Reactive Processing of RDF Streams of Events

Jean-Paul Calbimonte and Karl Aberer

Faculty of Computer Science and Communication Systems
EPFL, Switzerland.
firstname.lastname@epfl.ch

Abstract. Events on the Web are increasingly being produced in the form of data streams, and are present in many different scenarios and applications such as health monitoring, environmental sensing or social networks. The heterogeneity of event streams has raised the challenges of integrating, interpreting and processing them coherently. Semantic technologies have shown to provide both a formal and practical framework to address some of these challenges, producing standards for representation and querying, such as RDF and SPARQL. However, these standards are not suitable for dealing with streams for events, as they do not include the concpets of streaming and continuous processing. The idea of RDF stream processing (RSP) has emerged in recent years to fill this gap, and the research community has produced prototype engines that cover aspects including complex event processing and stream reasoning to varying degrees. However, these existing prototypes often overlook key principles of reactive systems, regarding the event-driven processing, responsiveness, resiliency and scalability. In this paper we present a reactive model for implementing RSP systems, based on the Actor model, which relies on asynchronous message passing of events. Furthermore, we study the responsiveness property of RSP systems, in particular for the delivery of streaming results.

1 Introduction

Processing streams of events is challenging task in a large number of systems in the Web. Events can encode different types of information at different levels, e.g. concerts, financial patterns, traffic events, sensor alerts, etc., generating large and dynamic volumes of streaming data. Needless to say, the diversity and the heterogeneity of the information that they produce would make it impossible to interpret and integrate these data, without the appropriate tools. Semantic Web standards such as RDF[1] and SPARQL[2] provide a way to address these challenges, and guidelines exist to produce and consume what we know as Linked Data. While these principles and standards have already gained a certain degree of maturity and adoption, they are not always suitable for dealing with data streams. The lack of order and time in RDF, and its stored and bounded characteristics contrast with the inherently dynamic and potentially infinite nature

[1] RDF 1.1 Primerhttp://www.w3.org/TR/rdf11-primer/
[2] SPARQL 1.1 http://www.w3.org/TR/sparql11-query/

of the time-ordered streams. Furthermore, SPARQL is governed by one-time semantics as opposed to the continuous semantics of a stream event processor. It is in this context that it is important to ask *How can streaming events can be modeled and queried in the Semantic Web?*. Several approaches have been proposed in the last years, advocating for extensions to RDF and SPARQL for querying streams of RDF events. Examples of these RDF stream processing (RSP) engines include C-SPARQL [4], SPARQL$_{stream}$ [6], EP-SPARQL [3] or CQELS [11], among others.

Although these extensions target different scenarios and have heterogeneous semantics, they share an important set of common features, e.g. similar RDF stream models, window operators and continuous queries. There is still no standard set of these extensions, but there is an ongoing effort to agree on them in the community [3]. The RSP prototypes that have been presented so far focus almost exclusively in the query evaluation and the different optimizations that can be applied to their algebra operators. However, the prototypes do not consider a broader scenario where RDF stream systems can reactively produce and consume RDF events asynchronously, and deliver continuous results dynamically, depending on the demands of the stream consumer.

In this paper we introduce a model that describes RSP producers and consumers, and that is adaptable to the specific case of RSP query processing. This model is based on the *Actor Model*, where lightweight objects interact exclusively by interchanging immutable messages. This model allows composing networks of RSP engines in such a way that they are composable, yet independent, and we show how this can be implemented using existing frameworks in the family of the JVM (Java Virtual Machine) languages. In particular, we focus on specifying how RSP query results can be delivered in scenarios where the stream producer is faster than the consumer, and takes into account its demand to push only the volumes of triples that can be handled by the other end. This dynamic push delivery can be convenient on scenarios where receivers have lower storage and processing capabilities, such as constrained devices and sensors in the IoT. The remainder of the paper is structured as follows: we briefly describe RSP systems and some of their limitations in Section 2, then we present the actor-based model on Section 3. We provide details of the dynamic push delivery on Section 4, and the implementation and experimentation are described in Section 5. We present the related work on Section 6 before concluding in Section 7.

