





Article

Dynamic Linked Data: A SPARQL Event Processing Architecture

Luca Roffia ^{1,*} , Paolo Azzoni ², Cristiano Aguzzi ¹ , Fabio Viola ¹ , Francesco Antoniazzi ¹ 
and Tullio Salmon Cinotti ^{1,3}

¹ Department of Computer Science and Engineering (DISI), University of Bologna, I-40126 Bologna, Italy; cristiano.aguzzi@unibo.it (C.A.); fabio.viola@unibo.it (F.V.); francesco.antoniazzi@unibo.it (F.A.); tullio.salmoncinotti@unibo.it (T.S.C.)

² Eurotech S.p.a., Via Fratelli Solari 3/a, 33020 Amaro (Udine), Italy; paolo.azzoni@eurotech.com

³ Advanced Research Center on Electronic Systems “Erocole De Castro” (ARCES), University of Bologna, I-40126 Bologna, Italy

* Correspondence: luca.roffia@unibo.it; Tel.: +39-051-209-5423

Received: 12 February 2018; Accepted: 16 April 2018; Published: 20 April 2018



Abstract: This paper presents a decentralized Web-based architecture designed to support the development of distributed, dynamic, context-aware and interoperable services and applications. The architecture enables the detection and notification of changes over the Web of Data by means of a content-based publish-subscribe mechanism where the W3C SPARQL 1.1 Update and Query languages are fully supported and used respectively by publishers and subscribers. The architecture is built on top of the W3C SPARQL 1.1 Protocol and introduces the SPARQL 1.1 Secure Event protocol and the SPARQL 1.1 Subscribe Language as a means for conveying and expressing subscription requests and notifications. The reference implementation of the architecture offers to developers a design pattern for a modular, scalable and effective application development.

Keywords: dynamic Linked Data; publish-subscribe; Semantic Web; SPARQL; event processing; protocols; distributed Web applications; interoperability; security

1. Introduction

In the last couple of decades, the World Wide Web, initially conceived of to share human-friendly information, has been growing into a new form: the Semantic Web [1], intended as a global Web of Data, which can be interpreted and processed directly by digital machines. In 2006, Tim Berners-Lee published a document on Linked Data claiming to be “the unexpected re-use of information the value added by the Web” (Tim Berners-Lee, Linked Data, 2006, <https://www.w3.org/DesignIssues/LinkedData.html>) and providing a set of rules on how to publish data on the Web (such as URI (<https://tools.ietf.org/html/rfc3986>), HTTP (<https://tools.ietf.org/html/rfc2616>), RDF (<https://www.w3.org/TR/rdf11-concepts/>) and SPARQL (<https://www.w3.org/TR/sparql11-overview/>)). In 2009, the Linked Data concept and relative technologies, along with an assessment of the Web of Data evolution, have been further described in [2]. By that time, research efforts on developing Linked Data have been made in several directions, including standards (e.g., SPARQL 1.1 Protocol (<https://www.w3.org/TR/sparql11-protocol/>), SPARQL 1.1 Query language (<https://www.w3.org/TR/sparql11-query/>), SPARQL 1.1 Update language (<https://www.w3.org/TR/sparql11-update/>), JSON-LD (<https://json-ld.org/spec/latest/json-ld/>), Linked Data Platform 1.0 (<https://www.w3.org/TR/ldp/>)), ontologies and vocabularies (e.g., OWL (<https://www.w3.org/TR/owl2-primer/>), CIDOC-CRM (<http://www.cidoc-crm.org/Version/version-6.2.1>), W3C OWL Time (<https://www.w3.org/TR/owl-time/>), Semantic Sensor Network Ontology (<https://www.w3.org/TR/vocab-ssn/>), schema.org

(<http://schema.org/>), RDF Data Cube Vocabulary (<https://www.w3.org/TR/vocab-data-cube/>), just to mention a few), RDF stores and SPARQL endpoints (e.g., Jena (<https://jena.apache.org/index.html>), Fuseki (<https://jena.apache.org/documentation/fuseki2/>), Virtuoso (<https://virtuoso.openlinksw.com/>), Blazegraph (<https://www.blazegraph.com/>), Stardog (<https://www.stardog.com/>) and Amazon Neptune (<https://aws.amazon.com/neptune/>)), tools (e.g., for ontologies editing, like Protegè (<https://protege.stanford.edu/>) or RDF graphs visualization, like LodLive (<http://en.lodlive.it/>) and Visual Data Web (<http://www.visualdataweb.org/index.php>)) and Semantic Web portals (e.g., DBpedia (<http://wiki.dbpedia.org/>), Wikidata (https://www.wikidata.org/wiki/Wikidata:Main_Page)).

The adoption of Semantic Web technologies follows two main trends: the first tries to exploit their potential to create autonomous systems, in its widest meaning, while the second focuses on solutions that take advantage of the flexible RDF data model. The use of Semantic Web technologies in autonomous systems enables machines to generate, publish and consume new information autonomously (e.g., from simple autonomous systems adopted in IoT applications, to complex systems for the autonomous analysis of a huge amount of data that would be impossible for a human operator). At the same time, the adoption of a flexible data model is intended to support the applications where it is difficult to define the knowledge-base and its model once and for ever. In fact, the adoption of a static model is often not reasonable, because information is living and evolves in time. While relational databases intrinsically do not provide this level of flexibility, the flexibility offered by RDF ensures that the unanticipated and unexpected evolution of the knowledge base can be integrated on the fly. These two trends appear to be interdependent, and in this paper, we propose an enabling technology that could allow their convergence.

Focusing on data, the heterogeneity of data produced and consumed on the Web is very high, and the amount of data globally processed is huge (i.e., Big Data): social data collected from social networks (e.g., Facebook, Twitter), medical data (e.g., Medical Subject Headings RDF (<https://id.nlm.nih.gov/mesh/>), W3C Semantic Web Health Care and Life Sciences Interest Group (<https://www.w3.org/2001/sw/hcls/>), HealthData.gov (<https://www.healthdata.gov/>)), scientific data from several science fields (e.g., NASA Global Change Master Directory (<https://gcmd.gsfc.nasa.gov/>), Science Environment for Ecological Knowledge (<http://seek.ecoinformatics.org/>), NeuroCommons project (http://neurocommons.org/page/Main_Page), NASA Semantic Web for Earth and Environmental Terminology (<https://sweet.jpl.nasa.gov/>), Bio2RDF (<http://bio2rdf.org/>)), e-commerce information (e.g., Best Buy, Sears, Kmart, O'Reilly publishing, Overstock.com), governmental data (e.g., Opening Up Government (<http://data.gov.uk/>), Tetherless World Constellation - Linking Open Government Data (<https://logd.tw.rpi.edu/>), Data-gov WiKi (<https://data-gov.tw.rpi.edu/wiki>)) and human knowledge (e.g., Wikipedia). Also from this perspective, the use of Semantic Web technologies for Big Data processing and analysis follows the previously-mentioned trends: on the one hand, software platforms research, analyze and harvest with different levels of autonomy the Web to provide enhanced and enriched results; on the other hand, specific applications are developed to consume, publish or combine information for the purpose of a vertical domain. The solution we present in this paper is information agnostic and able to manage data sources gathered from different and heterogeneous domains, providing support for standard Web protocols (i.e., HTTP and WebSocket). Furthermore, we extend the concept of Web of Data to applications that become the actionable extension of the Semantic Web.

Nowadays, the Web of Data is becoming a Web of Dynamic Data, where detecting, communicating and tracking the evolution of data changes play crucial roles and open new research questions [3,4]. Moreover, detecting data changes is functional to enable the development of distributed Web of Data applications where software agents may interact and synchronize through the knowledge base [5,6]. The need for solutions on detecting and communicating data changes over the Web of Data has been emphasized in the past few years by research focused on enabling interoperability in the Internet of Things through the use of Semantic Web technologies (like the ones shown in [7–10] just to cite a few). Last but not least, an attempt made by W3C is represented by the Web of Things (<https://www.w3>

org/WoT/) working group and by the Linked Data Notifications (<https://www.w3.org/TR/ldn/>) released in 2017 that provide the recommendation to enable notifications over Linked Data.

In this paper, we propose a decentralized Web-based software architecture, named SEPA (SPARQL Event Processing Architecture) built on top of the authors' experience acquired developing an open interoperability platform for smart space applications [11–21]. SEPA derives and extends the architecture presented in [22] through the use of standard Linked Data technologies and protocols. It enables the detection and communication of changes over the Web of Data by means of a content-based publish-subscribe mechanism where the W3C SPARQL 1.1 Update and Query languages are fully supported respectively by publishers and subscribers. SEPA is built on top of the SPARQL 1.1 Protocol and introduces the SPARQL 1.1 Secure Event protocol and the SPARQL 1.1 Subscribe Language as a means for conveying and expressing subscription requests and notifications.

In particular, assuming an event as “any change in an RDF store”, SEPA has been mainly designed to enable event detection and distribution. The core element of SEPA is its broker (see Figure 1): it implements a content-based publish-subscribe mechanism where publishers and subscribers use respectively SPARQL 1.1 Updates (i.e., to generate events) and SPARQL 1.1 Queries (i.e., to subscribe to events). In particular, at subscription time, subscribers receive the SPARQL query results. Subsequent notifications about events (i.e., changes in the RDF knowledge base) are expressed in terms of added and removed query results since the previous notification. With this approach, subscribers can easily track the evolution of the query results (i.e., the context), with the lowest impact on the network bandwidth (i.e., the entire results set is not sent every time, but just the delta of the results). The SEPA broker design is detailed in Section 4.

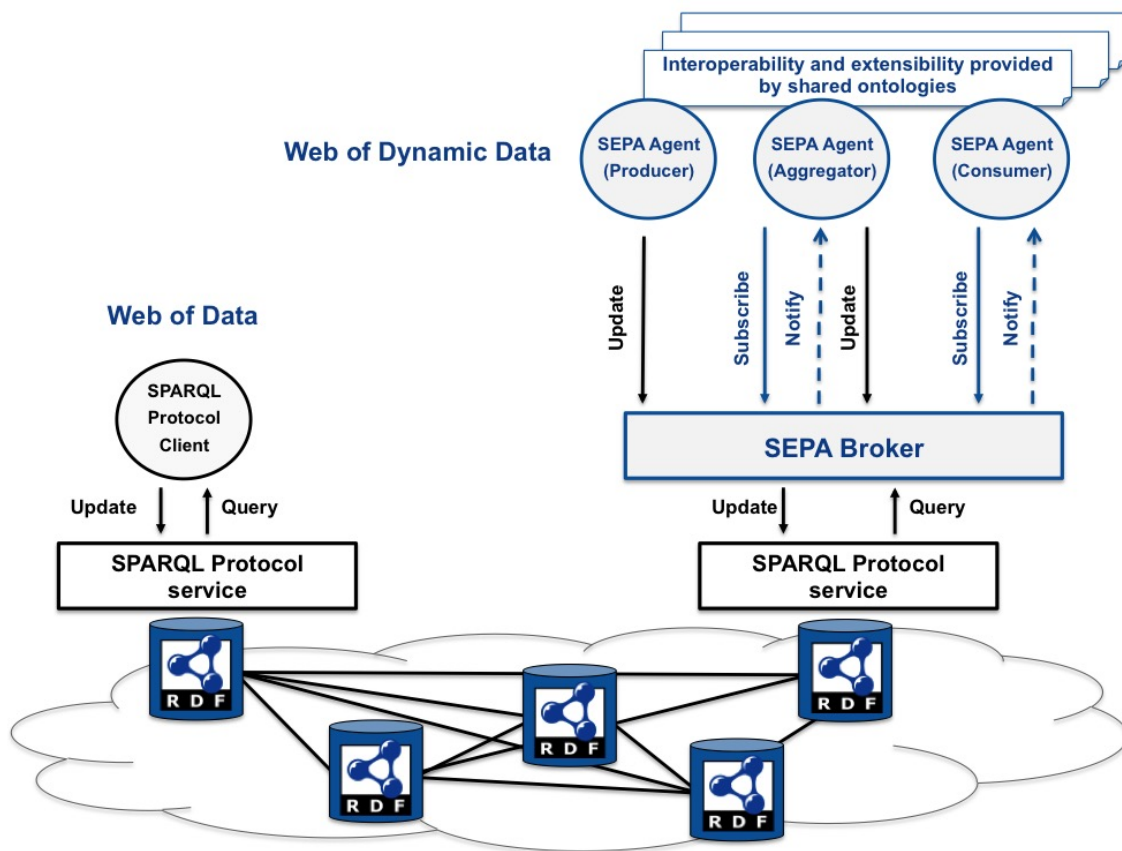


Figure 1. From the Web of Data to the Web of Dynamic Data. SEPA, SPARQL Event Processing Architecture.

In this paper, we propose the SPARQL 1.1 Secure Event (SE) Protocol and the SPARQL 1.1 Subscribe Language presented, along with the mechanisms to support client and server authentication, data encryption and message integrity, in Section 3. The SPARQL 1.1 SE Protocol allows agents to interact with the broker like with a standard SPARQL Protocol service (also known as the SPARQL endpoint), but at the same time, it allows one to convey subscriptions and notifications expressed according to the SPARQL 1.1 Subscribe Language. SEPA provides developers with a design pattern where an application is constituted by a collection of agents. As shown in Figure 1, each agent plays a specific role within an application (i.e., producer, aggregator or consumer) and can be shared among different applications. While a producer publishes events by means of a SPARQL 1.1 Update, a consumer is subscribed to specific events through a SPARQL 1.1 Query. An aggregator plays both roles: it is subscribed to events and generates new events based on the received notifications. The application design pattern introduced by SEPA, along with the application domains that may benefit from the adoption of this model, are presented in Section 5. Section 2 frames our proposal within the technologies and alternative solutions we have identified in the literature, while overall conclusions are drawn in Section 6.

2. Related Work

The architecture presented in this paper can be framed within the research topics known as stream reasoning [23], linked stream data processing [24] and content-based publish-subscribe [25]. To the best of our knowledge, the Linked Data concept and relative technologies were formalized for the first time in [2]. As concerns Linked Data dynamics, [3] provides a deep insight into aspects like discovery, granularity level, description of changes, detection algorithms and notification mechanisms related to detect, propagate and describe changes in Linked Data. In particular, with reference to the notification mechanism, some approaches are compared and divided into two classes: pull-based and push-based. With reference to pull-based approaches, the most recent one seems to be the Linked Data Notifications protocol, for the first time presented in [26]. In [27], the authors discuss the Web of Data Streams and present a set of requirements to be met by an infrastructure to exchange RDF streams on the Web. In particular, the “keep the data moving” requirement states that architectures “must prioritize active paradigms for data stream exchange, where the data supplier can push the stream content to the actors interested in it”. SEPA meets this requirement, as well as all the other requirements stated in [27], and it provides a push-based mechanism (i.e., based on the SPARQL 1.1 Secure Event and SPARQL 1.1 Subscribe Language herein proposed) through the publish-subscribe paradigm. However, at the same time, the SEPA broker reference architecture is open for implementing also pull-based mechanisms (e.g., Linked Data Notifications) and providing adaptive push-pull techniques like the one presented in [28].

