Writing Linked Data Event Streams to a Solid Container

Wout Slabbinck(✉) [0000−0002−3287−7312], Ruben Dedecker[0000−0002−3257−3394], Ruben Verborgh[0000−0002−8596−222X], and Pieter Colpaert[0000−0001−6917−2167]

IDLab, Departement of Electronics and Information Systems, Ghent University - imec, Belgium
wout.slabbinck@ugent.be

Abstract. The Solid specification allows CRUD operations on top of RDF and non-RDF resources. Its ambition is to become the universal read/write specification for Linked Data knowledge graphs, encompassing features like authentication, access control, or telling apps how to structure data within a container. When performing a write-action to a resource today, the write will be permanent, and the history is not kept. Some use cases however need to be history-aware: being able to see what a certain value used to be, supporting undo-operations on information resource changes across apps, etc. In this paper, we design an interface in which clients that want to be version aware, and those who do not, can interoperate. We designed a Solid orchestrator – an intermediary server that can perform maintenance tasks on a Solid container – that can change the write-location the client will use, set access control for historic resources, and create a Linked Data Event Streams (LDES) description of this immutable log. Using this implementation, we learned that a client can automatically structure the history by writing to the configurable ldp:inbox location. In order to support symmetric read/write for clients that do not want to be version aware, we however need server functionality, as the current Solid protocols do not allow to do multiple operations in an atomic fashion. While the LDES specification does not specify how to write to an LDES, it is now enabled thanks to the Solid specification.

Keywords: Linked Data · Solid · Linked Data Event Streams (LDES) · Linked Data Platform (LDP)

1 Introduction

The Linked Data Platform (LDP) specification enables clients to perform create, read, update and delete (CRUD) operations on both non-RDF and RDF data on an LDP server. An LDP server can thus be used to read and write Linked data on the web.

Hosting a bare LDP server on the web, however, introduces several problems. For example, everybody can write anything on the server. It is thus possible for any client to update or even delete certain documents containing the RDF data. Furthermore, when other clients use the server as data storage, they would not

even know whether documents have changed. The Solid specification builds upon several existing web standards, including LDP, and overcomes that anybody can read/write to all resources by incorporating an access control specification. Using WebSockets, it is also possible to know whether an update has happened. However, it is unknown what has changed. On top of this, the subscription of the WebSocket is not on pod level but its scope is only on resource or container level.

For fast-moving data, like sensor or location data, not only the current value matters. All the versions of that data (i.e. the previous values) also matter, especially when further analysis is performed on the whole series of data.

It is thus necessary to have a data structure that is version-aware and where all its items are immutable. Therefore, the Linked Data Event Streams (LDES) specification is used. The specification defines how to structure a collection of (versioned) items and how to interpret them, but it does not specify a standardised way to write to an existing LDES.

This paper introduces a way to write an LDES in a Solid container. To facilitate clients writing to an LDES, an orchestrator is implemented that amongst other tasks takes care of creating new containers as a fragment for the LDES. Furthermore, guidelines are given for a server managed way to create an LDES in a Solid container that implicates that a client does not have to understand how to write to an LDES in LDP or LDES at all.

First, Section 2 introduces core technologies and specifications together with related work about this topic. In Section 3, an in-depth explanation is given about how to write an LDES in a Solid Container. And finally, this paper concludes in Section 4 and elaborates on future research on how to write LDESs.

2 Background

The Linked Data Platform [17] protocol provides a standardized way to expose CRUD operations on collections of RDF Resources.

An LDP stores data as resources, where each LDP Resource has its own URI. There is a distinction between two types: First, there are LDP Containers, which can be compared to a directory in a file system and are used to contain themselves or the second type, the document LDP Resources. They can be seen as files and thus those document stores contain data.

To constrain specific resources within an LDP API to specific users and applications, the novel Solid protocol [6,19] can be used to provide this authorization. Since next to LDP, the Solid protocol consists, among others, of Solid-OIDC [8] for authentication and Web Access Control (WAC) [2] for authorization.

