Published in: Knowledge-Based Systems journal.
This version of the article has been accepted for publication, after peer review and is subject to Elsevier terms of use.
The Version of Record is available online at: http://dx.doi.org/10.1016/j.knosys.2014.05.003

A Knowledge-Based Platform for Big Data Analytics Based on Publish/Subscribe Services and Stream Processing

Christian Esposito[a,1,*], Massimo Ficco[b,2], Francesco Palmieri[b,2], Aniello Castiglione[c,3]

[a]*Institute of High Performance Computing and Networking (ICAR), National Research Council, Via Pietro Castellino 111, I-80131 Napoli, Italy.*
[b]*Department of Industrial and Information Engineering, Second University of Naples, Via Roma 29, I-81031 Aversa (CE), Italy.*
[c]*Department of Computer Science, University of Salerno, Via Ponte don Melillo, I-84084 Fisciano (SA), Italy.*

Abstract

Big Data Analytics is considered an imperative aspect to be further improved in order to increase the operating margin of both public and private enterprises, and represents the next frontier for their innovation, competition, and productivity. Big Data are typically produced in different sectors of the above organizations, often geographically distributed throughout the world, and are characterized by a large size and variety. Therefore, there is a strong need for platforms handling larger and larger amounts of data in contexts characterized by complex event processing systems and multiple heterogeneous sources, dealing with the various issues related to efficiently disseminating, collecting and analyzing them in a fully distributed way.

In such scenario, this work proposes a way to overcome two fundamental issues: data heterogeneity and advanced processing capabilities. We present a knowledge-based solution for Big Data analytics, which consists in applying automatic schema mapping to face with data heterogeneity, as well as ontology extraction and semantic inference to support innovative processing. Such a solution, based on the publish/subscribe paradigm, has been evaluated within the context of a simple experimental proof of concept in order to determine its performance and effectiveness.

Keywords: Publish/Subscribe Services, Interoperability, Schema Matching, Semantic Search, Complex Event Processing, Big Data Analytics, Ontologies.

*Corresponding author.
Email addresses: christian.esposito@na.icar.cnr.it (Christian Esposito), massimo.ficco@unina2.it (Massimo Ficco), francesco.palmieri@unina.it (Francesco Palmieri), castiglione@ieee.org castiglione@acm.org (Aniello Castiglione)
[1]Christian Esposito is a fixed-term researcher at the Institute of High Performance Computing and Networking (ICAR), located in Napoli (Italy). Office Telephone Number: (+39) 081 6139508 - Fax Number: (+39) 081 6139531.
[2]Massimo Ficco and Francesco Palmieri are Assistant Professors at the DIII Department of the Second University of Naples (SUN), located in Aversa (Italy). Office Telephone Number: (+39) 081 5010505 - Fax Number: (+39) 081 5010203.
[3]Aniello Castiglione is Network and Security manager at the DIA Department of the University of Salerno. Office Telephone Number: (+39) 089 969594 - Fax Number: (+39) 089 969600.

1. Introduction

At the state of the art, large and complex ICT systems are designed by assuming a system of systems perspective, i.e., a large number of components integrated by means of middleware adapters/interfaces over a wide-area communication network. Such systems usually generate a large amount of loosely structured data sets, often known as Big Data, since they are characterized by a huge size and an high degree of complexity, that need to be effectively stored and processed [1, 2]. Some concrete examples can be taken from the application domains of environmental monitoring, intrusion/anomaly detection systems, healthcare management and online analysis of financial data, such as stock price trends. The analysis of such data sets is becoming vital for the success of a business or for the achievement of the ICT mission for the involved organizations. Therefore, there is the need for extremely efficient and flexible data analysis platforms to manage and process such data sets, sometimes on a on-line/timely basis. However, their huge size and variety are limiting the applicability of the traditional data mining approaches, which typically encompass a centralized collector, able to store and process data, that can become an unacceptable performance bottleneck. Consequently, the demand for a more distributed approach for the scalable and efficient management of Big Data is strongly increasing in the current business arena.

The well-known *MapReduce* paradigm [3] has attracted great interest, and is currently considered the winning-choice framework for large-scale data processing. Such a successful adoption both in industry and academia is motivated by its simplicity, scalability and fault-tolerance features, and further boosted by the availability of an open-source implementation offered by Apache and named Hadoop [4]. Despite such a great success and benefits, MapReduce exhibits several limitations, making it unsuitable for the overall spectrum of needs for large-scale data processing. In particular, as described in details in [5], the MapReduce paradigm is affected by several performance limitations, introducing high latency in data access and making it not suitable for interactive use. As a matter of fact, Hadoop is built on top of the Hadoop Distributed File System (HDFS), a distributed file system designed to run on commodity hardware, and more suitable for batch processing of very large amounts of data rather than for interactive applications. This makes the MapReduce paradigm unsuitable for event-based online Big Data processing architectures, and motivates the need of investigating other paradigms and novel platforms for large-scale event stream-driven analytics solutions.

Starting from these considerations, the main aim of this work is to design and realize a flexible architectural platform providing distributed mining solution for huge amounts of unstructured data within the context of complex event processing systems, allowing the easy integration of a large number of information sources geographically scattered throughout the world. Such unstructured data sources (such as Web clickstream data, social network activity logs, data transfer or phone calls records, flight tracking logs, etc.) usually do not fit into more traditional data warehousing and business intelligence techniques and tools and sometimes require timely correlation and processing triggered on specific event basis (e.g., in case of online analysis solicited by specific crisis conditions or emotional patterns). This implies the introduction new flexible integration paradigms, as well as knowledge-driven semantic inference features in data retrieval and processing to result in really effective business benefits. Publish/subscribe services [6, 7] have been proved to be a suitable and robust solution for the integration of a large number of heterogeneous entities thanks to their intrinsic asynchronous communication and decoupling properties. In fact, these properties remove the need of explicitly establishing all the dependencies among the interacting entities, in order to make the resulting virtualized communication infrastructure more scalable, flexible and maintainable. In addition, despite their inherently asynchronous nature, publish/subscribe services ensure timely interactions, characterized by low-latency message delivery features, between the corresponding parties, being also perfectly suitable in online event-driven data processing systems. Accordingly, we have designed our Big Data analytics architecture by building it on top of a publish/subscribe service stratum, serving as the communication facility used to exchange data among the involved components. Such a publish/subscribe service stratum brilliantly solves several interoperability issues due to the heterogeneity of the data to be handled in typical Big Data scenarios. In fact, most of the large-scale infrastructures that require Big Data analytics are rarely built ex-novo, but it is more probable that they are realized from the federation of already existing legacy systems, incrementally developed over the years by different companies in order to accomplish the customer needs known at the time of realization, without an organic evolution strategy. For this reason, the systems to be federated are characterized by a strong heterogeneity, that must be coped with by using abstraction mechanisms available on multiple layers [8, 9]. Therefore, such systems can be easily interconnected by means of publish/subscribe services, with the help of proper adapters and interfaces in order to overcome their heterogeneity and make them fully interoperable on a timely basis. We can distinguish the aforementioned heterogeneity both at the syntactic and semantic level. That is, each system is characterized by a given schema describing the data to be exchanged. Even in domains where proper standards have been issued and progressively imposed, the heterogeneity in the data schema is still seen as a problem. Such heterogeneity limits the possibility for applications to comprehend the messages received from a different system, and hence to interoperate. Specifically, publish/subscribe services use these data schemas to serialize and deserialize the data objects to be exchanged over the network. If the schema known by the destination is different than the one applied by the source, it is not possible to correctly deserialize the arrived message, with a consequent loss of information. Interoperability not only has to resolve the differences in data structures, but it also has to deal with semantic heterogeneity. Each single value composing the data to be exchanged can have a different definition and meaning on the interacting systems. Thus, we propose a knowledge-based enforcement for publish/subscribe services in order to address their limitations in supporting syntactic and semantic interoperability among heterogeneous entities. Our driving idea is to integrate schema matching approaches in the notification service, so that publishers and subscribers can have different data schemas and exchange events that are easy to be understood and processed.

In order to be processed online, in a fully distributed (and hence more scalable) way, Big Data are filtered, transformed and/or aggregated along the path from the producers to the consumers, to allow consumers to retrieve only what they are interested in, and not all the data generated by the producers. This allows avoiding the performance and dependability bottlenecks introduced by a centralized collecting and processing unit, and guarantees a considerable reduction of the processing latency as well as of the traffic imposed on the communication network (since processing is placed closer to the event producers), with considerable benefits in terms of network resource usage. For this purpose, we introduced on top of the publish/subscribe service an event stream processing layer [10], which considers data as an almost continuous stream of events. This event stream is generated by several producers and reaches its destinations by passing through a series of processing agents. These agents are able to apply a series of operations taken from the available complex event processing techniques portfolio [10] to filter parts of the stream, merge two or more distinct streams, perform queries over a stream and to persistently store streams. Hence, the first step of our work consisted in the definition and implementation of several primitive stream processing operators specialized as data processing agents and in the realization of a model-based prototype to assist Big Data analysts to easily create a stream processing infrastructure based on publish/subscribe services.

Furthermore, we also observed that traditional solutions for performing event stream processing are affected by two main problems limiting their applicability to Big Data analytics:

- *Stream Interoperability*, i.e., users are exposed to the heterogeneity in the structures of the different event streams of interest. In fact, users have to know the details of the event types in order to properly define query strings based on the stream structures, and to write different queries for streams whose structure varies;
- *Query Expressiveness*, i.e., events composing the streams are considered as a map of attributes and values, and the typical queries on event streams are structured as finding particular values in the events.

The construction of our platform on top of a publish/subscribe service model empowered with a knowledge-based solution for interoperability among heterogeneous event types allows us to easily resolve Stream Interoperability issues, leaving only the Query Expressiveness as an open problem. Recent research on event-driven systems, such as the works described in [11, 12], is speculating on the introduction of semantic inference in event processing (by realizing the so-called *Semantic Complex Event Processing* (SCEP) [13]), in order to obtain a knowledge-based detection of complex event patterns that goes beyond what is possible with current solutions. To this aim, we designed an agent that dynamically builds up a Resource Description Framework (RDF) [14] ontology, based on the incoming events, and applies queries expressed in the SPARQL query language [15] for semantic inference. Such dynamically-built ontology can be integrated with external knowledge, related to the domain or to the specific application within which the stream processing platform is used.