The publish-subscribe paradigm [25] is an essential brick in the development of modern, distributed applications. This communication paradigm contemplates the existence of two types of clients (i.e., publishers and subscribers), respectively devoted to produce data and consume only those matching the client declared interest. One of the main advantages of this paradigm is the full decoupling in space and time of the involved entities that do not need to know each other and be online at the same time. Furthermore, publishers and subscribers also benefit from a synchronization decoupling: they still perform their tasks, while asynchronously generating events or receiving notifications. For example, Baldoni et al. in [29] adopt the publish-subscribe paradigm as the communication means among a set of processes and propose a framework based on a limited set of primitives (i.e., publish, subscribe, unsubscribe and notify) that contains all the essentials of the paradigm.

However, the publish-subscribe paradigm itself is not enough to have a complete definition of the semantics of data. We believe that Semantic Web technologies could play a significant role in defining the data semantics, and among all the solutions we have been able to discover, in the following, we summarize the ones closest to our approach. Initial solutions for Semantic Web-based publish-subscribe are presented by Wang et al. [30] and Chirita et al. [31], while a first attempt to use

SPARQL as the subscription language is presented in [32]. Another early SPARQL-based RDF stream processing proposal is streaming SPARQL [33], where the authors propose to extend SPARQL with time-windows like in continuous SPARQL (C-SPARQL) [34], SPARQLStream [35], event processing SPARQL (EP-SPARQL) [36], continuous query evaluation over Linked Data streams (CQELS) [37] and Sparkwave [38]. A time-window specifies the triples for which the query is executed. It can be defined either by the number of triples (last triples from the stream) or the time (e.g., the last 15 min). The window specification defines also how often the window is updated and consequently the frequency of query evaluation. C-SPARQL is a language for expressing persistent SPARQL queries over RDF streams, and in addition to the extensions for windows, it extends SPARQL 1.0 with support for time management and aggregations. SPARQLStream differs from Streaming SPARQL and C-SPARQL in three ways. First, the SPARQLStream only considers time-based windows, whereas Streaming SPARQL and C-SPARQL support also windows defined by a concrete number of triples. Second, the SPARQLStream enables windows to be defined into the past in contrast to Streaming SPARQL and C-SPARQL, where the windows always start from the present. The third difference is that the SPARQLStream proposes window-to-stream operations that are used to transform a stream of windows into a stream of RDF triples with timestamps. The EP-SPARQL focuses on the detection of RDF triples in a specific temporal order, and it proposes several binary operations that can be used to combine RDF graph patterns in a temporal-sensitive manner. EP-SPARQL does not enforce the use of windows, but instead, it provides an optional SPARQL function that can be used inside the FILTER pattern to create time windows by setting time intervals for which the query is active. CQELS uses its own native processing model in the query engine that transforms SPARQL into logic rules, Rete networks or data suitable for standard stream processing engines. This makes it possible to have full control of the query execution plan, and it is used in practice to dynamically reorder operators based on changes in the input data. Sparkwave is based on the Rete algorithm [39] that provides a generalized solution to perform pattern matching, where facts are matched against rules. In this case, the facts are presented with RDF triples and rules with persistent SPARQL queries. There are four notable aspects that differentiate SEPA from the window-based SPARQL streaming approaches presented above. First, the SEPA does not use windows to define the triples for which the query is evaluated (i.e., we concentrate on real-time evaluation of events within the whole system). Second, SEPA fully supports SPARQL 1.1 both to generate (i.e., update) and subscribe (i.e., query) to events, and it is based on the W3C SPARQL 1.1 Protocol (i.e., it would be transparent to clients performing SPARQL 1.1 Updates and SPARQL 1.1 Queries). Third, instead of processing individual RDF triples coming from specific RDF streams, SEPA is based on an interaction model where any agent can trigger events by modifying the context of the system with SPARQL 1.1 Update Language operations. Fourth, the SEPA broker detects how the results have changed from the initial query results, whereas the window-based approaches provide the whole result set whenever it is modified in any way. Because of these fundamental differences in the SPARQL event processing approaches, also the implementations of the event processing mechanisms and algorithms are totally different.

Other research works focusing on using SPARQL as a subscription language include the one by Groppe et al. [40], EventCloud [41,42], INSTANS [43,44], semantic event notification service (SENS) [5,6,45,46] and Smart-M3 [47] (i.e., Suomalainen et al. [48] proposed a secure broker, called RIBS. Galov et al. [49] have developed the CuteSIB, focusing on extensibility, dependability and portability. Viola et al. [12] proposed pySIB, targeted especially at resource-constrained computing platforms.). The approach presented in [50] focuses on providing a secure publish-subscribe mechanism for the management of complex supply chains in enterprises based on RDF. They use the same authorization framework used by SEPA (i.e., the OAuth 2.0 Authorization Framework, <https://tools.ietf.org/html/rfc6749>), suggesting also other solutions, like WebID (<https://www.w3.org/2005/Incubator/webid/spec/>). They adopt WebSub (<https://www.w3.org/TR/2018/REC-websub-20180123/>) (formerly PubSubHubBub) as a mean for conveying notifications, the same notification mechanism is used by Passant and Mendes [51].

Research efforts on developing Linked Data, and in particular detecting Linked Data changes, would have an impact on the development of real applications as much as Web standards will be adopted and promoted. The main Web players, including W3C, are pushing in this direction by promoting standards and defining ontologies and vocabularies. The SPARQL 1.1 Secure Event Protocol and the SPARQL 1.1 Subscribe Language proposed in this paper go in that direction.

3. SPARQL 1.1 Secure Event Protocol and Subscribe Language

The SPARQL 1.1 Secure Event (SE) Protocol (<http://mml.arces.unibo.it/TR/sparql11-se-protocol.html>) wraps the SPARQL 1.1 Protocol to support subscriptions and secure communications. The SPARQL 1.1 SE Protocol aims to transparently support the HTTP methods, provided by the SPARQL 1.1 Protocol, for conveying SPARQL 1.1 Queries and Updates. As shown in Figure 2, this is obtained by complementing these two languages with a third one: the SPARQL 1.1 Subscribe Language (see Sections 3.1–3.3). For instance, according to the SPARQL 1.1 Service Description (<https://www.w3.org/TR/sparql11-service-description/>), this could be specified by a SEPA broker through the `sd:languageExtension` and `sd:supportedLanguage` properties. Furthermore, subscriptions need a two-way asynchronous communication between subscribers (i.e., to issue subscribe and unsubscribe requests) and the SEPA broker (i.e., to provide notifications). SEPA is mainly focused on pushing notifications to agents in order to deal with high frequency changes and to provide high,y reliable agents synchronization. The SEPA broker reference implementation adopts the WebSocket protocol (<https://tools.ietf.org/html/rfc6455>) for conveying subscribe/unsubscribe requests and notifications, but other protocols like MQTT (<http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>) or COAP (<https://tools.ietf.org/html/rfc7252>) can be easily supported. Furthermore, according to [28], the push mechanism can be complemented by a pull mechanism (e.g., the one provided by Linked Data Notifications [26]).

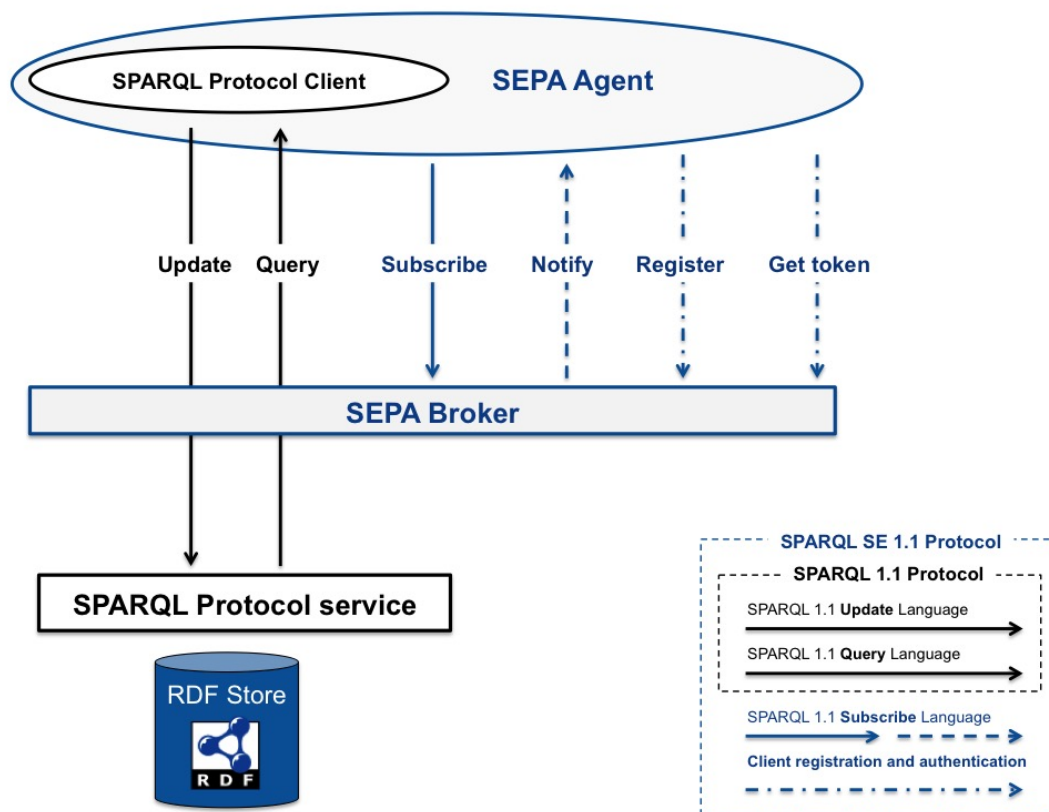


Figure 2. SPARQL 1.1 Secure Event Protocol.

The SPARQL 1.1 Subscribe Language (<http://mml.arces.unibo.it/TR/sparql11-subscribe.html>) complements the SPARQL 1.1 Update and Query Languages in order to express subscribe/unsubscribe requests and notifications. The language can be used by those agents who want to subscribe to changes in the content of an RDF store (i.e., aggregators or consumers, as shown in Figure 1 and detailed in Section 5). The following requirements have been assumed:

- The content of a subscribe request is the same as the one of a SPARQL 1.1 Query;
- At subscription time, an agent receives the complete set of query results;
- A notification includes the added and removed results since the previous notification;
- A client may activate more than one subscription at the same time;
- A notification refers to a specific subscription;
- Notifications related to the same subscription must be differentiated by an incremental index.

The language presented in this paper includes the minimum set of messages required to implement the above requirements and can be extended in the future if needed. This section presents a JSON (<https://tools.ietf.org/html/rfc7159>) serialization of the SPARQL 1.1 Subscribe Language. Other type of serializations may be implemented.

Concerning the security aspects, these are addressed by the SPARQL 1.1 SE protocol providing:

- Client authentication based on OAuth 2.0,
- Data encryption, server authentication and message integrity.

Figure 3 gives an overview of the interactions between a client and a broker. Every client requiring a secure communication must first register to obtain its own credentials (Step 1). Registration can be performed once, and policies on managing multiple registrations are considered application specific (please refer to Section 3.4 for more details). The client credentials are then used to obtain (or renew) a JSON Web Token (<https://tools.ietf.org/html/rfc7519>) (Step 2), as further described in Section 3.5. A valid token can be used by a client to: (i) perform updates and queries (Step 3), according to the SPARQL 1.1 Protocol; (ii) subscribe and unsubscribe (Step 4), using the SPARQL 1.1 Subscribe Language (see Sections 3.1–3.3).

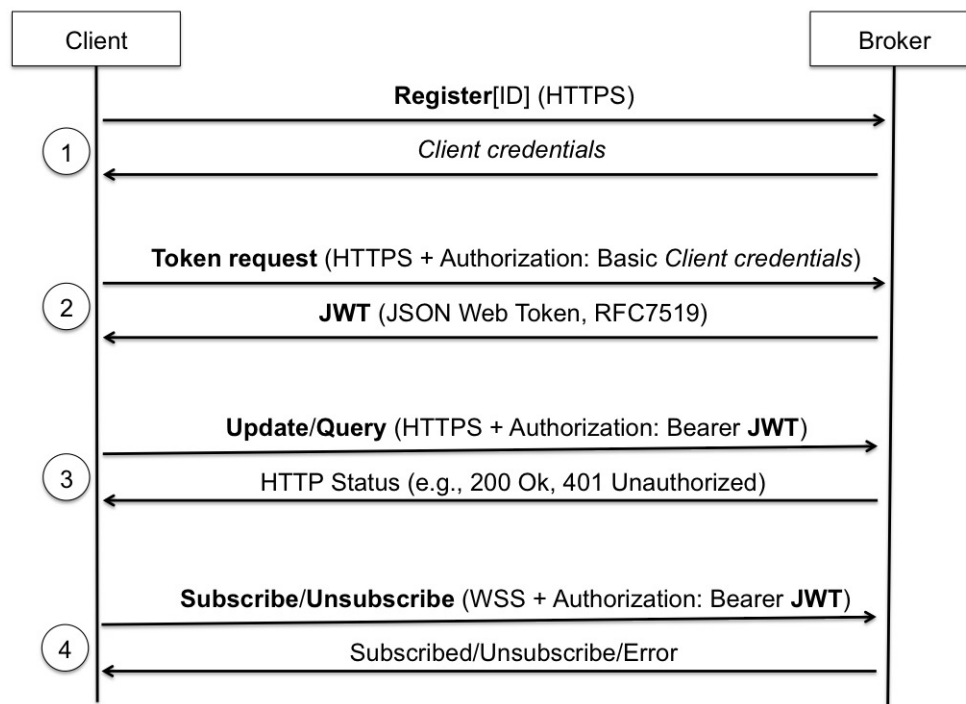


Figure 3. Registration, authentication and secure primitives invocation. JWT, JSON Web Token.

3.1. Subscribe Primitive

A subscribe request is expressed as shown in Listing 1. The value of the sparql member must be a SPARQL 1.1 Query; the value of the authorization member (if present) must be a Bearer JSON Web Token; and the value of the alias member (if present) is a string representing a friendly name of the subscription. The sparql member is required, while the other two are optional. The authorization member is only required for secure operations (see Section 3.6), and the value of the alias member, if present, will be included in the subscribe response.

If the subscribe request is successfully processed, the SEPA broker replies as shown in Listing 2. The value of the spuid member is an URI. It is used to identify the corresponding notifications, and it must be present. The alias member (if present) has the same value of the corresponding alias member of the subscribe request. The use of the alias member is recommended if the client sends multiple subscribe requests. In fact, subscribe responses are not supposed to arrive with the same order as the corresponding requests have been issued. Using an alias, the client will be able to relate each response to the corresponding request, and as a consequence, it will be able to relate each notification with the corresponding subscribe request. The value of the sequence member will be always zero (i.e., the response to a subscribe request can be considered the first notification). Eventually, the value of the addedResults member corresponds to the results of the SPARQL query specified in the request, according to the SPARQL 1.1 Query Results JSON format (<https://www.w3.org/TR/sparql11-results-json/>), while the value of the removedResults will be always empty (i.e., at subscription time, the results will be always new results).

In the case of error, the SEPA broker is expected to reply as shown in Listing 11.

3.2. Unsubscribe Primitive

A client may request to remove a specific subscription. This can be done by sending a message like the one shown in Listing 3. The spuid member value is the subscription URI provided by the subscribe response message. The value of the authorization member (if present) must be a Bearer JSON Web Token. The former member is required, while the latter is only required for secure primitives (see Section 3.6). The SEPA broker replies to an unsubscribe request with the message shown in Listing 4. In case of error, it is expected to reply as shown in Listing 11.

