Linked Data Notifications for RDF Streams

Jean-Paul Calbimonte[1]

[1]Institute of Information Systems,
University of Applied Sciences and Arts Western Switzerland,
HES-SO Valais-Wallis, Sierre, Switzerland
{name.surname}@hevs.ch

Abstract. Linked Data Notifications (LDN) is a W3C recommendation for interchanging notifications on the Web through a decentralized protocol. As LDN is not specific to any application domain, this paper analyzes how it can be used to enable a decentralized communication among senders, receivers and consumers of RDF streams. We propose extensions to the protocol for this particular use case, and we show the feasibility with an initial implementation of an LDN-based RDF stream interface.

1 Introduction

Linked Data Notifications (LDN) [7] is a new W3C Recommendation[1] for decentralized data interchange of notifications on the Web. The protocol specified by LDN has the potential to be used for virtually any type of notifications, including social media activity, sensor updates, or document updates, to name some examples. Even though the adoption of this recommendation is still to be assessed, its generality and simplicity make it an interesting option for different types of applications on the Web, for which extensions and/or profiles could be defined.

In this paper we focus on the case where the notification data has a streaming nature, and is represented using RDF [8]. Streams can be seen as potentially infinite sequences of data items, where recent items are usually more relevant that older ones. The requirements commonly related to processing data streams [13], establish among other things, that reactivity, data flow handling, scalabiltiy, and querying, should be guaranteed by a stream processor. In the case of RDF streams, RSP (RDF Stream Processing)[2] engines have been implemented with the goal of providing continuous querying, complex event processing and/or incremental reasoning [3, 10, 6, 9, 1]. However, in most of these cases, these engines stopped at the processing level, without indicating how the streams would be fed, and how the processing results would be consumed on the Web.

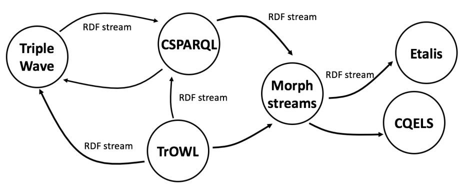

Fig. 1. Network of RSP actors communicating and sharing RDF streams with one another.

[1] https://www.w3.org/TR/ldn/
[2] For more details, see https://www.w3.org/community/rsp/

In this context, we explore the possibility of using LDN as the backbone protocol for sending, receiving and consuming RDF stream elements, which in this case would be seen as notifications. We first explore the feasibility of using the protocol as-is, with the assumption that underneath the LDN actors there might be RSP engines handling the RDF streams. This scenario was envisioned years ago [5], as depicted in Figure 1, and we believe that the protocol presented in this paper will contribute to its achievement. We show that indeed, the usage of LDN with certain extensions shows promising characteristics that indicate its possible adoption as a Web-based interchange mechanism for RDF streams. We describe the proposed specific adaptation to LDN, as well as a working prototype that encapsulates a well-known RSP engine in an LDN-enabled interface. With this feasibility experiment, we intend to contribute to the wider usage of RDF stream technologies on the Web.

2 Linked Data Notifications

The LDN protocol defines three basic types of actors: *sender*, *receiver*, and *consumer*, and the notifications refer to (or are about) a certain *target*. The target is detached of its *inbox*, which is the endpoint where notifications can be consumed or sent. As the name reflects it, senders may send notifications to an inbox, receivers may accept them and make them available, and consumers may retrieve them. The fact that a target is not necessarily attached to its inbox, makes it possible to separate a Web resource from the endpoint where notifications will be handled. As it can be seen in Figure 2 (left), a discovery process allows senders and consumers to retrieve the inbox location through a simple `GET`/`HEAD` HTTP request. Once the inbox location is known, senders can `POST` notifications to it, and consumers may `GET` the references to notifications contained in the inbox (see Figure 2, right).

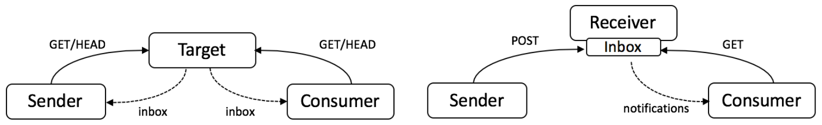

Fig. 2. LDN. Left: Discovery process of a target inbox from a sender and consumer. Right: sending and retreiving notifications from an LDN inbox.

