ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA

ARCES - ADVANCED RESEARCH CENTER ON ELECTRONIC SYSTEMS

SEMANTIC ARCHITECTURES LAB

SMART-M3 v.0.9: A semantic event processing engine supporting information level interoperability in ambient intelligence

A quick start tutorial

Francesco Morandi
Fabio Vergari
Alfredo D'Elia
Luca Roffia
Tullio Salmon Cinotti

Revised by Jussi Kiljander

Bologna, 7/10/2013

SMART-M3 v.0.9: A semantic event processing engine supporting information level interoperability in ambient intelligence

Francesco Morandi, Fabio Vergari, Alfredo D'Elia, Luca Roffia, Tullio Salmon Cinotti

Revised by Jussi Kiljander

ARCES

AlmaDL

Alma Mater Studiorum Università di Bologna

SMART-M3 v.0.9: A semantic event processing engine supporting information level interoperability in ambient intelligence/ Francesco Morandi, Fabio Vergari, Alfredo D'Elia, Luca Roffia, Tullio Salmon Cinotti. – Bologna: ARCES -; AlmaDL Alma Mater Studiorum Università di Bologna , 7 Ottobre 2013. -. P.27; 30 Cm.
ISBN: 978-88-98010-12-7

Quest'opera è stata realizzata con il patrocinio di:
Anna Ciampolini – Coordinator of the Laurea Magistrale in Ingegneria Informatica
Riccardo Rovatti - Coordinator of the Laurea Magistrale in Ingegneria Elettronica

This work was supported by EIT-ICT Labs – Action Line Smart Spaces – Activity OLDA - Open Local Data Applications (2013)

.

Foreword

Emerging computing paradigms such as *pervasive computing*, *Internet of Things*, and *cyber-physical systems* require novel approaches for managing interoperability between heterogeneous systems and devices. This tutorial describes in detail how Semantic Web technologies can be utilized in order to build systems interoperable at the semantic level. In particular, the tutorial focuses on systems where fast event processing is vital and introduces simple primitives for creating and subscribing to events with SPARQL. The tutorial also describes how to design such systems in modular and extendable way and provides the reader with important hands-on knowledge on application development.

The proposed software infrastructure is the result of European research projects with significant contribution from the University of Bologna and all the material here discussed is available as *open source* so the reader can instantly start experimenting by developing her/his own applications.

- Jussi Kiljander*, Research Scientist, VTT Technical Research Centre of Finland*

Contents

Foreword 1

Contents 3

Glossary 5

Introduction 7

SPARQL primitives for semantic event processing 9

- Delayed SPARQL update 10
- SPARQL Update Template 13
- SPARQL Subscription 14
- SPARQL Subscribe template 15

Programming approach 16

Application model and programming style 18

- Presence Sensor KP 20
- LampActuator KP 21
- LampManager KP 22

Hands on 25

- Downloading, installing and running the Semantic Event Broker 25
  - Downloading 25
  - Installing 25
  - Running 25
- Examples 25

References 27

Glossary

M3

A middleware architecture to share semantic information about the physical world in cross-domain multi-vendor, multi-device, multi-platform applications.

SMART-M3

The first open-source implementation of the M3 architecture (Sept 30, 2009, San Jose, CA).

Semantic Information Broker (SIB)

In SMART-M3 the SIB is the host and manager of the shared information.

Knowledge Processor (KP)

Any actor exchanging information with the SIB ;it is a producer and/or a consumer of information.

Smart Space Access Protocol (SSAP)

The protocol used by KPs to communicate with the SIB.

RedSIB

A SIB implementation (Release0.4) based on Redland triple store.

SEB

A SIB implementation (Release0.9) supporting *delayed SPARQL update*. SEB stands for Semantic Event Broker.

Resource Description Framework (RDF)

The World Wide Web Consortium standard for information modeling and conceptual description. SMART-M3 information is represented in RDF.

Web Ontology Language (OWL)

The World Wide Web Consortium language for authoring ontologies.