3.3. Notification

As described in Section 3.1, at subscription time, an agent receives the entire SPARQL query result set (i.e., initial bindings). In order to keep track of the following changes in the result set, the agent is notified of the bindings added and removed since the previous notification (i.e., or since subscription time in case of the first notification). The effect of this is two-fold: the network overhead is optimized (i.e., it is unnecessary to send the entire result set every time), and the notification processing is simplified (i.e., the agent only needs to add and remove from its result set the added and removed bindings, respectively). The notification content is expressed as shown in Listing 2. The value of the spuid member is the URI of the subscription that generates the notification, and it corresponds to the one received by the agent at subscription time (see Section 3.1). The value of the sequence member is a number, initialized at zero at subscription time and incremented by one at every new notification of the same subscription. Eventually, the values of the addedResults and removedResults members are both in the form of the SPARQL 1.1 Query Results JSON Format (see Section 3.1).

3.4. Client Registration

Registration allows a client to obtain the credentials needed to request (or renew) a JSON Web Token. Every SEPA implementation must support the client credentials authorization grant. Other authorization grants and registration mechanisms may be supported. To obtain the credentials, a client must own an application-specific identifier, known as client_identity. The client_identity may

correspond to the device serial number, the MAC address, the Electronic Product Code, an e-mail or any other sort of identifier defined by the application, even a TPM key (<https://www.iso.org/standard/66510.html>). This allows devices to register also without human intervention. For the scope of this paper, registration can be done once. Multiple registration requests (i.e., using the same client_identity) are not allowed: multiple registration policies and mechanisms are out of the scope of this paper and are considered implementation dependent.

A client can issue a registration request with an HTTP POST over TLS (<https://tools.ietf.org/html/rfc5246>), as shown in Listing 5. A SEPA broker must provide a JSON response as shown in Listing 6. The JSON object contains the client credentials (client_id and client_secret) and the signature. The signature shall be used by the client to verify the JWT.

3.5. Client Authentication

Once a client has registered and holds the credentials, it can request a JWT by sending an HTTP POST like the one shown in Listing 7. The authorization header uses the HTTP basic authentication scheme (<https://tools.ietf.org/html/rfc2617>) having as the value the base64 encoding (<https://tools.ietf.org/html/rfc4648>) of the string "client_id:client_secret". A SEPA broker implementation must respond to a token request with a JSON object like the one shown in Listing 8. The response contains the following members: access_token is the JWT, token_type to specify the token type (i.e., the default is bearer) and expires_in, representing the number of seconds after which the token will expire. Once a token is expired, the client can request a new token by using its credentials. Requesting a token while the current one is not expired generates an error.

3.6. Secure Primitives: Query, Update, Subscribe and Unsubscribe

Secure requests are authorized through JWT and sent over TLS connections (e.g., HTTPS or WSS). For HTTPS requests (i.e., updates or queries), clients must add to the SPARQL 1.1 Protocol request the authorization header as shown in Listing 9. For WSS requests (i.e., subscribes and unsubscribes), the JWT is assigned to the authorization member of the request as shown in Listing 3 (i.e., the same applies for unsubscribes; see Listing 1).

3.7. Error Responses

In the case of error, the SEPA broker replies with a JSON object like the one shown in Listing 11. If it applies, the use of the HTTP status codes is recommended.

4. Broker Design

The modular design of the SEPA broker allows one to support new protocols, mechanisms and algorithms for enabling subscriptions over the Web of Data. A reference implementation of the broker is available on GitHub (<https://github.com/arces-wot/SEPA>). As shown in Figure 4, the broker architecture is layered in three parts: gates, scheduler and core. The following sections provide details on the different parts of the broker.

4.1. Protocols And Dependability

The gates layer implements the SPARQL 1.1 SE Protocol (see Section 3): it create requests (i.e., update, query, subscribe and unsubscribe) for the scheduler and delivers responses and notifications. As shown in Figure 4, the SEPA broker reference implementation provides two gates: HTTP(S) and WS(S), both supporting also the Transport Layer Security (TLS) Protocol. The former processes updates and queries according to the SPARQL 1.1 Protocol, while the latter uses the WebSocket protocol for conveying subscription requests and notifications. The HTTP gate is based on the non-blocking, event-driven I/O model based on Java NIO, provided by the Apache HTTP Components (<https://hc.apache.org/>), while the WebSocket gate is based on the Java WebSockets

library by TooTallNate (<https://github.com/TooTallNate/Java-WebSocket>). Other protocols like MQTT, COAP or Linked Data Notification can be supported by implementing the corresponding gates.

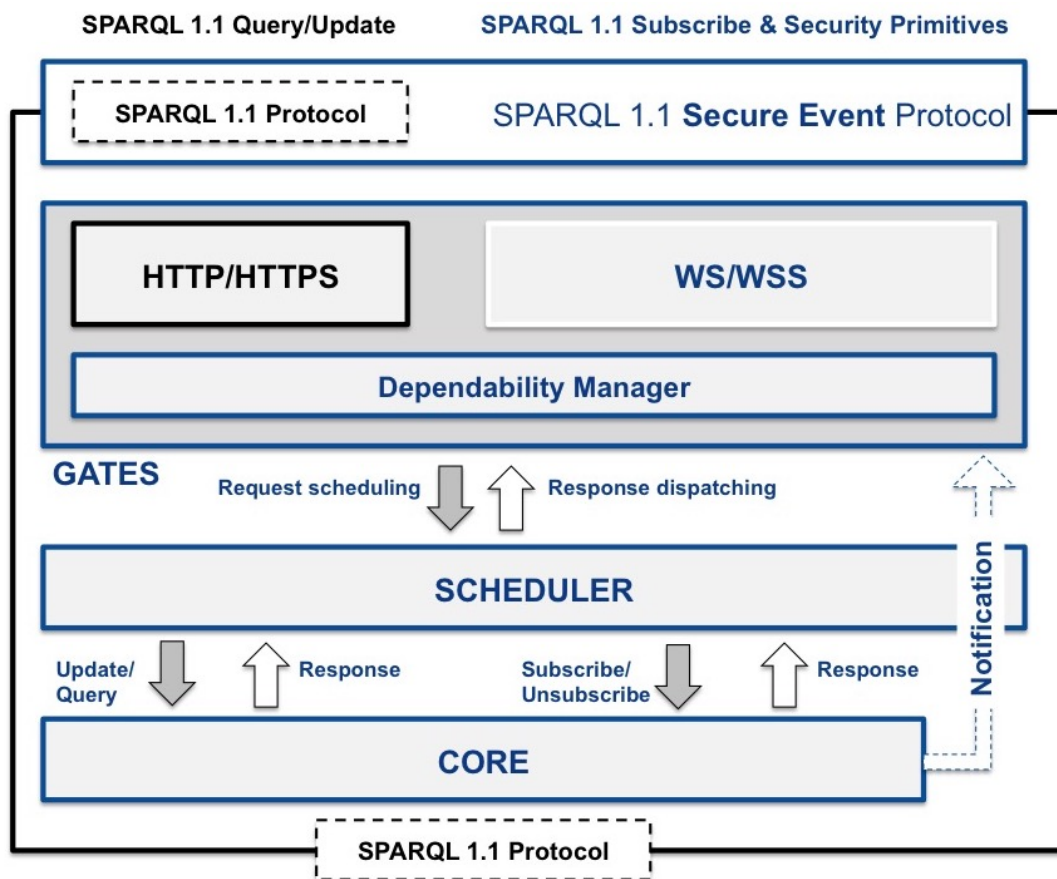


Figure 4. Broker reference architecture.

SEPA aims to provide a minimum level of dependability [52] through the Dependability Manager. On the one hand, it implements the security policies and mechanisms presented in Section 3. In a real-world scenario, the OAuth 2.0 Authorization Server would be different from the Resource Server (i.e., the SEPA broker). Clients who need secure access to the SEPA broker would register and get a valid token from such an external service (e.g., <https://auth0.com>). However, at the same time, to provide an off-the-shelf solution for testing SEPA security, the reference implementation of the Dependability Manager implements the client credentials grant type and uses JSON Web Tokens (JWT) (i.e., the reference implementation uses the APIs provided by Connect2Id, <https://connect2id.com/products/nimbus-jose-jwt>). On the other hand, reliability is achieved with simple, but effective methods of failure detection in the communication between the broker and subscribed agents. This property is important to grant the general dependability of the connection, but also, it is functional, on the broker side, for cleaning unused resources. In this sense, the WebSocket protocol used by the reference implementation embeds a failure detection mechanism. In fact, thanks to the ping-pong controls frames, the broker and the agents can recognize a failure (i.e., a broken connection) and react accordingly to some fault-tolerant policies. Furthermore, the broker can exploit this information to free unused resources created by agents that have been disconnected in an unexpected way, avoiding a negative impact on the performance. Overall, the protocol and the broker architecture must support the development of distributed applications with some degree of availability and resilience. Therefore, future implementations of new gates should at least implement a basic fault detection mechanism to recognize disconnections or node failures.

4.2. Requests Scheduling and Responses Dispatching

The scheduler layer implements the scheduling mechanisms and policies. In the reference implementation, requests are scheduled as FIFO, and the scheduler can be configured with a maximum number of pending requests (i.e., the size of the FIFO queue). The scheduler implements also the dispatching of responses coming from the core layer by forwarding the correct response to the correct gate, which will then send back the response to its client. The same applies to notifications. The requests coming from the gates layer may also be scheduled according to load balancing policies (e.g., processing may performed also on a different machine), and the scheduler may deny requests due to a high number of pending requests (e.g., for quality of service purposes). Requests may also have different priorities, or some requests may be avoided in some application contexts (e.g., the use of time-consuming queries may be avoided in highly synchronized and reactive environments).

4.3. Processing

The core of the broker processes the requests coming from the scheduler (i.e., update, query, subscribe and unsubscribe). As shown in Figure 5, the main building blocks of the core are: the Query processor, the Update processor the Subscription Processing Unit (SPU) manager and a main thread holding a FIFO queue of update requests.

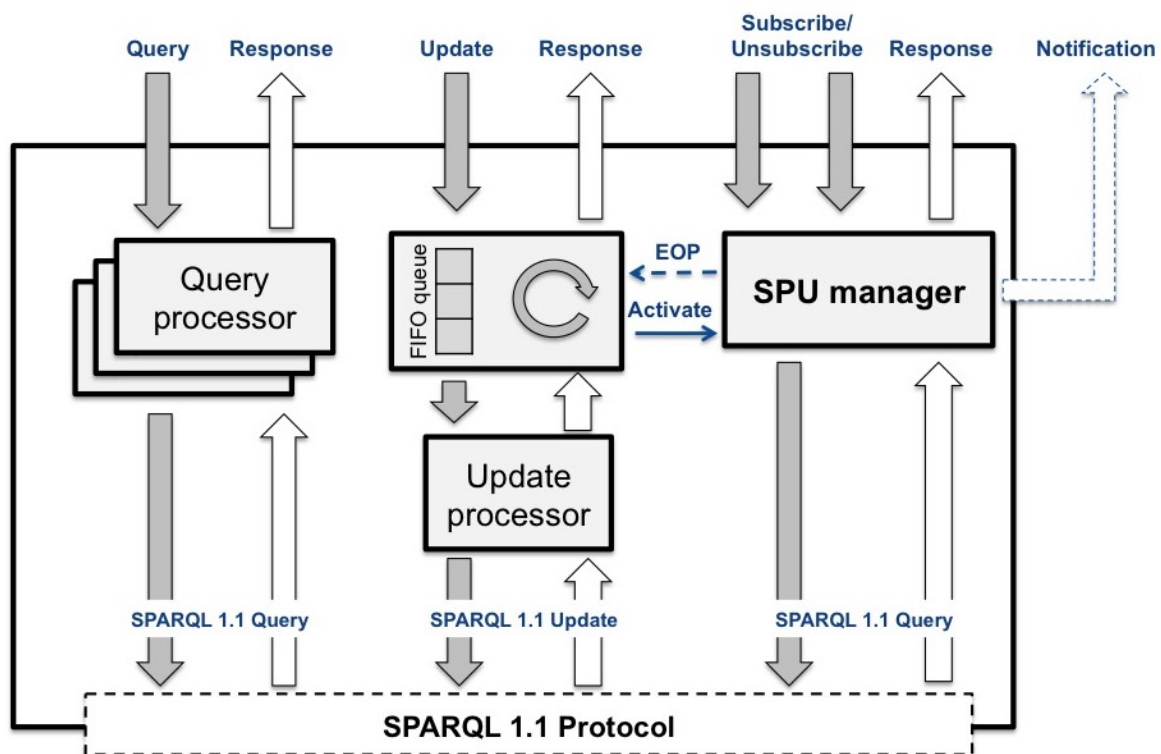


Figure 5. Core of the broker architecture. EOP, end-of-processing; SPU, Subscription Processing Unit.

While queries can be processed in parallel (i.e., multiple Query processor instances can run concurrently), updates are sequentially processed through a FIFO queue (i.e., only one instance of the Update processor can be active). As soon as a query arrives, it is sent to the underpinning SPARQL endpoint, and the decision on when to process such a query is made there. As most of the SPARQL endpoints are supposed to be able to process multiple requests in parallel, queuing together queries and updates could result in a substantial decrease of the performance. On the one hand, this means that the coherence of query processing (with respect to updates) is not granted, but on the other hand, this allows one to take advantage of all the processing power of the underpinning SPARQL endpoint.

Instead, the sequential processing of updates is a fundamental requirement to grant coherence on subscriptions' processing. In fact, as updates change the content of the RDF store, all the active subscriptions must be checked on the same RDF store snapshot. Because of that, a new update can only be processed after the processing of all subscriptions ended. More in detail, the Update processor and SPU manager are synchronized as follows. The core thread sends an update request to the Update processor and waits to receive a response. Once received, the response is forwarded to the publisher. It should be noted that, in this way, the publisher receives a response on the effective status of its update (i.e., the response to the publisher is not provided as soon as the request has been inserted into the FIFO queue, but once the response from the SPARQL endpoint has been received). At the application level, this allows to implement the synchronization mechanisms that are fundamental for the development of distributed applications. In case of a successful response, the core thread activates the SPU manager and waits to receive an end-of-processing (EOP) indication. The EOP indicates that all the active subscriptions have been processed. The core thread can so extract from the FIFO queue the next update request (if present) and send it to the Update processor.

With this approach, the update processing will never overlap with the subscription processing. This can be avoided only if the SPU manager does not need to perform queries on the SPARQL endpoint during subscription processing. There are two possibilities to implement this: (i) the SPU manager holds a local RDF store (i.e., cache) for each subscriptions (i.e., this is referred to as the Context Triple Store in [22]); (ii) the SPU manager implements a subscription algorithm that does not require access to the knowledge base, like the one presented in [53] (based on the Rete algorithm [39]). In both cases, the Update processor must return the triples that have been added or removed so that the SPU manager can track the evolution of the RDF store caches or the Rete network. In this scenario, the SPU manager is expected to indicate the EOP as soon as it receives an activate request (i.e., the request can be added to a synchronized queue), and the core can immediately start processing the next update request.

