On a Web of Data Streams

Daniele Dell'Aglio[1,a], Danh Le Phuoc[2,b], Anh Le-Tuan[3,c], Muhammad Intizar Ali[3,d], Jean-Paul Calbimonte[4,e]

[1]University of Zurich, Zurich, Switzerland,
[2]Technichal University of Berlin, Berlin, Germany,
[3]Insight Center for Data Analytics, National University of Ireland, Galway, Ireland,
[4]University of Applied Sciences and Arts Western Switzerland, HES-SO Valais-Wallis, Sierre, Switzerland
[a]dellaglio@ifi.uzh.ch, [b]danh.lephuoc@tu-berlin.de,
[c]anh.letuan@insight-centre.org, [c]ali.intizar@insight-centre.org,
[d]jean-paul.calbimonte@hevs.ch

Abstract. With the growing adoption of IoT and sensor technologies, an enormous amount of data is being produced at a very rapid pace and in different application domains. This sensor data consists mostly of live data streams containing sensor observations, generated in a distributed fashion by multiple heterogeneous infrastructures with minimal or no interoperability. RDF streams emerged as a model to represent data streams, and RDF Stream Processing (RSP) refers to a set of technologies to process such data. RSP research has produced several successful results and scientific output, but it can be evidenced that in most of the cases the Web dimension is marginal or missing. It also noticeable the lack of proper infrastructures to enable the exchange of RDF streams over heterogeneous and different types of RSP systems, whose features may vary from data generation to querying, and from reasoning to visualisation. This article defines a set of requirements related to the creation of a web of RDF stream processors. These requirements are then used to analyse the current state of the art, and to build a novel proposal, WeSP, which addresses these concerns.

Keywords: RDF stream • RDF stream processing • Interoperability • Stream processing

1 Introduction

The Web of Data (WoD) vision considers the Web as a distributed database: a vast—endless—amount of datasets to be accessed and queried. The Linking Open Data (LOD) cloud is one of the most successful examples of such a vision: more than a thousand of datasets, owned and managed by different organisations, exposing interconnected data to be freely accessed. The data

can be accessed in different ways, and the two most popular solutions are through dereferentiation and SPARQL. In the latter case, SPARQL is both a query language for the RDF data exposed in the LOD [14], and a protocol to exchange such data on the Web [11].

Since the late 2000s, we have observed a trend that substantially increased the velocity aspect of part of this data. The Internet of Things (IoT) well exemplifies this: sensors create *streams*, defined as continuous sequences of data generated at very high frequencies, where its value is usually strictly associated to recency, i.e. the quicker the data is processed, the more valuable it is. It is therefore natural to ask if the existent WoD-related technologies are good enough to cope with such data. Looking at the research in the database area, it is possible to observe that the typical database approaches show several limitations while coping with data streams. Stonebraker et al. [22] explain which are the requirements for processing streams, and highlight the limitations of DBMS approaches, e.g. their passive processing model works poorly with data characterised by extreme velocity; SQL does not support operators to cope with streams; and it is challenging to predict the behaviour of such systems while coping with streaming data. It is easy to observe that these limitations also affect SPARQL and existent WoD solutions.

As a possible solution, Stream Processing Engines (SPEs) emerged in the area of data management research to cope with streaming data. They introduce new paradigms and processing models that fit better the requirements and scenarios involving data streams. As a result, new languages have been designed and developed, such as CQL [2], EPL and StreamSQL, with engines and systems designed with the specific task of processing streams.

In the Web of Data, these novel paradigms inspired RDF Stream Processing (RSP): it builds on top of SPEs to mitigate the data heterogeneity issues by exploiting Semantic Web technologies. As the name suggests, such systems are designed to process data streams modelled through RDF, and they offer a wide set of operators, from typical SPE operations (e.g. filters, aggregations, event pattern matching) to reasoning features.

Table 1. Comparison between data management and web of data paradigms

	DB	SPE	WoD	WoDS
Processing	SQL	StreamSQL (et al.)	SPARQL	RSP-QL (et al.)
Data exchange	-	-	HTTP, SPARQL	?

