A Semantic Publish-Subscribe Architecture for the Internet of Things

A Semantic Publish-Subscribe Architecture for the Internet of Things

Journal:	*IEEE Internet of Things Journal*
Manuscript ID	IoT-0998-2016.R1
Manuscript Type:	Regular Article
Date Submitted by the Author:	07-Jun-2016
Complete List of Authors:	ROFFIA, LUCA; University of Bologna, DISI Morandi, Francesco; University of Bologna, ARCES Kiljander, Jussi; VTT Technical Research Center of Finland, D'Elia, Alfredo; University of Bologna, DISI Vergari, Fabio; University of Bologna, ARCES Viola, Fabio; University of Bologna, ARCES Salmon Cinotti, Tullio; University of Bologna, DISI; University of Bologna, ARCES Bononi, Luciano; University of Bologna, DISI
Keywords:	Cyber-Physical Systems < Sub-Area 3: Services, Applications, and Other Topics for IoT, Mobile and Ubiquitous Systems < Sub-Area 3: Services, Applications, and Other Topics for IoT, Semantic Data and Service < Sub-Area 3: Services, Applications, and Other Topics for IoT, Smart Cities < Sub-Area 3: Services, Applications, and Other Topics for IoT, Smart Environment < Sub-Area 3: Services, Applications, and Other Topics for IoT, Service Middleware and Platform < Sub-Area 3: Services, Applications, and Other Topics for IoT

A Semantic Publish-Subscribe Architecture for the Internet of Things

Luca Roffia, Francesco Morandi, Jussi Kiljander, Alfredo D'Elia, *Member*, *IEEE*,
Fabio Vergari, Fabio Viola, *Member*, *IEEE*, Luciano Bononi, and Tullio Salmon Cinotti, *Member, IEEE*

***Abstract*— This paper presents a publish-subscribe architecture designed to support information level interoperability in smart space applications in the Internet of Things (IoT). The architecture is built on top of a generic SPARQL endpoint where publishers and subscribers use standard SPARQL Updates and Queries. Notifications about events (i.e., changes in the RDF knowledge base) are expressed in terms of added and removed SPARQL binding results since the previous notification, limiting the network overhead and facilitating notification processing at subscriber side. A novel event detection algorithm, tailored on the IoT specificities (i.e., heterogeneous events need to be detected and continuous updates of few RDF triples dominate with respect to more complex updates), is presented along with the envisioned application design pattern and performance evaluation model. Eventually, a reference implementation is evaluated against a benchmark inspired by a smart city lighting case. The performance evaluation results show the capability to process up to 68K subscriptions/s triggered by simple single-lamp updates and up to 3,8K subscriptions/s triggered by more complex updates (i.e., 10 to 100 lamps).**

***Index Terms*— Interoperability, Internet of Things, Performance evaluation, Publish-Subscribe, Semantic Event Processing, Smart Space Applications, SPARQL**

I. Introduction

Many research programs in information and communications technologies (ICT) are motivated by the need to close the growing gap between demand and offer of services in our cities, often affected by challenging urbanization processes. Urban facility management, out-of-hospital preventive care, smart grid based energy efficiency, urban mobility and cultural development are some of the main domains calling for services that could change citizen's life, as well as, the attractiveness and development models of our cities. Public administrations, industries, research organizations and opinion makers share the vision that by integrating computing, networking and interaction with physical processes, ICT will enable such services, usually referred as *smart city services* [*1*] [*2*] [*3*]. In general, *smart city services* rely on *event processing infrastructures* that, operating in a closed loop: i) react to changes in the physical environment (i.e., event detection), ii) reason on the system status (i.e., event processing) and iii) actuate changes in the controlled environment. Dealing with heterogeneous interconnected devices distributed into physical environments, such event processing infrastructures can be framed within the *Internet of Things* (*IoT*) domain [*4*] [*5*] [*6*]. *IoT* can be considered as an abstraction of the physical world, where people and devices mutually interact but also interact with natural and artificial physical entities (i.e., things). These entities, in order to be monitored and/or controlled by other devices, must be uniquely identified, and may also be searched through a set of relevant properties or a set of relations with other entities.

The level of interoperability, dynamicity, flexibility, expressivity and extendibility required in *IoT* could be provided by Semantic Web [*7*] based interoperability platforms like the Task Computing Environment (TCE) [*8*], Context Broker Architecture for Pervasive Computing (CoBrA) [*9*] [*10*], Semantic Space [*11*], Semantic middleware for IoT [*12*], Smart objects awareness and adaptation Model (SoaM) [*13*], Amigo [*14*], SPITFIRE [*15*], OpenIoT [*16*] and Smart-M3 [*17*], to name a few. The main drawback of Semantic Web technologies concerns the low level of performance that makes it difficult to achieve responsiveness and scalability required in many *IoT* applications. The main reason for the poor performance is that Semantic Web technologies have been designed to process data sets consisting of big amounts of Resource Description Framework (RDF) [*18*] triples (e.g., Open Linked Data project [*19*]) that evolve constantly but at a much slower rate compared to the rate of elementary events occurring in the physical environment.

To address this limitation, we propose a novel *Semantic Publish-Subscribe* (*SPS*) *Architecture* for the *IoT* built on mainstream research results in Semantic Web technologies and smart spaces [*20*]. This architecture is intended to become the core component of edge computing nodes, supporting IoT gateway and local processing functions, and thus implement interoperability in IoT.

In our earlier work we have proposed a Semantic Interoperability Architecture for Internet of Things [*21*], which divides large-scale Semantic Web-based IoT systems into distributed knowledge bases in order to achieve scalable solution for semantic event processing. The *SPS Architecture* complements this architecture by focusing on enabling real-time semantic event processing within a single knowledge base. Additionally, the *SPS Architecture* builds on top of the authors experience on the development of an open interoperability platform for smart space applications [*22*] [*23*] [*24*] [*25*] [*26*] and it has been heavily influenced by the Smart-M3 semantic interoperability platform [*17*] . The Smart-M3 platform has been adopted, evaluated and extended in past and

current EU projects (e.g., SOFIA, CHIRON, IoE, RECOCAPE, IMPReSS, ARROWHEAD) in partnership with industrial players and the novel *SPS Architecture* presented in this paper is a research result of this work. The SPS Architecture improves the original Smart-M3 platform with a modular system architecture and an efficient *SPARQL Subscription (SUB) Engine.* The key differences of the *SPS Architecture* compared to the Smart-M3 and other state-of-the-art SPARQL subscription processing platforms are presented in more detail in Section VII. The main characteristics of the *SPS Architecture* are following:

- A semantic event is defined as a change in the RDF knowledge base
- Clients are divided into three groups: producers, consumers and aggregators
- Clients use SPARQL updates [*27*] and queries [*28*] (without any extension) respectively to generate and subscribe to semantic events
- A notification includes only the added and removed SPARQL binding results since the previous notification (i.e., the entire set of SPARQL binding results is returned to the subscriber when the subscription is registered)
- Event negation (i.e., the not occurrence of an event within a time interval) is supported
- The architecture is natively parallel

A central component in the *SPS Architecture* is the *SUB Engine* that implements a novel event detection algorithm. The *SUB Engine* has been designed for *IoT* systems where the environment status continuously evolves with frequent fine-grain asynchronous changes (i.e., events) and low latency reactions to these events are required. The *SUB Engine* is the core of the *SPS Architecture* and its architecture represents a major research contribution of this paper. In addition, the novel contribution of the paper includes an application design pattern and a method to evaluate the performance of a *SUB Engine* implementation. The proposed evaluation method consists of a performance model, a set of performance indicators and a benchmark. A prototype implementation of the *SUB Engine* is the result of an engineering effort and it was considered relevant to concretely evaluate the research outcome.

The paper is structured as follows: in Section II the *SPS Architecture* is introduced; in Section III the underpinning *SUB Engine* architecture is presented along with the event detection algorithm; Section IV suggests a performance evaluation method of the *SUB Engine*; Section V provides the details of the reference implementation evaluation and Section VI summarizes the evaluation outcome; in Section VII existing approaches to semantic event processing are summarized and compared with our approach. Conclusions are drawn in Section VIII. A glossary and the details about the performance tests are provided in two appendixes.

II. Architecture

A. Overview

The proposed architecture is inspired by the Smart-M3 concept [*17*] and it consists of a *processing infrastructure*, a set of *primitives* and *clients* (Fig. 1). It is a reduced primitive set architecture, as only two primitives are in principle enough to implement an application: UPDATE and SUBSCRIBE.

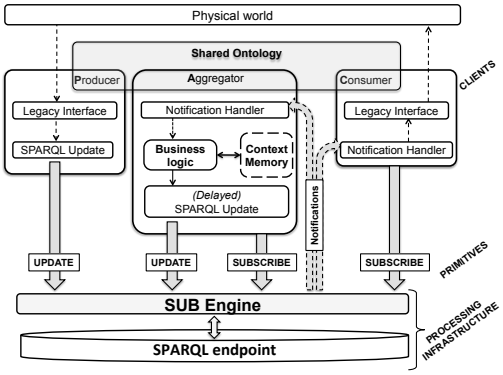

Fig. 1. Semantic Publish-Subscribe Architecture

The UPDATE primitive is a *SPARQL 1.1 Update*. It provides the mean for creating events by inserting, removing, or modifying information inside the SPARQL endpoint RDF store, optionally at a specific time in the future (i.e., namely *Delayed SPARQL Update*). The *Delayed SPARQL Update* is a new contribution with respect to the Smart-M3 original idea. As it will be explained in Section II.C, the SUB Engine will execute the primitive at the specified time and the client (i.e., usually an aggregator) has not to implement any time management mechanism (e.g., starting timers). This allows to simplify the design of the application business logic and to grant the time synchronization of clients through the SUB Engine.

The SUBSCRIBE primitive, represented with a *SPARQL 1.1 SELECT Query* form, provides the mean for a client to be notified on specific events. When it is invoked, it returns the SPARQL binding results. Then, if an UPDATE primitive triggers a notification, to avoid re-transmitting the entire results, the notification (denoted with *R*) contains only the added and the removed SPARQL binding results since the previous notification (i.e., this approach is similar to Istream and Dstream operators used in the SQL based continuous query language proposed by Arasu *et al.* [*29*]):

$$R = \{R_t \setminus R_{t-1},\ R_{t-1} \setminus R_t\}$$

where R_{t-1} and R_t are respectively the binding results before and after the UPDATE primitive that triggered the notification. The advantages of this approach can be appreciated by considering a simple (but at the same time very common) example: an IoT service (i.e., acting as a consumer) plots on a GUI the trend of 1 Million pollution sensors. If a sensor updates its measure, the service is notified with just that

single measure and so it can easily add a new point to the plot. Otherwise, the service would receive 1 Million values and it should compare all of them with the current ones to understand which value has changed. Furthermore, sending one single result, instead of 1 Million results, dramatically reduces the network overhead.

B. Application design pattern

In order to achieve modular, extensible and cost-effective solutions (i.e., by enforcing a clean separation between the physical world and its digital representation) the proposed application design pattern follows:

- Clients are categorized in three sets: *producers*, *consumers* and *aggregators*.
- *Producers* and *consumers* should be kept as simple as possible.
- *Aggregators* implement the application business logic.
- Vocabulary and rules for the semantic coupling of clients are defined by a shared OWL ontology [*30*].

Fig. 1 clarifies the roles of *producers*, *consumers* and *aggregators* in the proposed application design pattern. All three types of clients benefit from the expressivity of SPARQL 1.1. Thus, the engine inherently supports the generation, detection and notification of events of various complexity and granularity levels.

Producers and *consumers* are the bridge between the *physical world* and its digital representation (i.e., that is stored into the *SPARQL endpoint* RDF store). The role of *producers* is to collect and input *physical world* information into the *SPARQL endpoint* RDF store through the *SUB Engine*. *Consumers*, on the other hand, subscribe to events detected and notified by the *SUB Engine* and provide feedback to the *physical world*. *Producers* and *consumers* exchange information with the *physical world* through a *legacy interface*, which is strictly dependent on the physical sensors and actuators technologies. They have no internal memory (i.e., the local memory may only be used for local processing strictly related to the interfaced device/service). The *legacy interface* of a *producer* gathers data from the interfaced sensors (or other input devices/services) and, after some optional local processing, encapsulates these data into an UPDATE primitive (i.e., here is where the raw data is transformed into the semantic format). A *consumer*, on the other hand, waits for notifications (see *Notification handler*) from which it extracts the raw data and forwards it to the *legacy interface* in order to control a specific actuator (or other output devices/services).

The role of *aggregators* is to 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 *business 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*). In both cases it is possible to specify whether an UPDATE primitive has to be executed immediately or at a specific time in the future (see *Delayed SPARQL Update*).

The advantages of the proposed design guidelines are twofold: 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 that are typical in IoT.

C. Time management and event negation

The *SUB Engine* grants time management through the following functional elements:

- A SPARQL function to retrieve the current time (i.e., the Unix time extended to μs). As the time is retrieved on the *SUB Engine* side, this allows the events generated by clients and the notifications sent by the engine to be time stamped with a unique clock (i.e., the function can be used within a UPDATE or SUBSCRIBE primitive). If a hard timing is required on data produced by a client, this is left to the client itself and external synchronization mechanisms have to be implemented.
- The *Delayed SPARQL UPDATE* primitive that allows the clients to schedule a SPARQL Update execution on the *SUB Engine* side at a specified time.

The *SUB Engine* has also the capability to handle *event negation* (i.e., the notification of events that did not occur within a specified time interval). This is a relevant feature for example in supervising systems [*31*] where the "not-occurrence" of an expected event is itself the event to be detected and notified. In the proposed design pattern, this feature is provided by a *Delayed SPARQL UPDATE*: if a client invokes a *Delayed SPARQL UPDATE* primitive at time t, then at time $t+\Delta t$ the *SUB Engine* will execute the UPDATE primitive and it will generate results if and only if the expected event has not occurred. For example, detecting if the transition of a variable *var* to a desired *value X did not occur* within a specified time interval Δt can be achieved using a *Delayed SPARQL UPDATE* having in the WHERE clause two triple patterns referring to the variable under control: one representing the *value* itself (e.g., <*var*, hasValue, *X*>) and the other one representing the *timestamp* of its last update (e.g., <*var*, hasTimeStamp, *timestamp*>).

D. Application design and event negation example

In order to clarify the concepts and the application design pattern above discussed, a simple example follows: in a smart lighting scenario where each lamp-post is equipped with a presence sensor, every lamp must be turned on when the associated sensor detects a presence, while it must be switched off when *no presence* has been detected for Δt seconds (i.e., this is an event negation example). Fig. 2 shows the sequence

diagram of this application implemented with the following three clients:

1. *Presence sensor* (*producer*): it keeps up-to-date in the SPARQL endpoint RDF store both the presence sensor status (TRUE/FALSE) and its timestamp.
2. L*amp actuator* (*consumer*): it is subscribed to changes in the lamp status (ON/OFF).
3. *Smart lighting* (*aggregator*): it implements the application business logic. It is subscribed to changes in the presence sensor status and it updates the lamp status accordingly: if a presence is detected (i.e., presence=TRUE), the status of the lamp is updated with the value ON, otherwise (i.e., presence=FALSE) a delayed UPDATE primitive is issued to turn OFF the lamp after Δt seconds.

