Semantic Web 0 (0) 1
IOS Press

Decentralized Messaging for RDF Stream Processing on the Web

Jean-Paul Calbimonte [a,*],
[a] *Institute of Information Systems, University of Applied Sciences and Arts Western Switzerland, HES-SO Valais-Wallis, Sierre, Switzerland*
E-mail: jean-paul.calbimonte@hevs.ch

Abstract. The presence of data streams on the Web is constantly increasing in terms of volume and relevance, for a large number of applications domains and use-cases. As a consequence, there is a growing need for coherent data and processing models for streams, including data and metadata semantics that can help integrating, interpreting, and reusing them. RDF Stream Processing (RSP) introduced theoretical foundations and concrete technologies to deal with these issues, ranging from continuous query processors to stream reasoners. However, most of these efforts lack support for communication and interaction at the Web level. This paper proposes a decentralized model and implementation, RSP actors, for enabling Web interactions among RSP engines, based on the actor paradigm. The RSP actors proposed in this work, use a message-passing mechanism for asynchronous communication of RDF streams and metadata, and is designed to encapsulate the functionalities of existing RSP query engines, Complex Event Processors, or stream reasoners. Furthermore, we have used and extended the Linked Data Notifications recommendation of W3C, as a building block for a specific HTTP-based implementation of the model. The RSP actors code-base is open-sourced, and provides three concrete implementations of well known RSP/stream reasoners developed by the RSP community, to show the feasibility of the approach.

Keywords: RDF Streams, Stream Processing, Linked Data Notifications, Actors, Semantic Streams

1. Introduction

Data on the Web is increasingly dynamic, and the velocity at which it is produced, processed and consumed, is raising new challenges to the research community. Examples are prevalent in different domains. For instance, for wearable and fitness data management, streams of physiological data, along with user annotations, flow from IoT devices to Web services and cloud processing environments in a continuous manner [1]. Similarly, for use cases in energy production and consumption [2], large and heterogeneous streams of data are generated on a daily basis, at the same time that integration and query processing mechanisms are needed to produce valuable insights. Also on the financial domain, events processing is used to help in decision making and to analyze trends in the stock market [3]. Other countless examples could be drawn from domains such as environmental monitoring, criminality surveillance, robotics, or personalized health, to name some. They all share the need for technical solutions to cope with the dynamicity and velocity of data streams, and the complexity of managing heterogeneity. In addition, they require these aspects to be considered in the context of the Web, which is inherently decentralized, distributed, and linked.

This general challenge has gathered the attention of the Web research community, which in the last decade answered through different initiatives that can be placed under the *Stream Reasoning* umbrella [4]. Processing models, query languages, stream systems, stream reasoning engines, among others, have been produced as a result, with a certain number of common characteristics: (i) the usage of foundational standard semantic models such as RDF [5] and OWL [6], with extensions for the case of data streams; (ii) the definition of extensions for standard query processing languages (e.g. SPARQL [7]); and (iii) the design of al-

*Corresponding author. E-mail: jean-paul.calbimonte@hevs.ch.

gorithms and techniques for optimizing and scaling stream processing. Examples of the first aspect include RSP-QL [8], LARS [9], and RSEP-QL [10], which provide foundational models that capture and represent data streams and/or complex events, within RDF. For the second aspect, different query languages such as CQELS [11], EP-SPARQL [12], C-SPARQL [13], SPARQLStream [14], TEF-SPARQL [15], and others, have been designed and implemented, focusing on multiple (and to some extent overlapping) cases of RDF stream query processing. Finally, the third aspect includes query optimization techniques[11, 16], stream materialization[17], Ontology-based data access approaches[18], ontology evolution [19], etc.

Even if this family of scientific outcomes constitutes a notable progress, some of the original challenges were not sufficiently addressed. A key issue that falls in this category is the availability of these RDF streams (and their processing engines and reasoners) on the Web, and through Web standards. In the same way that the Linked Data [21] principles provided general guidelines to publish and consume RDF stored data, emerging approaches have started to propose analogous solutions for the provision of RDF streams on the Web [22–24]. However, these ideas still need to be materialized into concrete specifications, and consequently implementations that validate the overall concept.

In this paper, we propose a decentralized model for exposing RDF streams and RDF Stream Processing (RSP) engines on the Web, based on the *actor* paradigm [25]. This abstraction provides a well-defined framework for message exchange among RSP engines, which can be encapsulated as actors that receive/send RDF stream messages, and perform different types of processing over them. Furthermore, we show how this model can be further specialized for the case where the actor interactions are governed by a concrete protocol, more precisely the Linked Data Notifications (LDN) [26] recommendation of W3C. Through this protocol, actors are characterized as either senders, receivers and consumers of notifications –in this case RDF stream items– using standard RDF vocabularies and a well-defined set of HTTP-based interactions.

The RSP actors model proposed in this work, provides a simple but effective way of defining interactions among multiple RDF stream processors, reasoners or producers, allowing the definition of communication channels and workflows on the Web. Therefore, this model fills an important gap for RSP engines, given that in most cases, these engines stop at the processing level, without indicating how the streams would be fed, and how the processing results would be consumed on the Web. This scenario follows the vision presented years ago in [20], as depicted in Figure 1, and we believe that the RSP actors model and protocol presented in this paper will contribute to its achievement. In this context, the usage of LDN as a standard protocol for sending, receiving and consuming RDF stream elements, opens the possibility of enabling wide-interoperability of RSP engines, regardless of their implementation. As described in this paper, and following a first assessment on the matter [27], the usage of LDN with certain extensions has a certain potential for adoption as a Web-based interchange mechanism for RDF streams.

To validate the feasibility of this approach, we provide an openly available implementation of the actor-based model, which includes a generic interface that can be specialized for concrete RSP engines and/or stream reasoners. As examples, we provide implementations of interfaces for well-known engines (CQELS and C-SPARQL), as well as an OWL-API-based stream reasoner (TrOWL [28]). We also include in the open-sourced package the LDN implementation , which extends the actor interfaces through an asynchronous HTTP façade.

The remainder of the paper is structured as follows: we introduce basic definitions and concepts in Section 2. Then, we describe in details the RSP actors model in Section 3. We provide details on the LDN-based communication in Section 4. Implementation details are presented in Section 5, and evaluation results in Section 6. Section 7 describes related works before a discussion in Section 8 and conclusions in Section 9.

2. Preliminaries

The foundations of the work presented in this paper are: RDF stream data and processing models, the actor paradigm, and Linked Data Notifications. We briefly introduce them in the following sections.

2.1. RDF Stream Processing

RDF Stream Processing (RSP) emerged as a response to the increasing need to model, integrate, query, reason upon, and process streams of data using semantic-enabling models such as RDF. RDF streams can be characterized as potentially infinite time-ordered sequences [29] of RDF data elements. Each of these elements may be annotated with a timestamp, which could be a point-in-time annotation, or an interval [30]. Al-

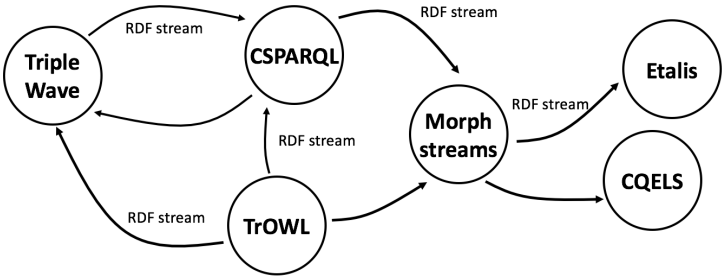

Fig. 1. Network of RSP actors communicating and sharing RDF streams with one another. The vision of a decentralized Web of streams requires common agreements to enable interoperability. [20].

though different variations of this RDF model have been used in the literature, the W3C RSP Community Group[1] has defined an abstract model that we will refer to in this paper[2]. This model introduces the notion of an RDF stream as a sequence of timestamped graphs. Each graph can have different time annotations, identified through timestamp predicates. Considering the case where each graph has a single time annotation, an RDF stream S is defined as a sequence of pairs (G,t) where G is an RDF graph, and t is the timestamp: $S = (G_1,t_1),(G_2,t_2),(G_3,t_3),(G_4,t_4),\ldots$. An RDF stream can be identified by an IRI, allowing it to be referenced for querying, reuse, reasoning, integration, etc.

