# RDF Stream Processing: Let's *React*

Jean-Paul Calbimonte

Faculty of Computer Science and Communication Systems, EPFL, Switzerland.
jean-paul.calbimonte@epfl.ch

**Abstract.** Stream processing has recently gained a prominent role in Computer Science research. From networks or databases to information theory or programming languages, a lot of work has been dedicated to conceive ways of representing, transmitting, processing and understanding infinite sequences of data. Nevertheless, there are still aspects that need time to reach a mature state. In particular, heterogeneity in stream data management and event processing is both a challenging topic and a key enabler for the rising Web of Things, where smart devices continuously sense properties of the surrounding world. Different proposals on RDF and Linked Data streams have shown promising results for managing this type of data, while keeping explicit semantics on the data streams, and linking them to other datasets in a web-friendly way. With time, these efforts led to the emergence of initiatives such as the RDF Stream Processing (RSP) W3C community group, aiming at specifying a base RDF stream model and query language for that model. Although these works produced interest results in defining overarching model definitions, there are still multiple orthogonal challenges that need to be addressed. In this work we identify some of these challenges, and we link them to the characteristics of what are nowadays called *reactive* systems. This paradigm includes natively supporting event-driven asynchronous message passing, non-blocking data communication and processing through all layers, and on-demand flexible scalability. We argue that RDF stream systems, combined with reactive techniques can lead to powerful, resilient and interoperable systems at Web scale.

## 1 Introduction

Streams of data are one of the main sources of data today. The application domains where they play a capital role include mobile wearable sensors, internet of things, environmental monitoring or stock market analysis, to name just a few. All these streams of data, or infinite flows of information, already exist and are available in our streaming world [6], but they are of no use unless something or someone processes and makes sense out of them. The dynamicity, volume and velocity of these data make it challenging to effectively process, query and derive results from them. In the area of databases, these research challenges led to the emergence of data stream and complex event processing systems, including data models, query languages, algebra and operational semantics for them [1, 3].

Nevertheless, research opportunities in this area are far from being exhausted. The imminent realization of the Internet of Things and the abundance of new sources of streaming data raise a set of new challenges, especially dealing with the variety and heterogeneity of the data. Clear foundations are required to solve problems such as data integration and real-time analytics, added to the need for better understanding the meaning and the value of streaming data on the web. Several attempts have been made to approach some of these challenges using the theoretical foundations and the tools of Semantic Web research. These works have resulted in systems that tackle different issues, including continuous query processing [4, 9], stream reasoning [11], event detection [2], ontology maintenance [13] or ontology-based data access [5]. In all these heterogeneous works, a common pattern can be found, in the fact that they generally process streams of RDF data in some form. The RDF Stream Processing (RSP)[1] community that has been formed around these research initiatives, has gradually grown and started to also produce datasets, benchmarks, systems, and compare them in terms of features, performance, correctness, and other criteria. However, it is not clear if current RSPs are capable of meeting the real-life requirements of stream processing, and if they do, to what extent.

In this paper we provide an analysis of these requirements in the context of RSP systems, and we argue that for today's standards the concept of *reactivity* prevails and is a major driver for designing and implementing such systems. The remainder of the paper is structured as follows: first we explain the main concepts related to reactivity and the typical requirements of stream systems in Section 2. Then we identify and discuss the current issues and opportunities for designing reactive RSP systems in Section 3, before concluding in Section 4.

## 2 Reactivity in Stream Systems

There is not a single way of characterizing stream processing and the systems that implement it. Different views touch different angles of the same problem as in any other research topic. However, in the case that concerns us, the database community has explored and detailed the challenges and issues of stream processing in a systematic way. One of the key works in this direction is the one of Stonebraker et al. [12], which identifies 8 major requirements for such systems. These are summarized as the following rules:

1. Keep the data moving
2. Query using SQL on Streams
3. Handle stream imperfections
4. Generate predictable outcomes
5. Integrate stored and streaming data
6. Guarantee data safety and availability
7. Partition and scale automatically
8. Process and respond instantaneously

These requirements have helped shaping stream processing systems, not only in the area of databases but also in the semantic web community. However, stream

[1] `http://www.w3.org/community/rsp`

processing can go beyond these principles, and nowadays we notice that stream processing systems have become a necessity in a wider range of scenarios. For this reason, some of the rules described before (e.g. query using SQL) can be debated, as we experience that alternatives such as no-SQL querying are gaining wide use. What we can observe in stream processing nowadays is that the emphasis goes more and more in supporting the *reactivity* of the system as a whole. Reactivity refers not only to *real-time processing* but to a more comprehensive concept that can be summarized as the *ability of a system to react* to different conditions and stimuli. A commonly adopted definition of reactivity[2] identifies fours main characteristics of a reactive systems:

1. **Message-driven**. A system reacts to events, through message-passing communication between loose-coupled components.
2. **Elastic**. A systems reacts to dynamic and varying workload, adjusting the resources allocated for processing and adapting through distribution or replication.
3. **Resilient**. A system reacts to failure gracefully, maintaining service availability, ensuring recovery and localizing failures.
4. **Responsive**. A system reacts in a timely manner to users and requests, guaranteeing quality of service and overall usefulness and utility of the system output.

It is clear to see that these characteristics of reactive systems are related in many ways to the preceding requirements. In the specific case of RDF stream processing, we can evidence that most systems are still in their infancy and the available prototypes lack one or more of these characteristics, preventing their wider use in real-life scenarios and applications. As we will see, many of the pitfalls in RSP engines are due to a mismatch between traditional persistence RDF databases and streaming systems: the supporting programming and design patterns, techniques and paradigms for both of them have substantial differences. We argue that the *reactivity* traits can help us building systems that can help us using the appropriate tools for building RSP engines.

## 2.1 Message-driven Processing

Big data processing needs massive parallel and distributed processing. In stream processing this is also the case, so it is important to avoid models that go into the opposite direction. This includes shared mutable state between components that leads to multi-threaded non-deterministic stream processing. Another example to avoid is blocking operators. For example a synchronous call to read from a web stream of data can block an entire stream processing workflow, hurting the performance and responsiveness of the system. One way to avoid these problems is adopting event and message-based communication, which can be achieved in different ways. One of the most widely used paradigms for this is the *actor* model.

[2] The reactive manifesto `http://www.reactivemanifesto.org/`

Actors are lightweight objects that communicate through messages in an asynchronous manner, with no-shared mutable state between them (see Figure 1). This allows providing a loose-coupled architecture where blocking operators are avoided, as each actor is solely responsible of maintaining its local state, and communicate with other actors via messages in a mailbox. The behavior of an actor is defined by the way it reacts to asynchronous messages that arrive to its mailbox[3].

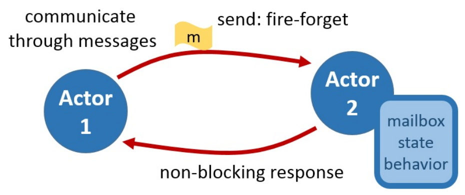

**Fig. 1:** Actor model: each actor communicates through asynchronous messages that arrive to its mailbox. Actors keep their own state and can modify their behavior upon arrival of new events.

## 2.2 Elasticity

Stream processing systems feed from multiple and heterogeneous sources, which can dynamically vary in terms of throughput and produce sudden load bursts. Processing these streams requires adaptive scalability and scheduling over a set of distributed computing units capable of reacting to theses continuous changes. Using the actor model is one of the possible ways of achieving this. Given their loose-coupled nature, actors can be deployed in one or many cores, or in an array of servers. Asynchronous message passing can occur either locally or remotely without changing the share-nothing overall design, as in Figure 2. This transparent remoting feature can be combined with routing policies, actor clustering and scheduling that gracefully adapts to workload variations.

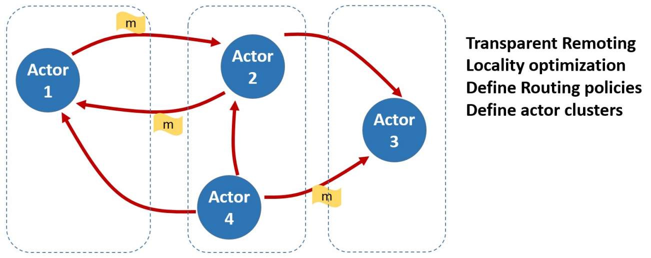

**Fig. 2:** Actors are distributed by nature, and can be deployed in one or many cores, or in different servers, without changing their communication and interaction model.

## 2.3 Resilient to Failures

Any system is exposed to failures, either internal or external. In our case, streams add the additional difficulty of dealing with extemporaneous events, delays, noisy data, and data loss. Again, there are several ways of dealing with failures. For example, in the case of the actor model, thanks to the decentralized nature

[3] An implementation example of this model is Akka:`http://akka.io`

of actors it is possible to isolate failures and avoid a complete degradation of the system. Actors allow defining hierarchies in which actors can supervise other children actors, and handle exceptions locally, as in Figure 3. Different strategies can be implemented depending on the type of failure and children actors can be set up to restart, resume, stop, escalate, etc. if needed. Moreover, the system can start new actors or re-schedule tasks to guarantee availability and response times during the system life-span.