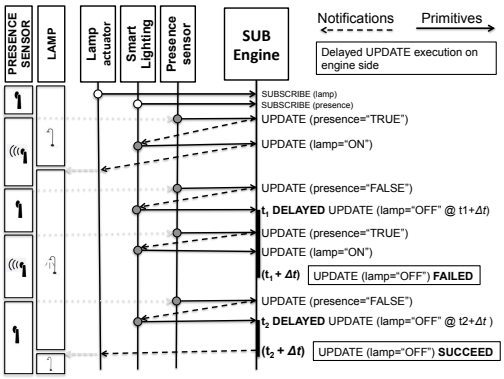

Fig. 2. Sequence diagram of the smart lighting example

As shown in Fig. 2, when the presence sensor returns FALSE for the first time, a first delayed UPDATE primitive is issued by the *Smart lighting* client at time t1. The *SUB Engine* executes this UPDATE primitive at time t1+Δt but the status of the lamp is not updated at time t1+Δt because the presence sensor status (therefore its timestamp) changed meanwhile. On the other hand, when the *SUB Engine* executes the second delayed UPDATE primitive at time t2+Δt, as neither the presence sensor status nor its timestamp have been modified in the interval [t2, t2+Δt], the lamp status is updated (OFF) and consequently the *Lamp actuator* is notified. The actual syntax of the UPDATE and SUBSCRIBE primitives for this example are in [*32*]. This mechanism allows the clients to delegate the time management to the *SUB Engine,* thus avoiding any further action on the client side once a delayed UPDATE primitive has been issued (i.e., no timers activations/deactivations are needed on the client side).

III. SPARQL SUBSCRIPTION ENGINE

Two major research contributions of this work are presented in this section: the SPARQL Subscription Engine (SUB Engine) internal architecture and the implemented event detection algorithm. More in detail, the SUB Engine architecture, how the SUB Engine processes the UPDATE and SUBSCRIBE primitives and the event detection algorithm (along with an analysis of its time complexity) are all presented in this section.

As Fig. 3 shows, the SUB Engine consists of the following main components:

- A *scheduler* listening for requests incoming from two FIFO queues: the *UPDATE Request Queue* (URQ) and the *SUBSCRIBE Request Queue* (SRQ).
- One SPARQL Processing Unit (SPU) for each SUBSCRIBE Request received.

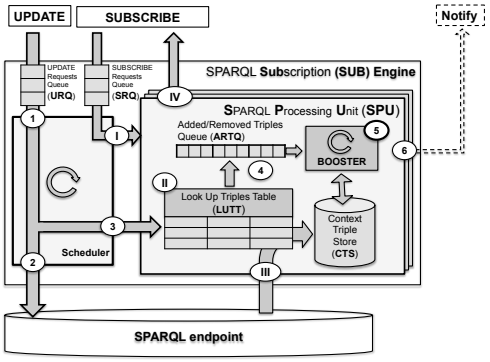

Fig. 3. SPARQL Subscription Engine Architecture

A SPU implements the event detection algorithm (see BOOSTER in Fig. 3) and notifies just the subscriber originating the request. Each SPU holds its own *Context Triple Store* (CTS). This is a subset of the entire SPARQL endpoint RDF store and it is defined as the "*union of all RDF triples matching at least one of the triple patterns of the SUBSCRIBE graph pattern"*. In the proposed architecture, the CTS is related to the SPARQL endpoint RDF store as the cache memory is related to main memory in a traditional computer, while each SPU corresponds to a processor with its own cache in a traditional multiprocessor system. The role and importance of the CTS can be better appreciated considering the following common IoT scenario. With reference to the example shown in Section II.D, let us consider a smart city equipped with presence sensors, one for each lamp-post in the city (e.g., 50K sensors). The SPARQL endpoint RDF store contains at least 50K RDF triples, each of them representing the status of a presence sensor. A smart space application is interested in monitoring the presence of cars close to a specific lamp-post (e.g., at the entrance of a tunnel). The application is so subscribed to a specific presence sensor and consequently the CTS will contain just one RDF triple. As it will be better appreciated in the following sections, this will boost the SPU performance, allowing a SPU to search for changes in a very reduced set of RDF triples (e.g., one RDF triple).

A. SUBSCRIBE primitive processing

The scheduler allocates a new SPU, and therefore a CTS, whenever a SUBSCRIBE request is extracted from the SRQ (see I in Fig. 3). Each SPU includes a Look Up Triples Table (LUTT). The purpose of the LUTT is to filter out, as early as

possible, those triples that are not relevant for the given subscription. In order to build the LUTT, the SPU extracts all the triple patterns from the SUBSCRIBE query pattern. Then the variables contained in the extracted triple patterns are substituted with wildcards and are inserted into the LUTT (see II in Fig. 3). An example of a LUTT created from a simple subscription is shown in Fig. 4.

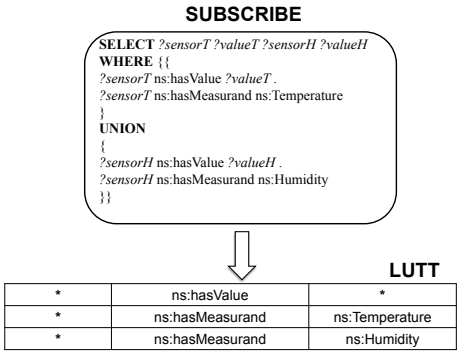

Fig. 4. A SUBSCRIBE primitive example and the associated LUTT

Furthermore, the SPU splits the SPARQL query graph pattern into basic graph patterns. Each basic graph pattern is then associated to a new SELECT query (i.e., we define this as a *sub-query*), which includes the references to its own basic graph pattern variables only (Fig. 5).

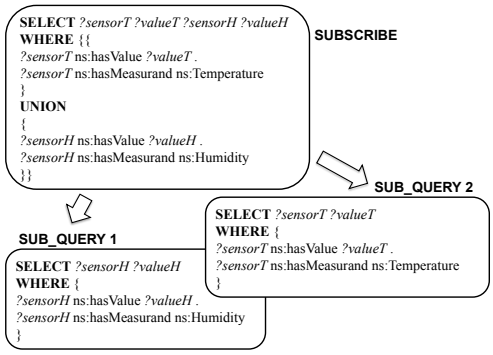

Fig. 5. SUBSCRIBE queries are split into sub-queries based on basic graph patterns. Each sub-query refers only to the variables included in its own basic graph pattern

All the RDF triples matching the LUTT are retrieved from the SPARQL endpoint and stored into the CTS (see III in Fig. 3). Then the SPARQL query is issued by the SPU on the CTS and the SPARQL binding results are sent to the subscriber (see IV in Fig. 3). In this way the subscriber is aware of the initial context and it is therefore in the position to track the evolution of such context through all subsequent notifications.

B. UPDATE primitive scheduling and processing

Fig. 6 is a stepwise view of the UPDATE handling workflow, which is partitioned into two phases: scheduling and processing. The workflow can be carried out in sequential or in parallel mode.

The sequential mode is useful for extracting performance indicators of each workflow step (see Section IV). It also enables semantic event processing to be deployed on low cost single core platforms (e.g., on a Raspberry Pi).

In parallel mode these two phases are pipelined. Furthermore, all SPUs run concurrently and within each SPU the processing phase steps are executed sequentially. Thus, with $n+1$ cores, n active subscriptions could be processed in parallel (i.e., one core for the scheduler and one core for each active SPU). In principle, with this processing model, scheduling may shadow processing, so that the proposed engine could not induce any overhead on the SPARQL endpoint updates processing.

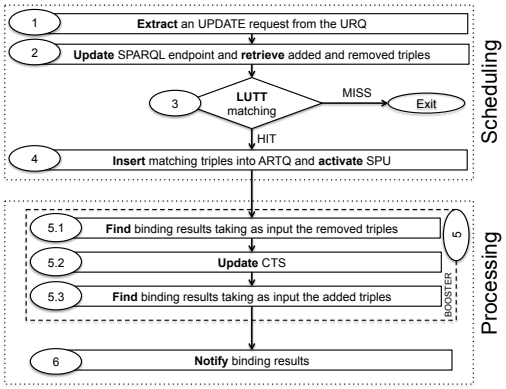

Fig. 6. UPDATE primitive scheduling and processing: SUB Engine workflow

As shown by Fig. 6 and with reference to Fig. 3, the scheduling phase goes through the following steps: a new UPDATE request is fetched from the input FIFO queue URQ (1), then the SPARQL endpoint RDF store is updated and the added and removed triples (if any) are retrieved (2) and filtered through the LUTT (3). Filtering is performed as a simple string matching on the added and removed triples against the LUTT content (i.e., wildcards mean "any string"). If no match is found, the processing ends (MISS) otherwise all added and removed triples matching the LUTT (HIT) are inserted into the *Added/Removed Triples Queue* (ARTQ) and the associated SPU is activated (4). Then the BOOSTER component of the activated SPU implements the event detection algorithm (5) taking as input the ARTQ triples. Eventually, all detected events (if any) are notified to the subscriber (6).

C. Event detection algorithm

The event detection algorithm implemented by the BOOSTER has been designed to match the typical profile of

Internet of Things applications, which mostly react to physical processes and share the following specificities:

- Selective updates of few triples dominate with respect to more complex updates (typical examples of this kind are continuous and asynchronous updates of sensor values).
- Large sets of heterogeneous events need to be detected.
- Average CTS size is significantly smaller compared to the SPARQL endpoint RDF store size.
- Multi-domain scenarios (i.e., smart cities) become increasingly relevant, implying that only a small subscriptions fraction is likely to be involved by the same update.

Thus, according to the *most important and pervasive principle of computer design "make the common case fast"* quantified by the Amdahl's Law [*33*] [*34*], selective updates need to be processed as fast as possible by selective subscribes, while less frequent updates, such as run time ontology extensions or massive RDF data uploads, are allowed for a much longer processing time. Accordingly, the following requirements were assumed for the event detection algorithm and for the SUB Engine:

- At the occurrence of an update, every SPU has to check if any of the updated triples hits the LUTT. Therefore, this check should be as fast and effective as possible: fast because it concerns all subscriptions and effective to minimize both the number of subscriptions and triples that are candidate to produce notifications.
- If an update hits the LUTT of a SPU, then the reflected context change may trigger a notification. In this case, only the added and removed SPARQL binding results since the previous notification must be found and sent to the subscriber.
- The engine performance should scale with the fraction of subscriptions concerned with the updates. Highly parallel architectures may help in meeting this requirement, as subscriptions are logically independent. Therefore subscriptions may be processed in any order, and, particularly, they might run in parallel as long as they do not compete for the same resource (e.g., CPU or memory bus). As every SPU has its own CTS, each subscription "hit" by an update (i.e., that needs to be processed) may be mapped onto a separate processing unit or thread of a distributed or multicore architecture. The price to be paid is measured in terms of CTS size and number of processing units required.

Once activated, for each triple extracted from the ARTQ, a SPU makes a separate query on the CTS and the triple content is used to bind as many variables as possible before performing the query. By doing so, the query processing is optimized in two ways: the number of variables is reduced and, furthermore, if the query produces results these are for sure bindings to be notified (i.e., these are removed bindings if the triple is a removed one (or an added one matching a FILTER NOT EXITS or MINUS pattern) or added bindings if the triple is an added one (or a removed one matching a FILTER NOT EXITS or MINUS pattern)). Therefore, there is no need to compare the current bindings with the previous ones to find out how the results have changed. Avoiding this comparison is very relevant for the algorithm performance, particularly if just few bindings out of many changed as a consequence of an update (i.e., this is very common in IoT like systems where selective updates of few triples, like continuous and asynchronous updates of sensor values, dominate with respect to more complex updates).

More in detail, during the processing phase, a SPU goes through the following steps (see Fig. 6):

1. It finds the binding results using the removed triples as input (step 5.1).
2. It updates the CTS with the added and removed triples (step 5.2).
3. It finds the binding results using the added triples as input (step 5.3).
4. It notifies the subscriber on the binding results (if any) (step 6).

The pseudo-code of the algorithm implemented by the BOOSTER to find both the removed and added binding results (steps 5.1 and 5.3) follows. For each procedure called by the algorithm (see lines 3,4 and 5), a description is given along with an analysis of its time complexity. Eventually, a discussion on the overall algorithm time complexity is presented in the next subsection.

Algorithm: **Find** binding results

Input: set of SPARQL queries on basic graph patterns (*queries*) with cardinality $N_{QUERIES}$
Input: set of RDF triples (*triples*) with cardinality $N_{TRIPLES}$
Output: SPARQL binding results (*results*)

Definitions:
- *triple*: an RDF triple
- *query*: a SPARQL query on a basic graph pattern
- *bindings*: SPARQL binding results

```
0.  results = Ø
1.  For each query in queries do
2.      For each triple in triples do
3.          <query*, bindings*> = Match (query, triple)
4.          bindings = Query (query*)
5.          results = Merge (results, bindings, bindings*)
6.      End For
7.  End For
8.  Return results
```

Procedure: **Match** (*query*, *triple*)

In order to bind as many variables as possible (see *bindings** at line 3), a string matching is performed on each triple pattern of the *query* against the *triple*. For example, if the triple pattern is *<?class* , rdf:type , rfds:Class> and the *triple* is *<ns:ClassURI* , rdf:type, rdfs:Class> then the procedure binds and replaces (within the *query*) the variable *?class* with the value *ns:ClassURI*. All the bindings and the corresponding modified query are returned (see *<query**, *bindings**> at line

3). The procedure time complexity is, in the worst case, equal to the time complexity of a string comparison up to three times (i.e., subject, predicate and object) for each *triple pattern* of the *query*.

Procedure: **Query** (*query*)

The simplified SPARQL *query* obtained in the previous step (see *query** at line 3) is executed on the CTS. In the best case (i.e., all the variables have been bound at line 3), the SPARQL query is replaced by an *ASK query*. The algorithm assumes that the time complexity of a *query* grows with the number of variables (i.e., the same query is faster if some variables have been bound), with the number of *triple patterns* and with the number of *bindings* results (i.e., a higher time is also required to receive the results over the communication channel). It is not possible to give a priori estimation of the SPARQL query time complexity as it depends on many factors, like the RDF store size, the form of the query and the SPARQL endpoint implementation.

Procedure: **Merge** (*results*, *bindings*, *bindings**)

The *bindings** found at line 3 and the *bindings* found at line 4 are merged together with the current binding *results* (see *results* at line 5). These are removed bindings if the input *triples* are removed ones (or added ones matching a FILTER NOT EXITS or MINUS pattern) or added bindings if the input *triples* are added ones (or removed ones matching a FILTER NOT EXITS or MINUS pattern). The time complexity of this procedure is the time to insert the two bindings set (see *bindings* and *bindings** at line 5) into the *results* bindings set.