RSP fills an important gap between the current Web of Data and a Web of Data Streams (WoDS): it introduces models and languages to process streams on the Web, most of them captured by reference models as RSEP-QL [10] and LARS [7]. However, an important element is still missing, and it is a technical infrastructure to manage decentralized data exchange among RSP engines. As depicted in Table 1, data can be exchanged through HTTP (following the

linked data principles) or via the SPARQL protocol. These solutions perfectly suit the cases where engines follow passive processing models, i.e. they pull the data when needed, but not when engines adopt active processing models.

This study addresses the following research question: *what is a suitable data exchange infrastructure to perform stream processing on the Web?* In other words, this study intends to provide the bulding blocks of an infrastructure that enables the scenario depicted in Figure 1: a network of RSP engines, distributed on the Web, able to exchange and process RDF streams among them.

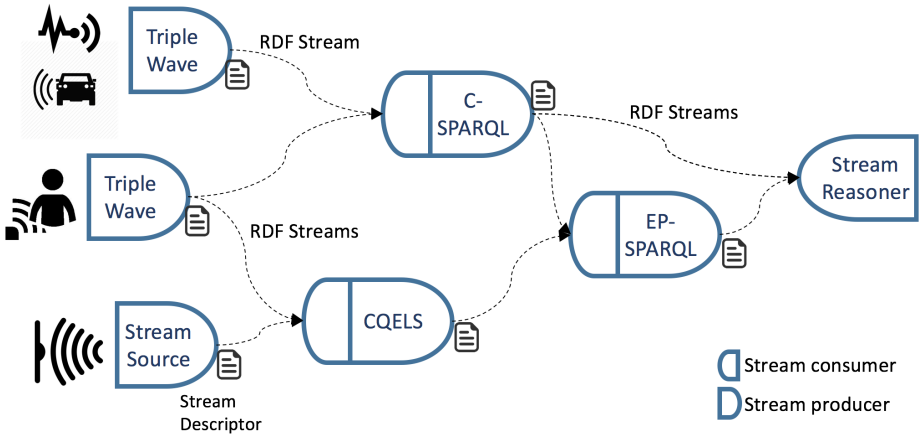

Fig. 1. The vision of WeSP: a network of RSP engines distributed on the Web, able to exchange RDF streaming data among them.

The solution should take into account the nature of the streams and the engines, as well as the technical characteristics of data exchange on the Web. It should indeed consider that existing RSP engines are heterogeneous in terms of operators and APIs to control them, as well as the nature of the Web, that is distributed and relies on existing technologies and standards. We envision a scenario as the one depicted in Figure 1: interconnected RSPs that perform several tasks such as querying to reasoning, exchanging the results and producing sophisticated analyses.

In the following, we present WeSP, a framework to build complex networks of RSP engines on the Web. WeSP defines a communication protocol to initialize the connection between two RSP engines and to manage the data exchange. This is implemented through two interfaces for the elements that produce and consume streams, and a vocabulary to annotate the streams. The design and implementation of WeSP takes into account existing previous results such as the W3C RSP Community Group (RSP-CG) reports, existing RSP engines, protocol and standards. We empirically show the feasibility of

the WeSP approach for the construction of a Web of Data Streams, by proposing an implementation of its components with exiting RSP publishers and engines.

The remainder of the article is structured as follows. We identify a set of requirements for communication of Web stream processors in Section 2. Section 3 presents WeSP, describing its components and presenting an existing implementation in Section 4. In Section 5, we review the state of the art and we discuss the solutions available so far, before concluding with remarks and future directions in Section 6.

2 Requirements

To design an infrastructure to exchange RDF streams on the Web, we first define a set of requirements. Three of them are extensions of Stonebraker et al.'s rules for building stream processing engines [22], while others are elicited from real-world use cases.