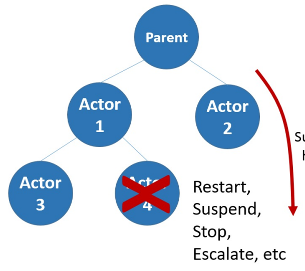

**Fig. 3:** Supervision in actor hierarchies. Different strategies canbe applied depending on the type of failure, but in general the faults are kept local.

### 2.4 Responsive Systems

Responsiveness in stream systems is capital and probably the number one requirement in the perspective of users. Real-time decisions and outcomes are expected in most use cases, and it is oftentimes admissible to loose in terms of precision and soundness in order to offer timely responses. It is clear to see that the previous three traits naturally help to achieve responsiveness in streams systems. Asynchronous communication and avoiding blocking operators is a key enabler for making data stream answers flow, but it requires to carefully look into all stacks of the system. If at any layer, a component is operating in a blocking fashion, all efforts at other levels are compromised and the system as a whole may not be responsive anymore. Elasticity is also important, as it allows providing flexible resource allocation and adaptive distributed and parallel processing. Finally, resiliency is key to localize failures, while keeping the system available even in the event of errors.

## 3 Reactive RSP: Issues and Opportunities

The reactive traits detailed above open a set of challenges for existing RSP engines. This translates into opportunities for embracing these ideas and applying them, so to design reactive RDF stream processors. In this section, we match and adapt the eight requirements of stream processing to existing challenges for RSP engines.

### 3.1 Event-driven RSP

Most, if not all of the existing RSP engines rely on tightly-coupled components in which RDF streams are implemented as mutable collections of objects. In many implementations, these objects are transmitted via notifications, usually following the observer pattern. For instance, consider the code in Listing 1. An RDF stream is both a data construct and a runnable object (something that executes on a thread), and is tightly attached to listeners that are notified each time something is *put* in the stream. This model clearly mixes data structures, execution and communication, in a way that makes it complicated to do scale, perform remote communication, distribute load, subscribe to stream events, etc.

```
public class SensorsStreamer extends RdfStream implements Runnable {
  public void run() {
    ..
    while(true){
      ...
      RdfQuadruple q=new RdfQuadruple(subject,predicate,object,
                                      System.currentTimeMillis());
      this.put(q);
    }
  }
}
```

**Listing 1:** Example of generation of an RDF stream in C-SPARQL. RDF quads are generated in the body of a thread executor.

A more suitable model for RDF streams can be designed based on events, transmitted as asynchronous messages between RSP actors that operate as producers and/or consumers. These streams should be immutable in nature, serializable and distributable over a network of RDF stream processors.

## 3.2 Extending SPARQL for streams

Several existing extensions of SPARQL have shown that it is possible and practical to query RDF streams using rich declarative languages [4, 5, 2]. These languages have shown to incorporate an interesting set of operators, and even challenge the non-RDF languages in tasks such as reasoning and data integration. This is mainly due to the fact that they rely on well defined semantic models, represented as vocabularies and ontologies. However, existing systems need to revisit the way in which they provide answers to these queries. For instance the code in Listing 2 shows a CQELS [9] query execution code excerpt, attaching a listener to an RSP query.

```
ExecContext context=new ExecContext(HOME, false);

String queryString =" SELECT ?person ?loc "
ContinuousSelect selQuery=context.registerSelect(queryString);
selQuery.register(new ContinuousListener()
{
  public void update(Mapping mapping){
    String result="";
    for(Iterator<Var> vars=mapping.vars();vars.hasNext();){
       result+=" "+context.engine().decode(mapping.get(vars.next()));
       System.out.println(result);
    }
  }
});
```

**Listing 2:** Example of generation of an RDF stream in CQELS.

Although this design allows obtaining updates on the query results (through a mapping that contains the bindings), it makes it complicated to distributed the results, respond asynchronously or remotely dispatch notifications to other components. RSP systems can rely on asynchronous message passing for delivering results and notify subscribers to a continuous query. Moreover, using dynamic push-pull mechanisms, subscribers can proactively specify their demand needs, and switch form pull to push mode and viceversa depending of the load (Figure 4).