D. Event detection algorithm time complexity

In a common IoT application, we can assume the following:

- $N_{QUERIES}$ = 1 (i.e., the SUBSCRIBE primitive has no UNION construct)
- $T_{MATCH} + T_{MERGE} << T_{QUERY}^{*}$ (i.e., the former two refer to string comparisons in memory and merging of in memory data structure, while the latter is related to executing a query on a RDF store)

The algorithm time complexity can be so estimated as:

$$T_{ALG} \sim N_{TRIPLES}\left(T_{MATCH} + T_{QUERY^*} + T_{MERGE}\right) \sim N_{TRIPLES} T_{QUERY^*} \quad (\boldsymbol{A1})$$

The optimization introduced by the algorithm can be appreciated with reference to the naïve Smart-M3 algorithm (i.e., the former implementation described in [*22*]). In order to detect changes in the RDF store and notify such changes to the subscriber, the naïve algorithm queries the SPARQL endpoint, retrieving all the bindings results (i.e., N), and it compares these results with the current ones (i.e., N^2 comparisons of bindings are needed). Defining T_{QUERY} as the time to perform the query and T_{CMP} as the time required by a single binding comparison, the time complexity of such algorithm can be expressed as:

$$T_{NAIVE} = N^2 T_{CMP} + T_{QUERY} \quad (\boldsymbol{A2})$$

The speedup introduced by the proposed algorithm with respect to the Smart-M3 naïve algorithm can be estimated assuming the following:

- T_{QUERY} = Q T_{QUERY}* (i.e., where Q >> 1, as the same SPARQL query, but with a lower number of variables, is always faster than the original one).
- $N_{TRIPLES}$ << N (i.e., the update of a sensor data usually corresponds to 1-2 triples, while the number of bindings can be a very huge number equals to the number of sensors within the system, 10^4 – 10^6 or more).
- The algorithm is executed twice to find both the added and removed bindings results.

With reference to (A1), (A2) and the above hypotheses, the speedup can be expressed as:

$$Speedup = \frac{T_{NAIVE}}{2\,T_{ALG}} = \frac{N^2 T_{CMP} + Q T_{QUERY^*}}{2 N_{TRIPLES} T_{QUERY^*}} = \frac{N^2}{2\,N_{TRIPLES}} \frac{T_{CMP}}{T_{QUERY^*}} + \frac{Q}{2\,N_{TRIPLES}} \quad (\boldsymbol{A3})$$

The optimization introduced by the algorithm is proved with respect to the SUB Engine reference implementation evaluated in Section V. A first impression of the speedup can be provided considering the following: an IoT application aims at detecting changes in the status of any presence sensor among 10^5 sensors of the same type (i.e., this is optimistic as the sensors could be many more). Each sensor reading corresponds to the update of one RDF triple (i.e., $N_{TRIPLES}$ = 1). The bindings results returned by the query are in this case equal to 10^5 (N = 10^5). Supposing a reasonable case where $T_{CMP} \approx 10^{-3}$ T_{QUERY} (e.g., µs versus ms) and considering the worst case of Q = 1, the speedup results 5 x 10^6.

IV. Performance Evaluation Method

Semantic event processing can be applied to scenarios, which might greatly differ along several vectors including knowledge base size, events complexity, number of active subscribers and subscriptions granularity. In order to master this diversity and exploit the innovation potential of semantic event processing in the Internet of Things domain, simple methods are needed to design the workload and achieve the best workload/platform tradeoff.

Frameworks, benchmarks and methods for performance evaluation of Semantic Web systems, in general, and Semantic Publish-Subscribe systems, in particular, have been proposed in the literature. Unfortunately, these methods are not suitable for analyzing the performance of the SPS Architecture in detail. In fact, the formers (e.g., [*35*] [*36*] [*37*] [*38*]) are mainly designed to evaluate the performance of a SPARQL endpoint on answering a predefined set of queries with reference to several data sets and they do not include any SPARQL Update. The latter are focused on analyzing the performance of specific publish-subscribe systems (e.g., [*39*] and [*40*]) whose architecture and primitives differ from the one presented in this paper. To this end, a method is envisioned to compare and predict the performance of different

implementations of the SPS Architecture, focusing on the SUB Engine (i.e., the clients-engine communication is not evaluated in this paper) and on its specific features (i.e., the LUTT filtering effectiveness and the parallel processing support). The method is based on a benchmark, a performance model and a set of key performance indicators (KPIs).

A. Benchmark

In semantic event processing, notifications may be triggered by the occurrence of events of various granularity levels (being, in our solution, the expressivity of the SPARQL language the limit to specify such granularity). Events specification patterns may range from "a sensor matches a specific value range" to a mix of patterns concerning sensors spread anywhere (e.g., in a specific geographical area, in a street or over an entire city), or characterized by particular time and/or space relations over a specific set of data. The benchmark should mimic the fine events granularity expected in IoT applications typically characterized by frequent updates of a single data-property value (i.e., a sensor value) and less frequent concurrent updates of sets of properties. In particular, a benchmark is defined with reference to a specific OWL ontology and it is composed by a set of experiments, where each experiment is characterized by:

- An *Update Profile* (**U**), defined as a set of *n* UPDATE primitives U_i.
- A *Subscription Profile* (**S**), defined as a set of *m* SUBSCRIBE primitives S_j.
- A *Number of Updated Triples Profile* (**[*Nu*]**), defined as vector of *n* elements**,** where each element Nu_i is the amount of triples updated by U_i. The average number of triples updated by a single UPDATE primitive within an experiment can be accordingly expressed as:

$$\boldsymbol{Nu_{AVG}} = \frac{1}{n}\sum_{i=1}^{n} Nu_i \qquad (1)$$

- A *LUTT matrix* (**[*h*]**), defined as a Boolean matrix of *n* x *m* elements where a generic element $h_{i,j} = 1$ if at least one of Nu_i triples pass LUTT_j, otherwise $h_{i,j} = 0$. The LUTT matrix may be considered a sparse matrix and the amount of "zeros" increases while the addressed scenario becomes more and more multi-domain (e.g., as it is expected in smart cities). An indication of the LUTT filtering effectiveness in a particular scenario is given by the *LUTT Hit Rate* defined as:

$$\boldsymbol{LHR}\ (\%) = \frac{100}{m \text{ x } n}\sum_{i=1}^{n}\sum_{j=1}^{m} \boldsymbol{h_{i,j}} \qquad (2)$$

The lower the *LHR*, the larger is the LUTT contribution to the engine performance level in the addressed scenario.

B. Performance model

The performance model subdivides the elapsed time ($\mathbf{T}_{\text{ELAPSED}}$) of an experiment in its most relevant components in order to analyze the impact of each component on the overall performance, understand if an implementation meets the requirements of a specific application, identify possible bottlenecks, remove the observations overhead from the KPIs evaluation and predict the performance level achievable with a parallel computing infrastructure. When the engine runs in sequential execution mode, five not-overlapping timing components may be recognized in the elapsed time of an experiment:

$$\mathbf{T}_{\text{ELAPSED}} = \mathbf{T}_{\text{OVERHEAD}} + \mathbf{T}_{\text{UPDATE}} + \mathbf{T}_{\text{LUTT}} + \mathbf{T}_{\text{BOOSTER}} + \mathbf{T}_{\text{NOTIFY}} \qquad (3)$$

$\mathbf{T}_{\text{UPDATE}}$ is the total latency time of the SPARQL endpoint occurring when the *Update Profile* is applied.

Given the *Update* and *Subscription Profiles*, respectively with cardinality *n* and *m*, $\mathbf{T}_{\text{LUTT}}$ is the time paid to check *n* times all the *m* LUTTs.

$\mathbf{T}_{\text{BOOSTER}}$ is the time spent by all *m* BOOSTERs to search their CTSs for the results to be notified to the subscribed clients.

$\mathbf{T}_{\text{NOTIFY}}$ is the time spent by all *m* SPUs to forward the results to the communication interface.

$\mathbf{T}_{\text{OVERHEAD}}$ is the time required to control the experiment and to collect the timing information (i.e., we define. the net time of an experiment as $\mathbf{T}_{\text{TOTAL}} = \mathbf{T}_{\text{ELAPSED}} - \mathbf{T}_{\text{OVERHEAD}}$).

C. Key Performance Indicators (KPIs)

Five key performance indicators (KPIs) are proposed to compare different implementations and predict if the requirements of new scenarios can be met by a specific engine implementation. KPIs values depend on the benchmark and they can be estimated starting from the model parameters deduced by the experimental results. Given the *Update* and *Subscribe Profiles*, respectively with cardinality *n* and *m*, the proposed KPIs are shown in Table 1.

Table 1 Key Performance Indicators (KPIs)

Average number of updates processed per unit time	**Average number of triples processed per unit time**
$\boldsymbol{Ups} = \frac{n}{\mathbf{T}_{\text{TOTAL}}}$	$\boldsymbol{Tps} = Nu_{AVG}\ \boldsymbol{Sps}$
Average number of subscriptions processed per unit time	**Engine to SPARQL Endpoint impact factor**
$\boldsymbol{Sps} = m\ \boldsymbol{Ups}$	$\boldsymbol{E2E} = \frac{\mathbf{T}_{\text{TOTAL}} - \mathbf{T}_{\text{UPDATE}}}{\mathbf{T}_{\text{UPDATE}}}$
Event notification latency range	
$\boldsymbol{NL_{min}} = \boldsymbol{min}\{tl_{i,j}+tb_{i,j}+te_{i,j} \mid i = 1..n, j = 1..m \wedge te_{i,j} \neq 0\}$ $\boldsymbol{NL_{max}} = \boldsymbol{max}\{tl_i+tb_i+te_i \mid i = 1..n \wedge te_i \neq 0\}$	

Ups is an indicator of how many updates are processed per unit time in average, while ***Sps*** states how many subscriptions are processed per unit time and therefore it is directly related to the *Subscription Profile* cardinality. But neither ***Ups*** nor ***Sps*** considers events complexity. Therefore, as the time complexity of the algorithm presented in Section III.C depends on the number of triples processed, ***Tps*** is introduced to provide an indication of the computational load in terms of average number of triples processed per unit time.

The notification latency is also a relevant quality factor of a publish-subscribe engine. Therefore ***NL*** is proposed as a

measure of the time span between updates and notifications (i.e., if events are detected). ***NL*** lower bound (***NL$_{min}$***) may be reached when the engine works in parallel mode. In this case, the notification latency of SUBSCRIBE S_j triggered by UPDATE U_i cannot be less than the sum of the LUTT filtering time ($tl_{i,j}$), the BOOSTER time ($tb_{i,j}$) and the time required to send the result to the communication interface ($te_{i,j}$). On the contrary, ***NL*** upper bound (***NL$_{max}$***) occurs when the last SPU notifies its client in sequential mode.

Eventually, ***E2E*** is motivated by the consideration that the engine sits on top of a SPARQL endpoint. This KPI tells which performance penalty has to be paid in order to add the semantic event detection and notification capability to a SPARQL endpoint. The closer this KPI is to zero, the lower is the overhead introduced by the SUB Engine on the underneath SPARQL endpoint.

V. Reference Implementation Evaluation

The open source reference implementation [*41*] evaluated in this paper extends the Smart-M3 implementation released in 2012 [*22*] by introducing the following main novelties: the algorithm presented in Section III.C, the parallelization of SPUs, the delayed SPARQL primitive and the Virtuoso [*42*] support. The Context Triple Store (CTS) is based on RedLand [*43*] [*44*] running in RAM, while the SPARQL endpoint can be one of RedLand supported storages (e.g., hashes, Berkeley DB, Virtuoso). The Smart-M3 protocol (i.e., Smart Space Access Protocol (SSAP) [*17*]) has been extended to support the two new primitives (i.e., UPDATE and SUBSCRIBE), and otherwise was left unchanged to maintain backward-compatibility. Other suitable protocols could be, for example, the Knowledge Sharing Protocol (KSP) [*45*] and the Constrained Application Protocol (COAP) [*46*]. Developers can benefit from a set of open source APIs, available in several programming languages (i.e., Python, C, C#, Java, PHP, JavaScript) that make the proposed implementation multi-language and multi-platform.

A. Reference ontology

The benchmark designed to evaluate the reference implementation is inspired by a public lighting system of a small city with large, medium, small and very small roads (i.e., roads with up to 100, 50, 25 and 10 lamp-posts). Table 2 provides the details about the ontology and the SPARQL endpoint RDF store size (i.e., number of RDF triples).

Table 2 Benchmark knowledge base

OWL Ontology T-Box content				
Classes				27
Individuals				26
Object properties				16
Datatype properties				8
OWL Ontology A-Box content (lamp-posts instances)				
Road types	**N_{LAMP}/Road**	**Roads**	**Lamp-posts (Sensors)**	**RDF Triples**
Very small	10	100	1K (2K)	35K
Small	25	100	2.5K (5K)	88K
Medium	50	100	5K (10K)	175K
Large	100	10	1K (2K)	35K
	Total	**310**	**9.5K (19K)**	**334K**

Altogether the city has 9500 posts (i.e., represented by 334K RDF triples), and each post is supposed to be equipped with a lamp and two sensors (i.e., temperature and presence). Each road and each lamp within a road are identified by a URI respectively in the form: ROAD_URI_X and LAMP_URI_X_Y, where X is a road identifier (i.e., in the range from 1 to 310), while Y is a lamp identifier within a road (i.e., Y varies from 1 to N_{LAMP}, where N_{LAMP} is the amount of lamp-posts in road X). Each lamp is characterized by a status (i.e., ON, OFF, BROKEN), a dimming value (i.e., 0-100 %) and a type (i.e., LED, TRADITIONAL). Each post is identified by its geographical position (i.e., latitude and longitude), while each sensor is represented by a set of properties: the type (i.e., TEMPERATURE, PRESENCE), the unit of measurement (i.e., °C/°F, BOOLEAN), the value (e.g., "32", "True", "False") and a timestamp (i.e., expressed as Unix time extended to μs).

B. Experiments

The benchmark considers two types of UPDATE primitives (see U_{LAMP}(X,Y) and U_{ROAD}(X) in Table 3) and two types of SUBSCRIBE primitives (see S_{LAMP}(X,Y) and S_{ROAD}(X) in Table 3), where **X** indicates the index of a road, while **Y** indicates the index of a lamp post within a road.