R1: Keep the data moving. The first rule of [22] states that a SPE "should use an active (i.e., non-polling) processing model". Looking at the WoD, we can notice that most of the interactions follow a polling paradigm and are stateless. This is a direct consequence of the client-server paradigm at the basis of HTTP and the Web: servers supplying resources to client on demand. In this sense, the traditional WoD is not the most suitable environment to perform stream processing. Therefore, as first requirement, *WeSP must prioritize active paradigms for data stream exchange, where the data supplier can push the stream content to the actors interested in it.*

R2: Stored and streamed data. The fifth rule of [22] is about the "capability to efficiently store, access and modify state information, and combine it with live streaming data". We need to revisit this rule from a WoD point of view, in particular about the notion of stored data. In this context, this data is represented by any dataset that is exposed to the Web through SPARQL endpoints or through Linked Data technologies. Our second requirement states that *WeSP must enable the combination of streaming and stored data.* In this case combination has a broad meaning, and may refer to several operations, such as stream enrichment, stream storage or fact derivation.

R3: High availability, distribution and scalability. The sixth rule of [22] proposes to "ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures", while the seventh states that SPE "must be able to distribute its processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent". Even if such rules are mainly related to the stream processing engine architectures, we can infer useful indications for the design of WeSP. Our third requirement is that *WeSP*

must enable the possibility to build reliable, distributed and scalable streaming applications.

R4: Operations on the stream content. Every use case has special requirements on the way the streaming data should processed. Querying is an option when the goal is to aggregate, filter and look for relevant patterns of events. However, other alternatives are possible, such as deductive, inductive and other types of reasoning; stream generation and visualization. *WeSP must guarantee a wide range of operations over the streams.* Such a requirement is important, in particular on the Web perspective, where data may be accessed and used in unexpected ways by third-parties.

R5: Accessible information about the stream. While the nature of the stream content is highly dynamic and volatile, the stream itself is a resource that can be described. Such descriptions are needed in a number of cases. For example, statistics about the frequency and the size of the stream content may be used to enable query optimization in the RSP engines; information about the generation of the stream may be used to enable provenance-related reasoning; description of the features of the streams may be collected in registries to achieve search and discovery. Our fifth requirement states that *WeSP must support the publication of stream descriptions.*

R6: Stream variety support. The decentralised and bottom-up nature of the Web makes it hard—if not impossible—to define *the* model and serialization format for streams. Looking at the use cases in the RSP-CG wiki, it is easy to observe that: streams can have different velocity (e.g. one new item every minute vs every millisecond); time annotations can be composed by zero, one, two or more time instants, item schemata can be simple (e.g. numbers) or complex (e.g., graphs or tables). To preserve the heterogeneity that characterizes the Web, *WeSP should support the exchange of a wide variety of streams.*

R7: Reuse of technologies and standards. A common problem when building new frameworks and infrastructures is to find a good trade-off among what is new and what exists. On the one hand, creating everything from scratch is an opportunity to create a solution that perfectly fits the requirements, but on the other hand adopting existing specifications (i.e. standards and protocols) allows reusing existing tools and methods. This is particular true on the Web, where guaranteeing compatibility is mandatory in order to enable interoperability. Our last requirements states that the design of *WeSP should exploit as much as possible existing protocols and standards.*

3 A framework to exchange RDF streams on the Web

In this section we present our proposal to build an infrastructure to exchange RDF streams on the Web, WeSP. The design follows the requirements presented above, and it is composed by a data model for RDF streams, an API and a protocol to exchange the streams, and a model to describe the stream objects.

3.1 RDF stream: model and serialization

Following the requirements and the type of scenarios that we address in this study, it is needed to represent heterogeneous data streams, which can be shared by different processing engines, and whose data can be not only retrievable but interpretable and distributable among these engines. The RDF model lends itself as a natural candidate for representing different types of data, thanks to its flexibility in data modeling, and the usage of well established standards, designed for the Web. However, in order to be used in a streaming scenario, the RDF model may need extensions, as shown in previous studies such as [16, 5].

We can distinguish two kinds of data in the requirements, first the streaming data, which are by nature highly dynamic and labeled with time annotations. Second, the contextual or background data, which changes less often, provides information that enriches the streaming data (e.g. locations, profiles, producer data, system descriptions). For the latter, RDF graphs can be used as an underlying data model, while streams can be captured using an extended RDF stream data model.

To fulfil these goals, we adopt the notion of time-annotated RDF graphs as elements of RDF streams, following the abstract data model proposed by the W3C RSP Community Group.

The proposal introduces a notion of RDF stream as a sequence of time-annotated named graphs (*timestamped graph*). Each graph can have different time annotations, identified through *timestamp predicates*.