4.4. Subscription Processing Unit Manager

This section describes the internal structure of the Subscription Processing Unit (SPU) manager that processes the subscriptions. The SPU manager architecture is shown in Figure 6.

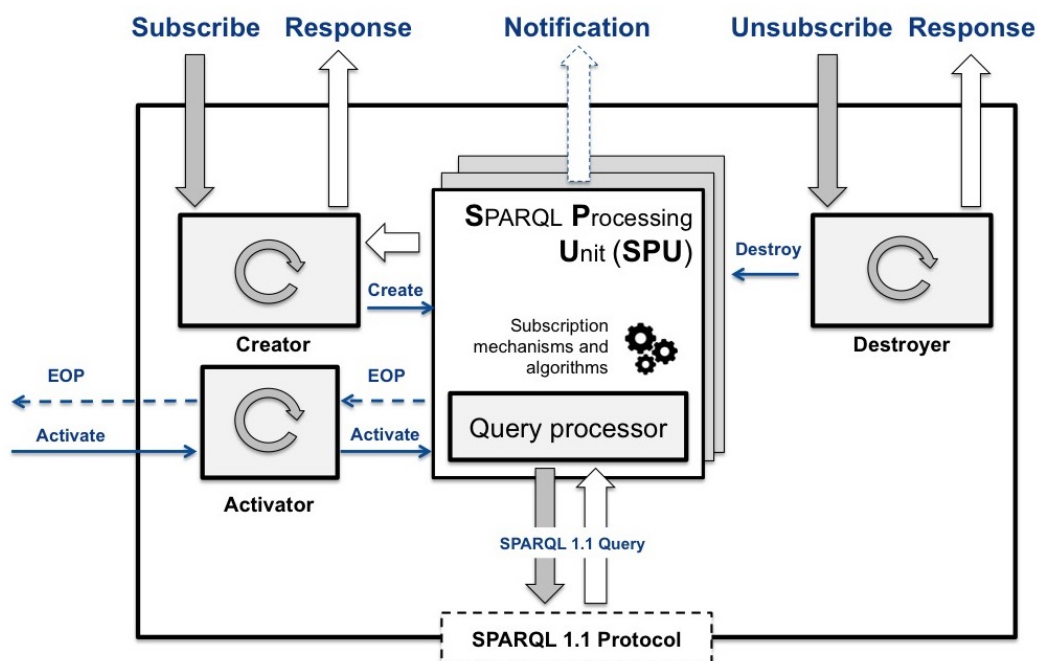


Figure 6. SPU manager architecture.

Each subscription is processed by an SPU that is instantiated by the Creator module when a subscribe request is received. The Creator module, by analyzing the SPARQL query, may instantiate a different kind of SPU (i.e., implementing a different algorithm) or link the subscription with an existing SPU (i.e., the SPARQL query is the same [54]). An SPU is deallocated by the Destroyer module when an unsubscribe request is received. This may be issued directly by a client (see Section 3.2) or by the Dependability Manager (see Figure 4) if a client connection has been lost.

On each update received by the broker, the SPU manager is activated (see Activate in Figure 6). The Activator module activates all the SPUs and waits for all of them to complete processing. Then, it signals to the main core thread (see Figure 5) the effective end-of-processing (EOP) so that the next update request can be processed. SPUs run in parallel, and each SPU may also run on a different machine in a distributed computing environment.

An SPU implements the subscription algorithm and notifies its subscriber (or subscribers if the SPU is shared by multiple clients) of changes due to the last update (if any). As each subscription must be processed at any update, subscription processing shapes the scalability level. Here, multi-resolution approaches where a fast coarse-grained step filters out most unaffected subscriptions leaving the burden of detecting the need for notification to a few candidates, turn out to be particularly effective, as shown by the performance evaluation sections in [11,22]. In particular, in [22], an algorithm is presented that speeds up the query processing and the results matching by (i) binding variables before sending the query to the SPARQL endpoint and (ii) performing a fast filtering stage based on look-up tables to reduce the amount of subscriptions that are candidates to produce notifications. Another option to optimize the subscription processing could be to implement a Rete network as described in [53]. Discussion on subscriptions processing optimization is out of the scope of this paper, but the reader can refer to [55] for a discussion on how performance can be evaluated.

5. Application Design Pattern

The aim of this section is to present the application design pattern implemented by SEPA to achieve modular, extensible and cost-effective solutions (see Section 5.2). Section 5.1 provides the reader with a plethora of examples of applications that may benefit from SEPA. In Section 5.3 is presented the JSON SPARQL Application Profile (JSAP), as a means to describe an application, while two intentionally simple examples are presented in Section 5.4 to clarify the proposed application design pattern and the role played by Semantic Web technologies in the design of interoperable and extensible applications. Eventually, these two examples are also considered to provide some preliminary results of the evaluation of the SEPA baseline implementation (see Section 5.5).

5.1. Application Domains

In the following, we present some examples of applications that we argue may benefit from SEPA. The most immediate application is the monitoring of the Semantic Web itself: the subscribe/notify mechanism and the living SPARQL queries allow one to monitor Linked Data over time. Monitoring Linked Data are extremely interesting in order to understand, for example, at which rate the Semantic Web grows, how frequently data and links are modified, which evolution patterns can be identified, how the structure of information evolves, what is the nature of stored information, what is the granularity and how long data are available before disappearing from the Semantic Web.

Web applications, like search engines, use meta-data and semantics to improve search results, both in terms of topic coherence, number of results, user satisfaction, business opportunities and market impact potentially generated by the results. The most important players in the Web industry are trying to promote the annotation of Web pages through the adoption of standards, in order to increase the potentialities of the Semantic Web: Microsoft, Google and Yahoo promote Schema.org; Facebook developed the Open Graph Protocol (<http://ogp.me/>); while the e-commerce industry frequently adopts the GoodRelations vocabulary (<http://www.heppnetz.de/projects/goodrelations/>).

A more restricted class of Web applications that could significantly take advantage of SEPA are recommender systems, applications that use information provided by users, users' profiles and meta-data to generate recommendations. A few examples of works in this wide domain are [56–60].

At the enterprise level, Semantic Web technologies can be adopted to develop inexpensive, scalable and incremental solutions for agile data integration and information classification [61–63]. Data integration is fundamental in enterprises in order to provide unified views of a large amount of information. Traditional solutions for data integration [64] are limited by the capabilities of database management systems, while the adoption of semantics can fill this gap in flexibility, ensuring system evolution and lowering maintenance costs.

Newspapers, televisions, governmental and public authorities, large enterprises and online global services rely on dynamic content management to produce and maintain information-rich Web sites and portals with very limited human intervention. Using technologies like ontology-driven reasoning [65] and automated topics aggregation/generation [66,67], dynamic content management allows one to create Web sites where the information dynamicity, the diversity of media format, the interactivity and automation cannot be achieved with traditional content management systems [68].

The flexibility of SEPA allows one to address also domains completely different from the previous ones, including supply chain optimization, financial data monitoring and security. Every company has to deal with the complexity of a supply chain: changes of suppliers, changes of data managed, a high volume of data and a significant effort to generate a unified view of information are required. The financial domain is a wide and heterogeneous ecosystem where a huge amount of complex and valuable information is exchanged every day. It is an extremely dynamic world in constant evolution, characterized by a lack of interoperability and rapid changes. Increasing the control on financial information and the reactivity on its changes have an important economical and societal impact: monitor the health of financial markets, anticipate financial crisis, avoid frauds, protect investments, identify profitable business and investments and perform risk analysis. A great effort is currently spent to find a flexible model shared by all financial stakeholders: in October 2017, the Enterprise Data Management (EDM) Council released a new version of the Financial Industry Business Ontology (FIBO) (<https://www.edmcouncil.org/financialbusiness>), trying to push the financial community to adopt Semantic Web technologies and promote interoperability. As a final use case, we mention the security domain where future solutions will be more oriented toward the automatic identification of threats, frauds, vulnerabilities and attacks. An automated approach relies on a knowledge base that models the system and its vulnerabilities, in conjunction with an initial set of security patterns, that evolves in time depending on the identification of new vulnerabilities and on the attacks that the system undergoes.

5.2. Software Framework and Application Design Pattern

Recalling Figure 1, a SEPA application is composed by agents (i.e., producers, consumers and aggregators), the interaction of which is mediated by vocabularies and rules defined through shared ontologies. While producers and consumers should be kept as simple as possible to be reused and shared by different applications, aggregators implement the application logic. Aggregators link the functionalities provided by producers and consumers in order to achieve the desired behaviors. To this end, they subscribe to events created by producers and create new events that may trigger actions of consumers or other aggregators. In general, the application logic implemented by an aggregator can be combinatorial (i.e., no context memory is needed) or sequential (i.e., the context evolution is stored into the aggregator internal context memory). Within SEPA, an aggregator plays the role played by an event processing agent within the event processing network presented in [69].

The advantages of the proposed design guidelines are two-fold. First, since producers and consumers are implemented independently from a specific use case, they can be shared between different applications (i.e., by modifying existing or implementing new aggregators, the overall system functionality can be modified or extended indefinitely). This, of course, leads to cost savings also when

new systems are deployed. Second, since the processing performed by consumers and producers is very simple, they may be implemented in resource-restricted devices.

SEPA provides the software framework shown in Figure 7. Interoperability at the information level is granted by RDF/RDFS/OWL ontologies: as a best practice, application developers are recommended to adopt standard and recognized ontologies. The framework offers an API in several programming languages and at different levels of abstraction, to allow developers to choose the desired one (also with reference to the class of the device as defined in RFC7228 (<https://tools.ietf.org/rfc/rfc7228.txt>): low-level APIs implement the SPARQL 1.1 Secure Event Protocol (see Section 3), while higher level APIs provide an application design pattern that can be followed to enhance reuse, modularization and interoperability (see APCI, Aggregator Producer Consumer API, in Figure 7). At the application level, SEPA promotes the reuse of software modules through the APCI: along with the adopted ontologies, an application can be characterized by the set of updates and subscribes issued by its agents. Within the APCI, each agent is linked to a specific primitive: a producer to an update, a consumer to a subscribe and an aggregator to a pair subscribe-update.

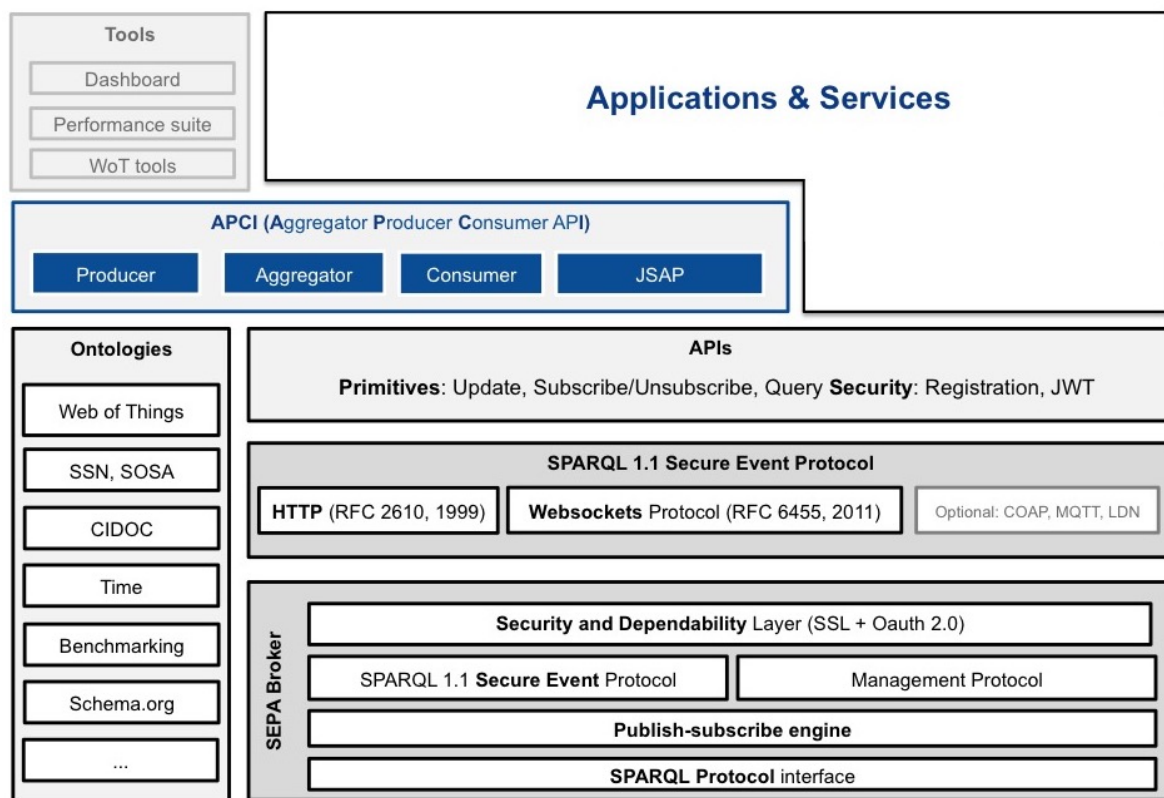


Figure 7. SEPA framework.

5.3. The JSON SPARQL Application Profile

SEPA introduces the JSON SPARQL Application Profile (JSAP) (<http://mml.arces.unibo.it/TR/jsap.html>) as a means to describe an application. JSAP is a JSON file, whose root structure is shown in Listing 12. JSAP allows describing a SEPA application by including the parameters needed to interact with one (or more) SEPA broker instance(s), along with all the SPARQL 1.1 Updates and Queries used by the application. Queries and updates can be written in a compact format thanks to the prefixed names for IRIs (<https://tools.ietf.org/html/rfc3987>). Prefixes are specified within the namespaces member (see Listing 12), and the PREFIX keywords are automatically appended by the API to the SPARQL issued to the broker. The graphs member includes the optional HTTP query string parameters as defined by the SPARQL 1.1 Protocol for queries (i.e., default-graph-uri and named-graph-uri) and

updates (using-graph-uri and using-named-graph-uri). The host member (see Listing 12) specifies the IP address of the broker (i.e., it can be overwritten by nested host members).

The sparql11protocol member (see Listing 13) and the sparql11seprotocol member (see Listing 14) allow one to configure the protocol (i.e., the former for updates and queries, while the latter for subscribes). The sparql11seprotocol member has two mandatory members: protocol and availableProtocols. The former specifies the protocol gate to be used (see Section 4.1). Such a protocol must be present as a JSON object within the availableProtocols member (i.e., the latter contains all the identifiers of the available protocol gates that can be used for subscriptions). The content of each member of "availableProtocols" is protocol specific. If present, the security member contains the parameters to connect to the OAuth service provided by the SEPA.

One of the main benefits of JSAP consists of the ability to create a template for an application, which can be fetched and modified at run-time to fit the application needs. This template acts as an identity card of an application. For example, a producer that updates the value of a temperature sensor will only need to fill a field in the template (i.e., the current value). Here is where the definition of forced bindings comes to help. A forced binding enables the developer to substitute at run-time a variable in a template with a custom value. To define forced bindings, the key forcedBindings must be used. The value is a JSON object. The variable of a forced binding is a key in that JSON object. Its value is again a JSON object containing the keys type and value. The type key is mandatory and must be one of uri, bnode or literal. The value key is optional and should be used to specify a default value for that variable.