SPARQL Protocol and RDF Query Language (SPARQL)

A query language to retrieve and manipulate RDF information.

Introduction

This document is a quick start tutorial. It describes how to create open ambient intelligence applications with Smart-M3 v.0.9 platform [1]. The reader is expected to have a basic knowledge about the following Semantic Web technologies: Resource Description Framework (RDF), RDF Schema (RDFS), Web Ontology Language (OWL) and SPARQL[3][4]. The proposed examples and programming templates are intended to be suitable for any popular programming language.

Smart-M3 v.0.9 provides new functionalities which will be enlightened in this guide.

Smart-M3 is an open-source middleware proposed by SOFIA, an European Project (2009-11) of the ARTEMIS framework, to enable information interoperability in cross-domain multi-vendor, multi-device, multi-platform applications. The platform implements the separation of concerns principle, decouples information producers and consumers, and inherently supports system modularity.

In M3 vision, all actors of the application (i.e. sensors, devices, services, actuators, etc.) cooperate by sharing information via common RDF database.

The architecture supports:

- information interoperability: interoperability is enabled by a shared data model which relies on Semantic Web technologies.
- prompt reaction to context changes: the functional architecture provides mechanism for subscribing to complex events.

Fig.1 shows the M3 functional architecture. The “legacy gate” is an interface to the external physical world. Many legacy gates may coexist in M3 based ecosystems.

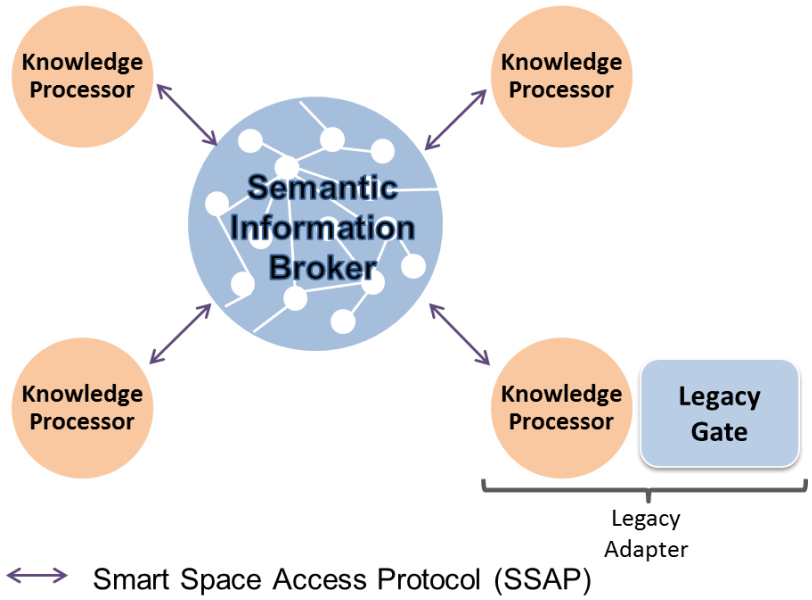

Fig. 1 - M3 architecture

The Semantic Information Broker (SIB) is the entity responsible for storing and governing information shared in M3 applications. Software agents exchanging information with each other via the SIB are called Knowledge Processors (KPs). The Smart Space Access Protocol (SSAP) defines to methods and syntax for KP-SIB communication; this protocol consists of XML messages transmitted over TCP/IP. Application Programming Interfaces (APIs) implement the SSAP for the KPs and are available in several programming languages.

Smart-M3 attempts to solve the information interoperability problem capitalizing on Semantic Web technologies. Semantic Web is a framework developed by the World Wide Web Consortium to enable data sharing and reuse across applications, enterprises and communities. Smart-M3 uses RDF to represent information. In RDF information is represented as triples consisting of subject, predicate and object. The RDF triples stored into the SIB form a directed labeled graph. It should be noted that the graph is not necessarily a connected graph however.