As an example, we can consider the case where each graph has a time annotation. We define a *timeline* T as an infinite, ordered sequence of time entities $(t_1, t_2, \ldots)$, where $t_i \in$ **TimeEntities** and for all $i>0$, it holds that t_i happened before t_{i+1}. When the time entities are restricted to instants, $t_{i+1} - t_i$ is a constant, i.e. the *time unit* of T. An RDF stream is a sequence of pairs (G, t), where G is an RDF graph and $t \in T$ is a time entity:

$$S = (G_1, t_1), (G_2, t_2), (G_3, t_3), (G_4, t_4), \ldots \quad (1)$$

where, for every $i > 0$, (G_i, t_i) is a timestamped RDF graph and $t_i \leq t_{i+1}$.

The following stream S is compliant with the above model:

$$S{=}(G_1, 2), (G_2, 4), (G_3, 6), (G_4, 8), (G_5, 10), \ldots,$$

where each G_i contains the depicted RDF triples in .

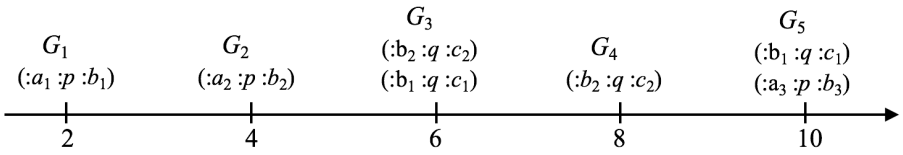

Fig. 2. An example of time annotated graphs on an RDF Stream.

This model can be serialized using current standards for RDF representation, as depicted in Listing 1. The listing shows a JSON-LD representation of a timestamped graph,annotated using the PROV *generatedAtTime* property. In this case the graph contents include the graph G_2 of Figure 2.

```
{"prov:generatedAtTime": "2015-06-30T04:00:00.000Z",
 "@id": "wesp:G2",
 "@graph": [
  { "@id": "wesp:a2",
    "wesp:p": {"@id":"wesp:b2"}
  }],
 "@context": {
    "prov": "http://www.w3.org/ns/prov#",
    "wesp": "http://streamreasoning.org/wesp/"
}
```

Listing 1. Example of a timestamped graph represented using JSON-LD.

In the remaining of this contribution, we consider this model where the time annotation is represented by one time instant. The choice is without loss of generality, since WeSP can be used to exchange other types of streams.

3.2 Communication protocol

WeSP defines two interfaces, namely *producer* and *consumer*. As the names suggest, the former is implemented by actors that distribute streams across the Web, while the latter is implemented by actors who want to receive streams.

Every actor involved in the processing may implement either interface, as well as both of them. This leads us to the following nomenclature. A *stream source* implements the producer interface only. Services exposing sensors data on the Web and TripleWave [18] are examples of stream sources. Similarly, a *stream sink* implements the consumer interface only, e.g. a dashboard visualising information for the final user. Finally, a *stream transformer* implements both the interfaces: it gets as input a stream and outputs another stream, e.g. a stream processor or a stream reasoner.

The principal responsibility of the producers is to make streaming data accessible. It is done by exposing *Stream Descriptors*, i.e. metadata about the streams. Examples of Stream Descriptor contents are the creator, the adopted vocabulary and statistical information about the stream. However, the most relevant data the Stream Descriptor brings is about how to effectively access the stream content. We detail the stream descriptor in Section 3.3. In the following, we describe how the producer/consumer communication develops.