All the SPARQL Updates and SPARQL Queries used by the application, along with their forced bindings, are respectively enumerated within the updates and queries members of the JSAP (see Listing 12), according to the structure shown in Listing 15. The scope of an update (or subscribe) can also be redefined with reference to the default protocol parameters (e.g., the host or the port). This can be achieved by overwriting one or more fields of the sparql11protocol member (for update and queries) or the sparql11seprotocol member (for subscribes). In this way, an application can interact with multiple SEPA brokers and/or SPARQL endpoints at the same time. For example, an aggregator can subscribe to a SEPA broker instance and publish on a SPARQL endpoint (i.e., storing events for later analysis).

5.4. Examples of the Design of a SEPA Application

In order to provide the reader with a better understanding of the SEPA application design pattern and the role of JSAP, this section presents two simple examples. The first can be considered as a minimum-working example aimed at explaining the proposed application design pattern (producer-aggregator-consumer) and the role of JSAP. It should be noted that this example shows how agents can interact and synchronize, fundamental requirements for the development of distributed applications. The second highlights the role of ontologies on defining a shared meaning, thus enabling interoperability at the information level. Raw sensor data gathered from an MQTT broker are contextualized and stored according to the W3C Semantic Sensor Network Ontology. Furthermore, these data are timestamped according to the W3C Time Ontology in OWL and inserted into a dedicated RDF Big Data store for future analysis.

The first example is a chat application where users exchange text messages. As shown in Figure 8, a client is composed of three agents: a producer (Sender) and two aggregators (Receiver and Remover). The JSAP fragment of the application is listed in Appendix C.

In Figure 8 is shown the sequence of updates and notifications triggered by the action of sending a message from Client #1 (identified by the schema:PersonURI1 URI) to Client #2 (identified by the schema:PersonURI2 URI):

- When the two clients join the chat, their Receiver and Remover agents subscribe to the SEPA broker by replacing respectively the receiver forced binding in the SENT query and the sender

- forced binding in the RECEIVED query with the client identifier (e.g., schema:PersonURI1 for Client #1 and schema:PersonURI2 for Client #2).
- When Client #1 sends a new message to Client #2, it invokes the SEND update (1), first replacing the Sender and Receiver forced bindings with the current one (i.e., respectively schema:PersonURI1 and schema:PersonURI2) and the text binding with the message to be sent. The effect of the update is to create the bold graph shown in Figure 8.
 - This update triggers a notification for Client #2 (2): its Receiver agent inserts the dotted part of the graph to specify the receiving time. This is done by invoking the SET_RECEIVED update (3), replacing the message forced binding with the corresponding one included in the notification.
 - This update triggers a notification on Client #1 (4): Client #1 knows at this time that Client #2 has received the message (i.e., clients are synchronized) and can delete the corresponding graph from the RDF store. This is done invoking the REMOVE update (5), replacing the message forced binding with the effective URI of the message to be removed. The effect of this update is two-fold: the RDF store is cleaned by all messages that have been received (i.e., the store contains only the messages that have been sent, but not yet received), and the Receiver agent is notified of the removed bindings (i.e., Client #2 is aware that Client #1 has been notified by the SET_RECEIVED update).

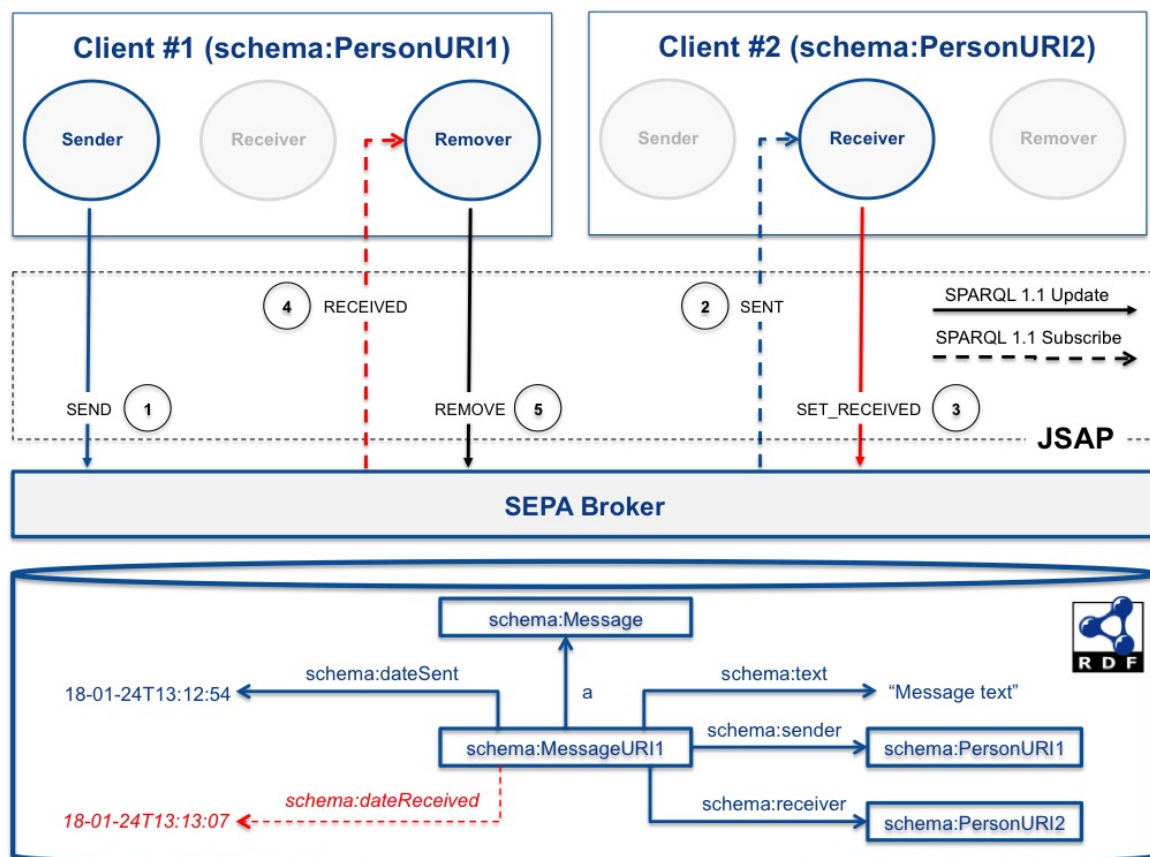


Figure 8. A simple chat application.

The second example is a typical Internet of Things application consisting of monitoring sensor data. The application consists of four agents (see Figure 9) and is described by the JSAP shown in Listing 18.

Heterogeneous sensor data (e.g., soil moisture and temperature and humidity of some server control rooms) gathered by heterogeneous wireless sensor networks (e.g., DASH7 [70,71], LoRa

(<https://www.semtech.com/technology/lora/what-is-lora>) and 6LowPan (<https://tools.ietf.org/html/rfc6282>) are sent over MQTT along with the temperatures of CPUs and hard disks of a set of servers. The application is an example of how raw data (i.e., the topic-value pairs provided by an MQTT broker) can be mapped into RDF triples. At the same time, it provides an overview of how to store temporal data for later analysis (i.e., based on SPARQL and semantic reasoning techniques [72]). Information level interoperability is achieved through the use of the following ontologies:

- W3C Sensor Network Ontology (<https://www.w3.org/TR/vocab-ssn/>)
- W3C Time Ontology in OWL (<https://www.w3.org/TR/owl-time/>)
- Quantities, Units, Dimensions and Data Types (QUDT) Ontology (<http://www.linkedmodel.org/doc/qudt/1.1/index.html>)

According to the core ontology SOSA (Sensor, Observation, Sample, and Actuator), the application maps each topic into an observation, providing the data value as the numeric value (i.e., qudt-1-1:numericValue) of a quantity (i.e., qudt-1-1:QuantityValue). The ontology has been extended with the arces-monitor:hasMqttTopic property that allows one to link an observation URI with a MQTT topic (see the “ADD_OBSERVATION” update in Listing 18 for details on how this mapping is performed). Mappings are dynamically added by the Topics Manager agent. A set of initial mappings can also be specified within the extended member of the JSAP (see Listing 12). An example of the content of an entry used to map the topic pepoli/6lowpan/network/NODO1/Temperature with the arces-monitor:Pepoli-6lowpan-Nodo1-Temperature observation URI is shown in Listing 17.

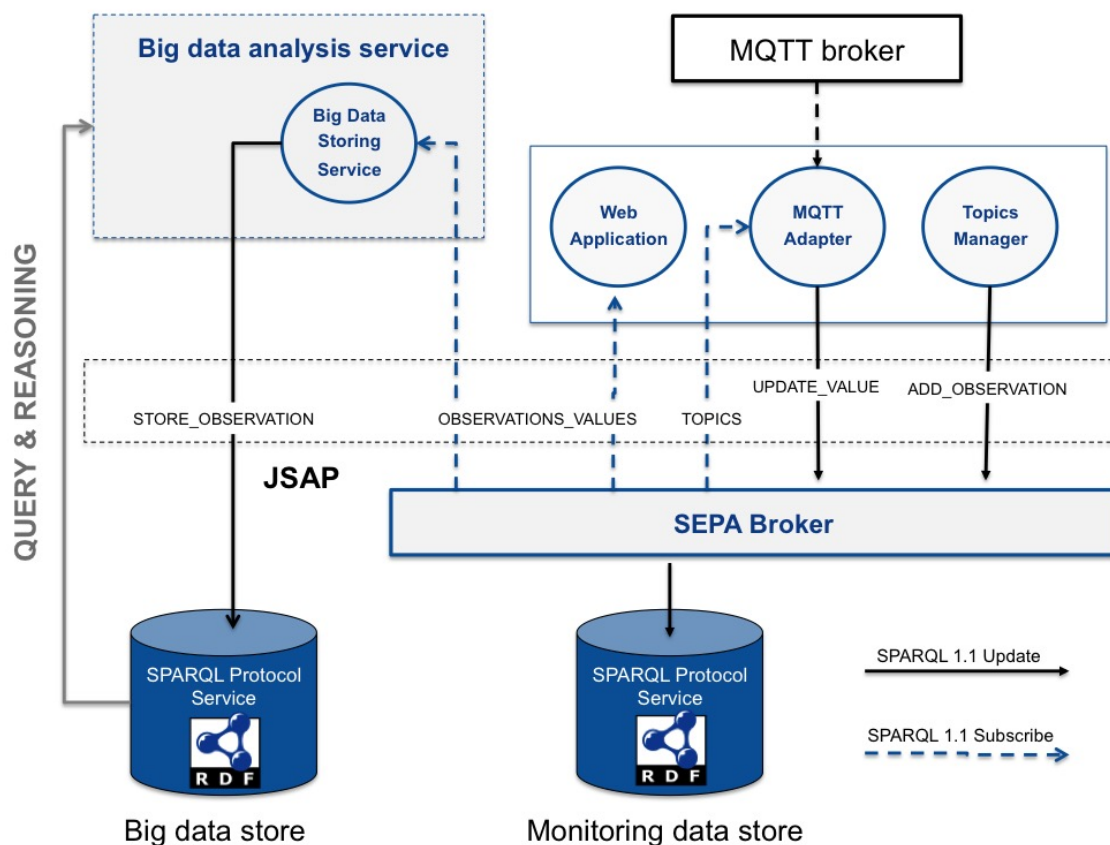


Figure 9. An MQTT sensors' monitoring application.

Data values are updated by the MQTT Adapter agent. On the one hand, the agent is subscribed to the MQTT broker (i.e., to all the topics using the “#” wildcard); on the other hand, it is subscribed,

through the “TOPICS” query, to the mappings created by the Topics Manager. On every notification coming from the MQTT broker (i.e., topic-value), the MQTT Adapter agent finds the observation URI that matches with the topic and extracts the numeric value of the observed quantity from the value (i.e., using regular expressions). This is where the lack of information level interoperability comes out: the value has no a predefined format, and its meaning is unknown. Moreover, in some cases, the data identifier is embedded within the topic, while in other cases, it is included within the value: an agreement with the MQTT broker provider is required in order to parse and understand the content of each topic-value pair. On the contrary, semantic information is all described in RDF, identified by URIs and their meaning represented through ontologies.

The events generated by the MQTT Adapter agent are consumed by two different agents (see “OBSERVATIONS_VALUES” in Figure 9): the Web Application and the Big Data Storing Service. The former, based on the Java Script APIs, visualizes the data in a Web browser as shown in Figure 13. The latter, through the “STORE_OBSERVATION” update, stores the events in a Big Data RDF store to be later analyzed. Events are time stamped according to the W3C Time Ontology in OWL. The discussion of the big data analysis techniques that could be used to extract meaningful information is out of the scope of this paper.

5.5. Experimental Evaluation and Preliminary Results

This section provides some preliminary results of the evaluation of the SEPA baseline implementation (<https://github.com/arces-wot/SEPA>) If on the one hand, a complete performance analysis like the one presented in [22] is out of scope, on the other hand, to provide evidence regarding the feasibility of the proposed architecture, some experiments have been done. The experiments are based on the two previously-presented examples. The SEPA broker here considered implements a naive subscription algorithm (i.e., on each update, all the subscriptions are evaluated and the results of the queries compared to the previous ones): the results represent a lower bound of the performance that could be achieved (e.g., a significant speed-up would be obtained by implementing smarter algorithms and approaches like the one presented in [11,22,53]). All the experiments have been run on the following configuration: (i) a server (Ubuntu 15.04, Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50 GHz, 64 GB RAM) hosting the SEPA broker and the Virtuoso Open Source SPARQL endpoint; (ii) a server (Ubuntu 16.04.2 LTS, QEMU Virtual CPU version (cpu64-rhel6) @ 2.2 GHz, 1 GB RAM) hosting the client applications.

The initial two experiments refer to the chat example. The first one aims at evaluating the overhead introduced by the SEPA broker on the underpinning SPARQL endpoint. A client performs the updates and queries used by the chat application (see Listing 16) in the same order shown in Figure 8. In particular, the client issues 10 SEND updates, followed by 10 SENT queries, 10 SET_RECEIVED updates, 10 RECEIVED queries and 10 REMOVE updates. The time spent by the broker to handle each request has been sampled along with the time required by the SPARQL endpoint to perform each single SPARQL request (i.e., update or query). As shown in Figure 10, in this experiment, the SEPA broker introduces an almost negligible overhead (i.e., 1.5%) to the update and query processing.

The second experiment provides the reader with an overview of the update time and the latency of notifications. It consists of a client sending to itself a stream of 100 messages. Messages are sent without any break, and each message produces the sequence of notifications and updates shown in Figure 8 (i.e., (1) SEND update; (2) SENT notification; (3) SET_RECEIVED update; (4) RECEIVED notification; (5) REMOVE update). As shown in Figure 11, the update time is almost limited by the SPARQL endpoint update time (attested around 50–100 ms, as shown in Figure 10). Figure 12 gives a first impression of the notification latency that can be provided by a SEPA broker. In this simple experiment, a notification is sent to a client with a latency of less than 60 ms.

The last experiment refers to the MQTT monitoring example. The system has been deployed and run for more than 45 days. A screenshot of the Web application developed to monitor the sensor data is shown in Figure 13. Some preliminary dependability indicators (see Table 1) have been

collected through a JMX console (see Figure 14). This experiment demonstrates the feasibility of the proposed approach and provides an indication of the level of reliability granted by the baseline SEPA implementation.