The *TREE* hypermedia specification [10] addresses navigating over a large collection of RDF Resources. The core concept of this specification is the TREE *Collection*, which is a set of members. To improve navigating a large collection, it may consist out of different *fragments* which are identifiable with their own URI. Another core concept is the TREE View, which can reach all the members of this fragmented collection via relations.

An extension of TREE hypermedia, Linked Data Event Streams (LDES) [9], defines the strategy to publish those collections. The Event Stream is a TREE Collection where each member is *immutable*. Furthermore, the *Version Materializations* in LDES allow for members to have multiple versions where each version has a timestamp.

When applications use data, they assume that certain conditions are met. Not conforming to those conditions may result in applications not working at all. With RDF Data shapes, it is possible to define a set of constraints to which a piece of data must conform. Two specifications Shape Constraint Language (SHACL) [12] and Shape Expressions (ShEx) [14] exist to declare such shapes and both have their own validators.

2.1 Related Work

Storing fast-moving data with a secure and authorised approach is thus done as follows: The dataset is stored as a time-based LDES. This event stream thus consists of all the resources and the versions of those resources. A Solid Container is used to store this LDES, thus interactions are done with the LDP API when sufficient access control is granted.

In the next paragraphs, similar approaches on storing multiple versions of the same resource on an LDP are presented.

First there is the *Fedora API Specification* [5], a solution for digital libraries and archives, and *Trellis Linked Data Server* [1], a horizontal scalable LDP implementation. Both the Fedora API specification and Trellis are built using multiple Web standards like LDP for resource management, WAC for access control and the Memento Protocol [16] for resource versioning. Memento facilitates obtaining representations of prior states of resources. Retrieving a list of all different versions is done in Memento with TimeMaps. While a TimeMap works well for resources at archives, where the data does not change often, it is not preferred for fast-moving data. There, a big amount of versions per resource are present, resulting in large TimeMaps per resource. Retrieving big files takes significantly more time, thus large TimeMaps impact the performance. Therefore, a better structure is required that does not have to download all the information of each version. An LDES is suited as a structure because it can be navigated with hypermedia.

Next, in [15] we tried creating an LDES that continuously stores sensor measurements, where each new measurement of a sensor is a new version. The LDES was stored in an LDP with shape support. Shape support was provided using an LDP with Shape Tree [13] support. The LDES was updated using HTTP `PATCH` requests and a service to fragment the LDES. Results showed that using PATCH at one endpoint for updating the LDES was not scalable as adding new data took longer when the size of the LDES grew bigger. The fragmentation service helped, however without a retention policy, that removes certain older parts of the LDES, it was still becoming slower when the LDES grew bigger. However, some use cases require keeping all the data. In that case, a better

approach, one that does not rely on a retention policy for performance, to create an LDES is necessary.

3 Writing Linked Data Event Streams

We designed two approaches to write Linked Data Event Streams. First there is the *Client-Aware approach* where clients write directly to a time-based LDES stored on an LDP. Next, there is the *Client-Agnostic approach*. This approach retains compatibility with the Solid specification where needed, enabling the server to virtualize a Solid environment over an LDES back-end. All the LDP operations are here internally translated to interactions with the LDES.

3.1 LDES in LDP

We coin the term *LDES in LDP* to describe a time-based fragmented LDES that is stored on a Linked Data Platform. Which allows to interact with LDESs using the LDP API.

An LDES is initialized in a data pod as a container and identifyable with an URI. In this container, the view (a root node) is found[1] which contains relations to fragments of the LDES. Finally, a fragment container consists of several LDP resources, indicated by *ldp:contains*[2]. Listing 1 is an example of a root, where `http://example.org/{container}/`) is the base URI of the LDES in LDP. This root consists of one *TREE hypermedia Relation*, where its class and properties indicate that all members which were created after December the 15th can be found by traversing to node `http://example.org/{container}/1639526400000/`.