One of the first types of processing studied for RDF streams is querying. Different languages have been proposed with this purpose, usually extending SPARQL [7] with features such as: continuous querying, windowing, CEP operators, etc. Examples of such RSP query languages and engines include C-SPARQL [31], SPARQLStream [14], EP-SPARQL [12], Strider [32], or CQELS [11], which share certain features, but differ on a number of syntax and semantics aspects. As a concrete example of these languages, the CQELS query in Listing 1 requests the average temperature (`avgtemp`) grouped by sensor in the last 2 hours, as recorded by the stream identified as `http://hevs.ch/streams/s1`.

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ex: <http://example.org/vocab#>
PREFIX qudt: <http://qudt.org/1.1/schema/qudt#>
PREFIX cf-property: <http://purl.oclc.org/NET/ssnx/cf/cf-
     property#>
SELECT (AVG(?temp) AS ?avgtemp) ?sensor
WHERE {
  STREAM <http://hevs.ch/streams/s1> [RANGE 2 HOUR] {
  ?obs sosa:hasResult ?result;
       sosa:observedProperty cf-property:air_temperature;
       sosa:madeBySensor ?sensor.
  ?result qudt:numericValue ?temp. }
} GROUP BY ?sensor
```

Listing 1: CQELS: Query the average temperature in the last 2 hours.

Another prominent form of stream processing is stream reasoning. Different approaches have addressed the challenge of continuously providing entailments over ontologies that dynamically change over time, with axioms that are added and removed constantly. As it is seen in Figure 2, an ontology may *evolve* over time, represented as consecutive $O_1, O_2, \ldots$. At each time, new ontology axioms can be added or removed form the ABox or TBox, and the corresponding entailments must be produced. Incremental materialization, truth maintenance systems, OBDA and other techniques [18, 28, 31] have been developed in order to address these cases in stream reasoners. Nevertheless, both RSP engines,

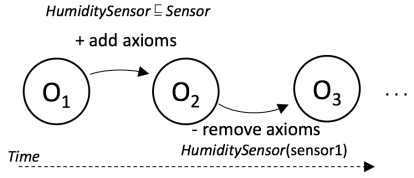

Fig. 2. Ontology stream: axioms are added and removed over time.

including those with CEP support, and stream reasoners, have mostly been designed without much considerations of how they integrate and interact with other entities on the Web. As explained in detail in Section 7, different efforts have emerged in order to fill this gap, although mostly targeting RDF stream publication, and architectures that need to be further specified and implemented.

[1] https://www.w3.org/community/rsp/
[2] https://w3id.org/rsp/abstract-model

2.2. The Actor Model

Actors were first thought as a model for concurrency computing, starting form the seminal work of Hewitt et al. [25]. Later on, this theory evolved and became the foundation of several implementations in different programming languages. In essence, actors are lightweight objects that encapsulate a *state* and a *behavior*. These objects share no mutable state among them, and in fact the only way to communicate is through message passing (See Figure 3). To manage the incoming messages, each actor has a mailbox, which serves as a queue to buffer and sequentially process them, as specified by its inherent behavior. This actor behavior may prescribe to modify its internal state, send messages to other actors, or create new actors. The asynchronous message-

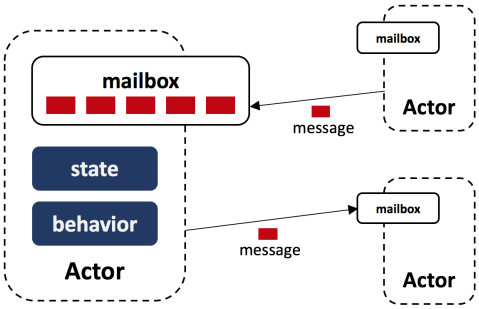

Fig. 3. Actors and their message-passing interactions.

passing communication that governs actor interactions is a key feature that allows providing a loose-coupled architecture where blocking operators are avoided, as each actor is solely responsible of maintaining its local state. These characteristics are particularly interesting for stream processing systems, especially for those where high scalability and parallel processing of streams are needed. Immutable state, no-sharing and asynchronous processing are common requirements for this type of systems, and examples of implementations that use some of these principles include programming languages such as Scala, Erlang, C#, etc.

2.3. Linked Data Notifications

The Linked Data Notifications (LDN) [26] W3C Recommendation[3] is a recent standardization effort 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.

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 4 (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 4, right).

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

3. RSP Actors: Decentralized Communication

In order to successfully make RDF streams available on the Web, a recent work [23] outlined a set of requirements, some of them derived from the more general guidelines for stream processors [33]. These can be summarized as: (1) prioritizing active paradigms for data exchange, (2) combination of streaming and stored data, (3) availability, distribution and scalability,

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

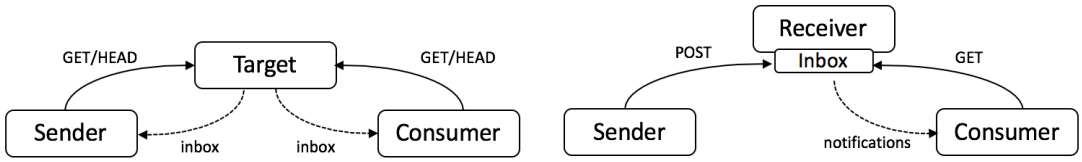

Fig. 4. LDN. Left: Discovery process of a target inbox from a sender and consumer. Right: sending and retrieving notifications from an LDN inbox.

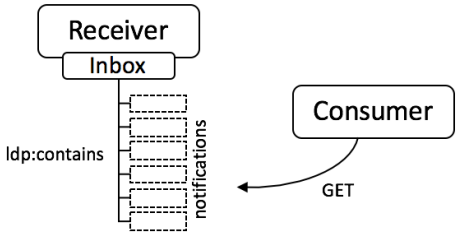

Fig. 5. Retrieving individual notifications from a receiver in LDN.

(4) wide range of stream operations, (5) availability of stream metadata, (6) support for a variety of streams, and (7) reuse of existing protocols and standards. While we endorse these requirements in general, in this paper we emphasize the need for guaranteeing that RSP engines interact among each other in a decentralized manner, following the nature of the Web. This implies a departure from the usual setting in previous RSP approaches, where a server-centric paradigm was usually followed. This is reflected, for instance, on the interaction patterns of a continuous query workflow, where the entire focus is solely on the query engine server.

We propose a model centered on what we call *RSP actors*, i.e. autonomous agents that can be deployed in a distributed fashion, and that are able to communicate and exchange RDF streams and their corresponding metadata (Figure 6). As described in the actor model (Section 2.2) each RSP actor encapsulates a state and behavior, and manages incoming messages through its inbox. Each actor may act as a sender or receiver of two main types of messages:

- *RDF stream elements*: these are RDF triples or graphs from a given RDF stream, as defined in Section 2.1. The stream delivery of these messages, either pulled or pushed, can be applied in different scenarios, e.g. feeding a stream, delivering query answers, pushing reasoning entailments, etc.
- *RDF stream metadata*: these are essentially metadata exchange messages required in order to perform tasks, such as: retrieving a stream description, declaring and RDF stream, filter a set of stream endpoints, declaring a query, etc.

Each actor has a unique identifier that can be used to find it, and it can have a set of endpoints, which can be used to reach the RSP actor resources, i.e. its streams and respective contents. The state of each RSP actor includes the metadata of the RDF streams it manages, as well as other information relative to them, i.e. back ground RDF datasets, RDF stream buffers, ontology TBoxes, RDF constraint rules, etc. The behavior of each actor defines how it proceeds at the arrival of incoming messages. This typically translates into the implementation of internal processing mechanisms, such as continuous query processing, complex event processing, stream reasoning, etc. To do, so the actor may emit new messages (e.g. response to a query), create new actors (e.g. a pushing actor, or a subscriber handler, etc.), or schedule other actions (see Figure 6).