Table 1. Some preliminary dependability indicators of the SEPA baseline implementation with reference to the MQTT monitoring application.

Up-time	>45 days
Processed requests	>30 M
Maximum active subscriptions	5
SPARQL endpoint query average time	28 ms
SPARQL endpoint update average time	26 ms
SEPA broker update processing average time	55 ms
SPU manager processing average time	30 ms

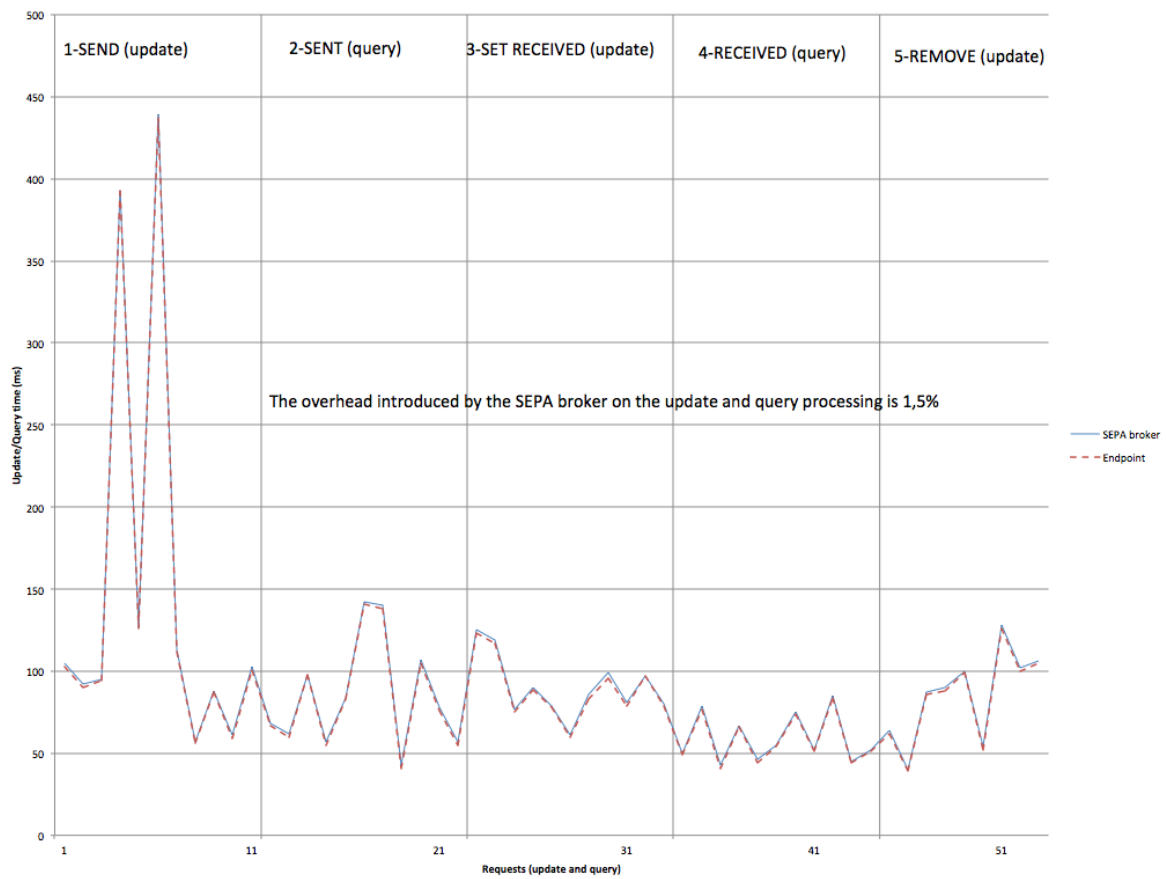


Figure 10. Analysis of the overhead introduced by the SEPA broker on the processing of updates and queries.

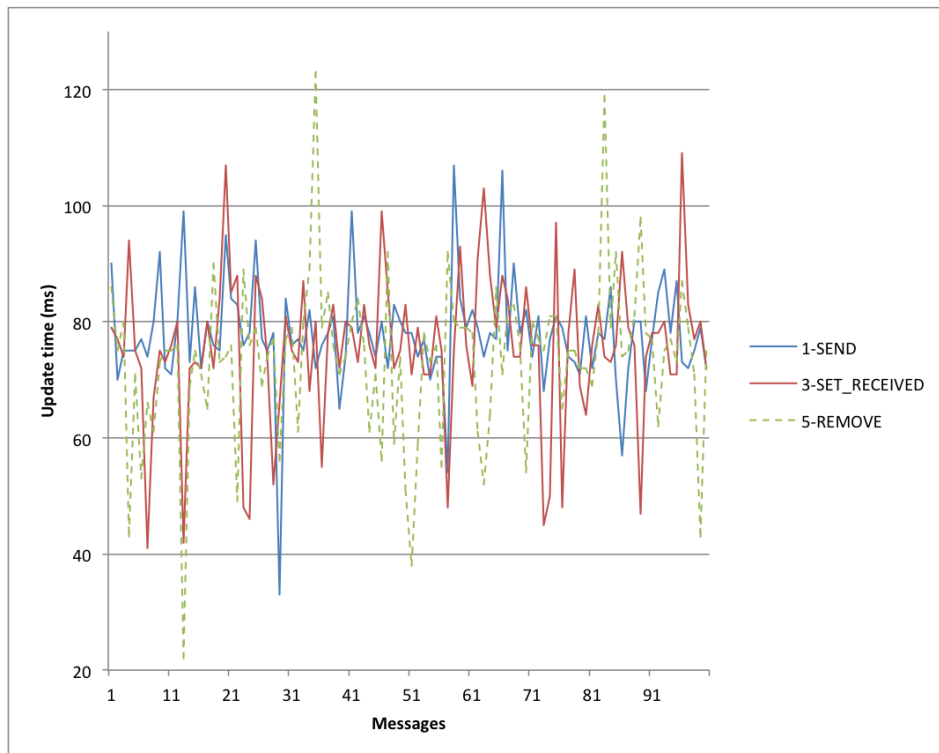


Figure 11. A client would be able to send up to 10 messages/s. This value is mainly limited by the SPARQL endpoint update time.

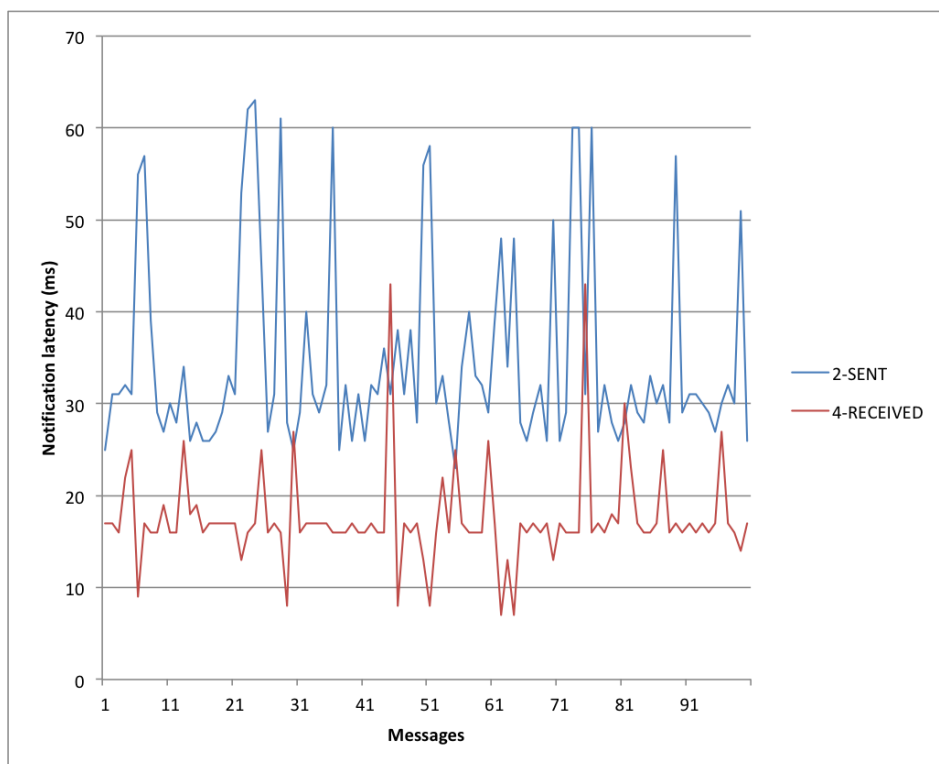


Figure 12. A client is notified within 10 and 60 ms after it has issued the update who triggered the notification.

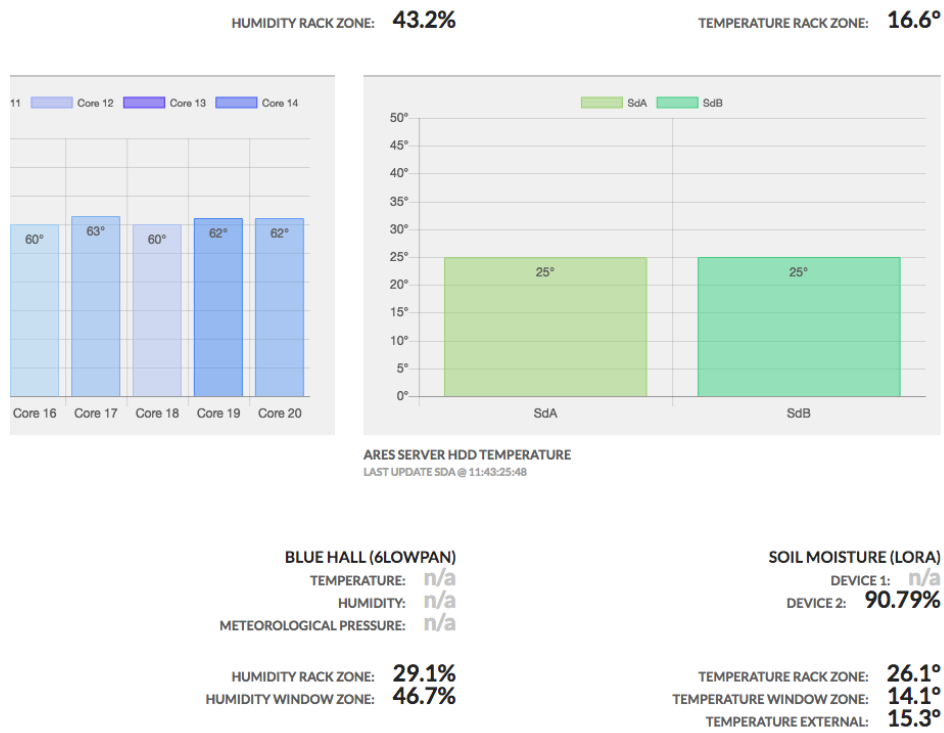


Figure 13. Dynamic Linked Data Web application: MQTT sensors' monitoring screenshot.

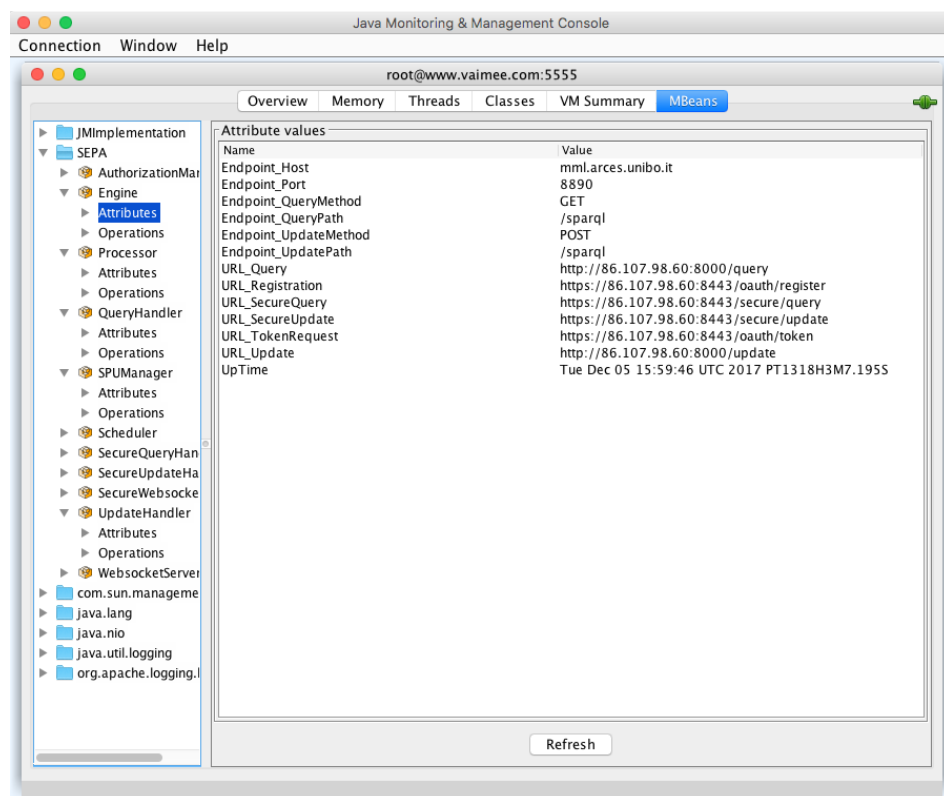


Figure 14. The JMX console of the SEPA broker.

6. Conclusions

Over the last decade, research on Linked Data has focused on unlocking the potential behind sharing data on the Web scale. If on the one hand, as evidenced by the Web of Things, the lack of interoperability across platforms and application domains can be tackled through Semantic Web technologies and standards, on the other hand, such technologies are not ready for the development of distributed, dynamic, context-aware and decentralized Web applications. Furthermore, detecting and notifying about any change within the Linked Data would be a fundamental building block, as confirmed by the ICT community with standards like Linked Data Notifications and WebSub. However, for those scenarios requiring security and dealing with high frequency data changes, solutions are still missing.

In this paper, we presented SEPA, a SPARQL Event Processing Architecture aimed at enabling the detection and communication of changes over the Web of Data by means of a content-based publish-subscribe mechanism where the W3C SPARQL 1.1 Update and Query Languages are fully supported respectively by publishers and subscribers. SEPA is built on top of the SPARQL 1.1 Protocol and introduces the SPARQL 1.1 Secure Event Protocol and the SPARQL 1.1 Subscribe Language as a means for conveying and expressing subscription requests and notifications. The SEPA Framework provides a development environment and an application design pattern that can be adopted to enhance reuse, modularization and interoperability. The framework, complemented by tools for the performance analysis, remote monitoring and control, offers developers a modular, extensible and cost-effective solution for implementing distributed, dynamic, context-aware, interoperable and secure Dynamic Linked Data applications and services.

Acknowledgments: The work presented in this paper is being developed within the EU project SWAMP (Smart Water Management Platform), Project ID: 777112, Funded under: H2020-EU.2.1.1.—INDUSTRIAL LEADERSHIP—Leadership in enabling and industrial technologies—Information and Communication Technologies (ICT).