Table 3 UPDATE and SUBSCRIBE primitives from the benchmark

UPDATE primitives
U_{LAMP}(X,Y)
INSERT {LAMP_URI_X_Y ns:hasDimmingValue "100"} DELETE {LAMP_URI_X_Y ns:hasDimmingValue ?dimming} WHERE {LAMP_URI_X_Y ns:hasDimmingValue ?dimming}
U_{ROAD}(X)
INSERT {?lamp ns:hasDimmingValue "100"} DELETE {?lamp ns:hasDimmingValue ?dimming} WHERE {?lamp ns:hasDimmingValue ?dimming . ?post ns:hasLamp ?lamp . ?road ns:isConnectedTo ?post . FILTER(?road = ROAD_URI_X)}
SUBSCRIBE primitives
S_{LAMP}(X,Y) (i.e., fine-grain)
SELECT ?dimming WHERE {LAMP_URI_X_Y ns:hasDimmingValue ?dimming}
S_{ROAD}(X) (i.e., coarse-grain)
SELECT ?lamp ?dimming WHERE { ?lamp ns:hasDimmingValue ?dimming . ?post ns:hasLamp ?lamp . ?road ns:isConnectedTo ?post . FILTER(?road = ROAD_URI_X)}

U_{ROAD}(X) is used to set to 100% the dimming value of all the lamps of road X and U_{LAMP}(X,Y) is used to set to 100% the dimming of lamp Y within road X. S_{LAMP}(X,Y) subscriptions (i.e., fine-grain) are sensitive to the update of the dimming value of lamp Y within road X, while S_{ROAD}(X) subscriptions (i.e., coarse-grain) are sensitive to the update of the dimming value of any lamp of road X. Table 4 compares the LUTT content and the CTS size of the two subscriptions.

Table 4 LUTT content and CTS size for fine-grain and coarse-grain subscriptions

LUTT Content			**CTS Size (Triples)**
S_{LAMP}(X,Y) (i.e., fine-grain)			**1**
LAMP_URI_X_Y	ns:hasDimmingValue	*	1

S_{ROAD}(X) (i.e., coarse-grain)			≈ 19K
ROAD_URI_X	ns:isConnectedTo	*	10,25,50,100[1]
*	ns:hasLamp	*	9500[2]
*	ns:hasDimmingValue	*	9500[2]

Two experiments (named **LAMP** and **ROAD**) based on two *Update Profiles* (named $\mathbf{U}_{LAMP}$ and $\mathbf{U}_{ROAD}$) with the same knowledge base (see Table 2) and the same *Subscription Profile* ***S*** are considered. ***S*** includes 1000 fine grain and 4 coarse grain subscriptions (i.e., m = 1004). The formal specification of the *Subscription Profile* ***S*** follows:

$$\boldsymbol{S} = \{S_j \equiv S_{LAMP}(X,Y) |\ j=10(X-1)+Y,\ X \in \{1..5\} \wedge Y \in \{1..10\}\}$$
$$\cup \{S_j \equiv S_{LAMP}(X,Y) |\ j=100+25(X-101)+Y,\ X \in \{101..104\} \wedge Y \in \{1..25\}\}$$
$$\cup \{S_j \equiv S_{LAMP}(X,Y) |\ j=200+50(X-201)+Y,\ X \in \{201..203\} \wedge Y \in \{1..50\}\}$$
$$\cup \{S_j \equiv S_{LAMP}(X,Y) |\ j=300+100(X-301)+Y,\ X \in \{301..307\} \wedge Y \in \{1..100\}\}$$
$$\cup \{S_j \equiv S_{ROAD}(X) | (j, X) \in \{(1001,6),(1002,105),(1003,204),(1004,308)\}\}$$

Table 5 shows that, with the above-defined *Subscription Profile*, a notification is triggered if the dimming value of any of the 1185 lamps is updated.

Table 5 Subscription profile

	Number of subscribers		
Road type	**S_{LAMP} (X,Y)**	**S_{ROAD} (X)**	**Monitored lamps**
Very small	50	1	60
Small	100	1	125
Medium	150	1	200
Large	700	1	800
Total	**1000**	**4**	**1185**

Both the experiments consist of 310 updates (i.e., $\boldsymbol{n}$ **= 310**). Therefore, even if the cardinality of both update profiles is the same, in the $\mathbf{U}_{LAMP}$ *profile* each producer updates one lamp per road (i.e., at the end of the experiment 310 lamps are updated) while in the $\mathbf{U}_{ROAD}$ *profile* each producer updates all the lamps of an entire road (i.e., at the end of the experiment 9.5K lamps are updated). The formal definition of the two *Update Profiles* follows:

$$\boldsymbol{U_{ROAD}} = \{U_i \equiv U_{ROAD}(i) \mid i \in \{1..310\}\}$$
$$\boldsymbol{U_{LAMP}} = \{U_i \equiv U_{LAMP}(i,1) \mid i \in \{1..310\}\}$$

For both the *Update Profiles*, the number of triples per update (i.e., **[*Nu*]** vector) is shown in Table 6.

Table 6 [***Nu***] vectors (number of triples per update of the two experiments)

	1...100	**101...200**	**201...300**	**301...310**
ROAD	10	25	50	100
LAMP	1			

Table 7 is a compact representation of two (310 x 1004) LUTT matrices. Columns are labeled with the subscription indexes and represent either a single subscription (for the *LAMP profile*) or a set of N_{LAMP} subscriptions (for the *ROAD profile*). Rows are labeled with the update indexes. The last row represents collectively all rows not explicitly labeled above. Same notation (last column) is used for the not explicitly labeled columns. From vector **[*Nu*]** and matrix **[*h*]**, the values of $\boldsymbol{Nu_{AVG}}$ and ***LHR*** defined in Section IV.A by (1) and (2) are:

- **ROAD** $\boldsymbol{Nu_{AVG}} = 31$ ***LHR*** = 0,72 %
- **LAMP** $\boldsymbol{Nu_{AVG}} = 1$ ***LHR*** = 0,40 %

Table 7 [***h***] block matrices (a compact representation of the LUTT matrixes of the two experiments)

ROAD	1...10	11...20	21...30	31...40	41...50	51...75	76...100	101...125	126...150	151...200	201...250	251...300	301...400	401...500	501...600	601...700	701...800	801...900	901...1000	1001...1004	...
LAMP	1	11	21	31	41	51	76	101	126	151	201	251	301	401	501	601	701	801	901		
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
101	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
102	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
103	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
104	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
201	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
202	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
203	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
301	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
302	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
303	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
304	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
305	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
306	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
307	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

These parameters show that the *ROAD profile* produces a higher computational load with respect to the *LAMP profile*. Given the *Update* and *Subscribe* profiles, the following amounts of notifications are sent to the subscribers during the reported experiments:

- **ROAD** 1004 notifications (all S_j are triggered)
- **LAMP** 23 notifications (19 S_{LAMP} subscriptions and 4 S_{ROAD} subscriptions are triggered)

C. Test bed and methods

Both the experiments run on a test bed consisting of:

- a machine hosting both the SUB Engine and the SPARQL endpoint. This machine is an Intel(R) Core(tm) i7-2630QM CPU @ 2.00GHz × 8 cores, 8 GB ram, Ubuntu 12.04; the SPARQL endpoint in all tests is Virtuoso
- a remote multithreading C# client application (i.e., simulator) running on a Windows XP Virtual Box machine (4 GB Ram, 1 CPU, execution cap 100%). The client machine is a MacBook Pro, Intel Core i7 2.2 GHz, 16 GB ram. The simulator is connected to the SUB Engine through a 100 Mbps LAN.

First of all, the ontology (see Table 2) is loaded on the RDF store. After that, the simulator starts all the 1004 subscribers (i.e., each of them running on a separated thread). Once all the subscribers are up and running, the simulator sequentially issues all the UPDATE primitives to the SUB Engine that is configured to run in sequential mode (i.e., one core is used for both the scheduler and all the SPUs). In this way, at the end of

[1] the number of triples depends on number of lamp-posts in road X
[2] each lamp-post has a lamp and each lamp has a dimming value

each experiment, it is possible to extract the timing profile, logged by the SUB Engine, related to a single UPDATE request. Each experiment has been repeated several times and average values of the timing components have been calculated to evaluate the parameters of the performance model (see Section IV.B).

VI. Evaluation Results Summary

A. KPIs

Table 8 reports the KPIs and summarizes the impact of all timing components on $\mathbf{T_{TOTAL}}$ (see (3) in Section IV.A). Interested readers can find a detailed evaluation analysis along with a statistical analysis of the measured timing components in Section X.

Table 8 Results summary for both experiments

		LAMP EXPERIMENT	ROAD EXPERIMENT
Experiment parameters			
Number of subscriptions	m	1004	
Number of updates	n	310	
Average triples/update	$\mathbf{Nu_{AVG}}$	1	31
LUTT Hit Rate	**LHR (%)**	0,40	0,72
Notifications/experiment		23	1004
KPIs			
Updates/s	**Ups**	68	3,8
Subscriptions/s	**Sps**	68K	3,8K
Triple/s	**Tps**	68K	117,3K
Notification latency	$\mathbf{NL_{min}}$	1,7 ms	1,3 ms
	$\mathbf{NL_{max}}$	9,3 ms	541,3 ms
Engine impact on Endpoint	**E2E**	1,0	1,8
Timing components (% on T_{TOTAL})			
	$\mathbf{T_{UPDATE}}$	49%	35%
	$\mathbf{T_{LUTT}}$	7%	3%
	$\mathbf{T_{BOOSTER}}$	44%	62%
	$\mathbf{T_{NOTIFY}}$	<< 1%	<< 1%

From the results here reported some preliminary conclusions could be drawn. Both the experiments (LAMP and ROAD) share the same number of subscriptions and updates but they present very different KPIs. We can so argue that the SUB Engine performance are not just influenced by the total number of subscriptions and/or updates itself but are mostly related to the specific content of such primitives. Moreover, the KPIs prove one of the fundamental hypothesis under which the notification algorithm has been designed: selective updates of few triples (i.e., the LAMP experiment) dominate with respect to more complex updates (i.e., the ROAD experiment). In fact, from Table 8, it is clear how the LAMP experiment outperforms the ROAD experiment (i.e., the maximum number of subscriptions that can be processed in one seconds (*Sps*) decreases from 68K to 3,8K) granting a lower notification latency (i.e., 9.3 ms compared to 541,3 ms) and with a lower overhead on the underpinning SPARQL endpoint (i.e., *E2E* moves from 1 to 1,8). Eventually, the results demonstrate the effectiveness of the LUTT: the LUTT matching requires no more than the 7% of the total experiment time.

B. LUTT filtering

An evidence of the LUTT filtering impact on the overall engine performance is provided by Table 9 where, for both the experiments, the timings components and the resulting KPIs are estimated without the LUTT support and are compared with those reported in Table 8.

Table 9 LUTT impact on engine performance (estimate). For reasons of simplicity and readability, all the values are approximated.

KPIs

	Ups	Sps	Tps	$\mathbf{Nl_{max}}$	E2E
LAMP EXPERIMENT					
LUTT On	68	68K	68K	0,09 s	1,0
LUTT Off	0,55	557	557	1,80 s	245
Ratio (On/Off)	122			5×10^{-3}	4×10^{-3}

	Ups	Sps	Tps	$\mathbf{Nl_{max}}$	E2E
ROAD EXPERIMENT					
LUTT On	3,8	3,8K	117,3K	0,54 s	1,8
LUTT Off	0,03	29,4	901	129 s	367
Ratio (On/Off)	130			4×10^{-3}	5×10^{-3}

Timing components ($\mathbf{T_{UPDATE}}$, $\mathbf{T_{NOTIFY}}$ and $\mathbf{T_{OVERHEAD}}$ are not shown as they are not affected by the LUTT)

	$T_{BOOSTER}$	T_{LUTT}	T_{TOTAL}
LAMP EXPERIMENT			
LUTT On	2 s	0,3 s	4,6 s
LUTT Off	9 min	-	9 min

	$T_{BOOSTER}$	T_{LUTT}	T_{TOTAL}
ROAD EXPERIMENT			
LUTT On	50,2 s	2,1 s	81,3 s
LUTT Off	176 min	-	176,5 min

The KPIs values reported in Table 9 show that the estimated LUTT impact on performance is quite similar in both the experiments (i.e., an improvement of two orders of magnitude). This estimate is not so surprising because, while the experiments seem significantly different to a human observer (310 versus 9500 context changes), they are similar from the LUTT based filtering point of view. In fact, in both experiments, 1000 out of 1004 LUTTs stop nearly all triples from progressing to the BOOSTER processing stage, drastically reducing the computational load in 99.6% of the cases.

C. Parallel execution

Some remarks about the parallel execution of active SPUs should also be added here. In sequential execution mode (N_{cores} = 1), according to (3), $\mathbf{T_{TOTAL}}$ is the sum of 4 separate timing components:

$$\boldsymbol{T_{TOTAL} = T_{UPDATE} + T_{LUTT} + T_{BOOSTER} + T_{NOTIFY}} \quad (4)$$

In the proposed implementation, $\mathbf{T_{UPDATE}}$ and $\mathbf{T_{LUTT}}$ do not overlap because all UPDATE primitives are serialized and because - for each UPDATE primitive – the update of the SPARQL endpoint and the LUTT filtering run sequentially. On the other side, if $N_{cores} > 1$, for each UPDATE primitive U_i, LUTT filtering and BOOSTER execution may be pipelined thanks to the interposed decoupling FIFO queue (ARTQ in Fig. 3). Moreover, all SPUs that find a core free may run in parallel. Thus, if tb_i is the BOOSTER execution time for U_i, while tu_{i+1} and tl_{i+1} are respectively the update time of the

SPARQL endpoint RDF store and the LUTT filtering execution time referred to U_{i+1}, then tb_i and $(tu_{i+1} + tl_{i+1})$ may overlap, drastically reducing $\mathbf{T}_{\text{TOTAL}}$, which so now becomes:

$$\boldsymbol{T}_{TOTAL} = \boldsymbol{T}_{UPDATE} + \boldsymbol{T}_{LUTT} \quad (5)$$

This is valid as long as ARTQ is not full and the scheduler does not share its core with any SPUs. For example, if for every U_i:

$$\mathrm{N}_{\text{cores}} \geq 1 + \sum_{j=1}^{m} h_{\mathrm{i,j}} \quad (6)$$

(i.e., one core for the scheduler and one for each SPU involved in U_i).

Fig. 7 compares the parallel execution mode (i.e., $N_{cores} = 8$) with the sequential execution mode (i.e., $N_{cores} = 1$) running an experiment with the following Update and Subscription profiles:

$$\boldsymbol{S} = \{ S_{ROAD}(\mathrm{X}) \mid \mathrm{X} \in \{1{,}100{,}200{,}300\} \}$$

$$\boldsymbol{U} = \{ U_{ROAD}(\mathrm{X}) \mid \mathrm{X} \in \{300..310\} \}$$

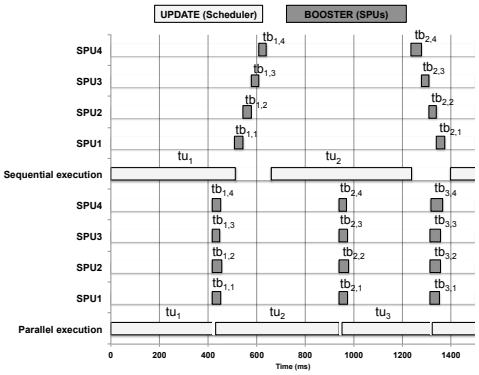

Fig. 7. Comparison between sequential and parallel execution modes (4 coarse-grain subscriptions react to 10 updates of 100 triples each). $\mathbf{T}_{\text{ELAPSED}}$ after 10 updates is respectively 6,0 s and 3,8 s (not shown).