Regarding the requirements mentioned previously, this abstract model addresses them in the following ways. For requirement 1, it natively supports asynchronous message passing, including the ability to push streams of messages if necessary. Concerning the combination of streams and stored data, the model takes a no-sharing approach for state information, so in principle any stored RDF data is only locally accessible and modifiable. The only allowed procedure to exchange it is through message delivery, which is fundamental to also guarantee scalability and distribution (requirement 3). The behavior of each RSP actor allow sufficient freedom and flexibility to implement different types of operators and processing mechanisms (requirement 4), while the explicit definition of RDF stream metadata covers requirement 5. The variety of streams is not restricted by the model (requirement 6), and the usage of standards, as we will see later with the usage of HTTP and LDN, is also advocated.

3.1. Messages and Notifications in RSP Actors

The RSP actor model places special importance to the messages that are exchanged, as they are the only

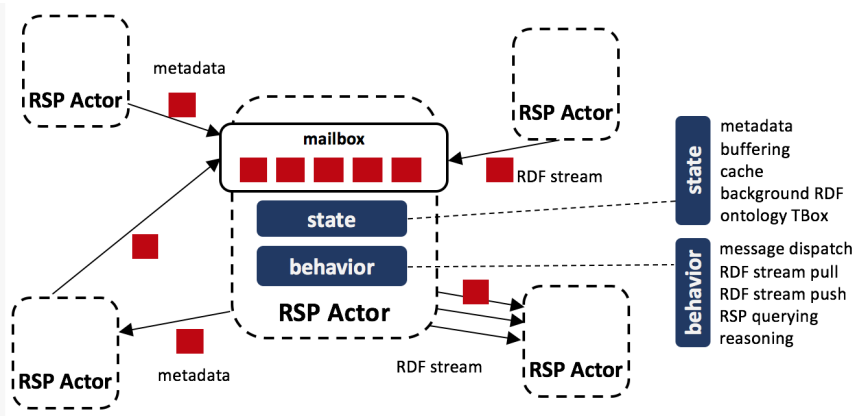

Fig. 6. RSP Actors. Eahc actor encapsulates a state and behavior, and manages incoming messages through its mailbox.

means for sharing information among them, and coordinating their interactions. The model takes special considerations about how the messages (also called notifications) are handled, given the differences of dealing with streaming vs. stored RDF data.

Format & vocabularies. Messages in RSP actors fall under two fundamental categories: RDF stream elements, and RDF stream metadata. In both cases RDF is the underlying data model used to represent the information that is exchanged, although there are minimal expectations on both cases. For RDF stream elements, these are expected to conform to the general RDF stream abstract model, as described by the W3C RSP Community Group (see Section 2.1). However, this abstract model provides high flexibility concerning the use of a particular vocabulary (e.g. using the SSN ontology for representing sensor streams, the Event ontology for streams of events, or the PROV ontology for provenance descriptions). Concerning the metadata, it would be advisable to provide a standard vocabulary for RDF stream descriptions, as proposed in [23, 24], although this goes beyond the scope of this paper.

Message storage. RSP engines are designed in such a way that RDF stream elements flow through them, and produce continuous results. Therefore, the stream is not stored anywhere, at least not in the way it is done in a traditional database or data store. In consequence, if RDF stream elements are matched to messages in the RSP actor model, the it follows that these messages are bound to *fade* as the time passes.

Message resolvability. As a consequence of the previous observation, it is hard to allow stream messages to be retrievable after they have been processed by an RSP actor. This differs to other RDF/Linked Data use case where data dynamics do not follow a streaming paradigm. As we will see later, this leads to the introduction of *input* and *output* RDF streams, which restrict RSP actors to either only write or read from a stream. In any case, resolving a particular stream element is of less importance in the context of RSP, than resolving the current contents of a stream, or a view over a stream.

Push message delivery. While on traditional Web standards, pulling is the primary method for delivering data (e.g. through HTTP `GET/POST` requests), it is not always the most suitable option for data streams. As it will be described later, our proposed RSP actor implementation provides alternative delivery methods, allowing the usage of WebSocket or HTTP Server-sent-events. The abstract description of the RSP actor leaves the message delivery method open for either of the options.

Querying. Given the ubiquity of query access patterns in RDF stream processing, it would be natural to include explicit interaction specifications for registering standing queries, as well as accessing their results as streaming notifications. This applies not only for window-based continuous queries, but also for Complex Event Processing, and for certain use cases in stream reasoning.

3.2. Stream Receivers, Senders and Consumers

The RSP actor model specifies three main types of actors: *Stream Receiver*, *Stream Sender*, and *Stream Consumer*. An RSP actor may play the role of one or all of these types.

3.2.1. Stream Receiver

The Stream Receiver is a profile for an RSP actor that is capable of receiving and processing the following types of messages:

- `RetrieveAllStreams`: to request metadata of all RDF streams registered in the receiver actor.
- `CreateStream`: to request the declaration of an RDF stream. This message includes the metadata of the RDF stream to be created.
- `RetrieveStream`: to request for the metadata of a given RDF stream in the receiver actor. The requested stream is identified by its IRI.
- `SendStreamItem`: to add a stream element to an existing RDF stream residing in the requested receiver actor. This message includes the stream element itself, as well as the RDF stream IRI.
- `RetrieveStreamItem`: to request for a specific stream element. The message includes the IRI of the RDF stream and the element itself. As in stream systems and element might be volatile, in the sense that it might not be de-referenceable after some time, this message includes also views over stream elements (e.g. based on time recency, etc.)
- `PushStreamItems`: to request for stream items to be pushed back. The message includes the RDF stream IRI.
- `CreateQuery`: to request for a continuous query to be registered. The query includes the reference to the stream IRIs to be used, and the IRI of the resulting stream of responses.

Following the actor model, the Stream Receiver will act upon arrival of any of the above messages to its inbox. We show in Algorithm 1 a sketch of how the actor reacts to these messages. The receive method of the actor is the interface used to indicate what action to take in each case.

The steps taken are self-explanatory in most cases. The Stream Receiver calls internal methods, for instance to create a stream (*postInputStream*) or to retrieve a stream item (*retrieveStreamItem*). In practice these methods will be implemented on top of exiting RSP, CEP or stream reasoners, binding their native implementations to this interface.

The Stream Receiver can manage a number of RDF streams registered within it. Streams may be of two different kinds:

- *Input streams*: these streams are essentially meant to only receive new items, but are not intended to be consumed by other actors other than the one that hosts it. Examples of such streams are those used as input for RSP queries: other RSP actors can feed these streams, but the query processor on the Stream Receiver is the only one that consumes it.
- *Output streams*: these are those RDF streams that are meant to be consumed by other RSP actors, but fed only by the actor that hosts it. An example of such stream is the continuous result of an RSP query engine.

RDF streams can also be available as both input and output.

Algorithm 1 Stream Receiver: receive function