Author Contributions: Luca Roffia is the principal investigator of SEPA. He conceived the architecture and developed the reference SEPA broker implementation and Java API. He designed and performed the experiments. Paolo Azzoni, as an industrial partner, mainly contributed presenting the most relevant application domains. He has also been involved in the definition of the structure of the paper and in its overall revision. Cristiano Aguzzi contributed to the design of the architecture, in particular, regarding dependability and extensibility. He mainly contributed to the definition of the SPARQL 1.1 Subscribe language and reviewed the presented examples. He is responsible for the software distribution and integration platform and GitHub repository management and the development of the JavaScript API. Fabio Viola contributed to the related work section and the overall revision of the paper. He is responsible for the implementation of the Python API and part of the subscription processing components. Francesco Antoniazzi contributed to the overall revision of the paper and with the implementation of the C API. Tullio Salmon Cinotti contributed to Sections 4.4 and 5.5 and to the final review.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. SPARQL 1.1 SE Protocol and Subscribe Language

```
{ "subscribe" : {  
  "sparql": "The SPARQL query",  
  "authorization": "Bearer Here goes the JWT", (optional)  
  "alias": "Friendly name" (optional) }}
```

Listing 1: SPARQL 1.1 Subscribe request.


```
{ "notification" : {
  "spuid": "Subscription URI",
  "alias": "Friendly name" (optional),
  "sequence": Incremental index,
  "addedResults": {JSON results},
  "removedResults": {JSON results}}
```

Listing 2: SPARQL 1.1 Notification.

```
{ "unsubscribe" : {
  "spuid": "Subscription URI",
  "authorization": "Bearer Here goes the JWT" (optional)}}
```

Listing 3: SPARQL 1.1 Unsubscribe request.

```
{ "unsubscribed" : {
  "spuid": "Subscription URI"}}
```

Listing 4: SPARQL 1.1 Unsubscribe response.

POST https://mml.arces.unibo.it:8443/oauth/register

Content-Type: **application/json**

Accept: **application/json**

Request body

```
{ "register" : {
  "client_identity": "Unique ID",
  "grant_types": ["client_credentials"]}}
```

Listing 5: Client registration request over HTTPS.

```
{ "credentials" : {
  "client_id": "...",
  "client_secret": "...",
  "signature": {
    "kty": "RSA",
    "e": "...",
    "x5t": "...",
    "kid": "sepacertificate",
    "x5c": ["..."],
    "n": "..."}}}
```

Listing 6: Client registration response.

POST https://mml.arces.unibo.it:8443/oauth/token

Content-Type: **application/json**

Accept: **application/json**

Authorization: **Basic base64(client_id:client_secret)**

Listing 7: Token request over HTTPS.

```
{ "token" : {
  "access_token" : "Here goes the JWT",
  "token_type" : "bearer",
  "expires_in" : Number of seconds }}
```

Listing 8: Token response.

```
POST https://mml.arces.unibo.it:8443/sparql
```

```
Authorization: Bearer Here goes the JWT
```

Listing 9: Secure query and update over HTTPS.

```
wss://mml.arces.unibo.it:9443/subscribe
```

Listing 10: Secure subscribe over WSS.

```
{ "error": {
  "body" : "SPARQL endpoint not found",
  "code" : 500}}
```

Listing 11: Error responses.

Appendix B. JSON SPARQL Application Profile

```
{ "host": "Host name or IP" ,
  "sparql11protocol": {...},
  "sparql11seprotocol": {...}},
  "namespaces" : {
    "prefix-1": "namespace URI1" ,
    ...,
    "prefix-N": "namespace URIn"},
  "graphs": { (all are optional)
    "default-graph-uri": "...",
    "named-graph-uri": "...",
    "using-graph-uri": "...",
    "using-named-graph-uri": "..."},
  "updates": {...},
  "queries": {...},
  "extended": {...} (optional) }
```

Listing 12: JSAP structure. A JSAP includes, along with the protocol configuration parameters, all the SPARQL updates and queries used by an application.

```

"sparql11protocol": {
"host": "Name or IP", (optional)
"protocol": "http",
"port": 8000,
"query": {
"path": "/...",
"method": "POST"
"format": "JSON" }
"update": {
"path": "/...",
"method": "URLENCODED_POST"
"format": "HTML" }}

```

Listing 13: JSAP sparql11protocol member. It allows to define the SPARQL 1.1 Protocol configuration parameters, including the overwriting of the host if needed.

```

"sparql11seprotocol": {
"host": "Name or IP", (optional)
"protocol": "Choose one among the available protocols",
"availableProtocols": {
"protocolID-1": {...},
...,
"protocolID-N": {...}},
"security": {...} (optional) }

```

Listing 14: JSAP sparql11seprotocol member. It allows to specify the SPARQL 1.1 SE Protocol configuration parameters, including the overwriting of the host if needed.

```

"IDENTIFIER" : {
"sparql": "SPARQL Update or Query" ,
"forcedBindings": { (optional)
"variable-1" : {
"type" : "literal" ,
"value" : "..."}, (optional)
"...": {...},
"variable-N" : {
"type" : "uri" ,
"value" : "..."}}, (optional)
"sparql11protocol": {...}, (optional, used by updates or queries)
"sparql11seprotocol": {...}, (optional, used by subscribes)
"graphs": {...} (optional, used by updates, queries or subscribes)
}

```

Listing 15: Template representing an update or query within a JSAP. It includes the forced bindings and it allows to overwrite some protocol parameters if required.

Appendix C. JSAP of the Chat and MQTT Examples

```

{"host": "...",
"sparql11protocol": {...},
"sparql11seprotocol": {...}
"namespaces": {
"schema" : "http://schema.org/" ,
"rdf" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
"updates": {
"SEND": {"sparql" : "INSERT {?message rdf:type schema:Message ;
schema:text ?text ; schema:sender ?sender ; schema:toRecipient ?receiver;
schema:dateSent ?time} WHERE {?sender rdf:type schema:Person .
?receiver rdf:type schema:Person
BIND (STR(now()) AS ?time)
BIND (IRI (CONCAT(\"http://schema.org/Message-\", STRUUID())) AS ?message)}",
"forcedBindings": {
"text": {"type": "literal"} ,
"sender": {"type": "uri"} ,
"receiver": {"type": "uri"}},
"SET_RECEIVED": {"sparql" : "INSERT {?message schema:dateReceived ?time}
WHERE {?message rdf:type schema:Message
BIND (STR(now()) AS ?time)}",
"forcedBindings": {
"message": {"type": "uri"}},
"REMOVE": {"sparql" : "DELETE {?message ?p ?o}
WHERE {?message rdf:type schema:Message . ?message ?p ?o}",
"forcedBindings": {
"message": {"type": "uri"}},
"queries": {
"SENT": {"sparql" : "SELECT ?message ?sender ?name ?text ?time
WHERE {?message rdf:type schema:Message ; schema:text ?text ;
schema:sender ?sender ; schema:toRecipient ?receiver ;
schema:dateSent ?time . ?sender rdf:type schema:Person ;
schema:name ?name . ?receiver rdf:type schema:Person} ORDER BY ?time",
"forcedBindings": {
"receiver": {"type": "uri"}},
"RECEIVED": {"sparql" : "SELECT ?message ?time
WHERE {?message schema:sender ?sender ; schema:dateReceived ?time ;
rdf:type schema:Message}",
"forcedBindings": {
"sender": {"type": "uri"}}}}

```

Listing 16: JSAP of the chat example

```

"semantic-mappings": {
...
"pepoli/6lowpan/network/NOD01/Temperature": {
"observation": "arces-monitor:Pepoli-6lowpan-Nod01-Temperature",
"unit": "qudt-unit-1-1:DegreeCelsius",
"location": "arces-monitor:Star",
"comment": "Server room temperature---Viale Pepoli (6LowPan)",
"label": "Server room temperature"},
...}

```

Listing 17: MQTT mapping.

```

{"host": "...",
"sparql11protocol": {Monitoring data store parameters},
"sparql11seprotocol": {Monitoring data store parameters},
"namespaces": {
"rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
"rdfs": "http://www.w3.org/2000/01/rdf-schema#",
"sosa": "http://www.w3.org/ns/sosa/",
"qudt-1-1": "http://qudt.org/1.1/schema/qudt#",
"qudt-unit-1-1": "http://qudt.org/1.1/vocab/unit#",
"arces-monitor": "http://mml.arces.unibo.it/monitor#",
"time": "http://www.w3.org/2006/time#",
"updates":
"ADD_OBSERVATION" : {"sparql" : "DELETE {?observation ?p ?o . ?q ?pl ?ol}
INSERT {?observation rdf:type sosa:Observation ; rdfs:label ?label ;
rdfs:comment ?comment ; sosa:hasFeatureOfInterest ?location ;
arces-monitor:hasMqttTopic ?topic; sosa:hasResult ?quantity .
?quantity rdf:type qudt-1-1:QuantityValue ; qudt-1-1:unit ?unit}
WHERE{
BIND (IRI (CONCAT (\ "http://mml.arces.unibo.it/monitor#QuantityValue-
\", STRUUID ())) AS ?quantity) . OPTIONAL {?observation rdf:type
sosa:Observation ; ?p ?o ; sosa:hasResult ?q . ?q ?pl ?ol }"}",
"forcedBindings": { "observation": {"type": "uri"},
"comment": {"type": "literal"}, "label": {"type": "literal"},
"location": {"type": "uri"}, "topic": {"type": "literal"},
"unit": {"type": "uri"}},
"UPDATE_VALUE" : {"sparql" : "DELETE {?quantity qudt-1-1:numericValue
?oldValue} INSERT {?quantity qudt-1-1:numericValue ?value}
WHERE {?observation rdf:type sosa:Observation ; sosa:hasResult
?quantity . OPTIONAL {?quantity qudt-1-1:numericValue ?oldValue}}",
"forcedBindings": {"observation": {"type": "uri"},
"value": {"type": "literal"}},
"STORE_OBSERVATION": {"sparql": "INSERT {?observation rdf:type
sosa:Observation ; sosa:hasResult ?quantity . ?quantity rdf:type
qudt-1-1:QuantityValue ; qudt-1-1:unit ?unit ; qudt-1-1:numericValue
?value; time:hasTime ?instant . ?instant rdf:type time:Instant ;
time:inXSDDateTimeStamp ?time} WHERE{
BIND (IRI (CONCAT (\ "arces-monitor:QuantityValue-", STRUUID ()))
AS ?quantity)
BIND (IRI (CONCAT (\ "http://www.w3.org/2006/time#Instant\",
STRUUID ())) AS ?instant)
BIND (STR (now ()) AS ?time)}",
"forcedBindings": { "observation": {"type": "uri"}, "unit": {"type": "uri"},
"value": {"type": "literal"}},
"sparql11protocol": {Big data store parameters}},
"queries": {
"TOPICS" : {"sparql" : "SELECT ?observation ?topic WHERE {?observation rdf:type
sosa:Observation ; arces-monitor:hasMqttTopic ?topic}"},
"OBSERVATIONS_VALUES" : {"sparql" : " SELECT ?observation ?location ?quantity
?label ?value ?unit WHERE {?observation rdf:type sosa:Observation ;
rdfs:label ?label ; sosa:hasFeatureOfInterest ?location ;
sosa:hasResult ?quantity . ?quantity rdf:type qudt-1-1:QuantityValue ;
qudt-1-1:unit ?unit ; qudt-1-1:numericValue ?value}"}
}

```

Listing 18: JSAP of the MQTT monitoring example.

References

1. Berners-Lee, T.; Hendler, J.; Lassila, O. The Semantic Web. *Sci. Am.* **2001**, *284*, 28–37. [\[CrossRef\]](#)
2. Bizer, C.; Heath, T.; Berners-Lee, T. Linked Data—The Story So Far. *Int. J. Semant. Web Inf. Syst.* **2009**, *5*, 1–22. [\[CrossRef\]](#)
3. Umbrich, J.; Villazön-Terrazas, B.; Hausenblas, M. Dataset Dynamics Compendium: A Comparative Study. In Proceedings of the First International Conference on Consuming Linked Data, Shanghai, China, 8 November 2010; CEUR-WS.org: Aachen, Germany, 2010; Volume 665; pp. 49–60.
4. Sanderson, R.; Van de Sompel, H. Cool URIs and Dynamic Data. *IEEE Int. Comput.* **2012**, *16*, 76–79. [\[CrossRef\]](#)
5. Murth, M.; Kühn, E. Knowledge-based interaction patterns for semantic spaces. In Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems, Krakow, Poland, 15–18 February 2010; pp. 1036–1043.
6. Murth, M.; Kühn, E. Knowledge-based coordination with a reliable semantic subscription mechanism. In Proceedings of the 2009 ACM Symposium on Applied Computing, Honolulu, HI, USA, 8–12 March 2009; ACM: New York, NY, USA, 2009; pp. 1374–1380.
7. Llanes, K.R.; Casanova, M.A.; Lemus, N.M. From Sensor Data Streams to Linked Streaming Data: A survey of main approaches. *J. Inf. Data Manag.* **2016**, *7*, 130–140.
8. Schade, S.; Ostermann, F.; Spinsanti, L.; Kuhn, W. Semantic Observation Integration. *Future Internet* **2012**, *4*, 807–829. [\[CrossRef\]](#)
9. Boulos, M.N.; Yassine, A.; Shirmohammadi, S.; Namahoot, C.S.; Brückner, M. Towards an “internet of food”: Food ontologies for the internet of things. *Future Internet* **2015**, *7*, 372–392. [\[CrossRef\]](#)
10. Alti, A.; Lakehal, A.; Laborie, S.; Roose, P. Autonomic semantic-based context-aware platform for mobile applications in pervasive environments. *Future Internet* **2016**, *8*, 48. [\[CrossRef\]](#)
11. D’Elia, A.; Viola, F.; Roffia, L.; Azzoni, P.; Salmon Cinotti, T. Enabling interoperability in the internet of things: A OSGi semantic information broker implementation. *Int. J. Semant. Web Inf. Syst.* **2017**, *13*, 146–167. [\[CrossRef\]](#)
12. Viola, F.; D’Elia, A.; Roffia, L.; Salmon Cinotti, T. A modular lightweight implementation of the Smart-M3 semantic information broker. In Proceedings of the 2016 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT), St. Petersburg, Russia, 18–22 April 2016; pp. 370–376.
13. D’Elia, A.; Viola, F.; Roffia, L.; Salmon Cinotti, T. A Multi-broker Platform for the Internet of Things. In Proceedings of the ruSMART 2015: Internet of Things, Smart Spaces, and Next Generation Networks and Systems, St. Petersburg, Russia, 26–28 August 2015; pp. 34–46.
14. Bedogni, L.; Bononi, L.; Di Felice, M.; D’Elia, A.; Mock, R.; Montori, F.; Morandi, F.; Roffia, L.; Rondelli, S.; Salmon Cinotti, T.; et al. An interoperable architecture for mobile smart services over the internet of energy. In Proceedings of the 2013 IEEE 14th International Symposium and Workshops on World of Wireless, Mobile and Multimedia Networks (WoWMoM), Madrid, Spain, 4–7 June 2013; pp. 1–6.
15. Morandi, F.; Roffia, L.; D’Elia, A.; Vergari, F.; Salmon Cinotti, T. RedSib: A Smart-M3 semantic information broker implementation. In Proceedings of the 12th FRUCT Conference, Oulu, Finland, 5–9 November 2012; pp. 86–98.
16. Roffia, L.; Bartolini, S.; Manzaroli, D.; D’Elia, A.; Salmon Cinotti, T.; Raffa, G. Requirements on System Design to Increase Understanding and Visibility of Cultural Heritage. In *Handbook of Research on Technologies and Cultural Heritage: Applications and Environments*; IGI Global: Hershey, PA, USA, 2011; Chapter 13, pp. 259–284.
17. Pantsar-Syväniemi, S.; Ovaska, E.; Ferrari, S.; Salmon Cinotti, T.; Zamagni, G.; Roffia, L.; Mattarozzi, S.; Nannini, V. Case study: Context-aware supervision of a smart maintenance process. In Proceedings of the 11th IEEE/IPSJ International Symposium on Applications and the Internet, Munich, Bavaria, Germany, 18–21 July 2011; pp. 309–314.
18. Vergari, F.; Salmon Cinotti, T.; D’Elia, A.; Roffia, L.; Zamagni, G.; Lamberti, C. An integrated framework to achieve interoperability in person-centric health management. *Int. J. Telemed. Appl.* **2011**, *2011*, 549282. [\[CrossRef\]](#)