VII. Related Work

The *SPS Architecture* presented in this paper can be framed within the research topics known as Stream Reasoning [*47*], Linked Stream Data Processing [*48*] and Content-Based Publish-Subscribe [*49*]. To the best of our knowledge, the first approaches for Semantic Web based publish-subscribe systems are presented by Wang *et al.* [*50*] and Chirita *et al.* [*51*]. Wang *et al.* propose an ontology based publish-subscribe system in which events are expressed with RDF graphs. Their contribution includes also a matching algorithm for event detection and a subscription language based on SquisQL [*52*], RDQL [*53*] and RQL [*54*] query languages. Chirita *et al.* in turn propose a solution to incorporate publish-subscribe capabilities in RDF-based peer-to-peer (P2P) network. Similarly to *Wang et al.,* they propose their own language for subscriptions. Due to the immaturity of Semantic Web query languages at the time, it is natural that all these approaches propose their own subscription languages. A similar approach is the one by Shi *et al.* [*55*]. However, since SPARQL was already gaining popularity in 2007, their subscription language is somewhat similar to SPARQL, although much more restricted. Because these early Semantic Web publish-subscribe approaches do not utilize SPARQL as subscription language, their processing engines and matching algorithms are completely different from the *SUB Engine* and its event detection and notification algorithm. This is also true for the solution presented in [*56*], which utilizes Semantic Web Rule Language (SWRL) [*57*] to represent subscriptions.

To the best of our knowledge, a first attempt to use SPARQL as the subscription language is presented in [*58*]. They present a simple architecture for content based publish-subscribe systems and evaluate the performance of their Jena [*59*] based reference implementation with a baseball related case study. However, they do not refer to any subscription related optimizations to the Jena query engine and the performance of the reference implementation is thus quite poor as shown by the evaluation results. Another important difference compared to our proposal is that they provide the whole result set whenever the SPARQL query results change, whereas our architecture sends the whole result set when the subscription is registered and then notifies clients about how the results set changes with new and obsolete bindings. Also the architecture described in [*58*] differs from the IoT focused *SPS Architecture* and its application design pattern in the following ways: 1) it does not include aggregators and 2) it does not specify the role of the agents in detail from the IoT system perspective.

Another early SPARQL based RDF stream processing proposal is Streaming SPARQL [*60*]. It differs from the above-presented approach in that it does not use standard SPARQL but extends it with *windows*. A 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 minutes). The window specification defines also how often the window is updated and consequently the frequency of query evaluation. The Streaming SPARQL engine is implemented within the ODYSSEYS framework.

Windows are typical in traditional stream processing and there are many other window-based RDF stream and event processing solutions, which focuses on different aspects of event processing and use different syntaxes for presenting windows: Continuous SPARQL (C-SPARQL) [*61*], $\text{SPARQL}_{\text{Stream}}$ [*62*], Event Processing SPARQL (EP-SPARQL) [*63*], Continuous Query Evaluation over Linked Data Streams (CQELS) [*64*], and Sparkwave [*65*].

C-SPARQL is a language for expressing persistent SPARQL queries over RDF streams. In addition to the extensions for windows, it extends SPARQL 1.0 with support for time management, and aggregations. The execution environment of C-SPARQL builds upon a standard SPARQL reasoner (which evaluates the static part of the query) and a Data Stream

Management System (which is responsible for the dynamic part of the query).

Similarly to Streaming SPARQL and C-SPARQL, $SPARQL_{Stream}$ is a language that introduces extensions to SPARQL to handle RDF streams. It differs from Streaming SPARQL and C-SPARQL in three ways. First, the $SPARQL_{Stream}$ only considers time-based windows whereas Streaming SPARQL and C-SPARQL support also windows defined by a concrete number of triples. Second, the $SPARQL_{Stream}$ 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 $SPARQL_{Stream}$ 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 is a SPARQL based language for Event Processing and Stream Reasoning. It focuses on the detection of RDF triples in a specific temporal order. To make this possible EP-SPARQL proposes several binary operations that can be used to combine RDF graph patterns in a temporal-sensitive manner. Another notable difference to the other window-based event processing approaches is that EP-SPARQL does not enforce the use of windows. 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. Because of this, the EP-SPARQL cannot be classified as a pure window-based solution and it is thus closer to the SPARQL event processing approach proposed in this paper. The system implementing the event processing functionalities defined by EP-SPARQL is called ETALIS [*66*]. It is implemented in Prolog and designed to process event-driven-backward-chaining (EDBC) rules [*67*] (i.e., EP-SPARQL expressions need to be translated into EDBC rules written in Prolog before they can be evaluated by ETALIS).

CQELS is a SPARQL based adaptive query engine for Linked Data and Linked Data Streams. The main difference with other window-based SPARQL event processing systems is that CQELS uses its own native processing model in the query engine – in contrast to other approaches that transform SPARQL into logic rules, Rete networks, or data suitable for standard stream processing engines. This makes it possible to have a full control of the query execution plan and it is used in practice to dynamically reorder operators based on changes in the input data.

To our knowledge, Sparkwave is the most recent window-based approach for continuous RDF data stream processing with SPARQL. The event processing in Sparkwave is based on the Rete algorithm [*68*] 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 the *SPS Architecture* from the window-based SPARQL event processing approaches presented above. First, the *SPS Architecture* 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, the *SPS Architecture* uses SPARQL, without any extensions, both to generate and subscribe to events. Third, instead of processing individual RDF triples coming from specific RDF streams, the *SPS Architecture* 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 SPARQL publish-subscribe engine (*SUB Engine*) detects how the results have changed from the initial query results whereas the window-based approaches provide the whole results 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 algorithms are totally different.

In addition to the window-based approaches and the early Semantic Web based publish-subscribe systems presented above, more similar approaches to the *SPS Architecture* have been presented in the literature. A common characteristic of these approaches, including Groppe *et al.* [*69*], EventCloud [*70*] [*71*], INSTANS [*72*] [*73*] [*74*], SENS [*40*] [*75*] [*76*] [*77*] and Smart-M3 [*17*], is that SPARQL is used as the subscription language.

Groppe *et al.* [*69*] propose algebra for handling RDF streams with SPARQL and suggest optimizations for its implementation. The SPARQL streaming engine implementing the algebra improves a SPARQL endpoint by 1) discarding irrelevant triples in the early state of processing, 2) creating indices only for triples relevant for the query, and 3) calculating partial results for the query as soon as possible. The *SPS Architecture* and the *SUB Engine* proposed in this paper differs from the approach prosed by Groppe *et al.* in three ways. First, the algebra proposed by Groppe *et al.* focuses only on streams, whereas we also provide a memory for the system. Having a central memory in the event processing engine makes the approach more suitable for IoT systems as the clients may join and leave the system dynamically at runtime. Second, the most notable difference in the approaches is that the engine proposed by Groppe *et al.* is designed to provide the whole result set (in similar way to normal SPARQL query) whenever it is modified in any way. Our approach in turn detects only how the result set change (i.e., new and obsolete bindings) and it is thus more suitable for situations where only small parts of the data set change during a publish operation. Because of this difference the actual algorithms and ways to detect the events are also completely different. Third, Groppe *et al.* do not propose any specific client design pattern whereas we propose a modular IoT application and system design pattern based on producers, consumers and aggregators.

In addition to the pioneer work of Chirita *et al.* [*51*], more recent approach for P2P publish-subscribe communication with Semantic Web technologies have been proposed by Pellegrino *et al.* [*70*] [*71*]. Their subscription processing system, called EventCloud, is based on P2P Content Addressable Network (CAN). As a subscription language, the EventCloud utilizes a subset of SPARQL. They also propose two matching algorithms, called Chained Semantic Matching

Algorithm (CSMA) and One-step Semantic Matching Algorithm (OSMA). There are four main differences between the EventCloud and the *SPS Architecture* proposed in this paper. First, EventCloud implementation is based on P2P networks whereas we implement our SPARQL subscription engine on top of a traditional SPARQL endpoint. Second, the subscription language used by the EventCloud supports only a limited subset of SPARQL features whereas we support SPARQL 1.1 as such. Third, the EventCloud provides the whole results set whenever the results change, whereas the *SPS Architecture* calculates how the results have changed. Fourth, Pellegrino *et al.* do not focus on the client side whereas we describe how IoT applications can be developed in a modular way by utilizing the client design pattern proposed in the paper.

INSTANS (Incremental eNgine for STANding Sparql) [*72*] [*73*] [*74*] is another Rete-based approach for SPARQL event processing. The INSTANS platform is implemented with Scala programming language and consists of Control, SPARQL parser, Rete network, RDF triple store and Garbage collector modules. In spite of the similarities, there are also many differences between the *SPS Architecture* and INSTANS. The first difference is that we provide a highly parallel architecture whereas in INSTANS the evaluation of SPARQL subscriptions is sequential. The second difference is that our solution introduces an optimized publish-subscribe mechanism on top of a generic SPARQL endpoint, whereas INSTANS bases its approach on the Rete algorithm. The third difference is that in INSTANS a notification includes all the query results when the results change, while in our solution only the delta in the SPARQL binding results is notified to the subscriber. The fourth difference is that INSTANS divides clients into two groups (i.e., event producers and agents) whereas we propose an application design pattern based on three types of clients in order to achieve modular design of IoT systems.

The Semantic Event Notification Service (SENS) proposed by Murth and Kühn [*40*] [*75*] [*76*] [*77*] is an event processing infrastructure that focuses on detecting when new knowledge emerges rather than detecting changes in the system status. Subscriptions are expressed with SPARQL basic graph patterns and it is also possible to create rules (represented with a subset of SPARQL CONSTRUCT) that create new knowledge to the knowledge base when specific events occur; this in turn can generate new knowledge events for the client. Their approach supports RDFS and OWL level reasoning making it possible to detect knowledge events that are not explicitly specified in the RDF updates but inferred based on the data. The earlier version, introduced in [*40*], is based on Jena framework and provides a Rete based forward reasoning engine. Later version, introduced in [*75*], utilizes OWLIM 2.9 [*78*] as the reasoning and query engine. The most important differences between the approach proposed in this paper and the SENS are the following. First, SENS approach takes ontological reasoning as part of the event processing whereas we focus solely on events expressed with SPARQL. Second, SENS only detects when new knowledge is inserted (i.e., knowledge can be removed but the event detection algorithm cannot detect that and the client will not be notified about it) whereas our approach also notifies clients about obsolete results. Third, SENS supports only a subset of SPARQL (i.e., basic graph patterns) whereas our approach provides a wider support for SPARQL. Fourth, SENS is not tailored for IoT systems and does not propose any specific system design pattern whereas we propose a modular client design pattern for IoT systems.

The Smart-M3 community has studied the feasibility of the Smart-M3 solution for device interoperability in pervasive computing and IoT, investigating application domains [*79*] [*80*], analyzing methodological aspects [*81*] [*82*] and reporting about the lesson learned [*83*] [*84*] [*85*]. The *Semantic Publish-Subscribe (SPS) Architecture* presented in this paper has been inspired by Smart-M3 [*17*] and its reference implementation named Smart-M3 RedSIB 0.9.2 [*41*] has been used as baseline for the *SPS Architecture* formalization. In particular, the novel *SPARQL Subscription Engine* (*SUB Engine*) aims at improving the performance of the previous Smart-M3 reference implementation in the following ways: it proposes a novel SPARQL event detection algorithm that calculates how each individual RDF triple modifies the results set obtained by the initial SPARQL query and it utilizes a parallel architecture where each subscription is executed by a separate processing unit (i.e., CPU core). In addition to the *SUB Engine*, the main contribution of the paper is the application design pattern, where only SPARQL subscription and update operations are used, including delay updates.

This paper is not the only one focusing on the Smart-M3 knowledge broker architecture. For instance, Suomalainen *et al.* propose a secure broker, called RIBS [*86*]. Galov *et al.* have developed the CuteSIB [*87*], focusing on extensibility, dependability, and portability. Viola *et al.* propose pySIB [*88*], targeted especially to resource constrained computing platforms. As can be seen, these existing architectures address different aspects (i.e., security, portability, extensibility and dependability) compared to the *SPS Architecture* (i.e., performance).

In our earlier work [*21*], we focused on the performance and scalability of Semantic Web enhanced IoT systems and proposed an architecture that partitions the IoT system into several parallel smart spaces. This work and the *SPS Architecture* are complementary contributions to enable real-time semantic information processing in large-scale IoT systems. In fact, the *SPS Architecture* focuses on the performance and scalability of a single smart space (i.e., knowledge broker) whereas the architecture presented in [*21*] describes how to achieve scalability by supporting several knowledge brokers with an infrastructure that enables clients to discover them.

VIII. Conclusion

Event detection and notification at several levels of abstraction are capabilities required by information processing systems since the very early stages of the computer history:

for example, back in 1946, Burks, Goldstine and von Neumann literally stated *"... simultaneous operation of the computer and the input/output organ requires additional temporary storage and introduces a synchronization problem..."* [*89*], motivating the need for such simultaneous operation with performance reasons. Performance, in fact, is often the primary requirement for event management in small size and application specific systems. But, when systems complexity and flexibility grow, while the heterogeneity and the distribution of their components become a dominant factor, unambiguous event interpretation and explicit specification of the context where the event has to be handled become requirements as challenging as the system reaction time. This is what happens, for example, when the Internet of Things (IoT) come into play: in general, IoT services are based on infrastructures that react to events recognized by scoping large information bases dynamically updated by ubiquitous and heterogeneous devices. The semantics of such information bases could conveniently be specified with a shared OWL ontology and their stores (i.e., SPARQL endpoints) can be searched with an appropriate standard query language like SPARQL. But SPARQL endpoints do not provide a facility to detect and notify events because they are mostly conceived to deal with huge amounts of RDF triples that evolve constantly but at a much slower rate compared to the rate of elementary events occurring in the physical environment. To fill this gap a semantic publish-subscribe architecture consisting of a SPARQL Subscription Engine sitting on top of a SPARQL endpoint is proposed. The architecture exposes two primitives, one to generate and one to subscribe to events. The engine is optimized for efficient detection and notification of events originated by frequent and small context updates as expected in IoT ecosystems. Specifically the engine entrusts its performance and scalability levels to three factors: i) the capability to recognize and remove from the semantic event processing pipeline out-of-context semantic events; ii) a novel algorithm which implements subscriptions with SPARQL queries and has the ability to notify only added and removed binding results since the previous notification; iii) the inherent subscription processing parallelism. A performance evaluation method is proposed and a reference implementation is evaluated. This implementation extends the core of Smart-M3, which is a semantic interoperability platform for smart spaces successfully demonstrated in several European research projects. Key performance indicators are provided and the impact of the above mentioned performance and scalability factors are analyzed with respect to a simple benchmark.