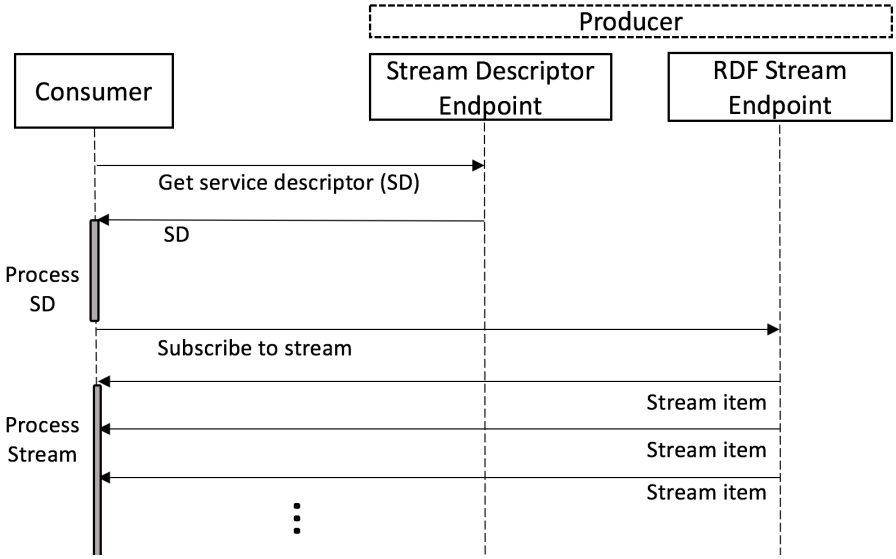

Fig. 4. Protocol to establish the communication between the producer and the consumer interfaces

The communication relies on two phases, depicted in Figure 3, and it starts on the consumer side (as in typical push-based communication). The consumer only needs to be aware of the IRI of the Stream Descriptor, e.g. the user provides it as an instruction to the constructor. This is what is needed for the first phase, where the consumer accesses the Stream Descriptor IRI (as in Figure 3) through an HTTP request, following the Linked Data principles.

In the second phase, the consumer starts to receive the stream (as in Figure 3). The Service Descriptor brings information about how to access the effective stream content. That means, it describes one or more ways to establish the connection between the consumer and the endpoint that is providing the stream content. This connection can happen with different protocols, based on push mechanisms. e.g. WebSocket and MQTT, as well as based on pull mechanisms, e.g. HTTP. Among the several options offered by the producer through the Service Descriptor, consumers choose the way they prefer. After the decision, the consumer can start to receive the stream content and to process it.

The choice to allow several channels to distribute the streams gives the freedom to adopt several channels to exchange the streams, relying on different technologies based on requirements of the specific scenario.

3.3 Stream Descriptor

The Stream Descriptor is an RDF document providing metadata of the RDF stream that is used by a *consumer* for bootstrapping its stream consuming process. Realizing that an RDF stream is similar to an RDF dataset, we extend the DCAT vocabulary to represent an RDF Stream as illustrated in Figure 4 (and available here).

The class RDFStream extends dcat:DataSet to inherit all properties of a RDF dataset, and it also offers properties specific to a stream, such as *streamrate* and *streamTemplate*. The *streamTemplate* property can be used to point to a document that specifies the template or the constraint of the RDF stream, for instance, the template can be an RDF constraint document described using SHACL. The template can be used as a hint of the data shape of the RDF stream for the consumer to optimize its processing. It also plays the role as a validation rule for consumer to verify if the incoming data is conformed with the template.

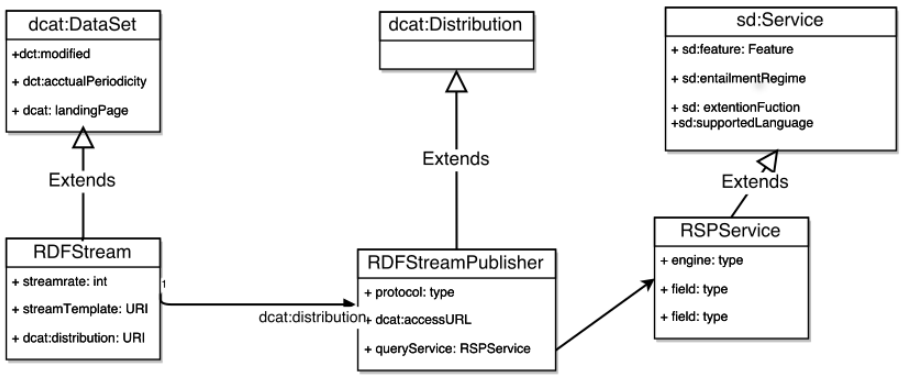

Fig. 5. Description of RDF Stream

Note that an RDF stream can be distributed via different stream channels using different protocols, like HTTP and Websocket. Therefore, a stream channel is an instance of the RDFStreamPublisher class which is linked to the RDFStream class via the property *dcat:distribution*. The RDFStreamPublisher is used to specify the protocol and the URL whereby the consumer can tap into to receive the stream. Also, RDFStreamPublisher is a subclass of the dcat:Distribution class which has several properties for the consumer to configure its parsers such as *dcat:mediaType* and *dcat:format*. An