While RDF provides the standard data model for information representation, use of an ontology language is essential to specify the information semantics. Ontology languages such as RDFS and OWL provide a common vocabulary. The use of a common ontology allows all actors (humans and machines) to mutually understand the semantics of information and cooperate with each other through the SIB. Smart-M3 is ontology agnostic and thus allows developers to choose the best way to model the information in order to satisfy the functional requirements of the addressed application domain.

Developers can write their own software (i.e. their own KPs) using APIs available in the following programming languages: Python, C, C#, Java, PHP, JavaScript.

SPARQL primitives for semantic event processing

The *Semantic Event Broker* (SMART-M3 v.0.9) was developed starting from the *RedSIB* (SMART-M3 v.0.4) implementation.

The *RedSib* is a Semantic Information Broker based on *Redland RDF store* [2] and supports the following operations:

Operation	
Join/Leave	-
Insert	RDF Triple
Remove	RDF Triple
Update	RDF Triple
Query	RDF Triple / SPARQL
Subscribe/Unsubscribe	RDF Triple / SPARQL

The *Semantic Event Broker* (SEB) supports all of the above mentioned primitives and introduces two new features:

- SPARQL update
- Time management

SPARQL 1.1 Update[4]is a W3C language to update RDF graphs. A basic operation in the SPARQL Update language is "DELETE/INSERT" that can be used to perform conditional update operations. A conditional update is an atomic operation consisting of: a query (i.e. a WHERE pattern) and an update (i.e. RDF triples to be inserted and removed). The update is executed only if the query provides result(s). In this case, if the RDF triples to be inserted and deleted contain variables, then the variables are substituted with the bindings obtained in the WHERE pattern matching.

Time management is a new feature that enables time sensitive behaviors. The SEB hosts the shared current time value and provides specific primitives to handle the time. KPs can retrieve the current time from the SEB to synchronize themselves by using the *get_sib_time()* function embedded within a SPARQL pattern. In addition to retrieve the current time, *time management* enables KPs to specify when the SPARQL update should be performed (*delayed SPARQL update*).

These new features simplify KP application logic design and encourage application programming style rethinking. KP lifetime logic (i.e. between *join* and *leave*), can be implemented only with *delayed SPARQL update* and *SPARQL Subscription*.

Delayed SPARQL update

The *Delayed SPARQL update* primitive performs a conditional SPARQL update at a specific time. This primitive has two input parameters: the SPARQL update itself and the desired time for its execution. If the time is not specified the operation is scheduled as soon as it is received by the SEB.

Underneath you'll find some examples of SPARQL messages where the *Delayed SPARQL update* is used. This primitive replaces and extends the *insert*, *remove* and *update* primitives originally available in the original SSAP protocol.

```
PREFIX unibo:<http://www.UniboExample/ExampleOntology.owl#>

DELETE  DATA {

        unibo:LampActuator_1 unibo:HasValue "False" .

        unibo:LampActuator_2 unibo:HasValue "True" .

        }

INSERT DATA {

        unibo:LampActuator_1 unibo:HasValue "True" .

        unibo:LampActuator_2 unibo:HasValue "False" .

        }
```

Example of SPARQL update data (concrete triples)

In the first line the PREFIX keyword is used to make the operation more compact and readable by defining a short label (i.e. *unibo*) for the ontology namespace: http://www.UniboExample/ExampleOntology.owl#. Typically the PREFIX keyword is used like this (i.e. for ontology namespaces), but it can be used also to shorten any other URI. In practice, the SPARQL engine will just replace each prefix label present in the operation with the URI string.

The SPARQL primitive includes also two fields specifying the RDF triples to be inserted in and removed from the RDF store. When this SPARQL update is sent to the SEB, the SEB performs the following atomic operation: two triples are deleted and two triples are inserted, with the following result: the instance *LampActuator_1* is set to on (it was off originally) and the instance *LampActuator_2* is set to off (it was on originally).