New benchmarks and case studies are now encouraged to provide the fuel for further improvements and for a more comprehensive validation of the proposed architecture. Meanwhile its application in IoT scenarios is expected to provide best practices in application design. The short-term plan is to incorporate the proposed architecture in a Multi-Network-Multi-Protocol IoT gateway, supporting protocols like MQTT, DASH7 and 6LoWPAN on the "Things" side and HTTP, COAP and NDN on the "Internet " side.

IX. Appendix A: Table of Terms

General terms
SPS Architecture (Semantic Publish-Subscribe Architecture): is the architecture proposed in this paper and is composed by: a processing infrastructure (SUB Engine + SPARQL endpoint), two primitives (UPDATE and SUBSCRIBE) and a set of clients (Consumers, Producers and Aggregators).
SUB Engine (SPARQL Subscription Engine): is the core component of the processing infrastructure. Its internal architecture and the model used to evaluate the performance of a generic implementation are part of the main research contributions of the paper.
URQ (UPDATE Request Queue), SRQ (SUBSCRIBE Request Queue): UPDATE and SUBSCRIBE requests are respectively stored into two FIFO queues. For each SUBSCRIBE request the SUB Engine instantiate a new SPU (SPARQL Processing Unit). The SUB Engine issues UPDATE requests to the SPARQL endpoint and the corresponding added and removed RDF triples are retrieved. These triples are forwarded to every active SPU.
SPU (SPARQL Processing Unit): the SUB Engine instantiate a SPU for each subscription. SPUs can execute in parallel on a multi-core architecture. A SPU implements the event detection algorithm (see BOOSTER).
CTS (Context Triple Store): each SPU maintains a local copy (i.e., like a cache) of the context represented by the set of triples that may contribute triggering a notification.
LUTT (Look Up Triples Table): each SPU has its own LUTT that is used to filter out triples that are out of the subscription scope
ARTQ (Added/Removed Triple Queue): is a FIFO queue that contains the triples that passed the LUTT.
BOOSTER: implements the event detection algorithm. It takes as input the triples extracted from the ARTQ and the CTS. The same triples extracted from the ARTQ are also used to update the CTS.
Benchmark parameters
U (Update Profile): a set of cardinality n of all the UPDATE primitives U_i of an experiment
S (Subscription Profile): a set of cardinality m of all the SUBSCRIBE primitives S_i of an experiment
[***Nu***]: a vector of n elements, where each element Nu_i is the number of triples updated by the UPDATE U_i
$\boldsymbol{Nu_{AVG}}$: average number of triples updated by a single UPDATE primitive within an experiment $$\boldsymbol{Nu_{AVG}} = \frac{1}{n}\sum_{i=1}^{n} Nu_i$$
[*h*] (LUTT matrix): a Boolean matrix of n x m elements [h], where a generic element $h_{i,j} = 1$ if at least one of Nu_i triples pass $LUTT_j$ check, otherwise $h_{i,j} = 0$.
LHR(%) **(LUTT hit ratio)**: is an indication of the LUTT filtering effectiveness in a particular scenario) $$\boldsymbol{LHR}\ (\%) = \frac{100}{m \text{ x } n}\sum_{i=1}^{n}\sum_{j=1}^{m}\boldsymbol{h}_{i,j}$$
Performance model (timing components)
$\mathbf{T_{UPDATE}}$ is the total latency time of the SPARQL endpoint occurring when the *Update Profile* is applied. tu_i is the time required by the SPARQL endpoint to complete U_i
$\mathbf{T_{LUTT}}$ is the time paid to check n times all the m LUTTs. tl_i is the time to check all the m LUTTs when U_i occurs, while $tl_{i,j}$ is the time required to check all Nu_i triples against $LUTT_j$.
$\mathbf{T_{BOOSTER}}$ is the time spent by all m BOOSTERs to search their CTSs for the results to be notified to the subscribed clients. tb_i is the total time required by all m BOOSTERs to detect all events triggered by U_i, while $tb_{i,j}$ is the time interval spent by the $BOOSTER_j$ to search the CTS for the events subscribed by S_i and triggered by U_i.
$\mathbf{T_{NOTIFY}}$ is the time spent by all m SPUs to forward the results to the communication interface. te_i is the time required by all m SPUs to forward all results triggered by Ui, while $te_{i,j}$ is the time required by SPU_j to forward the results produced by Ui.

KPI (Key Performance Indicators)
Ups (Average number of updates processed per unit time) $$Ups = \frac{n}{T_{TOTAL}}$$
Tps (Average number of triples processed per unit time) $$Tps = Nu_{AVG}\ Sps$$
Sps (Average number of subscriptions processed per unit time) $$Sps = m\ Ups$$
E2E (Engine to SPARQL Endpoint impact factor) $$E2E = \frac{T_{TOTAL} - T_{UPDATE}}{T_{UPDATE}}$$
NL_{min}, NL_{max} (Event notification latency range) $$NL_{min} = min\{tl_{i,j}+tb_{i,j}+te_{i,j} \mid i = 1..n, j = 1..m \wedge te_{i,j} \neq 0\}$$ $$NL_{max} = max\{tl_i+tb_i+te_i \mid i = 1..n \wedge te_i \neq 0\}$$

X. Appendix B: Detailed Evaluation Analysis

All measures were taken server side and Unix timestamps in µs were collected. In our test bed we do not have any PRET processor (Precision Timed Machine [*90*]), therefore the measurements are statistically affected by several factors including memory access time variability, interrupts and OS overhead. All the measures here reported are average values calculated from the collected timing components. For each measured timing component, the mean, the standard deviation of the sample (i.e., σ), the minimum and maximum values are reported.

A. Evaluation of SPARQL endpoint RDF store update

The time required to update the SPARQL endpoint RDF store and retrieve the added and removed triples is characterized by a vector of *n* elements **[*tu*]**, where each element tu_i is the time required by the SPARQL endpoint to complete U_i. The measured **[*tu*]** vectors of the two experiments are shown in Table 10. tu_i depends on the SPARQL endpoint, on the structure of U_i, on Nu_i (number of triples updated by U_i) and on the efficiency of the SUB Engine to retrieve from the SPARQL endpoint the Nu_i triples updated by U_i. Often, the same update U_α is applied to different portions of the SPARQL endpoint store, so that several U_i only differ for some constant value in their WHERE clause (i.e., an URI). For example if U_α is: *"switch on all public lights of a specific street in a specific city"*, different instances of U_α could be applied to different streets and different cities and so each of these instances may end up with different values of Nu_i. Fig. 8 shows the measured trend of tu_i versus Nu_i shared by all the instances U_i for both of the *Update profiles*.

Table 10 [***tu***] vectors (time values are in ms)

LAMP EXPERIMENT

i	Mean	σ	Min	Max
1…100	7,795	4,220	4,463	33,189
101…200	6,458	2,752	3,885	12,920
201…300	6,944	3,530	3,850	13,291
301…310	7,324	2,614	4,203	12,627

ROAD EXPERIMENT

i	Mean	σ	Min	Max
1…100	52,825	21,569	34,239	147,540
101…200	76,279	18,619	62,169	164,443
201…300	131,069	6,913	121,653	175,980
301…310	250,834	8,316	240,710	272,640

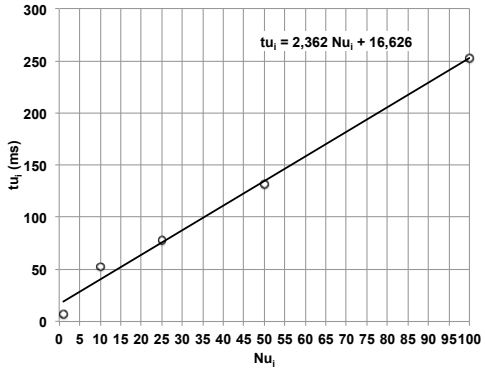

Fig. 8. The time (in ms) to update the SPARQL endpoint linearly grows with the amount of updated triples

B. Evaluation of LUTT filtering

The SUB Engine relies on the LUTT to filter out the triples that are out of a subscription context. The effect of LUTT filtering on the engine performance can be characterized by a matrix of *n* x *m* elements **[*tl*]**, $tl_{i,j}$ being the time interval required to check all the Nu_i triples updated by U_i against $LUTT_j$. A compact representation of the measured LUTT matrices for both experiments is shown in Table 11 (i.e., both are 310 x 1004 matrices). Fig. 9 plots the measured trends of the LUTT time $tl_{i,j}$ versus Nu_i: a fine-grain subscription (S_{LAMP}) is a subscription to a single lamp dimming value change, while a coarse-grain subscription (S_{ROAD}) is a subscription spanning all lamps of a road.

Table 11 [***tl***] LUTT matrices (µs)

LAMP EXPERIMENT

i	1…1000				1001…1004			
	Mean	σ	Min	Max	Mean	σ	Min	Max
1…100	1	1	0	111	7	3	3	19
101…200	1	1	0	160	6	7	3	144
201…300	1	1	0	54	6	3	3	27
301…310	1	1	0	13	7	3	3	15

ROAD EXPERIMENT

i	1…1000				1001…1004			
	Mean	σ	Min	Max	Mean	σ	Min	Max
1…100	3	1	2	35	40	15	22	141
101…200	5	1	4	115	68	16	46	183
201…300	10	2	9	131	133	14	101	214
301…310	22	7	19	303	255	24	213	317

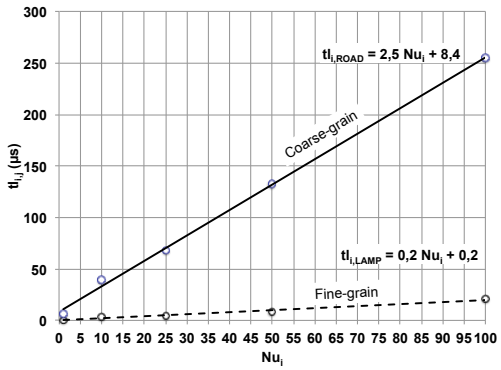

Fig. 9. Trend of LUTT time (in µs) versus Nu_i for a fine-grain ($tl_{i,LAMP}$) and for a coarse-grain ($tl_{i,ROAD}$) subscription

C. Evaluation of event detection (BOOSTER processing)

All triples that hit $LUTT_j$ have to be processed by the corresponding $BOOSTER_j$ (5 in Fig. 3 and Fig. 6). BOOSTER processing is characterized by two n x m matrices: **[*Nb*]** where each matrix element $Nb_{i,j}$ states how many triples out of Nu_i have to be processed by $BOOSTER_j$ (i.e., in general only a subset of the Nu_i triples will pass the LUTT filtering) and **[*tb*]** where each element $tb_{i,j}$ is the time spent by the $BOOSTER_j$ to search its CTS for the events subscribed by S_j and triggered by U_i. $Nb_{i,j}$ is zero if the corresponding LUTT element $h_{i,j}$ is zero. In all the other cases, $Nb_{i,j}$ depends on the *Update* and *Subscribe profiles*.

Table 12 shows both the **[*Nb*]** and **[*Ne*]** matrices as, in the proposed benchmark, they are the same (i.e., $Nb_{i,j}$ triples produce exactly $Nb_{i,j}$ binding results), while the measured **[*tb*]** matrices of the two experiments are shown in Table 13.

Fig. 10 shows the measured trend of $tb_{i,ROAD}$ versus $Nb_{i,ROAD}$ for a coarse-grain subscription S_{ROAD} of the proposed benchmark. As can be seen, the BOOSTER execution time linearly grows with the number of processed triples $Nb_{i,ROAD}$ and how fast is the BOOSTER in detecting an event, compared to corresponding time to update the SPARQL endpoint, can be perceived by comparing the first order coefficients in Fig. 8 and Fig. 10. At the same time, the LUTT effectiveness on filtering out of context triples can be perceived comparing Fig. 9 with Fig. 10 (i.e., µs versus ms).

Table 12 [*Nb*] and [*Ne*] matrices

LAMP EXPERIMENT

		j
		1…1004
i	1...310	1

ROAD EXPERIMENT

		j	
		1...1000	1001...1004
i	1...100	1	10
	101...200	1	25
	201...300	1	50
	301...310	1	100

Table 13 [*tb*] matrices (time values are in ms)

LAMP EXPERIMENT

		j							
		1...1000				1001...1004			
		Mean	σ	Min	Max	Mean	σ	Min	Max
i	1...100	1,816	1,030	1,073	3,799	1,580	0,556	1,023	4,446
	101...200	1,202	0,115	1,053	1,351	1,619	0,557	1,035	3,549
	201...300	2,268	0,261	1,938	2,573	1,528	0,630	1,013	4,671
	301…310	1,902	0,623	1,077	2,574	1,749	0,736	1,059	3,197

ROAD EXPERIMENT

		j							
		1…1000				1001…1004			
		Mean	σ	Min	Max	Mean	σ	Min	Max
i	1...100	1,206	0,087	1,121	1,569	13,398	2,189	12,137	26,517
	101…200	1,216	0,135	1,120	1,853	32,859	2,837	30,518	55,475
	201…300	1,199	0,086	1,126	1,729	66,496	5,105	61,218	104,513
	301…310	1,297	0,246	1,110	2,812	127,244	7,617	121,634	162,527

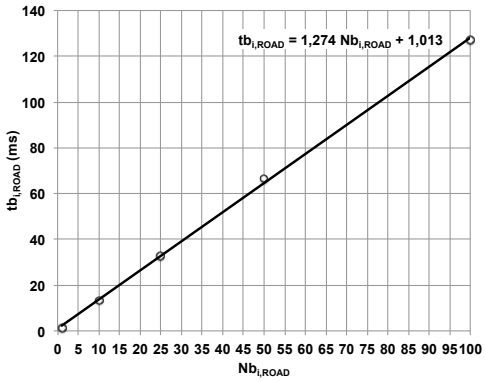

Fig. 10. BOOSTER execution time (in ms) for a coarse-grain subscription

D. Evaluation of event notification latency

Events detected by a SPU need to be passed to the hosting platform communication interface which is in charge of forwarding the notification to the appropriate subscriber. This step (see 6 in Fig. 3 and Fig. 6) is characterized by two n x m matrices: **[*te*]**, where each element $te_{i,j}$ is the time required by the SPU_j to forward the results produced by Ui to the communication interface and **[*Ne*]**, where each element $Ne_{i,j}$ states how many results are found by $BOOSTER_j$ and have to be notified when U_i occurs. $Ne_{i,j}$ is zero if the corresponding $h_{i,j}$ is zero. In all the other cases, $Ne_{i,j}$ depends on the *Update* and *Subscribe profiles*. Table 12 shows both the **[*Nb*]** and **[*Ne*]** matrices as, in the proposed benchmark, they are the same (i.e., $Nb_{i,j}$ triples produce exactly $Nb_{i,j}$ binding results).