The design principles of LDN imply that notifications are not merely transient messages that flow within a system, but Web resources that can be identified, accessed and shared across applications. In this way, notifications that were posted by one application can be retrieved by a different one, without the need of any inter-dependence between the two, or the notifications themselves, which reside in the inbox (Figure 3). The LDN specification does not provide further details on certain aspects, such as how the inbox contents should be handled, how notifications should be persisted, or how optimizations could be made for accessing and producing them. This openness leaves certain freedom

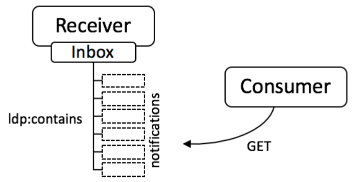

Fig. 3. Retrieving individual notifications from a receiver in LDN.

for implementers to use it as a core interaction model, upon which specific formats, profiles and constraints can be added.

3 Notifications for RDF Stream Processors

Handling data flow in RDF streams is fundamentally different from traditional RDF data management in different ways.

Notifications & storage. First, the streaming nature of the data implies that conventional RDF storage is not an option. RDF stream elements flow through an RSP engine that processes them as they arrive, and produces results continuously. This is the case, for instance, in an RSP query processor, where queries produce continuous answers over the stream of data. The stream is not stored anywhere, but it flows through the engines. If we visualize an RDF stream element as a notification, then we are confronted with the problem that the notification is almost certainly bound to *fade* as the time progresses, and therefore it might be impossible (but yet desirable) to reference it again.

Notification resolvability. A second important difference, which comes as a consequence of the first one, is that an incoming RDF stream element, once processed by an engine, is consumed and may not be retrievable again. Instead, an engine will produce a streaming response of elements, based on the input stream (or streams). These characteristics lead to a slightly different model, where a notification can be part of an RDF stream *input*, or an *output*. Still, nothing prevents from having and inbox of RDF stream notifications that acts as both input and output, which in this case would mean that the inbox is a special case of RSP engine that works as an identity function: it streams out the same input it receives.

Push notifications. A third observation is that the pull-based mechanism described in the LDN specifications is not always the most appropriate way of getting updates (i.e. notifications) from an RSP engine (e.g. continuous answers from a standing query). Although LDN explicitly mentions the possibility of using other mechanisms that implement a push approach, such as WebSocket, it does not develop further how this would be put in place, as it is regarded to be out of the scope of the generic definition of LDN. We consider that for the case of RDF streams, it would be important to explicitly describe how the push-based mechanism is specified, so that this type of interaction can be implemented in a compatible way among adopters.

Querying. Finally, in RSP engines a key access pattern is through querying, which may happen in windowed query engines, complex event processors, or even in stream reasoners. Therefore, it would be natural to include explicit interaction specifications for registering standing queries, as well as accessing their results as streaming notifications.

4 LDN for RDF streams

In this section, we present a proposal of how LDN could be used as a generic protocol upon which RSP engines could share RDF stream data among them as notifications. In this proposal we take into account the considerations made in the previous section, while keeping most of the principles behind LDN. We organize the presentation of our approach, according to different key aspects of it.

Stream identification. First, an RDF stream is uniquely identified by an IRI. This IRI is a Web resource, and it can be used to obtain information about the stream: what endpoints are available to retrieve its data, or to push data to it. An RDF stream is therefore a read/write Web resource detached from potentially multiple endpoints used to interact with its contents.

Endpoint discovery. The endpoints of an RDF stream are discoverable by performing a `GET` operation over the stream IRI, e.g.:

```
GET http://example.org/streams/my-stream
```

The response should then include metadata about the stream, including the endpoint information. For instance, if the response was requested with a JSON-LD header, it could include an inbox URI, as in LDN:

```
{ "@context": "http://www.w3.org/ns/ldp",
  "@id": "http://example.org/streams/my-stream",
  "inbox": "http://example.org/streams/my-stream/inbox" }
```

Similarly, this type of discovery could be performed using a HEAD request and a link HTTP header, as in LDN (See Figure 4).