With the *SPARQL update* the option exists to update information if and only if a set of conditions are satisfied. Going back to the previous example, let's suppose now that we want to switch on *LampActuator_1* and switch off *LampActuator_2* only if *SensorPresence_1* detects a presence i.e. only if the triple (*unibo:SensorPresence_1, unibo:HasValue, "True")* exists in the RDF store. This can be done with the following atomic SPARQL DELETE/INSERT operation:

```
PREFIX unibo:<http://www.UniboExample/ExampleOntology.owl#>
DELETE  {
        unibo:LampActuator_1 unibo:HasValue "False" .
        unibo:LampActuator_2 unibo:HasValue "True" .
        }
INSERT  {
        unibo:LampActuator_1 unibo:HasValue "True" .
        unibo:LampActuator_2 unibo:HasValue "False" .
        }
WHERE {
        Unibo:SensorPresence_1, unibo:HasValue, "True"
        }
```

Example of SPARQL update with WHERE pattern

Conditions are specified in the WHERE pattern (which specifies a condition check query). Only when there is a solution matching the WHERE pattern, the updates requested with the DELETE and/or INSERT sentences are performed.

The SPARQL update is logically an atomic operation consisting of a query and an update.

With the *Delayed SPARQL update* programmers can specify the time when their SPARQL update operation is executed. The assumed time base is *UNIX time* represented with six decimal digits. In the current implementation the scheduled time is specified inside the SPARQL update message by defining a specific PREFIX label (*scheduled_time*): this approach was adopted to maintain backward compatibility with the SSAP protocol.

Let’s consider now the following SPARQL update message:

```
PREFIX scheduled_time:<1378737852.770930>

PREFIX unibo:<http://www.UniboExample/ExampleOntology.owl#>

PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

DELETE  {

        ?LampActuator unibo:HasValue "False" .

        }

INSERT  {

        ?LampActuator unibo:HasValue "True" .

        }

WHERE {

        ?LampActuator, rdf:type, unibo:LampActuator .

        ?LampActuator, unibo:HasValue, “False” .

        }
```

Example of delayed SPARQL update with WHERE pattern and variables

The operation will be performed by the SEB only at the UNIX time expressed in the *scheduled_time* prefix (i.e. *1378737852.770930*). As already described, when the KP sends its SSAP message to the SEB, the SEB receives the requested primitive but it will execute it only at the specified time. At the specified SEB time, the update is performed. In the example at UNIX time *1378737852.770930* all lamps are switched on. More in detail, in this example all *LampActuator* class instances that are *off* are retrieved (by the query specified in the WHERE patterns *(?LampActuator, rdf:type, unibo:LampActuator . ?LampActuator, unibo:HasValue, “False”*), then they are switched on.

It is up to the programmer to compute the desired argument for the scheduled_time prefix.

SPARQL Update Template

PREFIX Namespace Definition

PREFIX scheduled_time:<op_time**>** (only for postponed operation)

DELETE

{

Subgraph to be removed

}

INSERT

{

Subgraph to be inserted

}

WHERE

{

Conditions based on graph pattern

}

SPARQL Subscription

M3 is a content-based publish/subscribe architecture. One of its main specificities is its ability to react to context changes according to the subscribe/notify paradigm. In Smart-M3 V.0.9, subscriptions are expressed with the *SPARQL subscription primitive* (see the example below) which consists of:

- The required set of namespace definitions each specified by a PREFIX keyword
- The "SELECT" keyword followed by the addressed variables.
- The "WHERE" keyword followed by the query pattern.

The SPARQL subscription works as follows:

- When a *SPARQL Subscription* request is sent to the SEB, the SEB immediately replies with the query result;
- any subsequent RDF database change which matches the SPARQL query pattern defined in the subscription is automatically notified to the subscribed KP: Notifications include added and removed results.

For instance, if a KP performs the subscription shown below, 8it will first receive initial results specifying the URIs of the *unibo:LampActuator* instances that have the *unibo:hasValue* attribute assigned with value "True". Afterwards the KP will be notified both whenever information matching the query is inserted and whenever triples are removed, so that the original results become obsolete. In this case, for example, the original results are made obsolete either by turning unibo:LampActuator "Off" or by completely removing the instance from the RDF database.