RDFStreamPublisher might provide a continuous query service which is specified by the RSPService class extended from the sd:Service defined by SPARQL 1.1 Service description.

```
{"@context": {
  "wesp": "http://streamreasoning.org/wesp/",
  "generatedAt": {
   "@id": "http://www.w3.org/ns/prov#generatedAtTime",
   "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
  }
 },
 "@type": "wsd:RDFStream",
 "dcat:accessURL": "ws://localhost/streams/S",
 "wsd:streamTemplate": {"@id":"http://purl.oclc.org/NET/ssnx/ssn"},
 "wsd:shacl": {
  "@list": [
   {
     "generatedAt": "2015-06-30T02:00:00.000Z",
     "@id": "wesp:G1"
   },{
     "generatedAt": "2015-06-30T04:00:00.000Z",
     "@id": "wesp:G2"
   },{
     "generatedAt": "2015-06-30T06:00:00.000Z",
     "@id": "wesp:G3"
   }
  ]
 },
 "sld:lastUpdated": "2015-06-30T06:00:00.000Z"
}
```

Listing 2. The Stream Descriptor associated to the stream in Figure 2.

4 Implementation and examples

In this section, we describe three implementations of WeSP. The first is a stream source, the others are stream transformers. They are built on top of existing frameworks, TripleWave and C-SPARQL [5] and CQELS [16]. All the related code is open source under Apache 2.0 Licence and is available online.

4.1 Stream source example: TripleWave

TripleWave [18] is a framework to create and expose RDF streams according to the producer interface described above. Triplewave can be fed with different input, such as non-RDF Web streams or RDF data containing temporal annotations.

In the former case, TripleWave lifts existing Web streams as RDF streams, such as the Wikipedia update stream and Twitter stream. The operation is enabled through R2RML transformations [9], that map relational to RDF data. The lifting operation enables the interoperability among the WeSP actors, addressing R6.

In the latter case, TripleWave streams out data stored in a repository. This mode is useful for benchmarking as well as experimental runs, where data has to be streamed multiple times across the network. Precondition to this mode is the existence of time information associated to the repository content: TripleWave uses them to compute the correct intervals between consecutive events and set delays accordingly.

TripleWave has been extended to natively support WeSP, as it implements the producer interface. It can spread the streams through WebSocket, MQTT and HTTP chunked encoding. Support for Server-Sent Events (SSE) will be released soon. This offers to consumers a wide choice on the protocol to be used to receive the data, according to the use case requirements and on the system needs. The endpoints related to the protocols, as well as the required information to access them, are described in RDF and embed in the Stream Descriptor, exposed on an HTTP address. The code and the instructions are available at the project page.

4.2 Stream transformer example: C-SPARQL and CQELS

It is possible to create WeSP actors out of existing stream processors by introducing adapters. We depict this possibility by presenting the cases of C-SPARQL [5] and CQELS [16], two engines able to processing streams through continuous query languages based on SPARQL [14].

These engines offer programmatic APIs to manage the communication, i.e. (i) declare RDF streams, (ii) push the stream items into the declared RDF Streams, (iii) register the query, (iv) register listeners to obtain and manage query results. The WeSP adapter should implement the producer and consumer interfaces according to such APIs.

The implementation of the consumer interface in the adapter should enable the connection to a stream source or transformer, according to the protocol described above. Given the nature of such engines, the control of the engine should happen in a query-driven way. That means, when a new query is registered, the engine should establish the connections for the streams needed to evaluate the query. The main problem is related to the discovery of the streams: how to let the engine know where the streams are? The solution we propose is based on the following consideration: both the languages adopted by the engines (respectively C-SPARQL and CQELS-QL) offer the possibility to declare the stream through a IRI. If such IRI denotes a Stream Descriptor address, engines would have the information they need to establish the connection.