2 RSP Engines, Producers and Consumers

In general RSP query engines can be informally described as follows: given as input a set of RDF streams and graphs, and a set of continuous queries, the RSP engine will produce a stream of continuous answers matching the queries (see Figure 1). This high-level model of an RSP engine is simple yet enough to describe most stream query processing scenarios. Nevertheless, this model,

[3] W3C RDF Stream Processing Community Group `http://www.w3.org/community/rsp`

and the existing implementations of it, does not detail how stream producers communicate with RSP engines, and how stream consumers receive results from RSP engines. This ambiguity or lack of specification has resulted in different implementations that may result in a number of issues, especially regarding responsiveness, elasticity and resiliency.

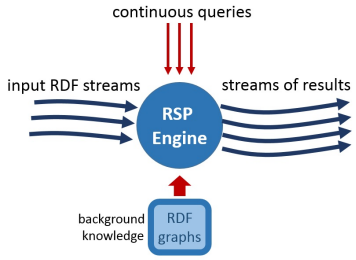


Fig. 1: Evaluation of continuous queries in RDF Stream Processing. The data stream flows through the engine, while continuous queries are registered and results that match them are streamed out.

2.1 RSP Query Engines

To illustrate these issues, let's consider first how streams are produced in these systems. On the producer side, RDF streams are entities to which the RSP engine subscribes, so that whenever a stream element is produced, the engine is notified (Figure 2). The issues with this model arise from the fact that the RSP engine and the stream producer are tightly coupled. In some cases like C-SPARQL or SPARQL$_{stream}$, the coupling is at the process level, i.e. both the producer and the engine coexist in the same application process. A first issue regards scalability: it is not possible to dynamically route the stream items from the producer to a different engine or array of engines, since the subscription is hard-wired on the code. Moreover, if the stream producer is faster than the RSP engine, the subscription notifications can flood the latter, potentially overflowing its capacity. A second issue is related to resilience: failures on the stream producer can escalate and directly affect or even halt the RSP engine.

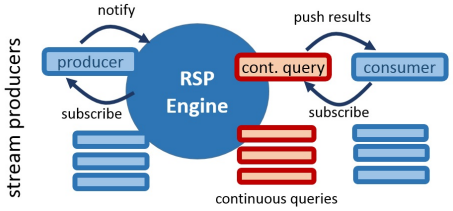


Fig. 2: Implementation of an RSP query engine based on tightly coupled publisher and subscribers.

Looking at the stream consumer side, the situation is similar. The continuous queries, typically implemented as SPARQL extensions, are registered into the RSP engine, acting as subscription systems. Then, for each of the continuous queries, a consumer can be attached so that it can receive notifications of the continuous answers to the queries (see Figure 2). Again, we face the problem

of tightly coupled publisher and subscribers that have fixed routing configuration and shared process space, which may hinder the scalability, elasticity and resiliency of the system. Added to that, the delivery mode of the query results is fixed and cannot be tuned to the needs of the consumer.

It is possible to see these issues in concrete implementations: for instance in Listing 1 the C-SPARQL code produces an RDF stream. Here, the stream data structure is mixed with the execution of the stream producer (through a dedicated thread). Even more important, the tightly coupled publishing is done when the RDF quad is made available through the `put` method. The engine (in this case acting as a consumer) is forced to receive quad-by-quad whenever the RDF Stream has new data.

```
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.

A similar scenario can be observed on query results recipient. The continuous listener code for the CQELS engine in Listing 2 represents a query registration (`ContinuousSelect`) to which one or more listeners can be attached. The subscription is tightly coupled, and results are pushed mapping by mapping, forcing the consumer to receive these updates and act accordingly.

```
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.

2.2 Results Delivery for Constrained Consumers

In the previous section we discussed some of the general issues of current RSP engines regarding producing and consuming RDF streams. Now we focus on the particular case where a stream consumer is not necessarily able to cope with the rate of the stream producer, and furthermore, when the stream generation rate fluctuates. As an example, consider the case of an RDF stream of annotated geo-located triples that mobile phones communicate to stationary sensors that detect proximity (e.g. for a social networking application, or for public transportation congestion studies), In this scenario the number of RDF stream producers can greatly vary (from a handful to thousands, depending on how many people are nearby in a certain time of the day), and also the stream rate can fluctuate.