```
1:  procedure RECEIVE(msg)
2:      sender ← msg.sender
3:  switch msg:
4:  case RetrieveAllStreams:
5:      send(getAllStreams) to sender
6:  case CreateStream:
7:      ack ← postInputStream(msg.body)
8:      send(ack) to sender
9:  case RetrieveStream:
10:     send(getStream(msg.uri) to sender
11: case SendStreamItem:
12:     postStreamItem(msg.uri, msg.body)
13: case RetrieveStreamItem:
14:     send(retrieveStreamItem(r.uri)) to sender
15: case PushStreamItems:
16:     handler ← pushStreamItems(msg.uri)
17:     handler.onReceive(data) :
18:         send(data) to sender
19: case CreateQuery:
20:     ackgetspostQuery(msg.body)
21:     send(ack) to sender
```

3.2.2. Stream Sender

This type of RSP actor characterizes those interactions related to sending RDF metadata , as well as RDF stream contents to another actor. The sender defines the following basic operations:

- *postStream*: send RDF stream metadata to declare it on a Stream Receiver. The sender emits a `CreateStream` message through this operation.
- *postStreamItem*: send and RDF stream element to an (input) stream on a given Stream Receiver. This is typically a feed stream message.
- *postQuery*: register a query on a Stream Receiver with a `CreateQuery` message.

Apart from these operations, a sender must also be able to discover the Stream Receiver endpoints. For this, it has a discover operation, which for a given stream IRI, requests the endpoint or endpoints available for sending (or consuming) stream elements. The sender also has operations to retrieve the metadata of a given RDF stream, or all available RDF streams on a receiver (`getStream`, and `getAllStreams`, respectively). These operations are common to a Stream Consumer, described below.

3.2.3. Stream Consumer

A Stream Consumer characterizes RSP actor interactions relative to receiving RDF stream data. It essentially defines two operations:

- *getStreamItem*: requests to consume an RDF stream item. Implementations of this operation can derive in different strategies for retrieving RDF stream contents. Given the dynamicity of streams, it is usually unfeasible to collect them one by one through their identifiers. Alternatively, these implementations may rather rely on stream views that may capture, for example, the latest stream items on a given window of time, or the ones complying to some filtering criteria.
- *pushStreamItems*: requests stream items to be pushed to the consumer. As opposite to the previous operation, which is essentially poll-based, this one requests the receiver to act as a sender as soon as there is an RDF stream element available for consumption.

As an example consider the Stream Receiver depicted in Figure 7. First, it receives a message on its inbox requesting the metadata of certain stream. The receiver dispatches back the metadata to the requester, which then post new elements to this stream. Then a consumer my also request a specific stream item to the receiver, through a corresponding message. To make these interactions possible, the actors need to have first the addresses of the other actors, and if necessary, discover their endpoint locations.

4. LDN for RSP actors

In this section, we show how RSP actors can be fronted by an LDN layer, used as a generic protocol through which RSP engines could share RDF stream data among them in the form of notifications. In this approach 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 its key aspects.

Stream identification. 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 endpoint(s) of an RDF stream are discoverable by performing a `GET` operation over the stream IRI, e.g.:

```
GET http://hevs.ch/streams/stream1
```

The response should 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://hevs.ch/streams/stream1",
  "inbox": "http://hevs.ch/streams/stream1/inbox" }
```

Similarly, this type of discovery could be performed using a HEAD request and a link HTTP header, as in LDN (See Figure 8).

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 the case of RSP actors with LDN, 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://hevs.ch/streams/stream1",
  "input": "http://hevs.ch/streams/stream1/input" }
```

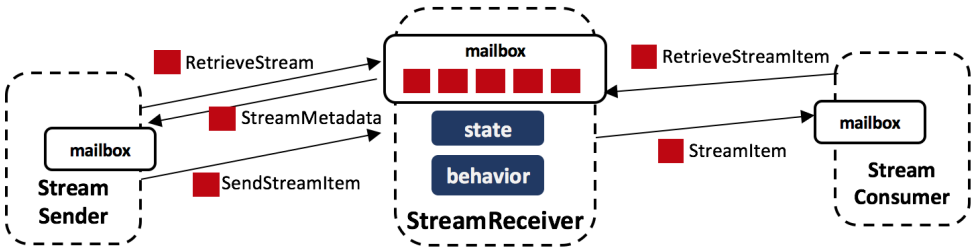

Fig. 7. RSP Stream Receiver, processes incoming messages from the sender that posts RDF stream items, and sends RDF stream items to a consumer.

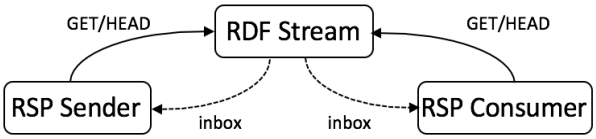

Fig. 8. The discovery interactions follow those of LDN: endpoint information is available at the target, which is an RDF stream.

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[4] vocabulary, we have used a new input term from a new vocabulary. This vocabulary is yet to be specified (see [24] for a work-in-progress in this direction), but we use the base IRI for the rest of the examples in this paper.

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 9). As an example consider the JSON-LD representation of a humidity observation posted as a timestamped graph:

```
POST /streams/stream1/input HTTP/1.1
Host: hevs.ch
Content-Type: application/ld+json

{"prov:generatedAtTime": "2017-09-22T05:00:00.000Z",
 "@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.

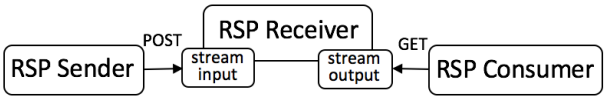

Fig. 9. RSP Sender sends a notification to a receiver stream input, and a consumer retrieves elements form a stream output.

Publicizing stream elements. As it is depicted in Fig 9, 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:

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

```
{ "@context": "http://www.w3.org/ns/ldp",
  "@id": "http://hevs.ch/streams/stream1/output",
  "contains": [
    "http://hevs.ch/streams/stream1/output/graph1",
    "http://hevs.ch/streams/stream1/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 method 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": "2017-09-22T05:00:00.000Z",
     "@id": "ex:Graph1",
     "@graph": [
       { "@id": "ex:humidityObservation",
         "ex:hasValue": 34.5 }]   },
   { "prov:generatedAtTime": "2017-09-22T06:00:00.000Z",
     "@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[5] 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://hevs.ch/streams/stream1`. Notice that this type of queries could include references to more than one input stream.