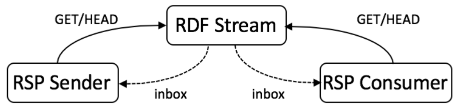

Fig. 4. The discovery interactions follow those of LDN: endpoint information is available at the target, which is and RDF stream.

Stream input and output. The inbox, as described in LDN, allows both senders and consumers to post and retrieve notifications through that web resource. In our case, we propose to constraint the inbox and specialize it in two distinct types: an input inbox and an output inbox. The rationale behind this choice is that some streams are published on the Web only with the intention of receiving notifications (i.e. to be fed) by senders. In this case the receiver is expected to process these streaming notifications, so that the stream is not meant to be consumed by other actors on the Web. Conversely, other streams are only meant to be consumed, as they are produced by an RSP engine. this is the case, for instance, of the results of a continuous query, or the output of a stream reasoner. As s result, the discovery process is similar, only that now instead of simply returning an inbox endpoint, the RDF stream may reference input and output endpoints. As an example, an input stream would be exposed as:

```
{ "@context": "http://w3id.org/rsp/ldn-s",
  "@id": "http://example.org/streams/my-stream",
  "input": "http://example.org/streams/my-stream/input" }
```

As it is specified in LDN, this type of write-only input could also be reflected in the response to an `OPTIONS` request, through an `Allow` header:

```
Allow: OPTIONS, POST
```

Notice that to differentiate from the inbox term of the LDP[3] vocabulary, we have used a new input term from a new vocabulary. This vocabulary is yet to be specified, but we use the base IRI for the rest of the examples in this paper.

[3] `https://www.w3.org/TR/ldp/`

Sending a stream notification An RSP Sender may `POST` notifications to an RDF stream input endpoint, in the same way that is specified in LDN. Essentially, the `POST` body should contain the stream element (e.g. and RDF graph) that will be fed to the stream (as in Figure 5). As an example consider the JSON-LD representation of a humidity observation posted as a timestamped graph:

```
POST /streams/my-stream/input HTTP/1.1
Host: example.org
Content-Type: application/ld+json

{"prov:generatedAtTime": "2017-07-22T05:00:00.000Z",
 "@id": "ex:Graph1",
 "@graph": [
   { "@id": "ex:humidityObservation",
    "ex:hasValue": 34.5}],
 "@context": {
   "prov": "http://www.w3.org/ns/prov#",
   "ex": "http://example.org#"}  }
```

The RSP receiver can respond with a `201 Created` or `202 Accepted` code, if successful. However, in this case, as there is no interest in sending a location header back to the sender, this part of the protocol would differ from standard LDN.

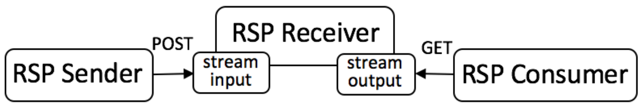

Fig. 5. RSP Sender sends a notification to a receiver stream input, and a consumer retrieves elements form a stream output.

Publicizing stream elements. As it is depicted in Fig 5, a consumer may `GET` stream elements from an RDF stream output endpoint. LDN specifies that performing a `GET` over an inbox should return the notification URIs listed as objects to the LDP `ldp:contains` predicate. Given that an RDF stream output endpoint behaves similarly to an inbox, this is also the expected behavior. However, as stream elements *fade* with time, depending on the stream fluctuations and the server configuration, the listed stream contents may progressively change. This means that if the stream updates are very frequent, when a consumer retrieves the list of notifications from the output endpoint, these may be quickly outdated when it tries to access one of them individually. In any case, it would be left to the implementations to configure properly how many and for how long the notifications should be kept. As an example, consider the following response JSON-LD containing the list of timestamped graphs:

```
{ "@context": "http://www.w3.org/ns/ldp",
  "@id": "http://example.org/streams/my-stream/output",
  "contains": [
    "http://example.org/streams/my-stream/output/graph1",
    "http://example.org/streams/my-stream/output/graph2" ] }
```