This article is structured as follows. In the next section, we provide a description of the fundamental problems addressed. Section 3 provides some background to support the proposal: in the first part it introduces publish/subscribe services, while in a second part it presents details on event stream processing. Section 4 describes in details the proposed solution for dealing with data heterogeneity and semantic inference in stream processing network infrastructures supporting Big Data analytics in a fully distributed scenario. Starting from the unsuitability of tree-based exchange formats, such as XML, we describe a knowledge-based solution to develop a flexible notification service. In addition, we show how it is possible to implement a stream processing network on top of a publish/subscribe service stratum. We conclude with details on how RDF ontologies can be dynamically built, and how SPARQL queries can be executed during event stream processing. Section 5 illustrates a proof-of-concept prototype used to assess our solution, as well as the outcomes of some performed experiments. Section 6 concludes the work by presenting some final remarks.

2. Problem Statement

The aim of this section is to describe in detail the two problems of data integration and semantic processing within the context of a platform for Big Data analytics.

2.1. Data Integration

Federating legacy systems, built by different companies at different times and under different regulation laws, requires the resolution of the interoperability issues imposed by the high potential heterogeneity among the systems belonging to a federated organization.

The first degree of heterogeneity is related to the programming environments and technologies used to realize the legacy systems to be federated. Nowadays, this heterogeneity is not felt as a research challenge anymore, but it is simply a matter of realizing the artifacts needed to bridge the different adopted technologies. The literature is rich of experiences on designing and implementing software for this technological interoperability, which have been summarized and formalized in the widely-known Enterprise Integration Patterns (EIP) [16], documenting the different communication ways according to which the systems are integrated and discussing how messages can be delivered from a sender to the correct receiver, by changing the information content of a message due to different data and information models and describing the behavior of messaging system end-points. As a practical example, let us consider two distinct systems, one implemented with CORBA and another with JBoss. In order to make them interoperable, i.e., the messages exchanged on CORBA are also received by destinations in the JBoss system and vice versa, a mediation/integration entity, such as a Messaging Bridge is needed, providing one instance for each system

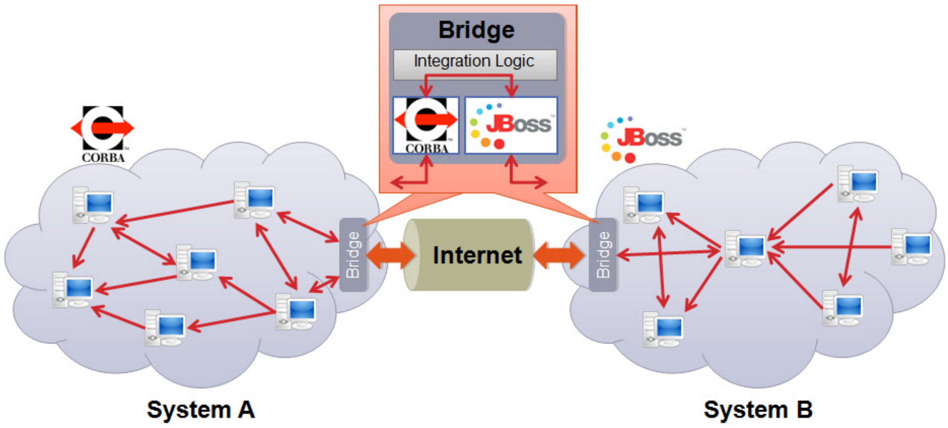


Figure 1: Bridging two heterogeneous systems

to be integrated. As clearly illustrated in Figure 1, such a component has a set of Channel Adapters, each in charge of sending and/or receiving messages on a particular platform (e.g., CORBA or JBoss), and an integration logic, responsible to map from one channel to the other ones by transforming the message format characterizing each communication channel into the other ones. For more details on integration issues and solutions, we refer interested readers to [17, 18, 19].

The second possible degree of heterogeneity is on the structural schema adopted for the exchanged data (referring to the organization of data in specific complex and simple data types), and raises the so-called *Data Exchange Problem* [20]. Specifically, let us consider an application, namely A_{source}, which is a data source characterized by a given schema, indicated as S_{source}, for the produced data, and another one, namely A_{dest}, which is a data destination and is characterized by another schema, indicated as S_{dest}. When the two schemas diverge, a communication can take place only if a mapping M between the two schemas exists. This allows the destination to understand the received message contents and to opportunely use them within its application logic. When the two schemas are equal, the mapping is simply the identity. On the contrary, when several heterogeneous legacy systems are federated, it is reasonable to have diverging data schemas, and the middleware solution used for the federation needs to find the mapping M and to adopt proper mechanisms to use it in the data dissemination process. Moreover, the communication pattern adopted in collaborative infrastructures is not one-to-one, but one-to-multi or multi-to-multi. So, during a single data dissemination operation, there is no single mapping M to be considered, but several of them, i.e., one per each destination. If we consider, for example, the position on the Earth of an aircraft for an Air Traffic Control framework, we can have different coordinate systems, all based on the concepts of latitude, longitude and altitude. A given system may measure latitude and longitude with a single number (i.e., expressing decimal degrees), or with triple numbers (i.e., expressing degrees, minutes and seconds). National institutions and/or standardization bodies have tried to find a solution to this problem by specifying a standard schema for the data exchanged in certain application domains. Let us consider two explicative examples, one in the context of aviation and the other from healthcare. EuroControl, the civil organization coordinating and planing air traffic control in Europe, has specified the structure of the flight data exchanged among Area Control Centers, called ATM Validation ENvironment for Use towards EATMS (AVENUE)[4]. Health Level Seven International (HL7), the global authority on standards for interoperability of health information technology, has issued a standardized application protocol for distributing clinical and administrative information among heterogeneous healthcare systems. These two standards have not resolved the syntactic heterogeneity in their domains of reference. In fact, a standard data format is likely to be changed over time for two main reasons. In the first case, it has to address novel issues by including more data. In fact, in the last three years AVENUE has been updated several times increasing the number of structures composing its schema. On the other hand, it has to evolve by adopting a different approach. In fact, there are two versions of the HL7 standard (i.e., HL7 version 3 and HL7 version 2.x), with the most recent one adopting an Object Oriented approach, and a well-defined mapping between them is missing [21]. Not all the systems may be upgraded to handle new versions, so systems with different versions have to co-exist. This brings back the Data Exchange Problem when a publisher produces events with a certain version of the standard data structure and subscribers can accept and comprehend only other versions.

Beyond the ability of two or more systems to be able to successfully exchange data, semantic interoperability is the ability to automatically interpret the data exchanged meaningfully and accurately as defined by the end users. For a concrete example, an integer value for the temperature of a transformer can have different interpretations, and cause different control decisions, if we consider it being expressed in Celsius, Fahrenheit or Kelvin degrees. Let us assume that, a control device receives a message from the temperature monitoring device, where the field $Temperature$ has the value 86. According to the IEEE C57.12.00-2000 standard [22], the transformer temperature should not exceed 65 Celsius degrees above ambient temperature when operated at its rated load (KVA), voltage (V), and frequency (Hz). Knowing that the ambient temperature is 20 Celsius degrees, if the value of $Temperature$ is expressed in Celsius degrees, the control device has to alarm system operators of a temperature limit violation in the transformer (i.e., the reported value of 86 Celsius degrees is greater than the threshold of 85 Celsius degrees). However, if it is expressed in another scale, such as Kelvin, an alarm should not be triggered, since the value of $Temperature$ is equal to -187.15 Celsius degrees. As a different example, in aviation the altitude can have several meanings, *e.g.*, (*i*) True altitude, i.e., the measure using the Mean Sea Level (MSL) as the reference datum; (*ii*) Absolute altitude or Height, i.e., the height of the aircraft above the terrain over which it is flying; or (*iii*) Flight Level, i.e., the value computed assuming an International standard sea-level pressure datum of 1013.25 hPa. Parties exchanging altitude information must be clear on which definition is being used.

[4] www.eurocontrol.int/eec/public/standard_page/ERS_avenue.html

2.2. Semantic Processing

Complex Event Processing (CEP) consists of collecting a series of data from multiple sources about what is currently happening in a certain system or environment (*i.e.*, events, and analyzing such events in a proper manner (*e.g.*, by looking for certain values or inferring specific event patterns) in order to detect the occurrence of certain situations or to generate new information by aggregating the available one. The data processing required by CEP is typically realized by writing computing rules based on the values exhibited by certain attributes of the received events. This may not be enough to spot the occurrence of complex critical situations, whose detection needs more data than the one carried by the exchanged events. Let us consider, for example, the following event raised by an aircraft:

```
--------------------------------------------------------------------
event AircraftState{
   ID = 01512fg5;
   Type = Boeing 757;
   Current_Position = (41 degrees, 54' North,  12 degrees, 27' East);
   Destination = Paris;
   Origin = Athens;
   Remaning_Fuel = 1000 gallons
}
--------------------------------------------------------------------
```

If we limit our analysis to the values assumed by the event attributes, we cannot notice that there is a serious trouble with this aircraft. In fact, the current position exposed by the aircraft is over the city of Rome. If we consider that a Boing 757 consumes 3 gallons of fuel per mile, then the total amount of fuel needed to cover the distance between Rome and Paris (i.e., about 687 miles) is equal to 2061 gallons, which is greater than the amount available in the aircraft. This simple example tells us that certain situations cannot be detected if we do not have the domain knowledge properly formalized and available to the processing agents in charge of analyzing the exchanged events. In our example, such a domain knowledge consists in the latitude and longitude of the main cities in Europe, and the fuel consumption of the main aircraft types flying in Europe.

Such a kind of semantic inference may be needed also when correlating events of different streams. Let us consider the case of the pilot noticing that the fuel is not suitable to reach its given destination, i.e., Paris, and asking for an authorization to land in a nearby local airfield, instead of a larger international one. Then, such a local airfield may publish this event:

```
---------------------------------------------------------------------
event LandingAuthorization{
   ID = P14254J;
   ICAO_ARC = 1;
   Position = (42 degrees, 25' North, 12 degrees, 6' East);
   Date = X Month 2013;
   Situation = Emergency }
---------------------------------------------------------------------
```

From this event it is not possible to assume that such an authorization is for the previous aircraft in a serious danger. However, if we infer that the position in the second event refers to the town of Viterbo (Italy), and we relate the fact that Viterbo is in the air sector of Rome, the same of the considered aircraft, we can infer that the pilot of the aircraft in danger decided to make an emergency landing in the local airfield of Viterbo. However, the minimum landing runaway length for a Boing 757 should be of 3,000 ft, but the airfield of Viterbo is classified as 1 in the ICAO Aircraft Reference Code (given to airports with a landing runaway with a length smaller than 800 meters). This allows the air traffic management system to rise an alert that the landing runaway is too small, and there is a high probability that an accident may occur. Such an alert can trigger the preparation for a rescue team to be ready at the place so as to provide assistance and save some lives.

