# A Restful Interface for RDF Stream Processors

Marco Balduini[1] and Emanuele Della Valle[1]

DEIB – Politecnico di Milano, Italy
marco.balduini@polimi.it, emanuele.dellavalle@polimi.it

**Abstract.** This poster proposes a minimal, backward compatible and combinable restful interface for RDF Stream Engine.

## 1 Introduction

A number of RDF Stream Processors exists (e.g., CQELS [1], SPARQL$_{stream}$ [2], ETALIS/EP-SPARQL [3], Sparkwave [4], INSTANS [5] and C-SPARQL Engine [6]), but **they do not talk each other**.

This hampers comparative evaluations: existing benchmark proposals [7, 8] had to create software adapters to test the various processors. In this condition, it is difficult to assess how much the benchmark results depend on the performances of the processors and how much on those of the adapters.

Moreover, the lack of a shared protocol to transmit RDF streams hinders the combined usage of those processors. For instance, a user may want: *a*) to deploy SPARQL$_{stream}$ to natively process data streams[1]; *b*) to semantically enrich the resulting RDF streams using Sparkwave (or INSTANS); *c*) to aggregate the enriched streams in events using the C-SPARQL Engine (or CQELS); and *d*), finally, to detect complex events with ETALIS/EP-SPARQL.
This poster proposes a restful interface for RDF Stream Processors that is:

1. **minimal** – more sophisticated interface can be envisioned, but in this attempt we would like to create a broad consensus, thus we avoid proposing controversial solutions.
2. **backward compatible** – we are reusing RDF and SPARQL standards wherever we can so to guarantee that adaptation of non-streaming clients for RDF and SPARQL is straight forward.
3. **combinable** – the proposed interface enforces that the output of a processor can serve as input to a processor (including the one that generates it).

The remainder of the paper is organised as follows. Section 2 briefly presents the background required to understand the proposed interface. Section 3 proposes the interface. Section 4 shortly discusses two requirements that are not considered for this minimal proposal and how the interface can be extended to cover them. A proof of concept implementation of the proposed interfaces for the C-SPARQL engine is available for download at `http://streamreasoning.org/download`.

[1] SPARQL$_{stream}$ rewrites continuous SPARQL queries issued against virtual RDF streams on continuous SQL-like queries on data streams.

## 2 Background

From a conceptual point of view, existing RDF Stream Processors are homogenous. They define the notion of *RDF Stream* – an unbound list of tuples $< t, \tau >$ where $t$ is an RDF triple and $\tau$ is a non-decreasing timestamp –, and *continuous SPARQL query* – a SPARQL query extended so that it can process RDF streams using continuous operators (e.g., windows to logically convert a portion of the infinite RDF stream in an RDF graph) and time-aware operators (e.g., sequence to ask that a graph pattern is detected before another one).

To the best of our knowledge, limited efforts was spent in defining a protocol for: *a*) transmitting RDF stream across RDF Stream Processor on separated machines, *b*) registering a continuous query in a processor, and *c*) observing the continuously evolving results. The only existing solution are proprietary. For instance, the C-SPARQL Engine is typically used within the Streaming Linked Data framework [9]. Similarly, CQELS is paired with the Super Stream Collider [10].

## 3 Services

A community effort is needed to propose a continuous SPARQL extension that can span across the existing proposals, but we believe this is the right time to propose a restful interfaces that processors can easily implement.

The following proposal specifies how to manage RDF streams, continuous SPARQL queries, and observers of continuous results (see Table 1 for details).

Complying to restful principles, users can register a new RDF stream $\sigma$ in the processor using the `PUT` method on the resource `/streams/`. As a result the RDF stream `/streams/`$\sigma$ becomes available in the processor. At this point, they can stream information on the RDF stream `POST`ing an RDF graph to `/streams/`$\sigma$) and they can unregister it using the `DELETE` method. The list of all registered stream is returned when `GET`ting the resource `/streams/`.

It is worth to note that, learning from flexible time management in data stream processors [11], we propose to avoid annotating the streamed RDF graphs with a timestamp. This complies to the expected input of best effort data stream processors (e.g. esper). We leave the annotation of the streamed RDF graphs with application timestamp to a future extension of this **minimal** protocol. Moreover, this design decision allows the proposal to be **backward compatible**. Any Semantic Web application can send information to an RDF Stream Processor simply posting an RDF graph.

