data with different applications. Note that Diaspora uses the term pod to refer to a data hosting server. A Diaspora "pod" is what Solid would refer to as a "pod server". Similar efforts include Micropub[21] and Pump.io[22].

Musubi [3] is a decentralized social network platform that enables users to share data from mobile devices. The focus of Musubi is exchanging secured messages (via public key encryption) among friends or groups without an intermediary. Safebook [2] is another social network platform that focuses on security and privacy by controlling how applications access the identities and private data of users. Safebook proposes a decentralized architecture for identity and

[20] https://diasporafoundation.org
[21] https://indiewebcamp.com/Micropub
[22] http://pump.io

authentication in which the identities of users are hidden from applications and multiple independent entities cooperate to grant access to user's data. The data exchanged on platforms like Musubi or Safebook can be sent and received by applications such as photo sharing or online games. However, these platforms do not specify how applications should store and access their data, so there is no simple way to replace one application with another that reuses the same data. That is, the data is not decoupled from the applications.

Some decentralized social network platforms decouple data from applications. Two notable examples are WebBox [18] and the distributed semantic social network architecture (DSSN) described in [16]. As Solid, both of these systems store user's data as Linked Data in a decentralized way. Also similar to Solid, both systems rely on WebID for decentralized identity, authentication and access control. In WebBox, each data storage service exposes a SPARQL endpoint, and applications manipulate the data via SPARQL queries and updates, or via HTTP GET requests. DSSN uses a publish/subscribe mechanism where applications subscribe to feeds and users publish data in these feeds. In contrast, Solid offers the full power of LDP for simple data interactions (e.g., hierarchical data organization, fine-grained manipulation) and additionally allows the use of link-following SPARQL for complex data retrieval. It also works as a generic storage platform upon which a significantly larger number of applications can be built.

Solid and similar platforms mentioned in this section provide mechanisms for users and applications to create and share data in a decentralized way. Thus, user's data is never stored in a "data silo". Another approach is to let the user's data reside in silos and provide a way for applications to access data in these silos. This can be done by providing services such as Gigya[23] and Janrain[24] that enable applications to access different social networks. It can also be done by providing an integration layer that imports data from different data sources – e.g., existing social networks, e-mail, web sites, and local files – and stores this data in a personal datastore that is accessed by users and applications (e.g., [15, 19]). Such approaches can be viewed as complementary to Solid, since Solid may benefit by gaining access to data that is stored in silos.

8 Conclusion

Solid uses Semantic Web technologies to decouple user data from the applications that use this data, which makes it easy for users to switch between applications that use the same data, and to switch between storage providers that host the data. We presented the protocols used by Solid for decentralized authentication and data access, and we described the necessary features required by a server that implements these protocols. Our experience in building several applications that use these servers demonstrates that Solid is a viable platform for social Web applications. We aim at creating a developer community for Solid applications.

[23] http://www.gigya.com
[24] http://janrain.com

References

1. M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proc. Int. Semantic Web Conf. (ISWC)*, pages 18–34, 2011.
2. L. A. Cutillo, R. Molva, and M. Önen. Safebook: A distributed privacy preserving online social network. In *Proc. Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–3, 2011.
3. B. Dodson, I. Vo, T. J. Purtell, A. Cannon, and M. S. Lam. Musubi: Disintermediated interctive social feeds for mobile devices. In *Proc. World Wide Web Conf. (WWW)*, pages 211–220, 2012.
4. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–HTTP/1.1, 1999.
5. D. Hardt. The OAuth 2.0 authorization framework. 2012.
6. I. Hickson. The WebSocket API. Candidate recommendation, W3C, Sept. 2012. http://www.w3.org/TR/2012/CR-websockets-20120920/.
7. T. Inkster, H. Story, and B. Harbulot. WebID-TLS Specification. 2013. http://www.w3.org/2005/Incubator/webid/spec/tls/.
8. A. Malhotra, J. Arwe, and S. Speicher. Linked Data Platform Specification. W3C Recommendation, 2015. http://www.w3.org/TR/ldp/.
9. E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee. A demonstration of the solid platform for social web applications. Demo at the World Wide Web Conf. (WWW), 2016.
10. N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. Openid connect core 1.0. *The OpenID Foundation*, 2014.
11. A. Sambra, A. Guy, S. Capadisli, and N. Greco. Building decentralized applications for the social web. Tutorial at the World Wide Web Conf. (WWW), 2016.
12. A. V. Sambra, S. Hawke, T. Berners-Lee, L. Kagal, and A. Aboulnaga. CIMBA - client-integrated microblogging architecture. In *Proc. Int. Semantic Web Conf. (ISWC)*, pages 57–60, 2014.
13. A. V. Sambra, H. Story, and T. Berners-Lee. WebID Specification. 2014. http://www.w3.org/2005/Incubator/webid/spec/identity/.
14. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *Proc. Int. Semantic Web Conf. (ISWC)*, pages 601–616, 2011.
15. S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam. PrPl: A decentralized social networking infrastructure. In *Proc. Workshop on Mobile Cloud Computing and Services*, 2010.
16. S. Tramp, P. Frischmuth, T. Ermilov, S. Shekarpour, and S. Auer. An architecture of a distributed semantic social network. *Semantic Web*, 5(1):77–95, 2014.
17. S. Tramp, H. Story, A. Sambra, P. Frischmuth, M. Martin, and S. Auer. Extending the webid protocol with access delegation. In *Proc. Workshop on Consuming Linked Data (COLD)*, 2012.
18. M. Van Kleek, D. A. Smith, N. R. Shadbolt, and mc schraefel. A decentralized architecture for consolidating personal information ecosystems: The WebBox. In *Proc. Workshop on Personal Information Management (PIM)*, 2012.
19. D. Vianna, A. Yong, C. Xia, A. Marian, and T. Nguyen. A tool for personal data extraction. In *Proc. Int. Conf. on Data Engineering (ICDE) Workshops*, 2014.