Moving on to **[*te*],** the time $te_{i,j}$, required by the SPU_j to send to the communication interface the appropriate notification due to U_i generally depends on the number of binding results to be notified (i.e., $Ne_{i,j}$). $te_{i,j}$ is different from zero only for those U_i where at least one triple hits $LUTT_j$ and produces at least one result through the associated $BOOSTER_j$. In the $\mathbf{U}_{LAMP}$ *profile*, each Ui updates a single dimming value (i.e., one triple), so that, in this case, $te_{i,j}$ will always be the same (i.e., 155 µs). On the other hand, in the $\mathbf{U}_{ROAD}$ *profile*, each U_i

updates all the dimming values of the lamps belonging to a specific road X. Therefore, in the $\mathbf{U}_{ROAD}$ *profile,* if a $S_{ROAD}(X)$ subscription is triggered, $te_{i,j}$ depends on the number of lamps (i.e., N_{LAMP}) belonging to road X (i.e., 10, 25, 50, 100), while it has the same value as in the *LAMP profile* if a $S_{LAMP}(X,Y)$ is triggered. Fig. 11 plots the measured trend of $te_{i,ROAD}$ (*ms*) versus $Ne_{i,ROAD}$ for a coarse-grain subscription S_{ROAD} of the proposed benchmark. In this case, the impact of $te_{i,j}$ on $\mathbf{T}_{TOTAL}$ is small as it can be perceived by comparing Fig. 11 with Fig. 8.

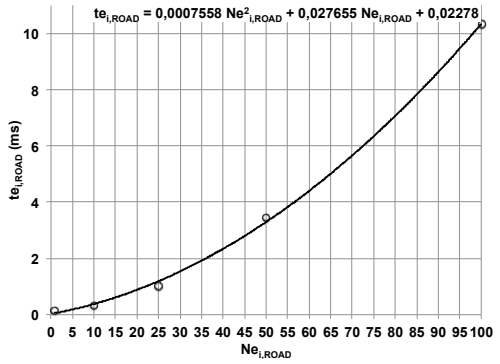

Fig. 11. Time to forward the results of a coarse-grain subscription to the communication interface

For both the *Update Profiles*, the measured $te_{i,j}$ values (for all the subscriptions *Sj* that trigger a notification) are*:*

Table 14 [te] notification latency matrices (time values are in µs)

LAMP EXPERIMENT

i \ j	1…1000 Mean	1…1000 σ	1…1000 Min	1…1000 Max	1001…1004 Mean	1001…1004 σ	1001…1004 Min	1001…1004 Max
1…100	136	99	72	327	228	0	228	228
101…200	83	8	72	91	84	0	84	84
201…300	203	9	193	215	232	0	232	232
301…310	164	71	75	270	79	0	79	79

ROAD EXPERIMENT

i \ j	1…1000 Mean	1…1000 σ	1…1000 Min	1…1000 Max	1001…1004 Mean	1001…1004 σ	1001…1004 Min	1001…1004 Max
1…100	74	10	66	116	74	10	66	116
101…200	73	10	66	112	73	10	66	112
201…300	70	5	65	93	70	5	65	93
301…310	76	21	65	272	76	21	65	272

E. Overhead time analysis

In the reported experimental setup, the overhead is assumed to be:

$$\boldsymbol{T}_{OVERHEAD} = \boldsymbol{T}_{MEAS} + \boldsymbol{T}_{PROT} \quad (7)$$

where $\mathbf{T}_{MEAS}$ is the time to collect all timing information, while $\mathbf{T}_{PROT}$ is the time required to handle the interaction between the simulator and the SUB Engine. $\mathbf{T}_{PROT}$ is defined as the sum of all time intervals between the end of processing of an UPDATE primitive and the start of processing start of the next one. During these time intervals, the SUB Engine is inactive as it is waiting for the next UPDATE primitive from the simulator. $\mathbf{T}_{PROT}$ is not related to the SUB Engine performance, but it depends on the network delay, on the protocol implementation and on the client reaction speed. Evaluating and optimizing $\mathbf{T}_{PROT}$ is not in the scope of this paper. The measured $\mathbf{T}_{OVERHEAD}$ values for both experiments are shown in Table 15.

Table 15 Impact of $\mathbf{T}_{OVERHEAD}$ on $\mathbf{T}_{ELAPSED}$ in the two experiments

	$\mathbf{T}_{MEAS}$ (s)	$\mathbf{T}_{PROT}$ (s)	$\mathbf{T}_{OVERHEAD}$ (s)	$\mathbf{T}_{ELAPSED}$ (s)	Overhead (%)
ROAD	1,08	3,30	4,38	86,87	5
LAMP	1,03	2,57	3,60	8,09	45

F. System scalability analysis

The scalability of the system with the number of subscriptions (i.e., N_{SPUs}) and the speed-up curves with the increasing number of available cores (i.e., N_{cores}) can be predicted using the performance evaluation method (see Section IV). The benchmark here considered uses the UPDATE and SUBSCRIBE primitives in 3 and the same knowledge base in Table 2, but it is based on two different experiments.

In the *first experiment*, all the publishers issue the $U_{LAMP}(1,1)$ UPDATE, while all the subscribers are subscribed with the $S_{LAMP}(1,1)$ SUBSCRIBE. This means that each publisher updates one RDF triple (i.e., Nu = 1) and all the updated triples pass the LUTT (i.e., LHR(%) = 100%). Moreover, all the subscribers are notified (i.e., each UPDATE changes the current value of the dimming) and a notification includes two bindings (i.e., the new and the old dimming value).

In the *second experiment*, all the publishers issue the $U_{ROAD}(300)$ UPDATE, while all the subscribers are subscribed with the $S_{ROAD}(300)$ SUBSCRIBE. This means that each publisher updates 100 RDF triples (i.e., Nu = 100) and, as in the first experiment, all the updated triples pass the LUTT. Like in the first experiment, all the subscribers are notified, but in this case each notification includes 200 bindings.

These two experiments evaluate the performance of the engine in a very unfair scenario (i.e., LHR(%) = 100%) and so the results provide a lower bound prediction of the performance in a real world scenario. Both the experiments consider an increasing number of subscribers (i.e., N_{SPUs}) and an increasing number of available cores (i.e., N_{cores}). Focusing on a single UPDATE primitive, the timing components of the two experiments are taken from the evaluation results (i.e., **tu** from Table 10, **tl** from Table 11, **tb** from Table 13 and **te** from Fig. 11) and shown in Table 15.

Table 15 Timing components of experiments designed to predict the scalability and the effect of parallelization

Timing components	**Estimated time (ms)** *First experiment*	*Second experiment*
tu (*SPARQL endpoint update time*)	7,302	252,573
tl (*LUTT filtering time*)	0,001	0,255
tb (*BOOSTER time*)	1,789	127,243
te (*Notification forwarding time*)	0,155	10,321

As the SUB Engine sequentially executes both the SPARQL endpoint update and the LUTT filtering (see Scheduling in Fig. 6), the maximum update frequency can be estimated as:

$$UPDATE_{MAX}(Hz) = \frac{1}{\boldsymbol{tu} + N_{SPUs}\boldsymbol{tl}} \quad (\mathbf{8})$$

On the other hand, the maximum throughput in terms of notifications/s can be expressed as:

$$THROUGHPUT_{MAX}(Hz) = \frac{1}{\boldsymbol{tb} + \boldsymbol{te}} N_{cores} \quad (\mathbf{9})$$

First experiment

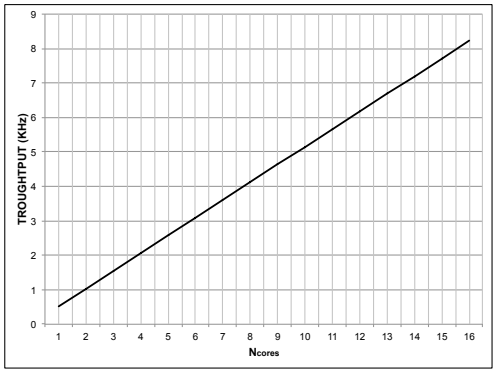

Fig.12. In a typical IoT scenario, where each client update a single triple (e.g., a sensor reading), the SUB Engine, running on a 16 cores system, is able to notify up to 8K subscribers in one second

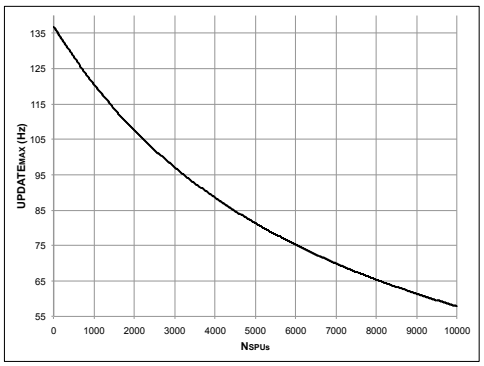

Fig. 13. The maximum update rate granted by the system decreases with the time needed to update the SPARQL endpoint (i.e., 7,3 ms) and with the time spent by the SUB Engine to implement the LUTT filtering (i.e., 1 µs)

Second experiment

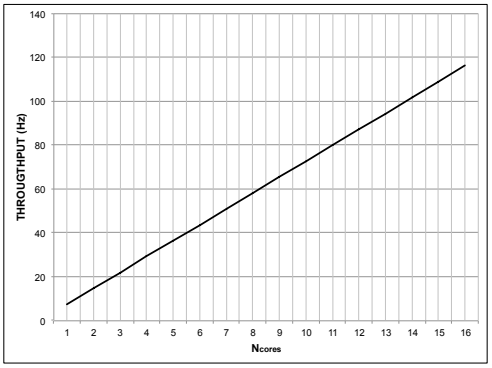

Fig. 14. In the worst case, where publishers update a large set of triples (i.e., 100 triples), the SUB Engine running on a 16 cores system is still able to notify (i.e., with a notification consisting of 200 bindings) more than 100 subscribers in one second

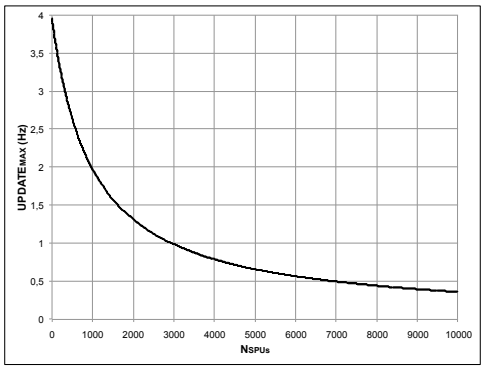

Fig. 15. The maximum affordable update rate of the SUB Engine is heavily affected by the time required by the SPARQL endpoint to update the RDF store (i.e., 252,7 ms to update 100 triples)

G. *Final remarks*

Some final observations about the engine performance can be drawn based on the experimental results. In our testing scenario and with the selected benchmark, the BOOSTER behavior is quite linear with respect to the number of triples involved in a specific event and detecting a desired event may require between 1 to well over 100 ms (see Fig. 10). As shown in Table 8, the latency of a notification spans from 2 to well over 500 ms. But the SUB Engine performance does not depend only on the BOOSTER algorithm. On the contrary, there are two additional contributions to improve the performance: LUTT based filtering and parallel execution of active SPUs.

Thanks to the LUTT based filtering, out-of-context triples are stopped from moving to the BOOSTER, and, as shown by Fig. 9 this happens quickly (i.e., less than 21 µs for fine-grain subscriptions and less than 255 µs for coarse-grain subscriptions). Comparing the first order coefficients in Fig. 9

and Fig. 10, it is easy to perceive up to which extent the impact of LUTT filtering on $\mathbf{T}_{\text{TOTAL}}$ may be neglected while the number of active subscriptions grows. Table 7 visualizes the impact of LUTT filtering on the overall performance: every '0' in the LUTT matrix states that an UPDATE primitive does not need to be processed by a specific SPU as it is out of its context. The LUTT impact is particularly valuable when the LUTT Hit Rate (LHR) is low, as expected in Internet of Things scenarios mostly dominated by fine grain updates.

Typically the SUB Engine will work in parallel execution mode but the number of available cores will not necessarily meet condition (6) (i.e., usually there will be less than one core per active SPU), therefore $\mathbf{T}_{\text{TOTAL}}$ will fall between (4) and (5).

We may thus conclude that the maximum event load affordable by a specific engine implementation is related to:

1. Its ability to instantly reject out-of-scope events (i.e., granted by the LUTT);
2. Its BOOSTER performance and the amount of memory available for the CTSs;
3. The number of cores available to the engine (N_{cores}).

ACKNOWLEDGMENTS

The authors are grateful to the M3 community and to the partners of all EU projects that inspired this paper: SOFIA, CHIRON, IoE, ARROWHEAD, RECOCAPE, Flex4Grid and the joint EU-Brazil project IMPRESS. The work was supported by the EU, the Italian Ministry of Education and Research (MIUR), the University of Bologna and VTT Technical Research Centre of Finland.

REFERENCES

[1] S. Helal, "IT Footprinting - Groundwork for Future Smart Cities," *Computer*, vol. 44, no. 6, pp. 30-31, June 2011.

[2] Jiong Jin, Jayavardhana Gubbi, Slaven Marusic Marusic, and Marimuthu Palaniswami, "An Information Framework for Creating a Smart City Through Internet of Things ," *IEEE Internet of Things Journal*, vol. 1, no. 2, pp. 112-121, April 2014.

[3] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi, "Internet of Things for Smart Cities ," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22-32, February 2014.

[4] A. Bassi et al., *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*.: Springer, 2013.

[5] John A. Stankovic, "Research Directions for the Internet of Things ," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 3-9, February 2014.

[6] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, February 2016.

[7] T. Berns-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, pp. 34-43, 2001.

[8] R. Masuoka, B. Parsia, and Y. Labrou, "Task Computing - the Semantic Web Meets Pervasive Computing," in *The Semantic Web - ISWC. Lecture Notes in Computer Science*.: Springer Berlin Heidelberg, 2003, pp. 866-881.

[9] H. Chen, T. Finin, and A. Joshi, "Semantic Web in the Context Broker Architecture," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*, 2004, pp. 277-286.

[10] H. Chen et al., "Intelligent agents meet the semantic Web in smart spaces," *Internet Computing, IEEE*, vol. 8, no. 6, pp. 69-79, 2004.

[11] X. Wang, J.S. Dong, C.Y. Chin, S.R. Hettiarachchi, and D. Zhang, "Semantic Space: an infrastructure for smart spaces," *Pervasive Computing, IEEE*, vol. 3, no. 3, pp. 32-39, 2004.

[12] Z. Song, A.A. Cárdenas, and R. Masuoka, "Semantic Middleware for the Internet of Things," *Internet of Things (IOT)*, pp. 1-8, 2010.

[13] J.I. Vazquez, De Ipina D.L., and I. Sedano, "SoaM: A Web-powered Architecture for Designing and Deploying Pervasive Semantic Devices," *International Journal of Web Information Systems*, vol. 2, no. 3, pp. 212-224, 2006.

[14] G. Thomson, S. Bianco, S.B. Mokhtar, N. Georgantas, and V. Issarny, "Amigo Aware Services," in *Constructing Ambient Intelligence, Communications in Computer and Information Science*.: Springer Berlin Heidelberg, 2008, pp. 385-390.

[15] D. Pfisterer et al., "SPITFIRE: toward a semantic web of things," *Communications Magazine, IEEE*, vol. 49, no. 11, pp. 40-48, 2011.