```
SELECT ?s ?p ?o
WHERE {
  STREAM <http://hevs.ch/streams/stream1> [RANGE 2s]
  {?s ?p ?o}
}
```

5. Implementation

We have implemented the RSP actors model and the LDN interfaces as a library available for JVM languages such as Java or Scala. The code is open-source, and it is available in Github[6]. The core of the RSP actors implementation is written in Scala, using the Akka Actors library[7]. Akka provides the essential programming abstractions to create actors, providing message dispatching, remoting, actor hierarchies, and other features.

For instance, the generic RSP trait from which all RSP receiver actors inherit, `ActorStreamReceiver`, extends from the Akka Actor trait, that defines a `receive` method where all messages from the mailbox are processed. As shown in the snippet below, it follows exactly the logic presented before in Alg. 1, checking on the type of message that is received, and acting upon.

```
trait ActorStreamReceiver extends Actor with StreamReceiver{
  def receive = {
    case req:RetrieveAllStreams=>
      val s = sender
```

[5] `https://www.w3.org/TR/eventsource/`

[6] `https://github.com/jpcik/ldn-streams`

[7] `http://akka.io`

```
    s ! (getAllStreams(req.range))

  case req:CreateStream=>
  ...
```

RSP actors defines *traits* (analogous to interfaces in Java and other languages) for its main types of objects. For instance, the `StreamReceiver` trait implements the receiver actor described in Section 3.2.1. These traits are independent of the communication layer, e.g. it has no dependencies with LDN. Two specialized traits, `ActorStreamReceiver`, and `LdnStreamRecevier` provide specific interfaces for both Akka-Actor communication, and LDN, respectively. The trait hierarchy is depicted in Figure 10.

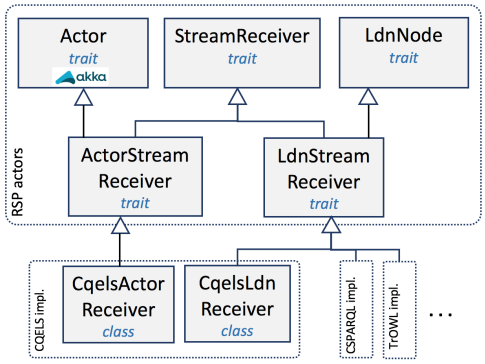

Fig. 10. RSP Actor traits. Implementing modules, such as CQELS, extend from ActorStremaReceiver, to implement an Actor-based interface, and from LdnStreamReceiver to support LDN.

Additional modules can be plugged to RSP actors, for instance for concrete implementations for particular RSP engines. In Figure 10 we illustrate this for a CQELS Actor and LDN classes, but we have also implemented classes for C-SPARQL and TrOWL.

In order to allow the integration with these existing engines, it suffices that they provide a JVM-compatible API. This is the case with the previously mentioned engines, as they have a Java API. RSP actor traits make use of abstract methods that need to be implemented for any specific extension. For example, the `StreamReceiver` trait defines abstract methods that allow: feeding an RDF stream with graph (`consumeGraph`), register a query (`query`), push data results (`push`), and terminate push and clean resources (`terminatePush`), as shown in the code snippet below.

```
trait StreamReceiver extends LdnEntity{
  def query(name:String,queryStr:String,insert:Map[String,Any
      ]=>Unit):Unit

  def consumeGraph(uri:Uri,g:Graph):Unit

  def push(queryId:String,insert:Map[String,Any]=>Unit):
      ResultHandler

  def terminatePush(queryId:String,handler:ResultHandler):Unit
```

Extended receivers for particular engines can easily implement these methods as shown below. First, any initialization code can be specified after declaring the inherited class or trait. In the example below, a CQELS trait inherits from `StreamReceiver`, and initializes the CQELS engine.

```
trait CqelsReceiver extends StreamReceiver{
  val cqelsCtx=new ExecContext("./tmp/cq_id",true)
  val cqels=new CQELSEngine(cqelsCtx)
```

Then, it is straightforward to implement the remaining methods. For instance, in CQELS, the `send` method, taking as input a triple, needs to be invoked to feed the engine, as shown below.

```
  override def consumeGraph(uri:Uri,g:Graph)={
    g.triples.foreach { t => cqelsCtx.engine send (uri.
        toString,t) }
  }
```

Similarly, to register a query in CQELS, the `registerSelect` method needs to be invoked, and afterwards the results are handled through a listener.

```
override def query(name:String,queryStr:String,
      insert:Map[String,Any]=>Unit)={

  val slct=cqelsCtx.registerSelect(queryStr)
  slct.register(new ContinuousListener{
    def update(map:Mapping)={
      val newMap=map.vars.asScala.map{v=>
        v.getVarName->cqels.decode(map.get(v))
      }.toMap
      insert(newMap)
    }
  })
}
```

For other engines, a similar process can be followed. For instance in C-SPARQL, to feed the stream, one should instantiate an `RdfQuadruple` class that represents a timestamped triple.

```
override def consumeGraph(uri:Uri,g:Graph)={
  g.triples.foreach { t =>
    csparql.getStreamByIri(uri.toString).put(
        new RdfQuadruple(t.subject,t.predicate,t.object,System
            .currentTimeMillis))
  }
}
```

An RSP actor application can use a simple API provided by the library. To instantiate an RSP actor, an Akka system object can be started, and then the specific type of actor can be instantiated (e.g. `CqelsActorReceiver` for CQELS):

```
val sys=ActorSystem("testSys")
implicit val serverIri="http://hevs.ch/streams"
implicit val ct:ContentType.NonBinary=`application/ld+json`
val cqels=sys.actorOf(Props(new CqelsActorReceiver(serverIri
     ,1)), "cqels")
```

Notice that all actors instantiated in this way (e.g. the `cqels` variable in the previous example) are of type `ActorRef`, i.e. a reference to the actor itself.

5.1. Clients

The API also includes tools to create clients in a simple manner. For instance, the code below posts a query to the previously defined CQELS actor:

```
 client.postStream(cqels, "s1")
val query="""
  SELECT ?s ?p ?o WHERE {
    STREAM <http://hevs.ch/streams/s1> [RANGE 2s]
      {?s ?p ?o}
  }"""
val client = createClient
client.postQuery(cqels,"q1", query, ct)
```

As a second example, the next code snippet posts a new item in JSON-LD to the stream identified by the IRI: `http://hevs.ch/streams/s1`.

```
val ev="""{ "@context": "http://schema.org/",
            "@type": "Event",
            "value": "Some Event"} """
client.postStreamItem(cqels, "http://hevs.ch/streams/s1", ev)
```

5.2. LDN Implementation

Extending the actor based implementation of RSP actors, we have developed an HTTP communication layer for the library, based on LDN, as explained in detail in Section 4. This implementation extends an LDN code-base that complies with most of the specification for a Sender, Receiver and Consumer, as indicated in the LDN implementation reports[8].

The implementation uses the Akka HTTP library[9], which provides a routing API, streaming support, marshalling, etc. For the stream receiver implementation, the main entry point is the routing declaration, which determines how incoming requests paths map to different actions on the RSP actors side. As an example, consider the snippet below, where the routing paths are defined using the Akka API:

```
val receiverRoute =
    path("streams") {
      pathGetAllStreams ~
      pathPostInputStream
    } ~
    path("streams" /  Remaining ){ name=>
      pathGetStream(name)
    } ~
    path("streams" / "query") {
      pathPostQuery
    }
    ...
```

For instance, any path starting with `"streams/"` and an identifier, will map to the `pathGetStream` method, that processes the incoming request and retrieves the corresponding stream metadata, according to the specified media type ranges, and other HTTP headers. Notice that for reusing the LDN trait for a specific RSP engine (e.g. CQELS, C-SPARQL, TrOWL, or others), developers do not need to deal with the routing details, which are provided by the library.

6. Experimentation

In this section we present a set of experimentation results of the implementation of RSP actors. The goal of these experiments is to show how the actor model implemented in the library can be used with existing engines. The project already provides implementations for CQELS, C-SPARQL, and TrOWL, but it can be used for any other processor. In the following experiments, the main indicator evaluated is the throughput, measured in terms of efficiency, i.e. the rate between the actual number of RDF stream elements processed per unit of time, and the maximum ideal number of processed elements. The choice of this metric is based first on the need to assess the behavior of the platform to different conditions of the input streams (e.g. number of streams/senders, velocity of the streams, number of parallel processors). The usage of a rate indicator is due to the fact that an absolute throughput is obviously variable depending on the input stream characteristics. Therefore the efficiency rate provides a normalized parameter. Queries and data have been adapted from SR-Bench [34], using an upgraded version of the datasets,

[8] `https://linkedresearch.org/ldn/tests/summary`

[9] `https://doc.akka.io/docs/akka-http/current/scala/http/`

using the new version of the SSN Ontology[10], and a synthetic generation stream feeder. The data consists of environmental sensor observations. All experiments were run on Ubuntu 16.04 LTS, Intel Core i7-7700U (3.60GHz, 8MB cache, Quad Core).

In the first set of experiments, we measured the throughput efficiency, for different input stream rates (1,10,10, and 1000 graphs/s), and for different number of concurrent senders, and a single receiver (Figure 11).

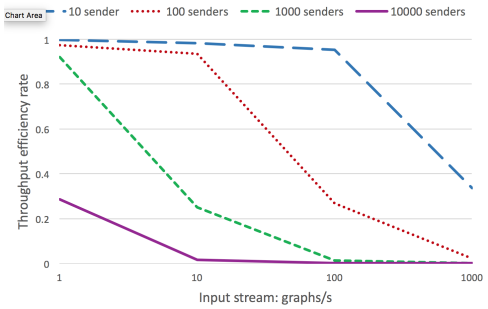

Fig. 11. Throughput efficiency vs. input stream rates, for different sets of concurrent senders, and one receiver.

As it can be seen, the efficiency decreases considerably as the input stream increases. A drastic increase produces a significant drop in efficiency, either if it is by increasing the number of senders or the input rate. This is basically due to the limitations of CQELS as the underlying engine.

The next experiment is set in exactly the same conditions, excepting that it uses 5 concurrent stream receiver actors instead of only 1 (Figure 12). As it can be seen, using more receiver actors already provides a higher throughput efficiency for a larger number of cases.

The next experiment provides more information on how a set of CQELS engines running as RSP actors can handle a total of 10K concurrent senders, each spitting 1 graph per second. The experiment is set for 1,5, 10, 20, 50, and 100 concurrent CQELS actor receivers. As it can be seen in Figure 13, with 5, 10 and 20 concurrent senders there is a considerable improvement of the throughput efficiency. However, increasing even more receivers produces a sustained decrease, as the CPU is not able to scale on its own to that number of engines. A distributed deployment would be required to scale in that case.

[10] https://www.w3.org/TR/vocab-ssn/

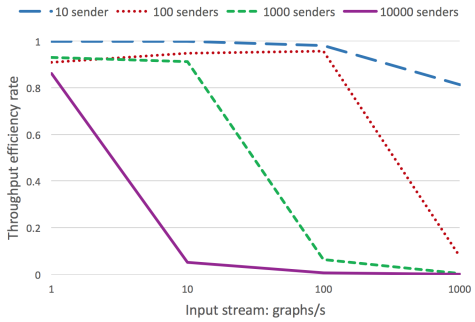

Fig. 12. Throughput efficiency vs. input stream rates, for different sets of concurrent senders, and five receivers.

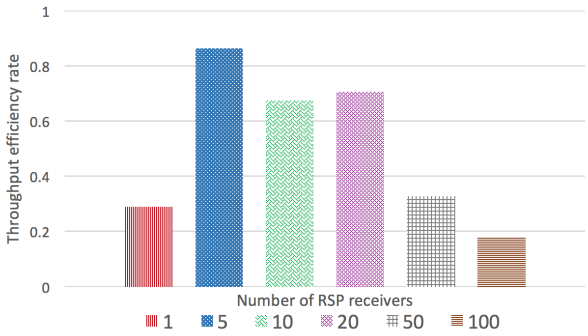

Fig. 13. Throughput efficiency vs. number of RSP receivers, for 10K concurrent senders.

The next experiment shows how RSP stream receivers (CQELS engine) respond to different number of senders (1, 10, 100, 1000, 1000). It shows results for input streams of 1, 10, 100, 1000 graphs/s (Figure 14). The graph shows the progressive efficiency decrease as the input stream (combined with the number of senders) increases. For instance, for 10K senders, and 1K graphs/s, the input load is of 10M graphs/s, which is too much for a single receiver instance.

In the next experiment, we evaluate a similar scenario, but this time adding CQELS RSP instances (1, 2, 5, 8, and 10 concurrent engines). The results are shown for different numbers of RSP concurrent senders (2K, 4K, 8K, 10K, and 16K). For very high input loads, the system is still capable of at least 0.5 efficiency. It is clear that at this point a cluster deployment would be required.

The final experiment was performed only for RSP actors ingesting but not processing data. It shows results for different number of RSP actors ingesting streams (1, 5, 10, 20, 50, and 100 concurrent actors), and for 100, 1000, and 10000 senders. As it can be seen, for 1K and

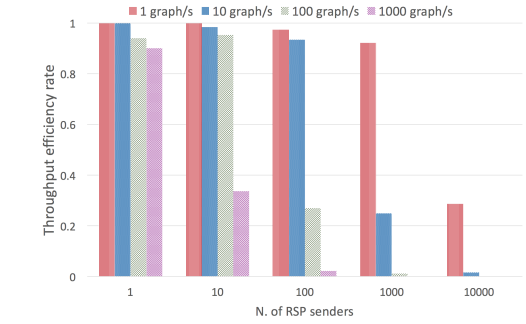

Fig. 14. Throughput efficiency vs. number of RSP senders, for different input stream rates.

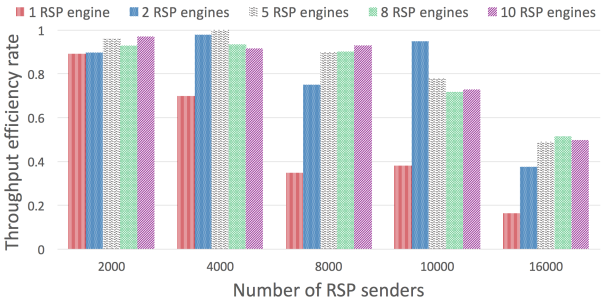

Fig. 15. Throughput efficiency vs. number of RSP senders, for different numbers of RSP receiver engines.

10K adding additional resources in general increases the overall efficiency (Figure 16).

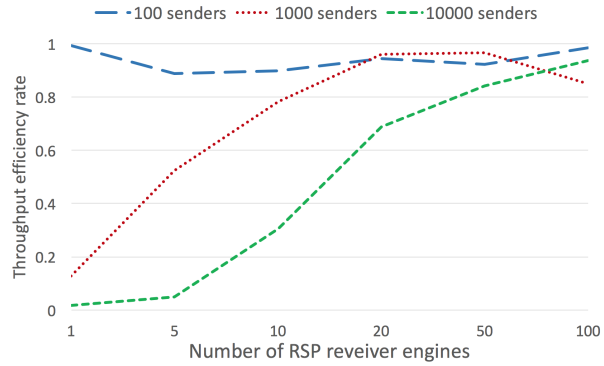

Fig. 16. Throughput efficiency vs. number of RSP receivers without processing, for different sets of concurrent senders.

7. Related Work

Several RSP engines have been developed in the last decade, focusing on the processing aspects of RDF streams, including incremental reasoning, continuous querying, complex event processing, among others [11, 12, 14, 32, 35, 36]. However, most of these RDF stream processors disregard to a certain degree the Web dimension, as they provide programming APIs and interfaces that are not immediately accessible from consumers on the Web.

Some initial attempts to provide generic service interfaces for streaming data were explored in [31, 37]. They both relied on HTTP REST-ful services, and in the second case targeting mainly sensor data systems. Although these early works were presented mainly as exploratory proposals, with limited or no implementation, they already pointed out the need for simple and programming-friendly WEb interfaces for RDF streams. A more recent development following this research line is the RSP Service Interface[11], which further develops the ideas presented in [31], providing a generic implementable programming API for continuous query engines. Limitations of this approach include the lack of a standard protocol or at least recommendation, in order to guarantee interoperability. Furthermore, it is too closely oriented to only query processing, while the RSP spectrum includes other types of interactions, as we have seen in RSP actors.

Another related approach targeting connectivity in networks of RSP engines is the SLD Revolution framework [38]. It optimizes a distributed workflow of RSP engines, using lazy-transformation techniques so that non-optimized RDF formats are used only when necessary.