Similarly to the consumption, the stream production is query driven. RSP engines stream out the query results and make it available to other agents (e.g. users, other engines, etc.). The WeSP adapter implements the producer interface as follows. For each query that is registered in the engine, the adapter performs two actions. First, it creates a Stream Descriptor and exposes it at an HTTP address. Second, it registers a listener to manage the answers. The current implementation provides the listeners for MQTT and WebSocket. The first forwards the results to a MQTT broker, while the latter sends the result to an internal component that manages the ongoing connections. In this way, several actors may access the results of the same query.

The code for C-SPARQL is available on GitHub. The library embeds the C-SPARQL engine in a Java class that implements the consumer and producer interfaces. The result is transparent, in the sense that the new main class exposes the same methods of the C-SPARQL original one. In this way, no additional effort is put on the user side, which can benefit of WeSP without learning to use new APIs.

The code for CQELS is available in the official CQELS repository. In this case, we opted for the integration of the WeSP interface directly in the main project. The current version supports the data stream exchange via WebSocket, but we plan to integrate other protocols as future work.

5 Related Work

RSP (RDF Stream Processing) engines emerged in recent years, with the goal of extending RDF and SPARQL to process RDF streams. They can be broadly divided into two groups. The first is of those influenced by Complex Event Processing (CEP), e.g. EP-SPARQL [1], Sparkwave [15] and Instans [19]. the second group is that of engines inspired by DSMS, which exploit sliding window mechanisms to capture a finite portion of the input data. Examples include C-SPARQL [5], CQELS [16], and $\text{SPARQL}_{\text{stream}}$ [8]. While these engines provide querying features, they generally lack the means to publish or reuse RDF data streams on the Web. Instead, they assume ad-hoc connection to RDF streams without any protocol of communication.

An inital effort targeting communication and interchange among RSP engines, named RSP Services, was proposed in [3]. The goal of this project is to enable remote control of RSP engines, which is achieved by introducing REST APIs that exposes the programmatic APIs of the RSP engines, and by Web Socket channels to connect engines. So far, an implementation for C-SPARQL exists. This project has limitations given the heterogeneity and difficulty of handling the engines in a uniform manner: not all the engines work based on queries, so it may not be enough to cover a wider range of RDF stream processing engines, such as stream reasoners. WeSP targets the

problem of exhanging RDF streams between engines, not only query engines; it supports a wider range of communication protocols and a richer description of the stream (through the stream descriptor).

Regarding the publication of streams on the Web using RDF technologies, we can mention [13, 24], which proposes the generation of RDF datasets out of unstructured streams. Other works focused on the usage of Linked Data principles for publishing streams [4,17], although they did not provide any further communication or interchange protocol beyond the standard principles used in static and stored data.

The stream publishing implementation of WeSP relies on TripleWave [18], a generic and RDF stream-oriented publication system tailored for the Web. It goes beyond previous works, providing a generic framework for RDF stream provision, including ingestion of non-RDF sources and time-annotated RDF datasets. TripleWave introduced the notion of stream descriptor. As explained in Section 4, in the context of this study we extended TripleWave with a new stream descriptor vocabulary and extension to new communication protocols such as MQTT.

The work of the W3C RDF Stream Processing Community Group (RSP-CG) provides a starting point towards a common model for representing, querying and exchanging RDF stream data. However, given the scope of the group, the published reports on Requirements and Design Principles, as well as the Abstract model, do not propose a concrete set of interfaces and serialization formats for enabling interchange and interoperability. In WeSP we take into consideration the abstract model and guidelines of the RSP-CG, taking it to the implementation level. Another related specification, the Linked Data Notifications (LDN) recommendation of W3C, targets a very generic scenario of Linked Data senders, receivers and consumers. While this specification is too generic for the streaming data requirements presented in Section 2, it might be worth to explore commonalities between LDN and this work.

Finally, outside of the RDF and Semantic Web technologies, different technologies and protocols have been developed for supporting data stream communication and exchange, often linked to the Internet of Things. Protocols include MQTT, WebSockets, or streaming HTTP-based solutions such as SSE (Server-Sent Events). In this work, these different options can be encapsulated under a higher-level protocol, allowing the co-existence of different underlying technologies.