[16] Anh Le Tu'n et al., "Global Sensor Modeling and Constrained Application Methods Enabling Cloud-Based Open Space Smart Services," in *Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC)*, Sept. 2012, pp. 196-203.

[17] J. Honkola, H. Laine, R. Brown, and O. Tyrkkiö, "Smart-M3 information sharing platform," in *IEEE Symposium on Computers and Communications (ISCC)*, Riccione, Italy, 2010, pp. 1041 – 1046.

[18] W3C Semantic Web. (2014, Febbruary) Resource Description Framework (RDF),. [Online]. http://www.w3.org/RDF/

[19] Linked Data Project. Connect Distributed Data across the Web. [Online]. http://linkeddata.org/

[20] E. Ovaska, T. Salmon Cinotti, and A. Toninelli, "The Design Principles and Practices of Interoperable Smart Spaces," in *Advanced Design Approaches to Emerging Software Systems: Principles, Methodologies and Tools*. Hershey, PA, USA: IGI Global, 2012, pp. 18-47.

[21] J. Kiljander et al., "Semantic interoperability architecture for pervasive computing and Internet of Things," *IEEE Access*, vol. 2, pp. 856-873, 2014.

[22] F. Morandi, L. Roffia, A. D'Elia, F. Vergari, and T. Salmon Cinotti, "RedSib: a Smart-M3 Semantic Information Broker Implementation," in *Proceedings of the 12th Conference of Open Innovations Association FRUCT*, Oulu, Finland, 2012, pp. 86-98.

[23] F. Vergari et al., "A Smart Space Application to Dynamically Relate Medical and Environmental Information," in *Design, Automation & Test in Europe (DATE)*, Dresden, Germany, 2010, pp. 1542-1547.

[24] F. Vergari et al., "An integrated framework to achieve interoperability in person-centric health management," *International Journal of Telemedicine and Applications*, vol. 2011, p. 10, 2011.

[25] R. Gazzarata, F. Vergari, T. Salmon Cinotti, and M. Giacomini, "A standardized SOA for clinical data interchange in a cardiac telemonitoring environment," *IEEE Journal of Biomedical and Health Informatics*, vol. 18, no. 6, pp. 1764-1774, November 2014.

[26] D. Manzaroli et al., "Smart-M3 and OSGi: the Interoperability Platform," in *IEEE symposium on Computers and Communications - First International Workshop on Semantic Interoperability for Smart Spaces (SISS 2010)*, Riccione - Italy, June 22, 2010, pp. 1053 - 1058.

[27] W3C Recommendation. (2013, March) SPARQL 1.1 Update. [Online]. http://www.w3.org/TR/sparql11-update/

[28] W3C Recommendation. (2013, March) SPARQL 1.1 Query Language. [Online]. http://www.w3.org/TR/sparql11-query/

[29] Arvind Arasu, Shivnath Babu, and Jennifer Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121-142, June 2006.

[30] T. Gruber, "Ontology," in *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu, Ed.: Springer-Verlag, 2008.

[31] S. Pantsar-Syväniemi et al., "Case Study: Context-aware Supervision of a Smart Maintenance Process," in *Second International Workshop on Semantic Interoperability for Smart Spaces (SISS 2011)*,

Munich; Germany, 2011, pp. 309-314.

[32] F. Morandi, F. Vergari, A. D'Elia, L. Roffia, and T. Salmon Cinotti, "SMART-M3 v.0.9: A semantic event processing engine supporting information level interoperability in ambient intelligence," ARCES-AlmaDL-Alma Mater Studiorum Università di Bologna, Bologna, Italy, Tutorial ISBN: 978-88-98010-12-7 DOI: 10.6092/unibo/amsacta/3877 Online: http://amsacta.unibo.it/3877/, 2013.

[33] J.L. Hennessy and D.A. Patterson, *Computer Architecture a Quantitative Approach*. San Mateo, USA: Morgan Kaufmann Publishers, Inc., 1990.

[34] G.M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS1967 Spring Joint Computer Conference*, Atlantic City (N.J.), 1967, pp. 483-485.

[35] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 158--182, October 2005.

[36] Yuanbo Guo, A. Qasem, Zhengxiang Pan, and J. Heflin, ""A Requirements Driven Framework for Benchmarking Semantic Web Knowledge Base Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 2, pp. 297-309, Feb 2007.

[37] Raúl García-Castro et al., Eds., *Web Semantics: Science, Services and Agents on the World Wide Web, Special Issue on Evaluation of Semantic Technologies*.: Elsevier, August 2013, vol. 21.

[38] Christian Bizer and Andreas Schultz, "The Berlin SPARQL Benchmark," *International Journal on Semantic Web & Information Systems*, vol. 5, no. 2, pp. 1-24, 2009.

[39] Martin Murth, Dietmar Winkler, Stefan Biffl, Eva Kühn, and Thomas Moser, "Performance Testing of Semantic Publish/Subscribe Systems," in *On the Move to Meaningful Internet Systems: OTM 2010 Workshops*, Robert Meersman, Tharam Dillon, and Pilar Herrero, Eds.: Springer Berlin Heidelberg, 2010, vol. 6428, pp. 45-46.

[40] M. Murth and E. Kühn, "A Semantic Event Notification Service for Knowledge- Driven Coordination," in *1st International Workshop on Emergent Semantics and cooperaTion in opEn systEMs*, Rome, Italy, 2008.

[41] Smart-M3. [Online]. https://sourceforge.net/projects/smart-m3/files/Smart-M3-RedSIB_0.9.2/

[42] E. Orri and I. Mikhailov, "RDF Support in the Virtuoso DBMS," *Networked Knowledge-Networked Media*, pp. 7-24, 2009.

[43] D. Beckett. (2014) Redland RDF Libraries. [Online]. http://librdf.org/

[44] D. Beckett, "The design and implementation of the Redland RDF application framework," *Computer Networks*, vol. 39, no. 5, pp. 577-588, August 2002, http://www.sciencedirect.com/science/article/pii/S1389128602002219.

[45] J. Kiljander, F. Morandi, and J.P. Soininen, "Knowledge Sharing Protocol for Smart Spaces," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 3, no. 9, 2012.

[46] Z. Schelby, K. Hartke, and C. Bormann. Constrained Application Protocol (CoAP). CoRE Working Group Internet-Draft. [Online]. http://datatracker.ietf.org/doc/draft-ietf-core-coap/

[47] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel, "It's a Streaming World! Reasoning upon Rapidly Changing Information," *Intelligent Systems*, vol. 24, no. 6, pp. 83 - 89, Nov-Dec 2009.

[48] Danh Le-Phuoc, Josiane Xavier Parreira, and Manfred Hauswirth, "Linked Stream Data Processing," in *Reasoning Web. Semantic Technologies for Advanced Query Answering*.: Springer Berlin Heidelberg, 2012, vol. 7487, pp. 245-289.

[49] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114-131, June 2003.

[50] Jinling Wang, Beihong Jin, and Jing Li, "An ontology-based publish/subscribe system," in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Toronto, Canada, 2004, pp. 232-253.

[51] P.A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl, "Publish/Subscribe for RDF-based P2P Networks," in *The Semantic Web: Research and Applications*.: Springer Berlin Heidelberg, 2004, pp. 182-197.

[52] L., Seaborne, A., Reggiori, A. Miller, "Three implementations of SquishQL, a simple RDF query language," in *Proceedings of the First International Semantic Web Conference on the Semantic Web*, 2002, pp. 423-435, http://dl.acm.org/citation.cfm?id=646996.711274.

[53] P., Broekstra, J., Eberhart, A., Volz, R. Haase, "A comparison of RDF query languages," in *The Semantic Web – ISWC 2004, Lecture Notes in Computer Science*, 2004, pp. 502-517.

[54] G. Karvounarakis et al., "Querying the Semantic Web with RQL," *Computer Networks*, vol. 42, no. 5, pp. 617-640, August 2003.

[55] DongCai Shi, Jianwei Yin, Yiyuan Li, Jianfeng Qian, and Jinxiang Dong, "An RDF-Based Publish/Subscribe System," in *Semantics, Knowledge and Grid, Third International Conference on*, 2007.

[56] K.E. Kjaer and K.M. Hansen, "Modeling and Implementing Ontology-Based Publish/Subscribe Using Semantic Web Technologies," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, 2010, pp. 63-71.

[57] I. Horrocks et al. (2004, May) SWRL: A semantic web rule language combining OWL and RuleML. [Online]. http://www.w3.org/Submission/SWRL/

[58] J. Skovronski and C. Kenneth, "Ontology-Based Publish Subscribe Framework," in *8th International Conference on Information Integration and Web-based Applications & Services (iiWAS2006)*, Jakarta, 2006.

[59] Apache Jena. [Online]. https://jena.apache.org/

[60] A. Bolles, M. Grawunder, and J. Jacobi, "Streaming SPARQL extending SPARQL to process data streams," in *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, Tenerife, Canary Islands, Spain, 2008, pp. 448-462.

[61] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for C-SPARQL queries," in *Proceedings of the 13th International Conference on Extending Database Technology*, Lausanne, Switzerland, 2010, pp. 441-452.

[62] J. Calbimonte, O. Corcho, and A. Gray, "Enabling Ontology-based Access to Streaming Data Sources," in *ISWC'10*, 2010, pp. 96-111.

[63] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "EP-SPARQL: a unified language for event processing and stream reasoning," in *Proceedings of the 20th international conference on World wide web*, Hyderabad, India, 2011, pp. 635 - 644.

[64] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth, "A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data," in *ISWC2011*, 2011, pp. 370-388.

[65] S. Komazec, D. Cerri, and D. Fensel, "Sparkwave: Conituous Schema-Enhanced Pattern Matching over RDF Data Streams," in *DEBS'12*, 2012, pp. 58-58.

[66] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream Reasoning and Complex Event Processing in ETALIS," *Semantic Web Journal, Special Issue: Semantic Web Tools and Systems*, 2012.

[67] D. Aninic et al., "A rule-based language for complex event processing and reasoning," in *RR'2010*, 2010, pp. 42-57.

[68] C.L. Forgy, "Rete: A fast algorithm for the many pattern/many object match problem," *Artificial Intelligence*, no. 19, pp. 17-37, 1982.

[69] S. Groppe, J. Groppe, D. Kukulenz, and V. Linnemann, "A SPARQL Engine for Streaming RDF Data," in *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, Shanghai, 2007, pp. 167-174.

[70] Laurent Pellegrino, Francoise Baude, and Iyad Alshabani, "Towards a Scalable Cloud-based RDF Storage Offering a Pub/Sub Query

Service," in *CLOUD COMPUTING 2012 : The Third International Conference on Cloud Computing, GRIDs, and Virtualization*, 2012.

[71] Laurent Pellegrino, Fabrice Huet, Francoise Baude, and Amjad Alshabani, "A Distributed Publish/Subscribe System for RDF Data," in *Data Management in Cloud, Grid and P2P Systems*. Prague , Czech Republic: Springer Berlin Heidelberg, 2013, pp. 39-50.

[72] H. Abdullah, M. Rinne, S. Törmä, and E. Nuutila, "Efficient matching of SPARQL subscriptions using rete," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 372-377.

[73] M. Rinne, E. Nuutila, and S. Törmä, "INSTANS: High-Performance Event Processing with Standard RDF and SPARQL," in *ISWC2012 posters and Demonstrations Track*, Boston, US, 2012.

[74] Haris Abdullah, Seppo Törmä, Esko Nuutila Mikko Rinne, "Processing Heterogeneous RDF Events with Standing SPARQL Update Rules," in *Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part II*, 2012.

[75] M. Murth and e. Kuhn, "Knowledge-based Coordination with a Reliable Semantic Subscription Mechanism," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, Honolulu, Hawaii, 2009, pp. 1374-1380.

[76] M. Murth and E. Kühn, "A Heuristics Framework for Semantic Subscription Processing," in *6th European Semantic Web Conference, ESWC 2009 Heraklion, Crete, Greece, May 31–June 4, 2009 Proceedings*, 2009, pp. 6-110.

[77] M. Murth and E. Kühn, "Knowledge-Based Interaction Patterns for Semantic Spaces," in *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2010, pp. 1036-1043.

[78] OWLIM Primer. [Online]. http://graphdb.ontotext.com/display/OWLIMv40/OWLIM+Primer

[79] S. Bartolini et al., "Reconfigurable natural interaction in smart environments: approach and prototype implementation," *Personal and Ubiquitous Computing*, vol. 16, no. 7, pp. 943-956, 2012.

[80] B. Vlist, G. Niezen, S. Rapp, J. Hu, and L. Feijs, "Configuring and controlling ubiquitous computing infrastructure with semantic connections: A tangible and an AR approach," *Personal Ubiquitous Computing*, vol. 17, no. 4, pp. 783–799, 2013.

[81] D.G. Korzun, S.I. Balandin, and A.V. Gurtov, "Deployment of Smart Spaces in Internet of Things: Overview of the Design Challenges," in *NEW2AN 2013 and ruSMART 2013. LNCS 8121*, 2013, pp. 48–59.

[82] J. Takalo-Mattila, J. Kiljander, M. Eteläperä, and J. Soininen, "Ubiquitous computing by utilizing semantic interoperability with item-level object identification," in *Mobile networks and management, lecture notes of the institute for computer sciences, social informatics and telecommunications engineering*.: Springer Berlin Heidelberg, 2011, pp. 198-209.

[83] D.G. Korzun, A.M. Kashevnik, S.I. Balandin, and A.V. Smirnov, "The Smart-M3 Platform: Experience of Smart Space Application Development for Internet of Things," in *NEW2AN/ruSMART 2015. LNCS 9247*, 2015, pp. 56–67.

[84] M. Etelapera, J. Kiljander, and K. Keinanen, "Feasibility evaluation of M3 smart space broker implementations," in *11th International Symposium on Applications and the Internet (SAINT)*, 2011, pp. 292-296.

[85] J. Kiljander, A. Ylisaukko-oja, J. Takalo-Mattila, M. Eteläperä, and J. Soininen, "Enabling semantic technology empowered smart spaces," *Journal of Computer Networks and Communications*, 2012.

[86] Suomalainen J., Hyttinen P., and Tarvainen P., "Secure information sharing between heterogeneous embedded devices," in *4th ACM European Conference on Software Architecture (ECSA)*, Copenhagen, Denmark, 2010, pp. 205–212.

[87] I. Galov, A. Lomov, and D. Korzun, "Design of semantic information broker for localized computing environments in the Internet of Things," in *17th Conference of Open Innovations Association FRUCT*, 2015, pp. 36–43.

[88] Viola F., D'Elia A., Roffia L., and Cinotti T., "A Modular Lightweight Implementation of the Smart-M3 Semantic Information Broker," in *18th Conference of Open Innovations Association FRUCT*, 2016.

[89] A. Burks, H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Study, Princeton, USA, 1946.

[90] S.A. Edwards and E.A. Lee, "The Case for the Precision Timed (PRET) Machine," in *in Proceedings of the 44th Design Automation Conference (DAC)*, San Diego, California, 2007.