These rather simple examples help to notice that the traditional data processing approaches, based on the values assumed by the event instances, are not sufficient to detect complex situations and/or obtain the advanced aggregate information needed by current applications. To this aim, such approaches have to be empowered by combining them with a semantic inference framework that can consider some knowledge on the domain and the semantics of the exchanged events.

3. Background

3.1. Publish/Subscribe Services

Publish/subscribe services are middleware solutions characterized by two types of processes: *Publishers*, which produce notifications, and/or *Subscribers*, which consume the notifications they are interested in, where such an interest is indicated by means of *Subscriptions* [6]. Specifically, there are different ways of specifying such an interest in a subset of the published events, which affect the architecture and algorithm adopted to implement the publish/subscribe platform. The most common one is indicated as *topic-based*, with publishers tagging outgoing notifications with a topic string, while subscribers use string matching to detect their events of interest. A different one is the *content-based*, where subscribers express complex predicates on the content of events, and only those events that satisfy such predicates are delivered.

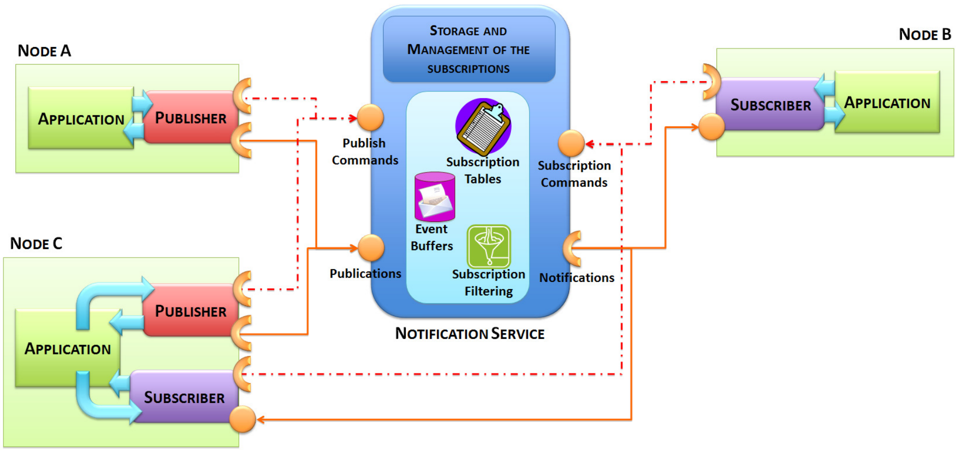


Figure 2: Schematic overview of a generic publish/subscribe service.

Subscriptions are not the only mechanism provided by publish/subscribe services to regulate event flows among applications. The basic architectural schema of this service also comprises the *Notification Service* (NS), as depicted in Figure 2, which plays the role of mediator between publishers and subscribers by giving strong decoupling features and offering the following functionalities: (*i*) storing subscriptions; (*ii*) receiving and buffering events; and (*iii*) dispatching received notifications to the interested subscribers. The NS is an abstraction that can be concretely implemented according to two main approaches: (*i*) Direct Delivery, i.e., each publisher also acts as a NS and takes the duty of delivering notifications to interested subscribers, or (*ii*) Indirect Delivery, i.e., the notification duties are shifted from the publishers to one or more networked brokers in order to improve scalability [23].

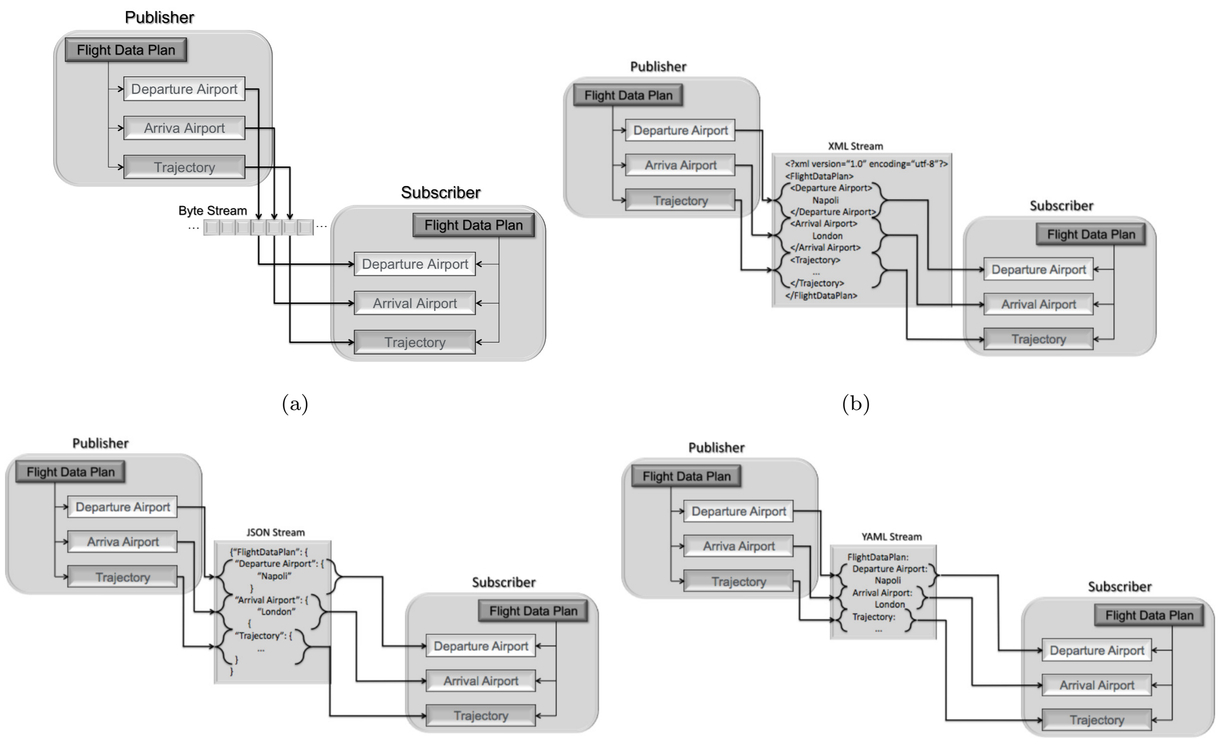
For our specific purposes, in this work we are interested in topic-based subscriptions with a broker-based implementation of the NS, but our findings can be easily adapted to any other kind of publish/subscribe services. We have decided to use topic-based publish/subscribe services, since they are the most popular within the industry community, and encompass all the main commercial products available on the market.

As depicted in Figure 2, NS provides two ways of interacting with publishers and two with subscribers. In the first case, NS can receive three kinds of commands from the publishers: (*i*) *create_topic* for the creation a new topic, (*ii*) *advertise* for informing that a publisher is about to send new events for a given existing topic; and (*iii*) *unadvertise* for informing that the given publisher is stopping to send new events for a given existing topic. NS can receive publications from the publishers, which are needed to be stored and forwarded to all interested subscribers. On the other hand, NS can receive two kinds of commands from the subscribers: (*i*) *subscribe* for informing NS which topics are of interest for the given subscriber, and (*ii*) *unsubscribe* for deleting a previous *subscribe* command and informing NS that the given subscriber is no more interested in receiving events associated to certain topics. The last interaction between subscribers and NS is the delivery of the notifications of interest. There are two different ways for subscribers to consume notifications from NS. Subscribers can wait and have NS push notifications to them, or they can continuously poll NS themselves to see if notifications are available.

3.2. Data Serialization Schemes

At the foundation of any middleware solution we find the serialization, and its dual operation named as deserialization. Respectively, the first operation takes an object as an input and returns a stream of bytes that can be conveyed by a network and delivered to a given destination, which performs the deserialization, i.e., it obtains the original object back from the received stream of bytes. A serialization format expresses the way to convert complex objects to sequences of bits. While some publish/subscribe services, as the ones compliant to the Java Message Service (JMS) specification, do not impose a particular format, leaving the decision to the application developers, other products, such as the ones compliant to the Object Management Group (OMG) Data Distribution Service (DDS) standard, use the Common Data Representation (CDR) [24]. They are based on a positional approach: serialization and relative deserialization operations are performed according to the position occupied by data within the byte stream (Figure 3(a)). Let us consider a concrete example with a publisher and subscriber exchanging a certain data instance. The publisher goes through all the fields of the given data instance, converts the content of each field in bytes, and sequentially stores it in a byte stream, treated as a FIFO queue. On the subscriber side, the application feeds data instances with information conveyed by received byte streams. Specifically, knowing that the serialization of the first field of type T requires a certain number n of bytes, the subscriber extracts the first n bytes from the byte stream. Then, it casts such n bytes in the proper type T and assigns the obtained value to the field in its own data instance. Such operation is repeated until the entire data instance is filled. This kind of serialization format has the drawback of weakening the decoupling property of publish/subscribe services, i.e., publishers and subscribers do not have to agree upon any detail of the communication. In fact, to be able to deserialize an object, the subscriber must be able to recreate the instance as it was when it was serialized, implying that the subscriber must have access to the same data schema, *i.e.* class, that was used to serialize the payload by the publisher. This introduces a strong coupling between publishers and subscribers, since they must share a common data schema for the exchanged notifications, which is unfeasible when integrating legacy systems in a complex scenario such as the Big Data Analytics one.