In this and other examples the assumption that all consumers can handle any type of stream load does not always hold, and RSP engines need to consider this fact. Some approaches have used load shedding, eviction and discarding methods to alleviate the load, and could be applicable in these scenarios [1, 9]. Complementary to that, it should be possible for stream producers to regulate the rate and the number of items they dispatch to a consumer, depending on the data needs and demand of the latter.

3 An Actor Architecture for RDF Stream Processing

A central issue in the previous systems is that several aspects are mixed into a single implementation. An RDF stream in these systems encapsulates not only the stream data structure, but also its execution environment (threading model) and the way that data is delivered (subscriptions). In distributed systems, one of the most successful models for decentralized asynchronous programming is the *Actor model* [2, 10]. This paradigm introduces *actors*, lightweight objects that communicate through messages in an asynchronous manner, with no-shared mutable state between them. Each actor is responsible of managing its own state, which is not accessible by other actors. The only way for actors to interact is through asynchronous and immutable messages that they can send to each other either locally or remotely, as seen in figure 3.

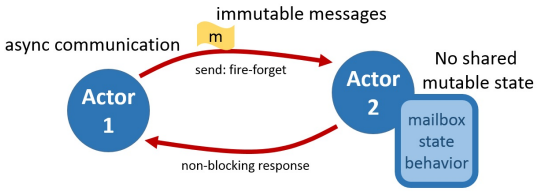


Fig. 3: Actor model: actors communicate through asynchronous messages that arrive to their mailboxes. There is no shared mutable state, as each actor handles its own state exclusively.

We can characterize an actor A as a tuple: $A = (s, b, mb)$, where s is the actor state, b is the actor behavior and mb is its message box. The state s is accessible and modifiable only by the actor itself, and no other Actor can either read or write on it. The mailbox mb is a queue of messages m_i that are received from other actors. Each message $m_i = (a_i^s, a_i^r, d_i)$ is composed of a data item d_i, a reference to the sender actor a_i^s, and a reference to the receiver actor a_i^r. The behavior is a function $b(m_i, s)$ where m_i is a message received through the mailbox. The behavior can change the actor state depending on the message acquired. Given a reference to an actor a, an actor can send a message m_i through the $send(m_i, a)$ operation. References to actors can be seen as addresses of an actor, which can be used for sending messages.

We propose a simple execution model for RDF stream processing that is composed of three generic types of actors: a stream producer, a processor and a consumer, as depicted in Figure 4. A producer actor generates and transmits messages that encapsulate RDF streams to the consumer actors. The processor actor is a special case that implements both a producer (producer of results)

and a consumer (consumes the input RDF streams), as well as some processing logic. Following the above definitions the data d_i of a message m_i emitted by a producer actor, or received by a consumer actor, is a set of timestamped triples. This model does not prevent these actors to receive and send also other types of messages.

In this model there is a clear separation of the data and the execution: the data stream is modeled as an infinite sequence of immutable event messages, each containing a set of RDF triples. Communication between producers and consumers is governed through asynchronous messaging that gets to the mailboxes of the actors. In that way, the subscribers are not tightly coupled with the producers of RDF streams, and in fact any consumer can feed from any stream generated by any producer. Moreover, this separation allows easily isolating failures in either end. Failures on consumers do not directly impact other consumers nor the producers, and vice-versa.