User can register a new continuous SPARQL query $\gamma$ in the processor using the `PUT` method on the resource `/queries/`. The proposed interface is agnostic w.r.t. the language used to declare the query and leaves to the processor to parse the query in the body of the request. Nonetheless, it requires the query to refer only to RDF streams already registered in the processor. If the user tries to register a query on streams that have not been registered, yet, the service must refuse to register the query. If the registration is successful, the processor starts the continuous execution of the query and the query `/queries/`$\gamma$ appears in

**Table 1:** The herein proposed restful interfaces for RDF Stream Processors. Along with restful principles, `GET`ing a resource returns what was `PUT`ed.

| **RDF Streams** | | | |
|---|---|---|---|
| Method | Address | Body | Description |
| PUT | /streams/<id> | | Register new stream |
| DELETE | /streams/<id> | | Delete specified stream |
| POST | /streams/<id> | RDF model | Stream new information |
| GET | /streams | | Get the list of streams |

| **Continuous SPARQL queries** | | | |
|---|---|---|---|
| Method | Address | Body | Description |
| PUT | /queries/<id> | query | Register new query |
| DELETE | /queries/<id> | | Delete specified query |
| POST | /queries/<id> | callback URL | Adds an observer |
| POST | /queries/<id> | Action [pause, restart] | Change query status |
| GET | /queries | | Get the list of queries |

| **Observers** | | | |
|---|---|---|---|
| Method | Address | Body | Description |
| DELETE | /queries/<id>/observers/<id> | | Delete specified observer |
| GET | /queries/<id>/observers | | Get observers list |

the list of queries that can be retrieved `GET`ting the resource `/queries/`. As for the RDF streams, the query `/queries/`$\gamma$ can be unregistered using the `DELETE` method. The method `POST` on the resource `/queries/`$\gamma$ is used to start observing the query results, to pause the query and to restart it.

Access to query results follows an observable-observer design pattern. In order to start observing the results of a query $\gamma$, a user has to `POST` a callback URL to `/queries/`$\gamma$. The created observer $\omega$ is identified by an URL of the form `/queries/`$\gamma$/observers/$\omega$. The user can stop observing the query by `DELETE`ing this resource. Multiple observers per query are possible. Whenever $\gamma$ computes new results, the processor notifies all the observers by invoking the provided callback URLs.

If the query is of the forms `SELECT` or `ASK`, results must be formatted according to SPARQL 1.1 query results[2], thus allowing for **backward compatibility** with existing SPARQL resultset parsers.

If the query is of the forms `CONSTRUCT` or `DESCRIBE`, the processor must `POST` an RDF graph containing the result. As a result our proposal is not only **backward compatibility** – it is conform to SPARQL 1.1 result formats –, but it is also **combinable** – the results of a query can be `POST`ed to another registered RDF stream. The callback URL passed as parameter in starting an observer simply has to be the URL of an existing RDF stream[3].

## 4 Conclusions

The proposal, being minimal, ignored important requirements w.r.t. time modelling, access control, and transmission overhead.

[2] Our implementation supports `http://www.w3.org/TR/2013/REC-sparql11-results-json/`

[3] In order to avoid the overhead to stream on HTTP an RDF stream that is consumed by the same processor, when a query $\gamma$ of the forms `CONSTRUCT` or `DESCRIBE` is registered, an RDF stream, whose identifier is `/streams/`$\gamma$, is automatically registered. The result of the query $\gamma$ is internally streamed on it.

Adding the application time to the protocol is only a matter to `POST` a timestamp together with the RDF graph. However, as explained in [11], in the case of multiple distributed sources `POST`ing to the same RDF stream, out-of-orders can appear due to lack of clock synchronisation and different network delays. In our future work, we will propose this extension and, at the same time, we will release an open-source package that includes the management out-of-orders.

The proposed interface lacks access control, but it is ready for HTTP-based access control. An HTTP server, between the user and the Restful service container, can handle access to `/streams` and `/queries`. Moreover, only the owner of a query $\gamma$ can start observing the results of $\gamma$ or is allowed to list all the observers (i.e., `GET`ting `/queries/`$\gamma$`/obsevers/`). However, investigating OAuth-based access-control is on our research agenda.

Last, but not least, we recognise that the transmission overhead of the proposed solution can reduce the processor throughput if the user frequently `POST`s RDF graphs containing only few triples. In our future work, we intent to explore the streaming of RDF triples in N-quads format on a Web-socket.

## References


1. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC. (2011) 370–388
2. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC. (2010) 96–111
3. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. (2011) 635–644
4. Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In: DEBS. (2012) 58–68
5. Rinne, M., Nuutila, E., Törmä, S.: Instans: High-performance event processing with standard rdf and sparql. In: ISWC (Posters & Demos). (2012)
6. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying rdf streams with c-sparql. SIGMOD Record **39**(1) (2010) 20–26
7. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/S-PARQL Benchmark. In: ISWC. (2012) 641–657
8. Le-Phuoc, D., Dao-Tran, M., Pham, M.D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: ISWC. (2012) 300–312
9. Balduini, M., Celino, I., Dell'Aglio, D., Valle, E.D., Huang, Y., il Lee, T.K., Kim, S.H., Tresp, V.: Bottari: An augmented reality mobile application to deliver personalized and location-based recommendations by continuous analysis of social media streams. J. Web Sem. **16** (2012) 33–41
10. Quoc, H.N.M., Serrano, M., Le-Phuoc, D., Hauswirth, M.: Super stream collider–linked stream mashups for everyone. In: Proceedings of the Semantic Web Challenge co-located with ISWC2012, Boston, MA, US (2012)
11. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: PODS, New York, New York, USA (2004) 263