To cope with this concern, a viable solution is to achieve *flexible communication*: the publisher does not care about the schemas known by the subscribers, and subscribers are able to introspect the structure of the received notifications and use such information to properly feed data instances. Such a flexible communication is achievable by adopting a certain serialization format that embodies in the serialization stream not only the data content, as the binary formats, but also meta-information about its internal structure. XML is the most widely-known example of a flexible serialization format, where the structure of data is specified by a combination of opening and closing tags, as shown in Figure 3(b). In fact, the demand

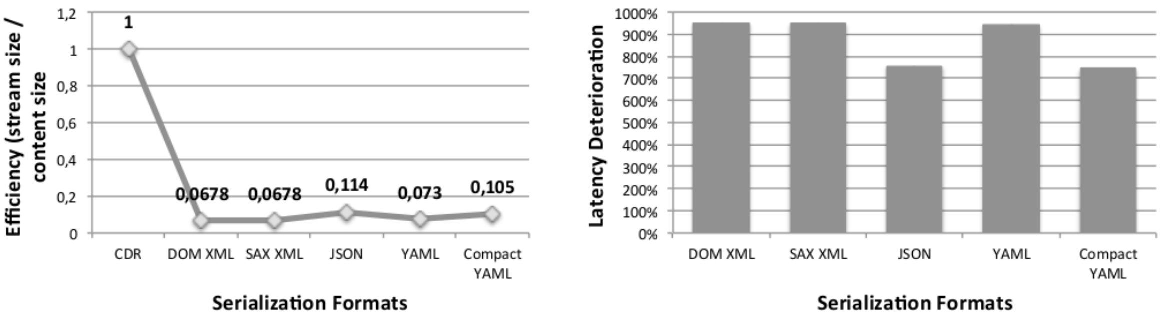


(a) (b) (c) (d)

Figure 3: Serialization and deserialization operated according to (*a*) CDR, (*b*) XML, (*c*) JSON and (*d*) YAML.

for flexible communications brought an increasing demand for XML-based pub/sub services, which support flexible document structures and subscription rules expressed by powerful languages, such as XPath and XQuery [25].

Unfortunately, XML syntax is redundant, and this redundancy may affect application efficiency through higher transmission and serialization costs. In fact, such solutions are affected by several performance problems, as studied in [26] and summarized in Figure 4. The redundant syntax of XML implies an increase of the bytes required to serialized a given event, with a consequent augment of the communication latency. In current literature, there are other formats available that exhibit a better balance between readability and compactness by being simpler than XML, while maintaining its flexibility guarantees. Java Script Object Notation (JSON) is a lightweight data-interchange format, based on a collection of name/value pairs and an ordered list of values, as illustrated in Figure 3(c), which allows saving bytes in the serialization stream [27]. In JSON, data is represented by two main structures: a collection of name/value pairs realized as an object, and an ordered list of values realized as an array. Objects start with a left brace and end with a right brace, and have a set of name/value pairs divided by commas, with names separated from the related values by a colon. Arrays begin with a left bracket and end with a right bracket, and contain only a set of values separated by commas. Values can be associated to native types, or even to user-defined data structures.



(a) (b)

Figure 4: Efficiency and performance of a given publish/subscribe service with a non flexible format, i.e., CDR, with flexible ones (taken from [26]).

YAML Ain't Markup Language (YAML) provides a data serialization format with data structure hierarchy maintained by outline indentation or a more compact version with brackets, shown in Figure 3(d) [28]. Data expressed in YAML is structured in data types common to most high-level languages: lists, associative arrays, and scalars, where the data structure is determined by means of line and whitespace delimiters. Names are separated from their relative values by colons.

We have proved in [26] that also such flexible formats are not suitable for the typical use cases in our specific environment, which exhibits considerable time constraints on the communication latency. In fact, as discussed for XML, the performance overhead caused by the use of flexible formats results high due to a larger number of bytes to send along the network, as proved in Figure 4(a), and could bring to the violations of the imposed time constraints.

The issue of jointly providing flexible communications and good performance is felt crucial for a more successful adoption of publish/subscribe services in complex integration projects such as the ones characterized by the need of handling Big Data. For this reason, the group responsible for managing the DDS standard within the OMG is actively discussing the topic of syntactic interoperability by issuing the OMG RFP mars/2008-05-03 on Extensible Topic Type System [29], with the intent of defining an abstract event schema system providing built-in support for extensibility and syntactic interoperability. The discussion within the OMG brought an addition to the OMG standard, named "Extensible and Dynamic Topic Types for DDS" [30], that specifies the structure of events, the formalism to express, such a structure, the protocol to serialize the events to have the byte stream conveyed by the network, and the API to define and manipulate data schema programmatically for dealing with syntactic heterogeneity. This new standard improved CDR in order to allow evolution of the event types. However, it supports only the addition of new fields, but is not suitable to treat the syntactical and semantic heterogeneity expressed in the previous section.

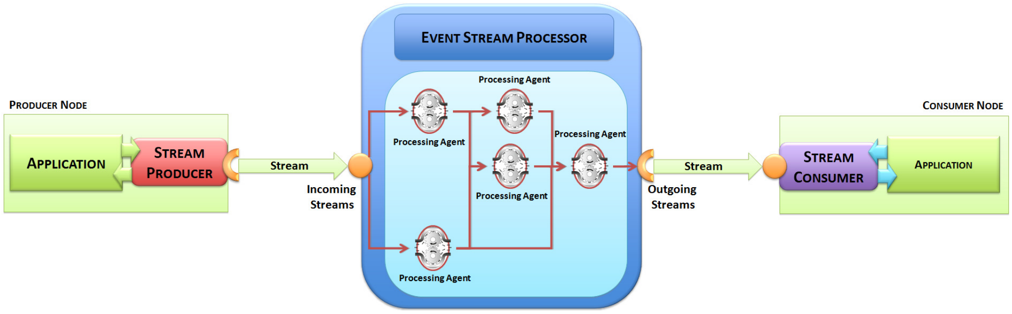


Figure 5: Schematic overview of a generic Event Stream Processing system.

3.3. Event Stream Processing

Publish/subscribe services have in our framework the only duty of distributing events from producers to consumers. On top of these services, it is possible to build an event processing facility, which has the goal of properly manipulating events, such as by means of filtering, correlating, transforming and/or aggregating them, to obtain new events and other useful derived information, according to the traditional Big Data analytics objectives. When events are seen as part of an almost-continuous stream, and operations consider such a stream-oriented perspective, then we talk about *event stream processing*. Figure 5 depicts a schematic overview of a generic solution for event stream processing: event producers and consumers are glued together by means of a Event Stream Processor (ESP), which is implemented by a network of processing agents, each applying a proper operation to the incoming streams to obtain the preferred outgoing streams. The behavior of each agent, in terms of operations to be applied to the incoming streams, can be specified in a proper event processing language [10], such as:

- *Trasforming Languages*, which indicate how to transform the incoming streams. These languages can be further classified in:
  - *Declarative Languages*, which consist in the extensions of the well-known SQL language from the Database field, and describe the outcome of the computation, rather than the exact flow of the execution of the needed operations to achieve such result. A concrete example is Event TrAnsaction Logic Inference System (ETALIS)[5] implemented in Prolog. Here's an example [31]:

```
-----------------------------------------------------------------
SELECT ?company WHERE
{ ?company hasStockPrice ?price1 }
SEQ { ?company hasStockPrice ?price2 }
SEQ { ?company hasStockPrice ?price3 }
FILTER ( ?price2 < ?price1 * 0.7 && ?price3 > ?price1 * 1.05
&& getDURATION() < "P30D"^^xsd:duration)
-----------------------------------------------------------------
```

[5]Event TrAnsaction Logic Inference System (ETALIS), available at: https://code.google.com/p/etalis/

Such a rule isolates the companies whose stock price has lowered by over 30%, and subsequently, increased by more than 5% within a time frame of 30 days.

– *Imperative Languages*, which describe the transforming rules as a proper combination of operators (i.e., implementing basic transformations on events) in series and/or in parallel. A concrete example is the Aurora's Stream Query Algebra (SQuAl) [32], and here's an example:

```
----------------------------------------------------------------------
Aggregate [Avg (Price),
Assuming Order(On Time, GroupBy Sid),
Size 1 hour,
Advance 1 hour]
----------------------------------------------------------------------
```

Such a rule computes a hourly average price (Price) per stock (Sid) over a stream of stock quotes that is known to be ordered by the time, the quote was issued (Time) by using the *Aggregate* and *Order* operators.

- *Pattern-based languages*, which specify a condition and an action to be triggered when the condition is verified. A concrete example is provided by the Tibco BusinessEvents, which provides event processing capabilities within the event-based platform by Tibco[6].

```
----------------------------------------------------------------------
rule Rules.FraudDetection {
  when {
    Temporal.History.howMany(account.Debits, DateTime.getTimeInMillis(
        DateTime.now())-FraudCriteria.interval, DateTime.getTimeInMillis(
        DateTime.now()), true) > FraudCriteria.num_txns;

    Temporal.Numeric.addAllHistoryDouble(account.Debits,
        DateTime.getTimeInMillis(DateTime.now())-FraudCriteria.interval)
        > FraudCriteria.debits_percent*account.AvgMonthlyBalance;

    account.Status!="Suspended"; }
  then {
    account.Status="Suspended"; }
}
----------------------------------------------------------------------
```

[6] Tibco BusinessEvents, available at: https://docs.tibco.com/products/tibco-businessevents-5-1-1

Such a rule analyzes the flow of events to detect a possible fraud by checking three conditions: (i) the number of debits in the verification interval is greater than a given threshold, (ii) the percentage of the average balance that was debited in the verification interval is greater than a given guard value, and (iii) the account is not yet suspected as being defrauded.

Another possible classification is based on the concept of state of the processing agent: in a stateless agent, the processing of an event is not influenced by the processing of past events; whereas, in a stateful agent, the processing of an event depends directly on the results of past processing.

Despite the differences among the available event processing languages, the common characteristic is to apply queries only on the value assumed by the attributes in the exchanged events, with no consideration of their semantics or any a-priori knowledge of the domain.

4. The Proposed Solution

While traditional online data mining/processing systems mainly assume that data has a unique schema and rigid semantics, when the volumes and heterogeneity of data sources and samples increase several inconsistencies may be introduced in data format, type or semantics by requiring the presence of new integration and semantic inference features when processing the data streams flowing from the involved sources.