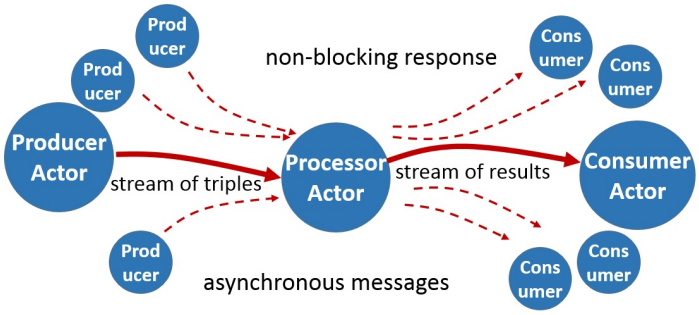


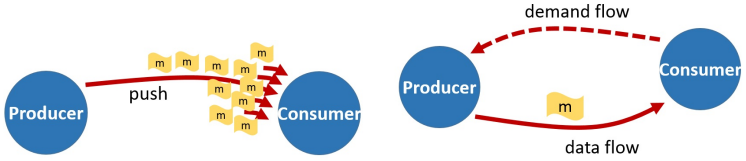
Fig. 4: RSP actors: RDF stream producers, processors and consumers. All actors send the stream elements as asynchronous messages. An RSP query engine is both a consumer (receives an input RDF stream) and a producer (produces a stream of continuous answers).

Event-driven asynchronous communication within RSP actors, as well as avoiding blocking operators, guarantees that the information flow is not stuck unnecessarily. Also, adaptive delivery of query results using dynamic push and pull, can prevent data bottlenecks and overflow, as we will see later. 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. The actor model results convenient for RDF stream processing, as it constitutes a basis for constructing what is commonly called a *reactive system*[4]. Reactive systems are characterized for being *event-driven*, *resilient*, *elastic*, and *responsive*.

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

4 Dynamic Push Delivery

In RSP engines there are typically two types of delivery modes for the stream of results associated to a continuous query: *pull* and *push*. In pull mode, the consumer actively requests the producer for more results, i.e. it has control of when the results are retrieved. While this mode has the advantage of guaranteeing that the consumer only receives the amount and rate of data that it needs, it may incur in delays that depend on the polling frequency. In the push mode, on the contrary, the producer pushes the data directly to the consumer, as soon as it is available. While this method can be more responsive and requires no active polling communication, it forces the consumer to deal with bursts of data, and potential message flooding. In some cases, when the consumer is *faster* than the producer, the push mode may be appropriate, but if the rate of messages exceeds the capacity of the consumer, then it may end up overloaded, causing system disruption, or requiring shedding or other techniques to deal with the problem (see Figure 5a).



(a) Push: overload if the producer pushes too fast.

(b) Dynamic push: demand on the side of the consumer.

Fig. 5: Delivery modes in RSP engines.

As an alternative, we propose using a dynamic push approach for delivering stream items to an RDF stream consumer, taking into consideration the capacity and availability of the latter (see Figure 5b). The dynamic mechanism consists in allowing the consumer to explicitly indicate its demand to the producer. This can be simply done by issuing a message that indicates the capacity (e.g. volume of data) that it can handle. Then, knowing the demand of the consumer, the stream producer can push only the volume of data that is required, thus avoiding any overload on the consumer side. If the demand is lower than the supply, then this behavior results in a normal push scenario. Otherwise, the consumer can ask for more data, i.e. pull, when it is ready to do so. Notice that the consumer can at any point in time notify about its demand. If the consumer is overloaded with processing tasks for a period of time, it can notify a low demand until it is free again, and only then raise it and let the producer know about it.

5 Implementing RSP Dynamic Push

In order to validate the proposed model, and more specifically, to verify the feasibility of the dynamic push in a RSP engine, we have implemented this mechanism on top of an open-source RSP query processor. We have used the Akka library[5], which is available for both Java and Scala, to implement our

[5] Akka: `http://akka.io/`

RSP Actors. Akka provides a fully fledged implementation of the actor model, including routing, serialization, state machine support, remoting and failover, among other features. By using the Akka library, we were able to create producer and consumer actors that receive messages, i.e. streams of triples. For example, a Scala snippet of a consumer is detailed in Listing 3, where we declare a consumer that extends the Akka Actor class, and implements a `receive` method. The receive method is executed when the actor receives a message on its mailbox, i.e. in our case an RDF stream item.

```
class RDFConsumer extends Actor {
  def receive ={
    case d:Data =>
      // process the triples in the data message
  }
}
```

Listing 3: Scala code snippet of an RDF consumer actor.