19. Manzaroli, D.; Roffia, L.; Salmon Cinotti, T.; Ovaska, E.; Azzoni, P.; Nannini, V.; Mattarozzi, S. Smart-M3 and OSGi: The Interoperability Platform. In Proceedings of the SISS 2010, IEEE First International Workshop on Semantic Interoperability for Smart Spaces, Symposium on Computers and Communications, Riccione, Italy, 22–25 June 2010; pp. 1053–1058.
20. Vergari, F.; Bartolini, S.; Spadini, F.; D’Elia, A.; Zamagni, G.; Roffia, L.; Salmon Cinotti, T. A Smart Space application to dynamically relate medical and environmental information. In Proceedings of the 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), Dresden, Germany, 8–12 March 2010; pp. 1542–1547.
21. D’Elia, A.; Roffia, L.; Zamagni, G.; Vergari, F.; Toninelli, A.; Bellavista, P.; D’Elia, A.; Roffia, L.; Zamagni, G.; Vergari, F.; et al. Smart Applications for the Maintenance of Large Buildings: How to Achieve Ontology-based Interoperability at the Information Level. In Proceedings of the SISS 2010, IEEE First International Workshop on Semantic Interoperability for Smart Spaces, Symposium on Computers and Communications, Riccione, Italy, 22–25 June 2010; pp. 1072–1077.
22. Roffia, L.; Morandi, F.; Kiljander, J.; D’Elia, A.; Vergari, F.; Viola, F.; Bononi, L.; Salmon Cinotti, T. A Semantic Publish-Subscribe Architecture for the Internet of Things. *IEEE Int. Things J.* **2016**, *3*, 1274–1296.[\[CrossRef\]](#)
23. Della Valle, E.; Ceri, S.; Harmelen, F.V.; Fensel, D. It’s a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intell. Syst.* **2009**, *24*, 83–89.[\[CrossRef\]](#)
24. Le-phuoc, D.; Parreira, J.X.; Hauswirth, M. Linked Stream Data Processing. In Proceedings of the Reasoning Web. Semantic Technologies for Advanced Query Answering: 8th International Summer School 2012, Vienna, Austria, 3–8 September 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 245–289.
25. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A.M. The many faces of publish/subscribe. *ACM Comput. Surv.* **2003**, *35*, 114–131.[\[CrossRef\]](#)
26. Capadisli, S.; Guy, A.; Lange, C.; Auer, S.; Samba, A.; Berners-Lee, T. Linked Data Notifications: A Resource-Centric Communication Protocol. In Proceedings of the ESWC 2017 The Semantic Web, Portorož, Slovenia, 28 May–1 June 2017; Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 537–553.
27. Dell’Aglia, D.; Le Phuoc, D.; Le-Tuan, A.; Ali, M.; Calbimonte, J.P. On a Web of data streams. In Proceedings of the ISWC2017 workshop on Decentralizing the Semantic Web, Vienna, Austria, 21–22 October 2017.
28. Bhide, M.; Deolasee, P.; Katkar, A.; Panchbudhe, A.; Ramamritham, K.; Shenoy, P. Adaptive push-pull: Disseminating dynamic Web data. *IEEE Trans. Comput.* **2002**, *51*, 652–668.[\[CrossRef\]](#)
29. Baldoni, R.; Contenti, M.; Tucci Piergiovanni, S.; Virgillito, A. Modeling publish/subscribe communication systems: Towards a formal approach. In Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003) Guadalajara, Mexico, 17 January 2003; pp. 304–311.
30. Wang, J.; Jin, B.; Li, J. An Ontology-Based Publish/Subscribe System. In Proceedings of the Middleware 2004, Toronto, ON, Canada, 18–22 October 2004; Jacobsen, H.A., Ed.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 232–253.
31. Chirita, P.A.; Idreos, S.; Koubarakis, M.; Nejd, W. Publish/Subscribe for RDF-based P2P Networks. In Proceedings of the Lecture Notes in Computer Science, Crete, Greece, 10–12 May 2004; pp. 1–15.
32. Skovronski, J. An Ontology-Based Publish-Subscribe Framework. Master’s Thesis, State University of New York at Binghamton, Vestal, NY, USA, 2006.
33. Bolles, A.; Grawunder, M.; Jacobi, J. Streaming SPARQL—Extending SPARQL to Process Data Streams. In Proceedings of the ESWC2008—The Semantic Web: Research and Applications, Tenerife, Canary Islands, Spain, 1–5 June 2008; Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 448–462.
34. Barbieri, D.F.; Braga, D.; Ceri, S.; Grossniklaus, M. An Execution Environment for C-SPARQL Queries. In Proceedings of the 13th International Conference on Extending Database Technology, Lausanne, Switzerland, 22–26 March 2010; ACM: New York, NY, USA, 2010; pp. 441–452.
35. Calbimonte, J.P.; Corcho, O.; Gray, A.J.G. Enabling Ontology-Based Access to Streaming Data Sources. In Proceedings of the Semantic Web—ISWC 2010, Shanghai, China, 7–11 November 2010; Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 96–111.

36. Anicic, D.; Fodor, P.; Rudolph, S.; Stojanovic, N. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In Proceedings of the 20th International Conference on World Wide Web, Hyderabad, India, 28 March–1 April 2011; ACM: New York, NY, USA, 2011; pp. 635–644.
37. 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 Proceedings of the Semantic Web—ISWC 2011, Bonn, Germany, 23–27 October 2011; Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 370–388.
38. Komazec, S.; Cerri, D.; Fensel, D. Sparkwave: Continuous Schema-enhanced Pattern Matching over RDF Data Streams. In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, Berlin, Germany, 16–20 July 2012; ACM: New York, NY, USA, 2012; pp. 58–68.
39. Forgy, C.L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artif. Intell.* **1982**, *19*, 17–37. [[CrossRef](#)]
40. Groppe, S.; Groppe, J.; Kukulenz, D.; Linnemann, V. A SPARQL Engine for Streaming RDF Data. In Proceedings of the 2007 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System, Shanghai, China, 16–18 December 2007; pp. 167–174.
41. Pellegrino, L.; Baude, F.; Alshabani, I. Towards a scalable cloud- based RDF storage offering a pub/sub query service. In Proceedings of the CLOUD COMPUTING 3rd International Conference on Cloud Computing and GRIDs Virtualization, Nice, France, 22–27 July 2012; pp. 243–246.
42. Pellegrino, L.; Huet, F.; Baude, F.; Alshabani, A. A Distributed Publish/Subscribe System for RDF Data. In Proceedings of the Data Management in Cloud, Grid and P2P Systems, Prague, Czech Republic, 28–29 August 2013; Hameurlain, A., Rahayu, W., Taniar, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 39–50.
43. Abdullah, H.; Rinne, M.; Törmä, S.; Nuutila, E. Efficient Matching of SPARQL Subscriptions Using Rete. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, Trento, Italy, 26–30 March 2012; ACM: New York, NY, USA, 2012; pp. 372–377.
44. Rinne, M.; Abdullah, H.; Törmä, S.; Nuutila, E. Processing Heterogeneous RDF Events with Standing SPARQL Update Rules. In Proceedings of the Confederated International Conferences on the Move to Meaningful Internet Systems: OTM 2012, Rome, Italy, 10–14 September 2012; Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 797–806.
45. Murth, M. A Semantic Event Notification Service for Knowledge-Driven Coordination. In Proceedings of the 1st Int'l. Workshop on Emergent Semantics and Cooperation in Open Systems (ESTEEM), Rome, Italy, 1 July 2008.
46. Murth, M.; Kühn, E. A heuristics framework for semantic subscription processing. In *The Semantic Web: Research and Applications*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 96–110.
47. Honkola, J.; Laine, H.; Brown, R.; Tyrkko, O. Smart-M3 information sharing platform. In Proceedings of the IEEE symposium on Computers and Communications, Riccione, Italy, 22–25 June 2010; pp. 1041–1046.
48. Suomalainen, J.; Hyttinen, P.; Tarvainen, P. Secure Information Sharing Between Heterogeneous Embedded Devices. In Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, Copenhagen, Denmark, 23–26 August 2010; ACM: New York, NY, USA, 2010; pp. 205–212.
49. Galov, I.V.; Lomov, A.A.; Korzun, D.G. Design of semantic information broker for localized computing environments in the internet of things. In Proceedings of the 2015 17th Conference of Open Innovations Association (FRUCT), Yaroslavl, Russia, 20–24 April 2015; pp. 36–43.
50. Frommhold, M.; Arndt, N.; Tramp, S.; Petersen, N. Publish and Subscribe for RDF in Enterprise Value Networks. In Proceedings of the Workshop on Linked Data on the Web co-located with 25th International World Wide Web Conference (WWW 2016), Montreal, Canada, 11–15 April 2016.
51. Passant, A.; Mendes, P.N. SparqlPuSH: Proactive Notification of Data Updates in RDF Stores Using PubSubHubbub. In Proceedings of the Sixth Workshop on Scripting and Development for the Semantic Web, co-located with the European Semantic Web Conference 2010 (ESWC 2010), Crete, Greece, 31 May 2010; Volume 699.

52. Avižienis, A.; Laprie, J.C.; Randell, B. Dependability and Its Threats: A Taxonomy. In Proceedings of the Building the Information Society, Toulouse, France, 22–27 August 2004; Jacquart, R., Ed.; Springer: Boston, MA, USA, 2004; pp. 91–120.
53. Rinne, M.; Nuutila, E.; Törmä, S. INSTANS: High-performance event processing with standard RDF and SPARQL. In Proceedings of the ISWC 2012 Posters and Demonstrations Track, Boston, MA, USA, 13–15 November 2012; Volume 914.
54. Dividino, R.; Gröner, G. Which of the Following SPARQL Queries Are Similar? Why? In Proceedings of the First International Conference on Linked Data for Information Extraction, Sydney, Australia, 21 October 2013; CEUR-WS.org: Aachen, Germany, 2013; Volume 1057, pp. 2–13.
55. Viola, F.; D’Elia, A.; Roffia, L.; Salmon Cinotti, T. Performance Evaluation Suite for Semantic Publish-Subscribe Message-oriented Middlewares. In Proceedings of the UBICOMM 2016, The Tenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, Venice, Italy, 9–13 October 2016; pp. 190–196.
56. Beel, J.; Gipp, B.; Langer, S.; Breitingner, C. Research-paper recommender systems: A literature survey. *Int. J. Digit. Libr.* **2016**, *17*, 305–338.[CrossRef]
57. Tanuja, L.; Sandhya, G.; Shilpi, A. Using Semantic Recommenders for Personalized Recommendations. *Int. J. Recent Innov. Trends Comput. Commun.* **2017**, *5*, 151–154.
58. Yang, R.; Hu, W.; Qu, Y. Using Semantic Technology to Improve Recommender Systems Based on Slope One. In *Semantic Web and Web Science*; Li, J., Qi, G., Zhao, D., Nejd, W., Zheng, H.T., Eds.; Springer: New York, NY, USA, 2013; pp. 11–23.
59. Felfernig, A.; Friedrich, G.; Jannach, D.; Stumptner, M.; Zanker, M. Configuration knowledge representations for Semantic Web applications. *Artif. Intell. Eng. Des. Anal. Manuf.* **2003**, *17*, 31–50.[CrossRef]
60. Felfernig, A.; Erdeniz, S.P.; Jeran, M.; Akcay, A.; Azzoni, P.; Maiero, M.; Doukas, C. Recommendation Technologies for IoT Edge Devices. In Proceedings of the 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017)/12th International Conference on Future Networks and Communications (FNC 2017)/Affiliated Workshops, Leuven, Belgium, 24–26 July 2017; pp. 504–509.
61. Ostrowski, D.; Rychtyckyj, N.; MacNeille, P.; Kim, M. Integration of Big Data Using Semantic Web Technologies. In Proceedings of the 2016 IEEE Tenth International Conference on Semantic Computing (ICSC), Laguna Hills, CA, USA, 4–6 February 2016; pp. 382–385.
62. Ordóñez de Pablos, P. *Cases on Open-Linked Data and Semantic Web Applications*; Information Science Reference; IGI Global: Hershey, PA, USA, 2013.
63. Garcia, R. *Semantic Web for Business: Cases and Applications*; Advances in E-Business Research: Information Science Reference; IGI Global: Hershey, PA, USA, 2008.
64. Lenzerini, M. Data Integration: A Theoretical Perspective. In Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin, 3–5 June 2002; ACM: New York, NY, USA, 2002; pp. 233–246.
65. Hohenecker, P.; Lukasiewicz, T. Deep Learning for Ontology Reasoning. *arXiv* **2017**, arXiv:1705.10342.
66. Raimond, Y.; Scott, T.; Oliver, S.; Sinclair, P.; Smethurst, M. Use of Semantic Web technologies on the BBC Web Sites. In *Linking Enterprise Data*; Springer US: Boston, MA, USA, 2010; pp. 263–283.
67. Raimond, Y.; Smethurst, M.; McParland, A.; Lowis, C. Using the Past to Explain the Present: Interlinking Current Affairs with Archives via the Semantic Web. In Proceedings of the Semantic Web–ISWC 2013, Sydney, NSW, Australia, 21–25 October 2013; Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 146–161.
68. Mauthe, A.; Thomas, P. *Professional Content Management Systems: Handling Digital Media Assets*; Wiley: Hoboken, NJ, USA, 2005.
69. Rinne, M.; Nuutila, E. Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL. *J. Data Semant.* **2017**, *6*, 57–69.[CrossRef]
70. D’Elia, A.; Perilli, L.; Viola, F.; Roffia, L.; Antoniazzi, F.; Canegallo, R.; Salmon Cinotti, T. A self-powered WSN for energy efficient heat distribution. In Proceedings of the SAS 2016—Sensors Applications Symposium, Catania, Italy, 20–22 April 2016; pp. 1–6.

71. Pizzotti, M.; Perilli, L.; del Prete, M.; Fabbri, D.; Canegallo, R.; Dini, M.; Masotti, D.; Costanzo, A.; Franchi Scarselli, E.; Romani, A. A Long-Distance RF-Powered Sensor Node with Adaptive Power Management for IoT Applications. *Sensors* **2017**, *17*, 1732. [\[CrossRef\]](#)
72. Maarala, A.I.; Su, X.; Riekkii, J. Semantic data provisioning and reasoning for the Internet of Things. In Proceedings of the 2014 IEEE International Conference on the Internet of Things (IOT 2014), Cambridge, MA, USA, 6–8 October 2014; pp. 67–72.



© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).