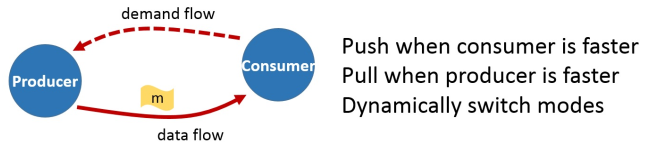

**Fig. 4:** Dynamic Pull-push. The consumer indicated its data demand so that the producer can push at a suitable rate. If the producer is too fast, then the data flow can switch to pull mode dynamically during the system lifespan.

## 3.3 Imperfect RDF Streams

Very few systems in the RSP scope deal with stream imperfections such as noise, out-of-order data and delays. While these issues are often described as out-of-scope or simply ignored, they are recurring in almost all real life streaming scenarios. This opens challenging questions, for instance regarding the RDF stream model used in most engines. These require timestamped data items, and assume that order among them is preserved at the time of arrival. Adapting this for out-of-order processing is an open question, as well as dealing with uncertainty in RSP continuous queries. Finally, in RSP engines all data items in a stream are considered to arrive instantaneously, and delays are not taking into account. Systems have to be ready to decide when to time-out, and how to deal with data delays, in order to prevent blocking the processing workflow.

## 3.4 Correctness in RSP

Recently, we have witnessed the emergence of RSP benchmarks that try to compare engines in terms of through put and query response time, among other criteria [14]. Nevertheless, it has been evidenced that these engines do not really behave in the same way, and throw different results to seemingly equivalent queries. This behavior has been studied and characterized in previous works [7], but it is still needed to adapt engines to consider these findings and be consistent and sound in query answering. Otherwise, benchmark results can be proven to be meaningless, and systems cannot really be reliable. To achieve this, it is needed to provide the theoretical foundations and models that describe precisely the operational semantics of an RSP system.

## 3.5 Stored and Streaming RDF

The RSP query languages already mentioned natively and elegantly support combining stored and streaming RDF (e.g. see Listing 3). Moreover, they allow in many cases to perform reasoning tasks based on persistent knowledge bases that contain domain knowledge expressed as an ontology. However, in most cases this knowledge is supposed to be static, and is often loaded once as a file, making it impractical to refresh data or update contents. Stored data should not be assumed to be static data, and on the other hand dynamic persistent data should not be confused with streaming data. In any case, to effectively combine the two, it is important to consider adaptive query operators and non-blocking access to persistent data stores, so to avoid creating bottlenecks on the stream of responses.

```
SELECT  ?person1 ?person2
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  GRAPH <http://deri.org/floorplan/>
    {?loc1 lv:connected ?loc2}
  STREAM <http://deri.org/streams/rfid> [NOW]
   {?person1 lv:detectedAt ?loc1}
}
```

**Listing 3:** Example of a CQELS query combining a stored RDF graph and and RDF stream.

## 3.6 Resilient RSP engines

Failures in RDF stream engines currently result in exception escalation and a general disruption in the processing pipeline. As we have seen, these engines are composed of tightly coupled components: e.g. streams attached to query listeners, and relying on synchronous communication between the query processor and the subscribers. For instance, if a stream generator thread fails, then its subscribers can fail as well. Or if the query engine fails, in many cases all registered queries are susceptible to fail too.

We can envision an RSP engine as a set of dynamic actors that are started, suspended or reused depending on the needs. These RSP actors can do different tasks, form stream acquisition to filtering, event processing, scheduling or data delivery. Using supervision strategies, failures can be handled locally, avoiding unwanted escalation or general service unavailability. Furthermore, at the processing level it is important to take into consideration the potential data bursts in the incoming streams. Load shedding and data eviction are possible alternatives that have started to be explored in RSP already [8].

## 3.7 Elastic and Scalable Processing

First steps towards parallel RSP systems have already been taken, although focusing on the deployment of RDF query processors on different nodes on a cluster [10]. There is still room for designing a more flexible RSP engine design that splits completely the query operators over a set of computing units, depending on the queries posed to the systems. Having RSP actors can allow realizing this vision, as these can communicate locally or remotely through asynchronous messages. Moreover, such an architecture can allow having different types of RSPs (query processors, reasoners, filtering units, etc.) in a workflow pipeline. Depending on the load and the number and difficulty of the queries, the system can see if it is necessary to adapt and use more or less nodes, and assign jobs to them. In specific systems, most notably for maintaining materialized ontologies, there have been attempts to engineer and optimize the computation by applying map-reduce parallelization techniques [13].