4.1. Self-describing Data Schema and Automatic Schema Mapping

We exploited the literature on schema matching in the field of data management in order to obtain a solution that keeps on using CDR as the serialization format without its limitations in handling heterogeneity in the event types among publishers and subscribers. Specifically, we indicate with S and T respectively the event type of the publisher (i.e., the source schema), and the event type of the subscriber (i.e., the target schema). In particular, a publisher is able to publish notifications whose content is an instance of the S scheme; while a subscriber is only able to consume instances of T domain. Both these schemas must be expressed by using a proper formalism that allows an easily processing by a computer program. Specifically, we have used a tree-based formalism with two kinds of nodes, one for representing entities, which can be nested, and one for their attributes, holding certain values from a given domain. In addition, we indicate with s a series of assertions that map a given attribute in an event schema into an interval of bytes within the serialization stream. s is formalized by a series of tuples of four elements as follows:

$$s = \cup < \nu, \pi_{begin}, \pi_{end}, \Delta >, \tag{1}$$

where $\nu \in S$ is an attribute whose values belong to the domain Δ; while π_{begin} and π_{end} are respectively the position within the stream of the first and last byte of the value of ν. The operations of serialization, and deserialization are parametrized with s. In the first operation, the value of a certain attribute is converted in bytes and placed in the position of the serialization stream starting from π_{begin}. In the opposite operation, the bytes from π_{begin} to π_{end} within the stream are assigned to the proper attribute ν after being casted in the appropriate domain Δ. The problem of flexible communication between a publisher and a subscriber (or several subscribers) can be formalized as defining the tuple $< S, T, s, c >$, where c is a set of assertions that allows to match a node $\nu \in S$ with a node $\mu \in T$.

$$c = \cup < \nu, \mu > \Rightarrow \nu \approx \mu, \tag{2}$$

where the symbol $\approx$ means that node ν matches the node μ. The combination of the assertions in s with the ones in c allows overcoming the heterogeneity within the system: certain bytes from π_{begin} to π_{end} within the received stream are extracted, then the right attribute μ belonging to T is fetched by querying the assertions in c by looking at the one that defines a matching with the attribute ν, that corresponds to the range from π_{begin} to π_{end} in s. Accordingly, the fundamental challenges to be faced with are the following: (i) how to compute the assertions in c?, and (ii) how to resolve the schema matching problem without introducing a coupling between publishers and subscribers?

To handle the first question, we have adopted one of the major approaches to schema matching used in the literature [33], while for the second question, we have enhanced the functionalities provided by NS in order to mediate among heterogeneous schemas. Specifically, the publisher application has to behave according to Alg. 1. First of all, it sends the two commands *create_topic* and *advertise* to create the topic of interest and to notify its intention of starting to publish new events for this topic. Later, it obtains a representation of its source schema S, and the rules s to be used to serialize the content of the events to be published. Both the representation of S and the rules s are sent to NS, which returns an identification of the registered schema. Then, for each of the events to be published, it makes a notification by serializing the content of the event, and assigns the obtained schema identification to the notification. At this point, the notification is ready to be published. When all the events have been published, the publisher can deregister its schema and sends an *unadvertise* command. Alg. 2, on the other hand, illustrates the operations of a given subscriber interested in getting notified of events related to a given topic. First, the subscriber has to send a *subscribe* command to NS in order to communicate its interest in a given topic. Then, it obtains a representation for its target schema T and registers it on NS. As long as it is active, the subscriber receives a notification (no matters if in a push- or pull-based manner). From such notification it obtains the identification of the source schema. If it is the first time receiving an event with this schemaID (rows 8-11), then the subscriber interrogates NS and obtains the deserialization rules, which are stored in a proper hash table with the schemaID as the key. If the subscriber has already received other events with this schemaID (row 7), then the deserialization rules are fetched from the mentioned hash table. In both cases, the obtained rules are applied to deserialize the notification and to have the original event as an instance of

Algorithm 1 Publishing primitive, which takes as input the events to be disseminated within the context of a given topic

```
do_publisher(Events, Topic):
create_topic(Topic);
advertise(Topic);
schema = obtainSchema(Topic);
rules = obtainSerializationRules(schema);
schemaID = registerSchema&Rules(schema, rules);
for each Event in Events do
    Set Notification = serialize(Event, rules);
    Notification.schemaID = schemaID;
    publish(Notification);
end for
deregisterSchema(schemaID);
unadvertise(Topic);
```

Algorithm 2 Subscribing and consuming primitive, which takes as input the topic of which the subscriber is interested of get notified.

```
do_subscriber(Topic):
subscribe(Topic);
schema = obtainSchema(Topic);
registerSchema(schema);
while subscriber_is_alive do
    Notification = receive();
    schemaID = Notification.schemaID;
    Set rules = find(schemaID);
    if rules == null then
        rules = obtainSchema(schemaID, sub_reference);
        load(rules, schemaID);
    end if
    Set Event = Notification.deserialize(rules);
    consume(Event);
end while
unsubscribe(Topic);
```

the target schema so that the application can consume it. When the subscriber is deactivated, the command *unsubscribe* is sent. These two algorithms show how the publisher and the subscriber are able to manage instances in their own schema, without being coupled to know the schema of the other party.

The key role in our solution is played by NS, which mediates between the publisher and the subscriber by resolving the heterogeneity in their schemas. This resolution is performed when the subscriber asks the deserialization rules to NS. The actions executed by NS to returns the deserialization rules to the subscribe are described in Alg. 3. From the received schemaID, NS obtains the source schema representation and serialization rules that have been previously registered by a publisher (if these elements are not found, then NS cannot construct the deserialization rule and null is returned, as indicated in rows 2-4). Afterwards, NS obtains the target schema representation from the reference of the subscriber (if this is not available, then NS cannot proceed and null is returned, as indicated in rows 6-8). When all the pieces are in place, NS is able to map the entities in the source schema to the ones within the target schema (row 9). Then, it constructs the deserialization rules from the serialization ones (row 10), by copying all the rules and substituting in each of them, the attribute belonging to the source schema, with the mapped one within the target schema. The two mapped attributes might not share the same domain for their values, so the deserialization rules

Algorithm 3 Primitive executed by NS to return the deserialization rules based on a given source schema and serialization rules

```
obtainSchema(schemaID, sub_reference):
Set source_schema, ser_rules ≥ fetch(schemaID);
if source_schema == NULL then
    Return null;
end if
Set target_schema = fetch(sub_reference);
if target_schema == NULL then
    Return null;
end if
mapping = do_mapping(source_schema, target_schema);
deser_rules = do_matching(ser_rules, mapping);
Return deser_rules;
```

also contains the indications of the converter to obtain an element in the domain of the target attribute from an element in the domain of the source attribute. Such converters can be simple objects to cast an integer value to a double one, but they can also be more complex when converting a value in Kelvin degrees to one in Celsius or a value of True altitude in one of Absolute altitude.

The mapping among attributes in the two schemas is determined based on the concept of similarity, considering their semantic relations in the form of equivalence ($=$), less general ($\sqsubseteq$), more general ($\sqsupseteq$) and disjointness ($\perp$). Specifically, we assume that:

$$\forall \nu \in S, \mu \in T : \nu \approx \mu \quad iff \quad \nu = \mu \vee \nu \sqsubseteq \mu, \tag{3}$$

in other words, μ maps ν if they are equivalent or if ν is less general than μ (since all the values of ν will belong to the domain of μ). If ν is more general than μ, they cannot be mapped (since some values of ν might not belong to the domain of μ).

4.2. Semantic Inference in Event Stream Processing

A platform for stream processing is typically built on top of a messaging middleware in order to define streams flowing between the involved systems and convey the data composing them. As concrete examples let us consider the open-source project S4[7] and the platform by EsperTech[8]. The first one is based on the Apache

[7] http://incubator.apache.org/s4/

[8] EsperTech, available at: http://www.espertech.com/products/esperee.php

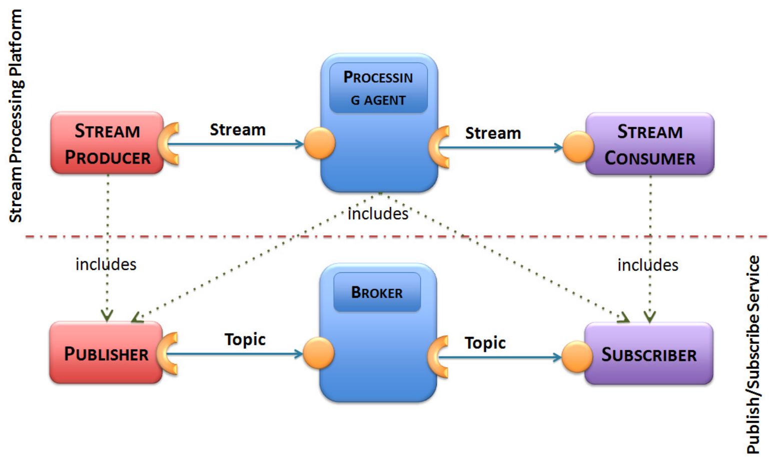


Figure 6: Mapping of the entities for a given stream processing platform with the ones of a publish/subscribe service.

ZooKeeper[9], a distributed, open-source coordination service for distributed applications; while, the second one is based on *Esper Data Distribution Services*, which is not related to the OMG DDS standard (as may be wrongly inferred by its name), but is built on a JMS-compliant product. Also in our case, we have built our platform on top of a messaging service. Specifically, we have used our empowered publish/subscribe service to deal with data heterogeneity. We have defined a mapping of the entities for a given stream processing platform with the ones of the publish/subscribe service, as depicted in Figure 6. As above mentioned, in a publish/subscribe service we have publishers and subscribers exchanging events through one or more brokers. We assume a topic-based subscription model, so that events have a topic string associated and subscribers subscribe to events by indicating their topic of interest. On the other hand, the stream processing platform is composed of stream producers and consumers interconnected by processing agents that manipulate the distributed streams. As shown in the figure, we directly map producers and consumers respectively with publishers and subscribers, in order to indicate an equivalence of a stream to a series of events with the same topic string associated. We do not map processing agents with brokers, but we assume them as applications with as many subscribers as the number of the incoming streams and as many publishers as the number of outgoing streams. Processing agents have their intelligence implemented in a stream engine that takes events from the subscribers of the agent, processes them by applying proper event operators and returns new events disseminated in an appropriate manner to the consumers through the agent publishers. Brokers can be employed to manage a large number of dependencies between producers, consumers and processing agents in order to improve the scalability in the communication among them.