To show that an RSP engine can be adapted to the actor model, we have used CQELS, which is open source and is written in Java, as it has demonstrated to be one of the most competitive prototype implementations, at least in terms of performance [12]. More concretely, we have added the dynamic push delivery of CQELS query results, so that a consumer actor can be fed with the results of a CQELS continuous query.

To show the feasibility of our approach and the implementation of the dynamic push, we used a synthetic stream generator based on the data and vocabularies of the SRBench [15] benchmark for RDF stream processing engines. As a sample query, consider the CQELS query in Listing 4 that constructs a stream of triples consisting of an observation event and its observed timestamp, for the last second.

```
PREFIX omOwl: http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#.
CONSTRUCT {?observation <http://epfl.ch/stream/produces> ?time}
WHERE {
   STREAM <http://deri.org/streams/rfid> [RANGE 1000ms] {
     ?observation omOwl:timestamp ?time
   }
}
```

Listing 4: Example of generation of CQELS query over the SRBench dataset.

In the experiments, we focused on analyzing the processing throughput of the CQELS dynamic push, compared to the normal push operation. We tested using different processing latencies, i.e. considering that the processing on the consumer side can cause a delay of 10, 50, 100 and 500 milliseconds. This simulates a slow stream consumer, and we tested its behavior with different values for the fluctuating demand: e.g. from 5 to 10 thousand triples per execution. The results of these experiments are depicted in Figure 6, where each plot corresponds to a different delay value, the Y axis is the throughput, and the X axis is the demand of the consumer.

As it can be seen, when the demand of the consumer is high, the behavior is similar to the push mode. However if the consumer specifies a high demand

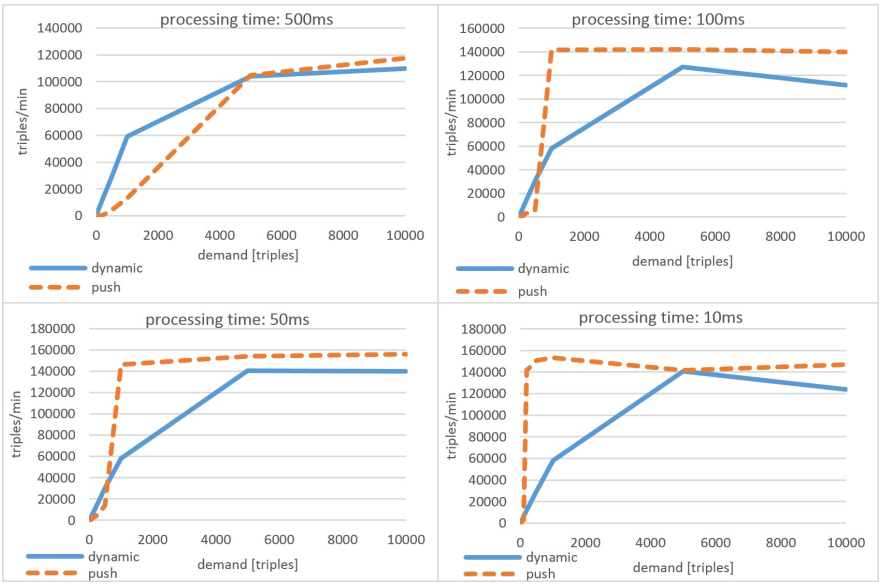


Fig. 6: Results of the experimentation: throughput of the results delivery after query processing for bot dynamic and normal push. The delay per processing execution is of 500, 100, 50, 10 milliseconds from left to right, top to bottom. The Y axis is the throughput, and the X axis is the demand.

but has a slow processing time, the throughput is slowly degraded. When the processing time is fast (e.g. 10 ms), the push delivery throughput is almost constant, as expected, although it is important to notice that in this mode, if the supply is greater than the demand, the system simply drops and does not process the exceeding items. In that regard, the dynamic push can help alleviate this problem, although it has a minor penalty in terms of throughput.