6 Conclusions

In this contribution, we presented WeSP, a framework to exchange RDF streams. WeSP moves a step towards the realization of an eco-system of Web stream engines, where engine has a broad meaning and can refer to stream sources, visualisers, continuous query systems and stream reasoners. In future

work we we aim at continuing the development of WeSP. In particular, we are interested in studying the relation with other studies, such as [23], LDN and Activity Streams, adopting and integrating them when possible.

We designed WeSP based on a set of requirements elicited from literature and real use cases, and as a result, it builds on existing technologies and recommendations, such as RDF, WebSocket and MQTT. We have shown the feasibility of the approach by presenting concrete implementations, available as open source projects. We believe that a wide availability of Web stream engines can enable research in future interesting work directions, such as federated query processing or, more in general, federated stream processing over the Web.

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. [online]
2. A. Arasu, S. Babu, J. Widom. The CQL continuous query language: semantic foundations and query execution. In *VLDB J.* 15(2), pages 121-142, 2006. [online]
3. M. Balduini and E. Della Valle. A restful interface for RDF stream processors. In *ISWC 2013 Posters & Demonstrations Track*, pages 209-212, 2013. [online]
4. M. Balduini, E. Della Valle, D. Dell'Aglio, M. Tsytsarau, T. Palpanas, and C. Confalonieri. Social Listening of City Scale Events Using the Streaming Linked Data Framework. In *ISWC*, pages 1-16. 2013. [online]
5. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a Continuous Query Language for RDF Data Streams. *Int. J. Semantic Computing*, 4(1):3-25, 2010.
6. D. F. Barbieri and E. Della Valle. A proposal for publishing data streams as linked data - A position paper. In *LDOW*, 2010. [online]
7. Beck, H., Dao-Tran, M., Eiter, T., and Fink, M. LARS: A Logic-Based Framework for Analyzing Reasoning over Streams. In *AAAI*, pages 1431-1438, 2015. [online]
8. 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. [online]
9. S. Das, S. Sundara, and R. Cyganiak. R2rml: RDB to RDF Mapping Language. W3C Recommendation, W3C, 2012. https://www.w3.org/TR/r2rml/
10. 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. [online]
11. L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres. SPARQL 1.1 Protocol. W3C Recommendation, W3C, 2013. https://www.w3.org/TR/sparql11-protocol/
12. J. A. Fisteus, N. F. Garcia, L. S. Fernandez, and D. Fuentes-Lorenzo. Ztreamy: A middleware for publishing semantic streams on the web. *J. Web Semantics*, 25:16-23, 2014. [online]
13. D. Gerber, S. Hellmann, L. Bühmann, T. Soru, R. Usbeck, and A.-C. N. Ngomo. Real-time rdf extraction from unstructured data streams. In *ISWC*, pages 135-150. 2013. [online]
14. S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3c Recommendation, W3C, 2013. https://www.w3.org/TR/sparql11-query/

15. S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *DEBS*, pages 58-68. 2012. [online]
16. 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. [online]
17. D. Le-Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, and M. Hauswirth. A middleware framework for scalable management of linked streams. *J. Web Semantics*, 16:42-51, 2012. [online]
18. 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. [online]
19. M. Rinne, S. Törmä, and E. Nuutila. SPARQL-based applications for RDF encoded sensor data. In *SSN*, volume 904, pages 81-96. 2012. [online]
20. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *ISWC*, pages 585-600. 2011. [online]
21. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: A federation layer for distributed query processing on linked open data. In *ESWC*, pages 481-486. 2011. [online]
22. M. Stonebraker, U. Cetintemel, and S. B. Zdonik. The 8 requirements of real-time stream processing. SIGMOD Record, 34(4):42-47, 2005. [online]
23. R. Taelman, P. Heyvaert, R. Verborgh, E. Mannens. Querying Dynamic Datasources with Continuously Mapped Sensor Data. In *ISWC (Posters & Demos)*, 2016. [online]
24. T.-D. Trinh, P. Wetz, B.-L. Do, A. Anjomshoaa, E. Kiesling, and A. M. Tjoa. A web-based platform for dynamic integration of heterogeneous data. In *IIWAS*, pages 253-261, 2014. [online]