For our specific purposes we have envisioned the processing agent as shown in Figure 6. The incoming streams are obtained from a series of subscribers, each interested to a topic characterizing an incoming

[9]ZooKeeper, available at: http://zookeeper.apache.org/

stream. The subscribers pass the received events composing the incoming streams to specific repositories, one for each incoming stream. When a new event is stored in a repository, the Complex Event Processing (CEP) Reasoner is executed to check if one of the registered predicates is verified and to obtain its result. The components described so far are typical of a traditional processing agent. In addition to them, we have designed the semantic counterpart side of our agent to perform semantic inference on the incoming streams. Specifically, the events obtained by the subscribers are also provided to a *Knowledge Extractor*, which has the role of extracting knowledge in terms of semantic metadata from the events of the incoming streams. Such a knowledge is stored in an appropriate repository with a given representation. Each time a new event is received, it is handled in both the *CEP Reasoner* and the *Semantic Reasoner*. Such a component takes as input the knowledge extracted from the received events and a proper domain knowledge (which has been provided to the agent by the administrator, which has collected some contributions from specialists and experts of the particular application domains of interest). The Semantic Reasoner evaluates the registered predicates and returns a result if it exists. A last component, called *Aggregator*, combines the results coming from the two reasoners in order to form one or more streams, which are distributed by proper publishers, one for each outgoing stream. The CEP Reasoner is not within the main scope of this work. However, in order to implement the presented solution, we adopted an open source event processing implementation based on Esper[10]. We mainly focused our interest on how to implement the side of the agent that makes semantic inference. Such a goal is faced by some of the papers published in the research community of distributed event-based systems, we adopted a different approach. Works in [34, 35] present ontologies for events by defining the set of common attributes they should have. Such a solution is too rigid for our context with high heterogeneity in the event definitions adopted in the different parts of a large-scale infrastructure. Also in the ENVISION project a semantic event processing has been proposed, and described in [36] where the chosen solution has been annotating events so as to allow their semantic enrichment and be able to conduct semantic inferences. In addition, a layered event ontology model has been presented: such a model comprises an Event-Observation ontology for describing the spatio-temporal occurrences of events, and a set of domain micro-ontologies (i.e., lightweight ontologies describing the concepts related to the domain). We share with this solution the usage of domain ontologies, but we do not consider useful a common ontology among events. Imposing a common ontology for exchanged events is considered as a failed attempt also by [13], which proposes to have events in the form of RDF triples and to model the semantic inferences as graph patterns to be verified on the RDF graphs of the received events. Also this solution is not applicable in our work, since it is not possible to have control on the stream producers in the different portions of a large-scale Big Data processing infrastructure and to impose such a representation of the produced events. However, we share with it the vision of semantic inference as patterns on RDF graphs.

[10]Esper, an open source event processing implementation, available at: http://esper.codehaus.org/

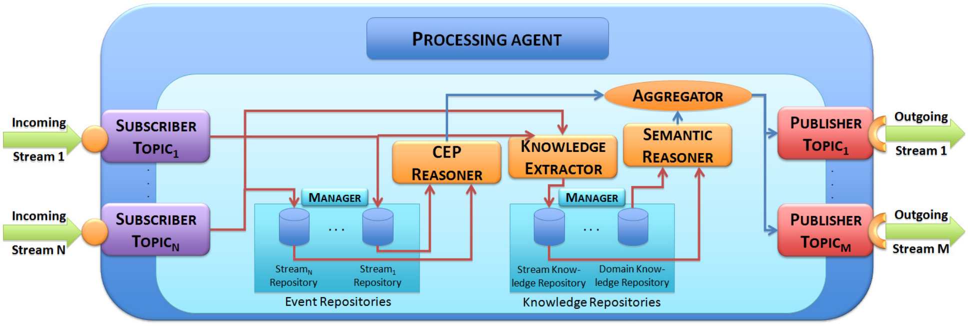


Figure 7: Internal architecture of the processing agent.

We preferred to extract ontologies written in RDF from the received events in order to have a knowledge more representative of the event content, without attempting of fitting the event content to a common ontology or imposing an RDF structure to the events. To this aim, we have followed an approach based on extracting an RDF-based ontology from the event type and relative instances received from the subscribers by the processing agent. Such an approach as been implemented within the above mentioned Knowledge Extractor, and formalized in Alg. 4, by drawing from the related experience on mapping relational databases and XML documents in RDF ontologies [37, 38]. Before describing our approach in detail, let us introduce some key concepts of RDF-based ontologies. RDF is a general-purpose language for representing a metadata data model of the information in the Web. Upon RDF, it is possible to build ontology languages as demonstrated by RDF Schema (RDFS) [14] or Web Ontology Language (OWL) [39]. In our work, we have described the ontologies in the processing agent by using RDFS to build a simple hierarchy of concepts and properties. An RDF class represents a set of individuals and models. RDF classes are structured in the hierarchy with relations of "subclass". To describe the attributes of these classes, proper properties can be defined. A last important entity is the predicate, which denotes a relationship between classes, or between a class and a data value. We directly mapped attributes and values carried out by events in these introduced RDF entities. Specifically, as Alg. 4 shows, when a new type of event is received for the first time, a new ontology is created (row 2), populated (rows 3-15), and stored in the Knowledge Repository (row 16). In particular, a new RDF class is created for the given topic name of the event (row 3); then, the structure of the complex attributes within the event type is mapped to a hierarchy of RDF classes rooted at the class named with the topic name. Specifically, for a given complex attribute (i.e., the one that contains no values but only other attributes), a new RDF class is created and defined as a sub-class of the class associated to the parent attribute (rows 6-8). In the case of simple attributes (i.e., the one containing a value of a given domain), a new RDF predicate is created, and its origin is the RDF class associated to the complex attribute containing the simple one (rows 12-13). Such new elements are inserted in the ontology (row 4, 9 and 14). If it is not the first time to receive an event belonging to the given topic, then the ontology is obtained from the repository (row 18). An RDF instance for the root class of the ontology is created with

Algorithm 4 Primitive executed by the Knowledge Extractor to obtain an RDF-based ontology from received event instances

obtainRDFOntology(topicName, event):

```
1: if !knowledgeRepository.exists(topicName) then
2:    ontology = new Ontology();
3:    class = ontology.createClass(topicName);
4:    ontology.insertClass(class);
5:    for all complexAttribute in event do
6:       class = ontology.createClass(complexAttribute.getName());
7:       parent = ontology.getClass(complexAttribute.getParent());
8:       class.associateParent(parent);
9:       ontology.insertClass(class);
10:   end for
11:   for all simpleAttribute in event do
12:      predicate = ontology.createPredicate(simpleAttribute.getName());
13:      parent = ontology.getClass(simpleAttribute.getParent());
14:      ontology.insertPredicate(parent, predicate);
15:   end for
16:   knowledgeRepository.insert(ontology);
17: end if
18: ontology = knowledgeRepository.obtain(topicName);
19: instance = ontology.createInstance(event.getKey(), topicName);
20: for all simpleAttribute in event do
21:    triple = ontology.createTriple();
22:    triple.setSubject(instance);
23:    triple.setPredicate(ontology.getPredicate(simpleAttribute.getName()));
24:    triple.setObject((simpleAttribute.getValue());
25:    ontology.setTriple(triple);
26: end for
```

the key of the event (i.e., the values that univocally identify the different instances of an event key). For each of the simple attributes an RDF triple is constructed, where the subject is always the newly created RDF instance (rows 21-22). The name of the simple attribute becomes the predicate of the triple (row 23); while, its value is the object (row 24). Also such triple is inserted in the ontology.

After the ontology has been created, it can be inferred by registering on the semantic reasoner's proper queries, which consist of inferring RDF data from a graph database perspective. In particular, we have considered the use of SPARQL [15], which is the well-known query language for RDF, as recommended by W3C. SPARQL is a graph-matching query language and can combine the information from different ontologies so as to complement the information carried out by the incoming events. A SPARQL query consists of several constituents: (*i*) a prefix (indicating the namespace for the used terms); (*ii*) a dataset definition clause (expressing where the data to be processed reside); (*iii*) a result clause (specifying the output to return to the user, such as how to construct new triples to be returned); (*iv*) pattern matching (such as optional, union, nesting, filtering); and (*v*) solution modifiers (such as, projection, distinct, order, limit, offset). There are three main query forms: (*i*) the SELECT form returns variable bindings, (*ii*) the CONSTRUCT form returns an RDF graph specified by a proper template, and (*iii*) the ASK form return a boolean value indicating the existence of a solution for a graph pattern. In our work we have used only SELECT forms. Let us consider the two examples at the end of Subsection 2.2: (1) how to find that the fuel in an enroute aircraft is not enough to reach a given destination, and (*ii*) how to detect that the aircraft is landing in an unsuitable airfield. We have noticed that these situations cannot be resolved with traditional CEP languages, since they are based only on the information carried out by the exchanged events. Let us consider three domain ontologies, one describing the positions of all the European cities, namely *cities*, one describing all the aircraft types, namely *airDesc*, and one describing the airfields in Europe, namely *fieldDesc*. Such ontology can be already existing in the Web or may be inserted in the agent by experts. Let us consider that the event AircraftState described in the mentioned Subsection arrives, and is converted in an RDF triples for the ontology $aircraftStates$. The SPARQL query that is able to resolve the example 1 is the following one:

```
---------------------------------------------------------------------
SELECT ?ID
FROM aircraftStates
WHERE {
        ?Type airDesc:consume ?fuelCons
        cities:i cities:hasName ?Destination;
                 cities:hasPosition ?destPos
        FILTER (?Fuel < ?fuelCons * (?destPos - ?CurrPos) )
}
---------------------------------------------------------------------
```

Such a query looks into the $aircraftStates$ ontology for the ID of the aircraft with not sufficient fuel based from the knowledge of how much fuel it consumers per miles, and the distance in miles between its current position and the destination. Let us consider now that, the event Landing Authorization described in the mentioned Subsection arrives, and is converted in an RDF triples for the ontology *landAutho*. The SPARQL query for the second example is the following one:

```
-----------------------------------------------------------------
SELECT ?ID
FROM aircraftStates
WHERE {
        ?ICAO_ARC fieldDesc:maxLenth ?maxLength
        ?Type airDesc:needLength ?needLength
        landAutho:i landAutho:hasPosition ?position
        FILTER (?needLength > ?maxLength
                && −5 < (?position − ?CurrPos) < 5)
}
-----------------------------------------------------------------
```

Such a query finds all the aircrafts that have a distance lower than 5 miles from an airfield, which exhibits a landing runaway shorter than the one required. Such examples prove that we are able to detect situations that current CEP languages fail to detect.