A particularly relevant topic in this context is the availability and publication of RDF streams on the Web. While for stored RDF data, there have been countless papers and implementations for publishing Linked Data, and other forms of RDF datasets, for streaming RDF data this has surprisingly not been the case. Existing works that touched this issue only tangentially include [39, 40], which describe techniques for producing RDF dataset based on unstructured streams as input. Following the trend of Linked Data publishing, other works developed frameworks and tools for the publication of streams following the Linked Data principles [41, 42]. However, these frameworks were not

[11] http://streamreasoning.org/resources/rsp-services

concerned with the issues related to the interchange of RDF streams on the web, or the establishment of standard protocols, as they rather focused on adherence to the Linked Data recommendations, which are essentially useful for stored data. As a consequence, there are currently very little number of streams available on the web as RDF.

This is exactly the problem addressed by efforts like TripleWave [22], which integrates tools for the transformation of non-RDF data streams, as well as time-annotates static RDF, into RDF streams. TripleWave allows the publication of these RDF streams so that they can be directly consumed or connected with applications that process them. As a continuoation to TripleWave, WeSP[12] is a recent proposal for producing and consuming RDF streams on the Web. The roadmap proposed by WeSP can be a starting point for a community-driven and pragmatic definition of RDF stream metadata vocabularies, interoperable interfaces, reuse of standard protocols, etc. In fact, the RSP actors proposed in this paper is a step into this direction, with specific technological and design choices for the communication among RSP engines. We consider that the simplicity and generality of LDN are valuable features that match the requirements for Web-wide RSP interactions.

It is also worth mentioning the efforts of the W3C RDF Stream Processing Community Group (RSP-CG), which intended to provide support for the development of a common data model for representing RDF streams, as well as defining guidelines for common principles for RSP query processing, data exchange, and reasoning. The documents produced so far include a set of requirements, design principles[13], and an abstract model[14]. RSP actors takes into consideration these results, and in fact adopts many of its recommendations and principles, as it can be seen throughout this paper.

Web technologies are increasingly including protocols, frameworks and tools for stream processing, given the increasing importance for IoT and social network data support. It will be important to follow existing trends, which are sometimes competing or overlapping, so that the RSP community is aligned to them. Examples of these technologies include WebSocket[15], MQTT[16], Server-Sent Events[17], etc.

8. Discussion and Potential Impact

The RSP actors model and implementation described in this work address an important issue in the area of RSP, linked to the need for appropriate tools that allow RSP engines to interact on the Web. RSP actors have the potential to solve many of the challenges of integrating RDF stream processing engines on the Web, given the following characteristics of its design:

- *IRIs:* RDF streams, their endpoints, stream elements, etc. are all Web resources identified with IRIs. This is a key principle for discovery, identification, reuse, etc.
- *Messaging:* Asynchronous message-passing communication is the primary way of exchanging both RDF stream data and metadata, targeting scalability and responsiveness.
- *Decentralization:* The design of RSP actors allows all participating instances to act as either producers or consumers of RDF streams, allowing the creation of decentralized networks of RSP engines.
- *Delivery:* Pull and push-based mechanisms are available depending on the application needs.
- *Implementation-agnosticism:* The generic model allows for concrete implementations to be plugged-in, allowing the addition of extensions for particular needs.
- *LDN:* The usage of LDN as communication protocol allows interoperability with other implementations, by adhering to the specification.

Nevertheless, the RSP actors approach still needs to face several challenges to gain adoption in the near future. We mention some of these challenges and current limitations.

- LDN itself is a new specification, and still has to show wider adoption, even outside of the RDF streams context.
- Implementation for other engines, reasoners and systems would provide further evidence of the applicability of this approach.

[12] http://w3id.org/wesp/web-data-streams
[13] https://w3id.org/rsp/requirements
[14] https://w3id.org/rsp/abstract-syntax
[15] https://www.w3.org/TR/websockets/
[16] http://mqtt.org/
[17] https://www.w3.org/TR/eventsource/

- The implementation of other types of communication mechanisms (e.g. through MQTT) would widen the range of potential use cases for which RSP actors would be of interest.
- The development of further functionalities targeting more specific interactions and features for stream reasoners (e.g. support for entailment regime parameters, materialization options, etc.) would provide a more comprehensive set of available functionalities.
- The integration with existing RDF stream publishing frameworks (e.g. TripleWave) and simulators, would facilitate the adoption and learning curve for RSP in general.

Finally, concerning the agreement on a certain protocol or model for RSP communication, we advocate for a bottom-up approach, reusing as much as possible existing standards. This is one of the reasons why we choose LDN as a protocol. All implementers can check the specification and claim compliance if they adhere to it. However, at some point we expect that the community will bring the topic to a wider discussion, and the RSP actors approach can serve as a possible starting point.

9. Conclusions

RDF provides a Web-native model that facilitates the integration and interpretation of data. This is a key motivations for using RDF to represent and process streams of data, going beyond traditional Linked Data production and consumption. However, while for stored datasets there exist well-established standards and recommendations for producing, publishing and consuming RDF, for data streams the situation is different. The fact is that there is still no agreed nor well-supported specification or model for exchanging RDF streams on the Web. Although recent efforts provide partial solutions to the problem – e.g. TripleWave for publication, or SLD Revolution for orchestration– there is a need for a standardized Web mechanism for communicating data among RSP engines in general.

RSP actors, as described in this paper, provides a generic model for integrating different RDF stream producers or consumers. The model is based on the actor paradigm, and uses synchronous message passing and a decentralized communication patterns. On top of this, it incorporates Linked Data Notifications as a native HTTP protocol for ensuring interoperability among RSP actors. The decentralized nature of LDN, along with its simplicity and extensibility, are positive arguments for advocating its use, as it was already proposed in a previous work [27].