Listing 1. LDES in LDP root Resource

```
@prefix : <http://example.org/{container}/root.ttl#> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix tree: <https://w3id.org/tree#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/{container}/root.ttl> rdf:type tree:Node ;
    tree:relation [
        a tree:GreaterThanOrEqualToRelation ;
        tree:node <http://example.org/{container}/1639526400000/> ;
        tree:path dct:modified ;
        tree:value "2021-12-15T00:00:00.000Z"^^xsd:dateTime
        ] .

```

[1] The view can be found at *containerURI/***root.ttl**

[2] `https://www.w3.org/ns/ldp#contains`

```
:Collection a <https://w3id.org/ldes#EventStream> ;
    tree:shape <http://example.org/{container}/shape> ;
    tree:view <http://example.org/{container}/root.ttl> .
```

Adding a Resource The method for adding a resource remains the same as for a normal LDP Resource creation: with an HTTP POST request. However, an application that adds a member to the Event Stream must know where to write to. To indicate the write location, a property already defined in the LDP specification is reused: the LDP Inbox[3] (*ldp:inbox*[4]). Thus a triple of the form *<baseContainer> <http://www.w3.org/ns/ldp#inbox> <locationURI>.* is added to the metadata of the base container. This location URI is retrieved via the Link Header as the link value to the relation *http://www.w3.org/ns/ldp#inbox* when sending a GET or HEAD HTTP request. Listing 2 shows such a request to the base URI and Listing 3 shows the Link Header response. Finally, a member can be added to the LDES with an HTTP POST request to the obtained location URI.

```
HEAD /{container}/ HTTP/1.1
Host: http://example.org
```

Listing 2: HEAD request to the inbox

```
HTTP/1.1 200
link: <http://example.org/{container}/1639526400000/>; rel="http://www.w3.org/ns/ldp#inbox"
```

Listing 3: HEAD response from the inbox

Improving Interoperability WS: @Ruben D, is het al beter leesbaar nu? When it is known a priori that the LDES will only have members with a certain predefined data model, it is possible to initialise the LDES in LDP with a shape. To enforce shape validation executed by the LDP, the validator requires to know which shape resource to use. Therefore, the constrained by property of LDP(*ldp:constrainedBy*[5]) will be used to encode an URI to the shape resource in the metadata of each fragment container. Since all requests to add data that does not conform to the shape will be rejected, the resulting Event Stream consists of members that all conform to the shape.

[3] The LDP Inbox was added in the Linked Data Notifications (LDN) specification [7].
[4] `https://www.w3.org/ns/ldp#inbox`
[5] `https://www.w3.org/ns/ldp#constrainedBy`

Basic LDES-Orchestrator To make a Linked Data Event Stream scalable, the ever-growing LDES was introduced. This complicates the writing process as it requires clients to create new containers, know the current write location and maintain the root. Furthermore, clients do not always have permissions to update certain parts of the LDES in LDP.

An active service component is required to orchestrate the writing process, which removes overhead for the clients and executes tasks that clients themselves can not perform. For this, we introduce a service called the Basic LDES-Orchestrator (henceforth shortened as Orchestrator). This Orchestrator has four roles:

- Creation of new LDP Container: when the current relation is deemed full, a new container is created with added metadata to indicate shape support
- Writable container indication: at the base container, update the metadata about the LDP Inbox
- Maintain the root of the LDES: add triples with TREE syntax to keep the view up to date
- Access control: when Solid is used, the Access Control List (ACL) files must be updated

Create Containers Downloading a document on the internet takes time proportional to the location of the server versus the location of the client, the bandwidth and the size of the document. Designing LDES in LDP while minimizing that time, results in controlling the size of documents where possible: the container size. When a container contains a large number of resources, the serialization of the information of that container is large as well. This results in a bottleneck for the performance as loading the container page takes longer.

To overcome this bottleneck, every time the current container page is deemed full, a new, empty container is created. Furthermore, when the LDES in LDP is initialised with a shape, metadata must be added to this container to further impose this constraint.

Writable Container Indication When a new container is created, the Inbox must be updated as well. Clients that want to add a member to the LDES can then find the container where they can write new resources. It is the responsibility of the Orchestrator to update that triple in the metadata.