Finally, our agent should be able to handle a large number of incoming events; therefore, it should be equipped with proper mechanisms to remove half events from its repositories in order to make space to the new ones. To this aim, the queries that the two reasoners are able to process defines a time windows for the events to be considered when evaluating the queries. A practical example is provided by the example provided in Subsection 3.3 for the declarative language ETALIS. In this query, a temporal filter is considered in order to isolate data within a time frame of 30 days. The removal of stored events or RDF triples from the two repositories is handled through a proper Garbage Collection facility by the manager of each repository. Specifically, a manager counts per each stored event or RDF triple how many queries should consider it when evaluated. If this number reaches 0, then the event or RDF triple is removed from the repository. In addition, such repositories have a Lifespan Policy configured by an administrator, which indicates how long the data written in the repository is considered valid and should remain stored. These removal mechanisms prevent the processing agent to run out of its memory with invalid and outraged data.

5. Evaluation

In order to perform functional validation and performance evaluation of the proposed framework, we developed a simple proof-of-concept prototype system, supporting both the data integration and semantic inference capabilities described in the previous sections, with an emphasis on the use of currently available open source components.

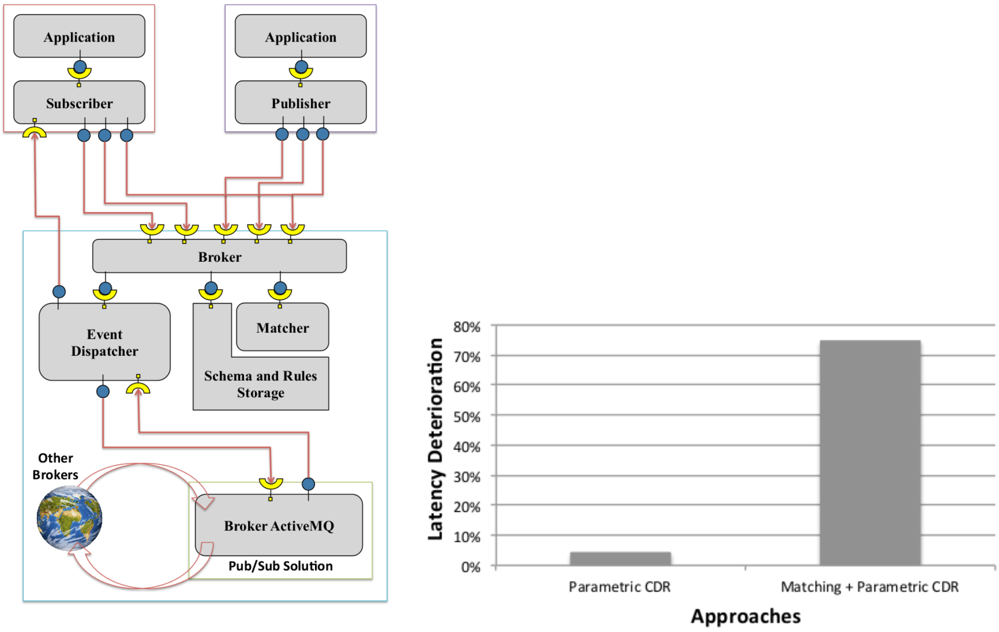
5.1. Data Integration

We have implemented the proposed data integration approach in a prototype based on a product compliant to the JMS standard, specifically Apache Active MQ[11], which provides a popular and powerful open source message broker. Apache Active MQ is adopted as the communication infrastructure to exchange notifications among the distributed brokers composing NS; but any other topic-based publish/subscribe solution can be easily adopted in our solution. On top of it, we have implemented the business logic of our broker, by realizing three components, as depicted in Figure 8(a): (1) an Event Dispatcher that takes as input a series of notifications and properly pass them to Apache Active MQ; (*ii*) a Schema and Rules Storage that receives schema representations and serialization rules in XML and stores them; (*iii*) and a Matcher that searches in the storage to obtain the schema and serialization rules, computes the mapping between the elements of the source and the target schema and returns the deserialization rules. We have used S-Match[12] as the semantic matching tool, but any other tools is integrable in our solution. In particular, we have used the original S-Match semantic matching algorithm [40], which is a rationalized re-implementation of the CTX match system [41]: (*i*) the labels of the nodes in the two schemes are extracted and translated into an internal language expressing concepts; (*ii*) all the possible semantic relations existing between any two concepts for the labels in the two trees are determined; (*iii*) the binding power of each identified semantic relation is tested; (*iv*) the relations with the strongest power are returned as matching rules to the user. The functionalities of our broker have been exposed as a Web service to the applications. Moreover, we have realized publisher and subscriber components that can be imported in the applications to Big Data event notification, which implements respectively Alg. 1 and Alg. 2. Last, we have implemented a component, named Parser, which takes data instances and returns, according to proper rules, byte streams to be sent along the network and vice versa.

We performed several experiments with the above prototype by relying on a large base of avionic data used in the Air Traffic Management (ATM) environment (collected within a month and made of 5 millions of flight plans), as well as on properly crafted test applications exchanging events that are structured according to the AVENUE type, which is characterized by a complex structure made of about 30 nested fields and a size of almost 100 KB. We have arbitrarily changed the schema at the subscribers by applying these variations: (*i*) removing/adding some parts of the AVENUE structures, (*ii*) changing the name of a structure with a synonym (we substituted 'Aircraft' with 'Airship' or 'Airplane'), (*iii*) changing the position of some structures, (*iv*) changing the domain of the values of some attributes (source schema used True altitude, whereas the target schema has Absolute altitude; distances in the source schema were in kilometers, changed in miles in the target schema). The experiments conducted for latency evaluation adopt

[11] Apache Active MQ, available at: http://activemq.apache.org.

[12] S-Match, available at: http://semanticmatching.org/s-match.html



(a) (b)

Figure 8: (*a*) Schematic overview of the prototype for the data integration. (*b*) Performance worsening comparison.

a "ping-pong" pattern. Specifically, the publisher feeds a data instance with randomly-generated content and passes it to the parser, which returns a byte stream that is passed to the broker constituting NS. On the subscriber side, the byte stream is received, deserialized and passed to the subscriber application. Then, the application immediately makes a copy of the received event and sends it back to the publisher, which receives the original event after the stream is passed through the parser component. The latency is computed as half of the time to perform the overall sequence of steps.

We obtained an efficiency of the serialization stream extremely close to 1, when compared with the CDR format. In addition to the event content, we have included a small number of bytes (e.g., we used only 10 bytes), representing the schemaID applied by the publisher. This efficiency allows our solution to obtain very similar performance to the optimal case when CDR is used. Specifically, as illustrated in Figure 8(b), we have to distinguish two cases: the first one is when the subscriber already has the deserialization rules, and the one in which the subscriber has to contact NS to obtain them. In the first case, the latency is 4% higher in average than the case with CDR, and this is the cost to pay for parameterizing the serialization and deserialization operations. In the second case, S-Match is able to match the two schema in about 1076 seconds, implying an increase of 75% of the overall latency. We have to remember that the second case occurs only once at the beginning of the conversation between the publisher and the subscriber, or in any circumstances when the source schema or the target one is changed. The precision of our solution strongly

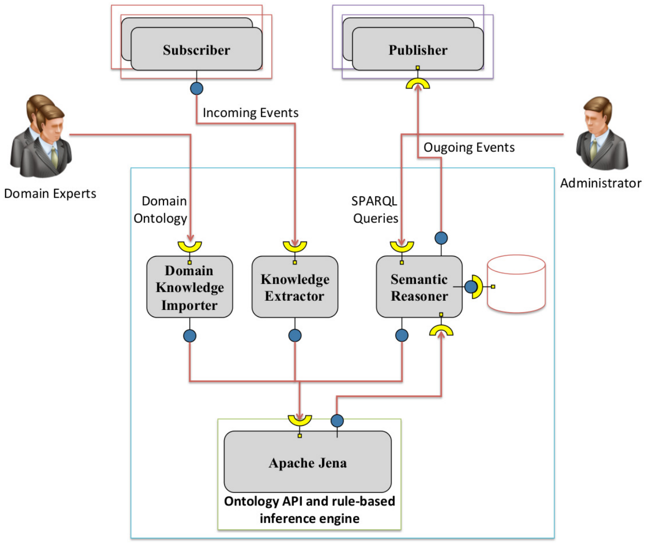


Figure 9: Schematic overview of the prototype for the semantic event stream processing.

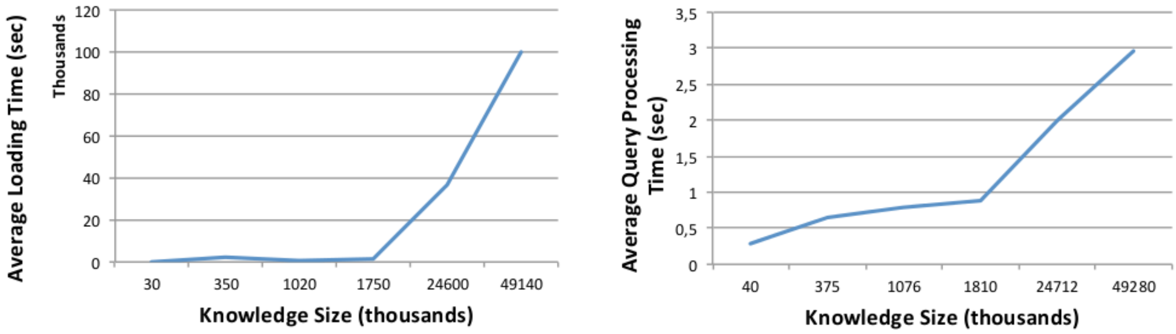
depends on the precision of the adopted schema matching tool. In our experiments S-Match was able to match all the attributes, so that the right data were in the right place at the instances of the target schema on the subscriber side.

Scalability is a fundamental challenge for Big Data analytics, and is straightforward to appreciate how the proposed architecture based on the publish/subscribe paradigm is able to seamlessly achieve scalability also in presence of very complex systems with thousands of data sources and processing sites.