### 3.8 Responsive Systems

Avoiding blocking operators (e.g. choosing right type of joins, adaptively acquiring stored data, using asynchronous messaging) from end-to-end in an RSP is key to achieve responsiveness. In general, responsiveness will benefit from the other characteristics described in the previous sections. Event driven asynchronous communication within RSP actors, as well as avoiding blocking operators guarantees that the information flow is not stuck unnecessarily. In the same way, adaptive delivery of query results using dynamic push and pull, can prevent data bottlenecks and overflow. Also, by handling stream delays, data out of order and reacting gracefully to failures, the system can maintain availability, even under stress or non-ideal conditions. Similarly, elasticity can boost the system overall responsiveness by efficiently distributing the load and adapting to the dynamic conditions of the system.

## 4 Discussion

Streams are here to stay and it is our responsibility to design and build systems that cope with them in an effective and usable way. In this paper we have seen how the RDF Stream Processing community has been addressing some of the main challenges in this area, although in some cases there remains a lot to investigate and develop. With a varying degree of success we can see that most of the requirements for generic stream processors are handled by RSP engines, and that in some cases we even cope with more complicated scenarios that include data integration, heterogeneity, data interpretation and reasoning. Nevertheless, there are many pitfalls in systems design that prevent most of RSP engines to be *reactive*, in the sense that they do not always incorporate the traits of resilience, responsiveness, elasticity and message driven nature. We strongly believe that these principles have to be embraced at all levels of RDF stream processing. Finally, we proposed an actor-based model of RSP engines that can communicate asynchronously using immutable streams, and that can be deployed in local or remote instances under supervision strategies for failure handling. This model can be a starting point for achieving interoperability in RDF stream systems, where different type of stream processing tasks can be delegated to a specialized engine, as depicted in Figure 5.

We are confident that this type of systems will soon be realized, given the strong response of the RSP community to the challenges of stream processing, including standardization, serialization, agreement on processing semantics, etc.. Thsi will lead to a common understanding of what we can call reactive RDF stream processors.

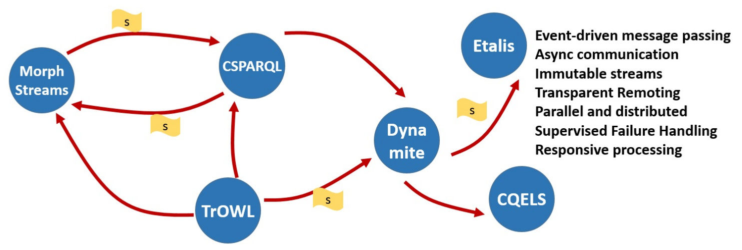

**Fig. 5:** Reactive RSP engines communicating through asynchronous messages in local or remote deployments, using immutable streams of RDF data.

**Acknowledgments** Partially supported by the SNSF-funded Nano-Tera OpenSense2 and D1namo projects[4].

## References

1. Aggarwal, C.C.: Data streams: models and algorithms, vol. 31. Springer (2007)
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW, pp. 635–644 (2011)
3. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford data stream management system. In: Data Stream Management: Processing High-Speed Data Streams. Springer-Verlag (2007)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: WWW, pp. 1061–1062 (2009)
5. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC, pp. 96–111 (2010)
6. Della Valle, E., Ceri, S., Harmelen, F.v., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems 24(6), 83–89 (2009)
7. Dell'Aglio, D., Calbimonte, J.P., Balduini, M., Corcho, O., Della Valle, E.: On correctness in rdf stream processor benchmarking. In: ISWC, pp. 326–342 (2013)
8. Gao, S., Scharrenbach, T., Bernstein, A.: The clock data-aware eviction approach: Towards processing linked data streams with limited resources. In: ESWC, pp. 6–20. Springer (2014)
9. 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, pp. 370–388 (2011)
10. Le-Phuoc, D., Quoc, H.N.M., Le Van, C., Hauswirth, M.: Elastic and scalable processing of linked stream data in the cloud. In: ISWC, pp. 280–297 (2013)
11. Ren, Y., Pan, J.Z.: Optimising ontology stream reasoning with truth maintenance system. In: CIKM, pp. 831–836 (2011)
12. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Record 34(4), 42–47 (2005)
13. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: Dynamite: Parallel materialization of dynamic rdf data. In: ISWC, pp. 657–672 (2013)
14. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC, pp. 641–657. Springer (2012)

[4] http://nano-tera.ch