Maintain the View The TREE Hypermedia specification states that a view of a collection must reach all its members. Therefore on each creation of a new container, which is a new fragment of the collection, the view must be updated. Thus a relation is added in the root by the Orchestrator for each new fragment.

Update ACL Files In case a Solid pod is used as a back-end, ACL resources are responsible for making sure that it is impossible to add new resources to containers that are not indicated as writeable. With an ACL resource in place in the current fragment container, it is enforced that only new resources may be

added there. This is done by providing read[6] and append[7] rights in the ACL resource of that container.

Architecture In Fig. 1 the LDP (here a Solid Community Server (CSS) [3]), the LDES in LDP and the Basic LDES-Orchestrator are merged together as the complete architecture. This overview shows clearly the structure of the LDES. The base container contains one or more *fragments* and the *root* of the LDES, a *metadata* resource and the *shape*. The *metadata* resource indicates the fragment that is writable. The *root* resource contains the Event Stream and the view with relations to all LDP containers (fragments) in {*container*}. Each fragment contains an *ACL resource*, a *metadata resource* and several other LDP Resources which are the members of the Event Stream. In the metadata, a constraint is placed to indicate that all members conform to the *shape*.

Finally, the Orchestrator is responsible for creating new fragments when the current one is deemed full.

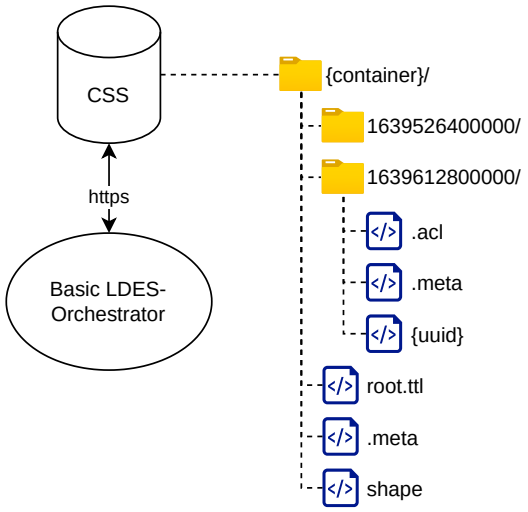

Fig. 1. Architecture of the LDES in LDP

Real Example At `https://tree.linkeddatafragments.org/announcements/`, a public, shape constrained LDES in LDP can be found which is used for publishing metadata of DCAT Application Profiles [4] about datasets or data services, or

[6] `http://www.w3.org/ns/auth/acl#Read`
[7] `http://www.w3.org/ns/auth/acl#Append`

metadata of a TREE View. As LDP, a CSS instance is used with shape support[8]. On the server where it CSS resides, the Basic LDES-Orchestrator runs where a fragment container is deemed full when 100 resources or more are present [9].

3.2 Symmetric Read/Write trough a Version Materialized View

PC: explain the difference between a version aware representation and the version materialized representation ==> why is it important?

With LDES in LDP it is possible to build up the history of a given resource using the Version Materializations of that resource[10], and synchronize and replicate this LDES with the use of LDES Action and LDES Client[11]. Unfortunately, not all clients can understand LDES in LDP. Therefore, a proposal is being presented that combines the simplicity to perform CRUD operations with RDF data on an LDP and the expressivity of an LDES in LDP. Here, the LDES in LDP is used as the source for the resources that are shown in the LDP.

Architecture Figure 2 shows the structure when the LDP is combined with the LDES in LDP. The resources that are present in the {*container*} are a view derived from the LDES in *feed*, which is stored as an LDES in LDP. More specifically a view that only contains the latest version of a given member in the Event Stream. Thus when a resource has multiple versions (e.g. due to it being edited) only the latest version will be shown in the LDP. However, the whole history of those resources can be retrieved from the feed.

With this proposal, reading from an LDES in LDP for clients who are not aware is thus solved. Though in order to use this feed as storage and let the LDP still appear as normal while creating, updating and deleting resources, additional modifications are required.