Pulling stream elements. While individual stream elements can be retrieved as notifications in LDN (i.e. with a `GET` to the resource URI obtained as described above), this methods is not too practical. First, it introduces the need of first fetching the list of available stream items (notifications), and only then fetching them individually. This strategy might be not too effective in common streaming data scenarios, so we propose a more direct approach, consisting in returning entire sequences of stream elements at once. The size or inclusion constraints of these sequences could be specified through parameters (e.g. the latest 10 minutes of data, or the latest 10 elements, etc.). As an example, consider the following time-annotated graphs about sensor observations, returned as JSON-LD for a given stream:

```
{"@context": {
    "prov": "http://www.w3.org/ns/prov#",
    "ex": "http://example.org#"},
 "@graph": [
   { "prov:generatedAtTime": "2017-07-22T05:00:00.000Z",
     "@id": "ex:Graph1",
     "@graph": [
       { "@id": "ex:humidityObservation",
         "ex:hasValue": 34.5 }]   },
   { "prov:generatedAtTime": "2017-07-22T06:00:00.000Z",
     "@id": "ex:Graph2",
     "@graph": [
       { "@id": "ex:humidityObservation",
         "ex:hasValue": 44.5 }]  } ]}
```

Pushing stream elements. While the previous data access method provides control to the consumer as when it will request the data from a stream, it is not always convenient, specially for applications that require immediate access to data that is produced on a stream. As an alternative, we propose using push based mechanisms to retrieve the data. One example is by using the Server-Sent Events[4] protocol, which is based on HTTP. Using this W3C Recommendation, it is possible to continuously push data, in this case RDF stream elements, from the server to the client, in a one-directional way (as opposed to bidirectional in WebSocket. Each data item is prefixed by the `data:` annotation. The usage of other push protocols could also be added, which in this case would mean to add an additional endpoint to the RDF Stream. In this regard, our proposal also diverges from LDN in that the latter can only advertise one inbox, while we propose having multiple endpoints for an RDF stream.

Register a query. One final aspect concerns query processing. Although this feature would be restricted to query-based RSPs, we consider important to include it, as these are one of the most prominent types of processors for RDF streams. An actor may `POST` a query to an RSP endpoint, considering that in the query, there must be a reference to a valid registered RDF stream. Also, the RSP endpoint should return the URI of the resulting output stream, so that its results can be retrieved, by either pulling or pushing. As an example, consider the CQELS query over the stream `http://example.org/streams/my-stream`. Notice that this type of queries could include references to more than one input stream.

```
SELECT ?s ?p ?o
WHERE {
  STREAM <http://example.org/streams/my-stream> [RANGE 2s] {?s ?p ?o}
}
```

[4] https://www.w3.org/TR/eventsource/

5 Implementation

We have developed a minimal implementation of the LDN protocol that complies with most of the specification for a Sender, Receiver and Consumer[5]. The implementation is in Scala, is based on the Akka Http library, and is available in Github[6]. We have also implemented the proposed behavior for the case of handling RDF streams. Although this is a preliminary implementation, it shows its feasibility at the same time that showcases its main features. We can summarize the main implementation characteristics as follows:

- A fully asynchronous HTTP processing mechanism has been chosen, as it avoids the usage of blocking operators, and is better suited for streaming HTTP responses. This mechanism is natively supported by the Akka library.
- Streams and their input/output endpoints are managed by the server implementation.
- A continuous query processor, CQELS [10], was used as an example of how an RSP engine can be encapsulated with ldn-streams.
- Push-based notifications have been implemented using Server-Sent Events (SSE).

6 Related Work

Most of the RSP engines developed to date [3, 6, 10, 9, 1] focus on the processing aspects of RDF streams (i.e. incremental reasoning, continuous querying, complex event processing, etc.), but disregard to a certain degree the Web dimension. Early attempts to design Web-based services on top of RDF streams were explored in [4, 12], although they focused mostly on interfaces for query processing engines, and only had partial implementations. A more recent development in this line is the RSP Service Interface[7], which further develops the ideas in [4], and provides a generic implementable programming API for continuous query engines.