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*. Furthermore, a community agreement would need to be made concerning vocabularies for the description of RDF stream metadata.

Also, it is important to consider the use of RSP actors for other use cases, targeting not only querying, but 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 and scalability reasons. This trend extends to the RDF processing world in general, and the same principles described in this work and, in RSP in general, could have an enormous impact even outside of the Semantic Web/RDF scientific communities.

References

[1] B.-R. Chen, S. Patel, T. Buckley, R. Rednic, D.J. McClure, L. Shih, D. Tarsy, M. Welsh and P. Bonato, A web-based system for home monitoring of patients with Parkinson's disease using wearable sensors, *IEEE Transactions on Biomedical Engineering* **58**(3) (2011), 831–836.

[2] D. Alahakoon and X. Yu, Smart electricity meter data intelligence for future energy systems: A survey, *IEEE Transactions on Industrial Informatics* **12**(1) (2016), 425–436.

[3] K. Teymourian, M. Rohde and A. Paschke, Knowledge-based processing of complex stock market events, in: *Proceedings of the 15th International Conference on Extending Database Technology*, ACM, 2012, pp. 594–597.

[4] E. Della Valle, S. Ceri, F. Van Harmelen and D. Fensel, It's a streaming world! Reasoning upon rapidly changing information, *IEEE Intelligent Syst.* (2009), 83–89.

[5] R. Cyganiak, D. Wood and M. Lanthaler, RDF 1.1 concepts and abstract syntax, *W3c Recommendation* **25**(02) (2014).

[6] B. Motik, P.F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler et al., OWL 2 web ontology language: Structural specification and functional-style syntax, *W3C recommendation* **27**(65) (2009), 159.

[7] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013. `https://www.w3.org/TR/sparql11-query/`.

[8] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte and O. Corcho, RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems, *International Journal on Semantic Web and Information Systems (IJSWIS)* **10**(4) (2014), 17–44.

[9] H. Beck, M. Dao-Tran, T. Eiter and M. Fink, LARS: A logic-based framework for analyzing reasoning over streams, in: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI Press, 2015, pp. 1431–1438.

[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*, 2016, pp. 145–162.

[11] 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*, 2011, pp. 370–388.

[12] D. Anicic, P. Fodor, S. Rudolph and N. Stojanovic, EP-SPARQL: a unified language for event processing and stream reasoning, in: *WWW*, 2011, pp. 635–644.

[13] 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) (2010), 3–25.

[14] 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** (2012), 43–63.

[15] J. Kietz, T. Scharrenbach, L. Fischer, A. Bernstein and K. Nguyen, TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams, Technical Report, Tech. Rep., Technical Report, University of Zurich, Department of Informatics, 2013.

[16] S. Gao, T. Scharrenbach and A. Bernstein, The clock data-aware eviction approach: Towards processing linked data streams with limited resources, in: *European Semantic Web Conference*, Springer, 2014, pp. 6–20.

[17] J. Urbani, A. Margara, C. Jacobs, F. Van Harmelen and H. Bal, Dynamite: Parallel materialization of dynamic rdf data, in: *International Semantic Web Conference*, Springer, 2013, pp. 657–672.

[18] J.-P. Calbimonte, J. Mora and O. Corcho, Query rewriting in RDF stream processing, in: *International Semantic Web Conference*, Springer, 2016, pp. 486–502.

[19] F. Lécué and J.Z. Pan, Predicting Knowledge in an Ontology Stream., in: *IJCAI*, 2013, pp. 2662–2669.

[20] J.-P. Calbimonte, RDF stream processing: let's react, in: *OrdRing*, 2014, pp. 1–10.

[21] C. Bizer, T. Heath and T. Berners-Lee, Linked data-the story so far, *Semantic services, interoperability and web applications: emerging concepts* (2009), 205–227.

[22] 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*, 2016, pp. 140–149.

[23] D. Dell'Aglio, D. Le Phuoc, A. Le-Tuan, M.I. Ali and J.-P. Calbimonte, On a Web of Data Streams, in: *Proc. ISWC Workshop on Decentralizing the Semantic Web DeSemWeb 2017*, 2017.

[24] Y.A. Sedira, R. Tommasini and E. Della Valle, Towards VoIS: a Vocabulary of Interlinked Streams, in: *Proc. ISWC Workshop on Decentralizing the Semantic Web DeSemWeb 2017*, 2017.

[25] C. Hewitt, P. Bishop and R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: *Proceedings of the 3rd international joint conference on Artificial intelligence*, Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[26] S. Capadisli, A. Guy, C. Lange, S. Auer, A. Sambra and T. Berners-Lee, Linked Data Notifications: a resource-centric communication protocol, in: *ESWC*, 2017, pp. 537–553.

[27] J.-P. Calbimonte, Linked Data Notifications for RDF Streams, in: *Proc. of the Web Stream Processing (WSP) Workshop at ISWC*, 2017.

[28] Y. Ren and J.Z. Pan, Optimising ontology stream reasoning with truth maintenance system, in: *CIKM*, ACM, 2011, pp. 831–836.

[29] A. Arasu, S. Babu and J. Widom, The CQL continuous query language: semantic foundations and query execution, *The VLDB Journal* **15**(2) (2006), 121–142, ISSN 1066-8888. doi:10.1007/s00778-004-0147-z.

[30] G. Cugola and A. Margara, Processing flows of information: From data stream to complex event processing, *ACM Computing Surveys* **44**(3) (2011), 15–11562.

[31] D.F. Barbieri and E. Della Valle, A Proposal for Publishing Data Streams as Linked Data - A Position Paper, in: *LDOW*, 2010. http://ceur-ws.org/Vol-628/ldow2010_paper11.pdf.

[32] X. Ren, O. Curé, L. Ke, J. Lhez, B. Belabbess, T. Randriamalala, Y. Zheng and G. Kepeklian, Strider: an adaptive, inference-enabled distributed RDF stream processing engine, *Proceedings of the VLDB Endowment* **10**(12) (2017), 1905–1908.

[33] M. Stonebraker, U. Çetintemel and S.B. Zdonik, The 8 requirements of real-time stream processing, *SIGMOD Record* **34**(4) (2005), 42–47.

[34] Y. Zhang, P.M. Duc, O. Corcho and J.-P. Calbimonte, SRBench: a streaming RDF/SPARQL benchmark, in: *The Semantic Web–ISWC 2012*, Springer, 2012, pp. 641–657.

[35] 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) (2010), 3–25.

[36] S. Komazec, D. Cerri and D. Fensel, Sparkwave: continuous schema-enhanced pattern matching over RDF data streams, in: *Proc. 4th ACM International Conference on Distributed Event-Based Systems DEBS*, ACM, 2012, pp. 58–68.

[37] J.F. Sequeda and O. Corcho, Linked stream data: A position paper, in: *SSN*, CEUR-WS. org, 2009, pp. 148–157.

[38] M. Balduini, E.D. Valle and R. Tommasini, SLD Revolution: A Cheaper, Faster yet more Accurate Streaming Linked Data Framework, in: *RSP*, 2017, pp. 1–15. http://ceur-ws.org/Vol-1870/paper-01.pdf.

[39] D. Gerber, S. Hellmann, L. Bühmann, T. Soru, R. Usbeck and A.-C.N. Ngomo, Real-time RDF extraction from unstructured data streams, in: *International Semantic Web Conference*, Springer, 2013, pp. 135–150.

[40] 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: *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, ACM, 2014, pp. 253–261.

[41] 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: *International Semantic Web Conference (2)*, Lecture Notes in Computer Science, Vol. 8219, Springer, 2013, pp. 1–16.

[42] 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** (2012), 42–51.