Creating An HTTP POST request is used to create a resource in an LDP. When a POST request is sent to an LDP Container three things happen: an identifier is created by the server[12] for the created LDP Resource, the body of the request becomes the LDP Resource and metadata is added in the container to indicate that it contains the new resource.

For LDES Write, the LDP behaviour for a POST is just the first step. After that, the body of the request is combined with two extra triples to indicate the version-specific representation. This is then added to the *feed* LDES.

[8] Branch feat/shape-support at `https://github.com/woutslabbinck/community-server`

[9] The Orchestrator uses the following package: `https://www.npmjs.com/package/@treecg/ldes-orchestrator`.

[10] Version Materializations defined in the LDES specification `https://semiceu.github.io/LinkedDataEventStreams/#version-materializations`

[11] `https://github.com/TREEcg/event-stream-client` [18]

[12] When a slug is provided in the Header of the request, a server can choose to use that slug as an identifier.

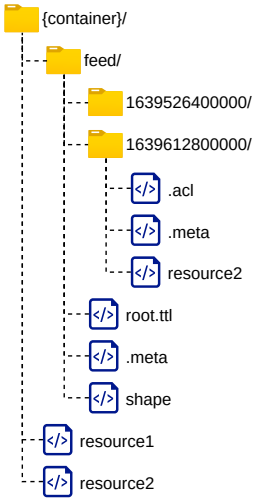

Fig. 2. Architecture of the version materialized view

The first triple is to indicate the time of the creation of the Resource, the second triple is the reference to the identifier of the resource. An example of those triples when the server chose *http://example.org/{container}/resource2* as identifier is shown in Listing 4.

Listing 4. Version specific triples added to an LDP Resource

```
1 @prefix dct: <http://purl.org/dc/terms/> .
2 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3
4 <http://example.org/{container}/feed/1639612800000/{uuid}> dct:issued
      "2021-12-16T10:00:00.000Z"^^xsd:dateTime .
5 <http://example.org/{container}/feed/1639612800000/{uuid}> dct:
      isVersionOf <http://example.org/{container}/resource2> .
```

Updating Updating LDP Resources can be done in two ways. First, there is an HTTP PUT request which replaces the resource with the body that accompanies the request. The second option is using an HTTP PATCH request that uses a SPARQL Update [11] query, where first the server applies the changes and then the result is stored as the updated resource.

An LDP with LDES Write support stores those updates in the *feed* as the newest version. Thus when using PUT, the whole body of the request together with the version-specific triples are added to *feed*. After applying the changes using a PATCH request, the resulting resource is accompanied with the version-specific triples and appended to the *feed*.

Deleting An HTTP DELETE request to an identifier of a resource results in the removal of that resource and its corresponding metadata in the parent container. In the *feed* however, all the versions are not removed because of two reasons. The first one is that an LDES is immutable, meaning that members can not be edited once they are in an LDES. The second reason is that the history of this resource would be removed as well.

Thus next to the LDP behaviour, an item consisting of three triples is added to the *feed* to indicate a resource has been removed. Two of those triples are the version-specific ones, the last one is just an indication that the resource has been removed from the LDP. For this last triple "*https://example.org/versionMemberURI ldes:state ldes:removed .*" is proposed.

LDES in LDP Aware Clients When an application knows how to append to an LDES in LDP, it may add updates of resources directly to the *feed*. As stated previously, this would also change the contents of the LDP Container because it is just a derived view of the *feed*.

4 Conclusion and Future Work

LDES in LDP presents a new approach to building a Linked Data Event Stream. Using a Solid server as LDP means that the creator can control who can view and update this LDES. The Basic LDES-Orchestrator service ensures that containers within an LDP contain a controlled amount of LDP Resources to improve the performance of interacting with the LDES in LDP. Extending the CSS with shape support improve the interoperability of the resources contained in a constrained container. As, then, only resources that conform to the defined shape can be stored.

Although working with versioned resources was already provided with Memento, an alternative to TimeMaps is desirable when working with fast-moving data. Furthermore, with this streaming approach to building an LDES, a novel option is introduced to publish LDESs, as currently datasets have to be transformed to an LDES, and maintain ones published already.