In a step forward targeting connectivity in networks of RSP engines we can mention the SLD Revolution framework [2], which optimizes a distributed workflow of RSP engines. One of the long-disregarded issues in RSP was also the availability of RDF streams on the Web. This theme was the main concern of efforts like TripleWave [11], which facilitates the publication of streams with a variety of possible modes and configurations. Furthermore, a more generic vision of how these streams could be not only published but consumed in the Web, was formalized in the WeSP proposal[8]. This paper actually follows the WeSP vision, while going into details with one possible implementation path. The simplicity and generality of LDN are, in our view, positive aspect that make it a good candidate for a comprehensive RSP interchange protocol.

7 Discussion

One of the key motivations for using RDF to represent and process streams of data, is that it provides a Web-native model that facilitates the integration and interpretation of data.

[5] `https://linkedresearch.org/ldn/tests/summary`
[6] https://github.com/jpcik/ldn-streams
[7] `http://streamreasoning.org/resources/rsp-services`
[8] `http://w3id.org/wesp/web-data-streams`

However, while for stored data the standards for producing, publishing and consuming RDF are well-established, there is not yet a well-supported and agreed specification. Although recent efforts provide partial solutions to the problem, e.g. TripleWave for publication or SLD Revolution for orchestration, there is an impending need for a standardized Web method for communicating among RSP actors in general. The work in progress described in this paper provides initial evidence that a very generic protocol as LDN could serve as a starting point towards this goal. The decentralized nature of LDN, along with its simplicity and extensibility, are positive arguments for advocating its use.

Nevertheless, it is important to consider that even in a generic case as LDN, there are certain assumptions about the data, in this case notifications, which are fundamentally different in the case of dealing with RDF streams. We believe that the proposed extensions, could be used to formalize an LDN *profile* which could be used for RDF streams in general. In contrast with previous works, the protocol that we propose is not targeted only towards querying, but to any type of processing over RDF streams, which can even include traditional SPARQL engines, reasoners, or even machine learning processors. The current trends in Big Data processing, show that even stored data (i.e. data that was not inherently represented as a stream) is now more and more processed in a streaming fashion, typically for efficiency reasons. This trend extends to the RDF processing world in general, and the lessons learned in the RSP community could be of great benefit to a larger audience.

References

1. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.
2. M. Balduini, E. D. Valle, and R. Tommasini. SLD revolution: A cheaper, faster yet more accurate streaming linked data framework. In *RSP*, pages 1–15, 2017.
3. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Intl. J. Semantic Computing*, 4(01):3–25, 2010.
4. D. F. Barbieri and E. Della Valle. A proposal for publishing data streams as linked data - A position paper. In *LDOW*, 2010.
5. J.-P. Calbimonte. Rdf stream processing: let's react. In *OrdRing*, pages 1–10, 2014.
6. J.-P. Calbimonte, H. Jeung, O. Corcho, and K. Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semantic Web Inf. Syst.*, 8:43–63, 2012.
7. S. Capadisli, A. Guy, C. Lange, S. Auer, A. Sambra, and T. Berners-Lee. Linked data notifications: a resource-centric communication protocol. In *ESWC*, pages 537–553, 2017.
8. D. Dell'Aglio, M. Dao-Tran, J.-P. Calbimonte, D. L. Phuoc, and E. Della Valle. A Query Model to Capture Event Pattern Matching in RDF Stream Processing Query Languages. In *EKAW*, pages 145–162, 2016.
9. S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *DEBS*, pages 58–68. 2012.
10. D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, pages 370–388. 2011.
11. A. Mauri, J.-P. Calbimonte, D. Dell'Aglio, M. Balduini, M. Brambilla, E. D. Valle, and K. Aberer. TripleWave: Spreading RDF Streams on the Web. In *ISWC*, pages 140–149, 2016.
12. J. F. Sequeda and O. Corcho. Linked stream data: A position paper. In *SSN*, pages 148–157. CEUR-WS. org, 2009.
13. M. Stonebraker, U. etintemel, and S. B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.