5.2. Semantic Inference in Event Stream Processing

In order to test and evaluate the knowledge repository and semantic reasoner, we have implemented the processing agent by using Apache Jena[13]. Specifically, Apache Jena is an open-source project to provide a collection of tools and Java libraries for (*i*) reading, processing and writing RDF data in various formats; (*ii*) handling OWL and RDFS ontologies; (*iii*) reasoning with RDF and OWL data sources; (*iv*) allowing large numbers of RDF triples to be efficiently stored on disk; and (*v*) expressing and evaluating queries compliant with the latest SPARQL specification. Specifically, on top of Apache Jena, we have built three components, as depicted in Figure 9: (*i*) a Domain Knowledge Importer, which receives domain ontologies from the domain experts and loads them into Apache Jena repository; (*ii*) a Knowledge Extractor, which implements the Alg. 4, takes as input events from the agent subscribers and calls the proper operations of

[13] Apache Jena, available at: http://jena.apache.org/index.html



(a) (b)

Figure 10: $(a - b)$ Performance evaluation of the semantic reasoner.

the Apache Jena API to create ontologies; and (*iii*) a Semantic Reasoners, which has a knowledge repository populated by SPARQL queries coming from the agent administrator, as well as new events, by evaluating this queries on the ontologies hold by Apache Jena, and disseminates results through the agent publishers.

We have conducted some experiments in order to evaluate the performance of our semantic reasoner. Specifically, we have realized the domain ontologies mentioned in the previous section and provided to the reasoner the queries expressed in SPARQL in the previous Section. We have defined two incoming streams, one with events related to the aircraft states and another of the airfield authorizations. The reasoner was running on a workstation with Intel Core i7-2600K Quad CPU 3.40 GHz 8GB of RAM (only a single dedicated CPU is used), running Windows 7 Professional. At the beginning of the test, domain knowledge is provided to the reasoner and then a total of 100 events has been produced per each stream. We have measured two different performance indexes: (*i*) the time needed to load the domain knowledge within the semantic reasoner, and (*ii*) the time needed to perform a query on the incoming events. In the first case, we have varied the size (in terms of number of RDF triples) of the domain knowledge to be loaded, and from Figure 10(a) we can notice that by increasing the knowledge size we have an augmented loading time. Such a dependency on the knowledge size (i.e., the domain knowledge and the dynamically built ontologies) has also been presented in Figure 10(b) where the average query time is shown.

6. Conclusions

Considering the Emerging Technologies Hype Cycle in 2012 by Gartner [42], we notice that the area of Big Data is approaching its peak of expectation. This means that such a technology is not yet mature and there are several possible research challenges that have to be yet faced with. Specifically, there is a demand for a suitable platform to collect and process the data generated within the different heterogeneous systems/components of a large-scale infrastructure.

When integrating multiple systems to construct large-scale infrastructures by means of publish/subscribe services, it is possible to encounter into some troubles due to the different data schema known by the legacy systems to be fudged. This is a major issue, since it can affect the successful notification of events towards subscribers with different schema than the publishers. We have shown in this paper how syntactic and semantic heterogeneity in publish/subscribe services can be treated by embedding a schema matching mechanism within the NS. We have experimentally proved that our solution is able to make the publish/subscribe service completely flexible in terms of data representation schema, while containing the latency worsening caused by such a flexibility.

Furthermore, when considering the event exchange scenario on the resulting Big Data processing infrastructure we also observed that traditional event processing methods compute queries based on the values that certain attributes assume at the event instances received by the processing agents. We have illustrated with concrete examples that such an approach is not able to detect complex situations when domain knowledge is required. We have resolved such an issue by proposing a method to dynamically building ontologies based on the received events, and computing semantic inferences by combining such ontologies with properly formalized domain knowledge. Such mechanism, based on a properly crafted semantic reasoner has been experimentally evaluated, by observing a satisfactory performance and functional behavior. The presented technological framework may become the core of an open source solution supporting the analysis processing of huge data volumes across multiple heterogeneous systems scattered throughout the Internet.

References

[1] D. Zeng, R. Lusch, Big Data Analytics: Perspective Shifting from Transactions to Ecosystems, IEEE Intelligent Systems 28 (2) (2013) 2–5.

[2] P. Hunter, Journey to the centre of big data, Engineering & Technology 8 (3) (2013) 56–59.

[3] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[4] T. White , Hadoop: The Definitive Guide, Oreilly & Associates Inc; 3 edizione, 2012.

[5] C. Doulkeridis, K. Norvag, A survey of large-scale analytical query processing in mapreduce, The VLDB Journal (2013) 1–26.

[6] P. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, The many Faces of Publish/subscribe, ACM Computing Surveys (CSUR) 35 (2) (2003) 114–131.

[7] C. Esposito, D. Cotroneo, S. Russo, On reliability in publish/subscribe services, Computer Networks 57 (5) (2013) 1318–1343.

[8] F. Palmieri, S. Pardi, Towards a federated metropolitan area grid environment: The scope network-aware infrastructure, Future Generation Computer Systems 26 (8) (2010) 1241–1256.

[9] C. Esposito, M. Ficco, F. Palmieri, A. Castiglione, Interconnecting federated clouds by using publish-subscribe service, Cluster Computing 16 (4) (2013) 887–903.

[10] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Computing Surveys 44 (3) (2012) 15:1–15:62.

[11] K. Teymourian, O. Streibel, A. Paschke, R. Alnemr, C. Meinel, Towards Semantic Event-Driven Systems, Proceedings of the 3rd International Conference on New Technologies, Mobility and Security (NTMS) (2009) 1–6.

[12] F. Heintz, J. Kvarnström, P. Doherty, Stream Reasoning in DyKnow: A Knowledge Processing Middleware System, Proceedings of the Stream Reasoning Workshop. In series: CEUR Workshop Proceedings.

[13] K. Teymourian, O. Streibel, A. Paschke, R. Alnemr, C. Meinel, Towards Semantic Event-Driven Systems, Proceedings of the 3rd International Conference on New Technologies, Mobility and Security (NTMS) (2009) 1–6.

[14] D. Brickley, R.V.Guha, RDF Vocabulary Description Language 1.0: RDF Schema, accessed: July 2013. URL `www.w3.org/TR/rdf-schema/`

[15] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, accessed: July 2013. URL `www.w3.org/TR/rdf-sparql-query/`

[16] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional; 1 edition, 2003.

[17] W. Hasselbring, Information system integration, Communication of the ACM 43 (6) (2000) 32–38.

[18] N. Erasala, D. Yen, T. Rajkumar, Enterprise application integration in the electronic commerce world, Computer Standards & Interfaces 25 (2) (2003) 69–82.

[19] W. He, L. Xu, Integration of distributed enterprise applications: A survey, IEEE Transactions on Industrial Informatics 10 (1) (2014) 35–42.

[20] P. G. Kolaitis, Schema Mappings, Data Exchange, and Metadata Management, Proceedings of the ACM Symposium on Principles of Database Systems (PODS) (2005) 90–101.

[21] J. Gillson, Interoperability between heterogeneous healthcare information systems, accessed: April 2013. URL `www.slideshare.net/technopapa/interoperability-between-heterogeneous-healthcare-information-systems-by-john-gillson`

[22] I.-S. S. Board, IEEE Standard General Requirements for Liquid-Immersed Distribution, Power, and Regulating Transformers, IEEE Std C57.12.00-2000 (2000) i–53.

[23] R. Meier, V. Cahill, Taxonomy of Distributed Event-Based Programming Systems, The Computer Journal 48 (4) (2005) 602–626.

[24] Object Management Group, Common Object Request Broker Architecture (CORBA), v3.0, OMG Document (2002) 15.4–15.30.

[25] J. Zhao, D. Yang, J. Gao, T. Wang, An XML Publish/Subscribe Algorithm Implemented by Relational Operators, Lecture Notes in Computer Science, Advances in Data and Web Management 4505/2007 (2007) 305–316.

[26] C. Esposito, D. Cotroneo, S. Russo, An Investigation on Flexible Communications in Publish/Subscribe Services, Software Technologies for Embedded and Ubiquitous Systems - Lecture Notes in Computer Science 6399 (2011) 204–215.

[27] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), RFC4627, accessed: September 2010. URL `http://www.json.org/index.html`

[28] O. Ben-Kiki and C. Evans , YAML Ain't Markup Language, accessed: September 2010. URL `http://www.yaml.org`

[29] A. Corsaro, OMG RFP mars/2008-05-03, accessed: April 2013. URL `http://www.omg.org/cgi-bin/doc?mars/2008-05-03(restrictedaccess)`

[30] OMG, Extensible and Dynamic Topic Types for DDS, accessed: April 2013. URL `http://portals.omg.org/dds/sites/default/files/x-types_ptc_11-03-11.pdf`

[31] D. Anicic, P. Fodor, S. Rudolph, N. Stojanovic, EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning, Proceedings of the Twentieth International World Wide Web Conference.

[32] D. A. et al., Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2).

[33] E. Rahm, P. Bernstein, A survey of approaches to automatic schema matching, The VLDB Journal 10 (4) (2001) 334–350.

[34] J. Aasman, Unification of geospatial reasoning, temporal logic, & social network analysis in event-based systems, Proceedings of the second international conference on Distributed event-based systems (DEBS) (2008) 139–145.

[35] S. Abdallah, Y. Raimond, The event ontology, accessed: July 2013.
URL motools.sourceforge.net/event/event.html

[36] A. Llaves, H. Michels, P. Maué, M. Roth, Semantic event processing in ENVISION, Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics (WIMS) (2012) 25:1–25:9.

[37] S. S. et al., A Survey of Current Approaches for Mapping of Relational Databases to RDF, accessed: July 2013.
URL www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport_01082009.pdf

[38] P. T. Thuy, Y.-K. Lee, S. Lee, XSD2RDFS and XML2RDF Transformation: a Semantic Approach, Proceedings of the 2nd International Conference on Emerging Databases (EDB).

[39] D. McGuinness, F. van Harmelen, OWL Web Ontology Language - Overview, accessed: July 2013.
URL www.w3.org/TR/owl-features/

[40] F. Giunchiglia, P. Shvaiko, M. Yatskevich, S-Match: an Algorithm and an Implementation of Semantic Matching, The Semantic Web: Research and Applications, Lecture Notes in Computer Science 3053 (2004) 61–75.

[41] P. Bouquet, L. Serafini, S. Zanobini, Semantic coordination: A new approach and an application, The Semantic Web - ISWC 2003, Lecture Notes in Computer Science 2870 (2003) 130–145.

[42] Gartner, Hype Cycles 2012, accessed: July 2013.
URL www.gartner.com/technology/research/hype-cycles/