In future work, when LDESs are used as base storage in a client-agnostic way, many derived views can be build on top of them by other parties. To facilitate this ecosystem, we introduce the idea of virtual containers. Virtual containers are like ... **WS**: todo which would allow for those derived views to always be up to date with the source LDES. Time-series analysis requires rapid retrieving of the LDES at any moment, which means that an even more efficient way to consume an LDES is required. For this aggregations of LDESs at certain points in time would facilitate retrieving the versions that are desired for analyses.

References

1. Trellis Linked Data Server, `https://www.trellisldp.org/`

2. Web Access Control (WAC), https://solidproject.org/TR/wac
3. Community Solid Server (Jan 2022), https://github.com/solid/community-server, original-date: 2020-05-19T09:00:39Z
4. Albertoni, R., Browning, D., Cox, S., Beltran, A.G., Perego, A., Winstanley, P.: Data Catalog Vocabulary (DCAT) - Version 3 (2020), https://www.w3.org/TR/vocab-dcat-3/
5. Armintor, B., Cowles, E., Lamb, D., Warner, S., Woods, A.: Fedora API Specification 1.0 (Nov 2018), https://fedora.info/2018/11/22/spec/
6. Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K., Bingham, J., Zagidulin, D.: The Solid Protocol, https://solidproject.org/TR/protocol
7. Capadisli, S., Guy, A.: Linked Data Notifications (2016), https://www.w3.org/TR/ldn/
8. Coburn, A., Pavlik, E., Zagidulin, D.: SOLID-OIDC, https://solid.github.io/authentication-panel/solid-oidc/
9. Colpaert, P.: Linked Data Event Streams spec (Feb 2021), https://w3id.org/ldes/specification
10. Colpaert, P.: The TREE hypermedia specification (Feb 2021), https://w3id.org/tree/specification
11. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update (Mar 2013), https://www.w3.org/TR/2013/REC-sparql11-update-20130321/
12. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL) (Jul 2017), https://www.w3.org/TR/shacl/
13. Prud'hommeaux, E., Bingham, J.: Shape Trees Specification, https://shapetrees.org/TR/specification/
14. Prud'hommeaux, E., Boneva, I., Gayo, J.E.L., Kellog, G.: Shape Expressions Language 2.1, http://shex.io/shex-semantics/
15. Slabbinck, W., Jeroen Hoebeke, Verborgh, R.: Interoperabiliteit tussen applicaties met behulp van Solid voor het beheren van sensordata in slimme woningen (2021), http://lib.ugent.be/catalog/rug01:003014963, publisher: 2021.
16. Sompel, H.V.d., Nelson, M., Sanderson, R.: HTTP Framework for Time-Based Access to Resource States – Memento (Dec 2013). https://doi.org/10.17487/RFC7089, https://rfc-editor.org/rfc/rfc7089.txt, issue: 7089 Num Pages: 50 Series: Request for Comments Published: RFC 7089
17. Speicher, S., Arwe, J., Malhotra, A.: Linked Data Platform 1.0 (Feb 2015), https://www.w3.org/TR/ldp/
18. Van Lancker, D., Colpaert, P., Delva, H., Van de Vyvere, B., Rojas Meléndez, J., Dedecker, R., Michiels, P., Buyle, R., De Craene, A., Verborgh, R.: Publishing base registries as Linked Data Event Streams. In: Brambilla, M., Chbeir, R., Frasincar, F., Manolescu, I. (eds.) Proceedings of the 21th International Conference on Web Engineering. Lecture Notes in Computer Science, vol. 12840, pp. 28–36. Springer (May 2021). https://doi.org/10.1007/978-3-030-74296-6_3, https://link.springer.com/chapter/10.1007/978-3-030-74296-6_3
19. Verborgh, R.: Re-decentralizing the Web, for good this time. In: Seneviratne, O., Hendler, J. (eds.) Linking the World's Information: A Collection of Essays on the Work of Sir Tim Berners-Lee. ACM (2022), https://ruben.verborgh.org/articles/redecentralizing-the-web/