6 Related Work & Discussion

RDF stream processors have emerged in the latest years as a response to the challenge of producing, querying and consuming streams of RDF events. These efforts resulted in a series of implementation and approaches in this area, proposing their own set of stream models and extensions to SPARQL [5, 11, 7, 3, 9]. These and other RSP engines have focused on the execution of SPARQL streaming queries and the possible optimization and techniques that can be applied in that context. However, their models and implementation do not include details about the stream producers and consumers, resulting in prototypes that overlook the issues described in Section 2.

For handling continuous queries over streams, several Data Stream Management Systems (DSMS) have been designed and built in the past years, exploiting the power of continuous query languages and providing pull and push-based data access. Other systems, cataloged as complex event processors (CEP), emphasize

on pattern matching in query processing and defining complex events from basic ones through a series of operators [8]. Although none of the commercial CEP solutions provides semantically rich annotation capabilities on top of their query interfaces, systems as the ones dexfibed in [14, 13] have proposed different types of semantic processing models on top of CEPs.

More recently, a new sort of stream processing platforms has emerged, spinning off the massively parallel distributed Map-Reduce based frameworks. Examples of this include Storm[6] or Spark Streaming[7], which represent stream processing as workflows of operators that can be deployed in the cloud, hiding the complexity of parallel and remote communication. The actor based model can be complementary to such platforms (e.g. Spark Streaming allows feeding streams from Akka Actors on its core implementation).

7 Conclusions

Event streams are one of the most prevalent and ubiquitous source of Big Data on the web, and it is a key challenge to design and build systems that cope with them in an effective and usable way. In this paper we have seen how RDF Stream Processing engines can be adapted to work in an architecture that responds to the principles of reactive systems. this model is based on the usage of lightweight actors that communicate via asynchronous event messages. We have shown that using this paradigm we can avoid the tight coupled design of current RSP engines, while opening the way for building more resilient, responsive and elastic systems. More specifically, we have shown a technique for delivering the continuous results of queries in an RSP engine through a dynamic push that takes into consideration the demand of the stream consumer. The resulting prototype implementation, on top of the well known CQELS engine, shows that is feasible to adapt an RSP to include this mode, while keeping a good throughput.

When processing streams of data, whether they are under the RDF umbrella or not, it is important to take architectural decisions that guarantee that the system aligns with the characteristics of a reactive system. Otherwise, regardless of how performant a RSP engine is, if it is not able to be responsive, resilient to failures and scalable, it will not be able to match the challenges of streaming applications such as the Internet of Things. we have seen that 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 to be successful.

As future work, we plan to extend the reactive actor model to all aspects of an RSP engine, including the stream generation, linking with stored datasets and dealing with entailment regimes. We also envision to use this architecture to show that different and heterogeneous RSP engines can be combined together, forming

[6] http://storm.apache.org/

[7] https://spark.apache.org/streaming/

a network of producers and consumers that can communicate via messaging in a fully distributed scenario.

Acknowledgments Partially supported by the SNSF-funded Osper and NanoTera OpenSense2 projects.

References

1. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (August 2003)
2. Agha, G.: Actors: A model of concurrent computation in distributed systems. Tech. rep., MIT (1985)
3. 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)
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. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: Proc. 7th Extended Semantic Web Conference, pp. 1–15 (2010)
6. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC, pp. 96–111 (2010)
7. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. International Journal On Semantic Web and Information Systems (IJSWIS) 8(1), 43–63 (2012)
8. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Computing Surveys 44(3), 15:1–15:62 (2011)
9. 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)
10. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 11–20. ACM (2009)
11. 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)
12. Le-Phuoc, D., Nguyen-Mau, H.Q., Parreira, J.X., Hauswirth, M.: A middleware framework for scalable management of linked streams. Web Semantics: Science, Services and Agents on the World Wide Web 16, 42–51 (2012)
13. Paschke, A., Vincent, P., Alves, A., Moxey, C.: Tutorial on advanced design patterns in event processing. In: DEBS. pp. 324–334. ACM (2012)
14. Taylor, K., Leidinger, L.: Ontology-driven complex event processing in heterogeneous sensor networks. In: ISWC, pp. 285–299. Springer (2011)
15. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC, pp. 641–657. Springer (2012)