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?LampActuator

WHERE  {

        ?LampActuator rdf:type unibo:LampActuator .

        ?LampActuator unibo:HasValue "True" .

        }
```

Example of SPARQL subscription (only "SELECT" queries are currently supported)

SPARQL Subscribe template

```
PREFIX Definition
SELECT Variables Definition
WHERE
        {
        Conditions
        }
```

Programming approach

M3 architecture supports event driven programming.

Consumers and producers are not directly connected, on the contrary, they are loosely connected through the SEB.

The M3 programming paradigm is inherently distributed and multi-language.

Interacting agents do not have to agree on their own proprietary data format as information semantics is specified by a shared ontology stored in the SEB semantic store.

M3 applications may react to both local and remote events, and the SEB may notify circumstances that relate local and remote contexts.

Producers and consumers are clearly decoupled and the application of the separation of concerns principle is straightforward.

The M3 architecture is inherently oriented to enable interoperability at information level in the emerging vision of the Internet of Things, and its implementations aim to provide frameworks for open Cyber Physical systems.

M3 applications interact with the physical world with sensors and actuators; while sensors produce new information to the SEB actuators needs to be reactive to context changes (typically detected by sensors) published to the SEB.

Sensors and actuators are considered “legacy devices”. They are interfaced to the SEB through KPs.

Beside acting as interface between legacy devices and the SEB, KPs may implement data abstraction functionalities, consisting in the creation of more meaningful information from row sensor data; in this case they are called *aggregator KPs*. Eventually they may implement the application business logic.

KPs interact with the SEB with multi-language APIs which handle the SSAP protocol and, particularly, convey the *delayed SPARQL update* and the *SPARQL subscription* primitives to the SEB.

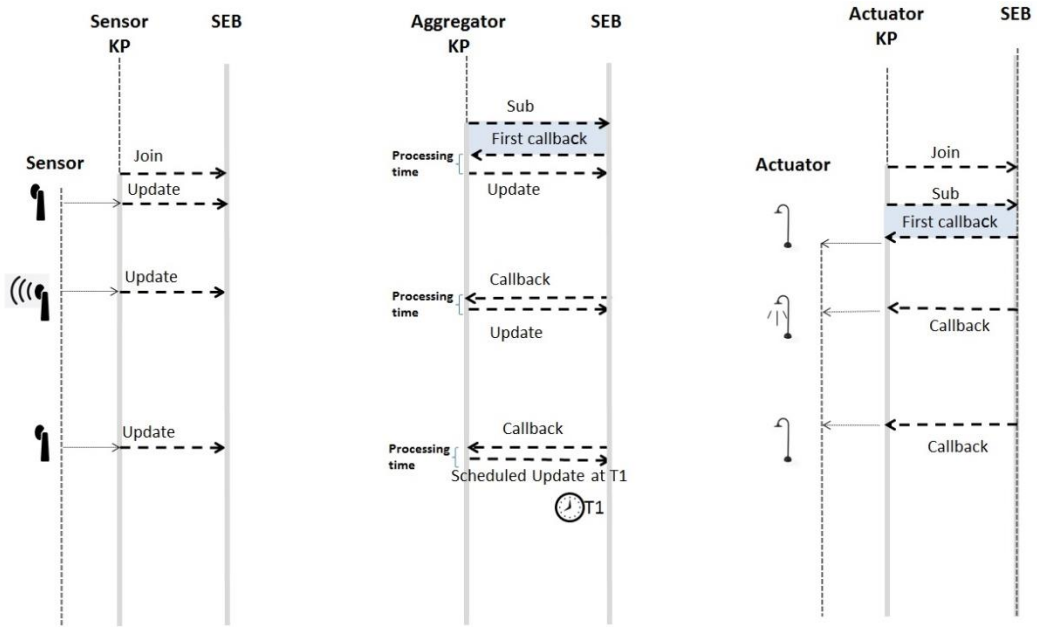

Fig. 2: Sensor, Aggregator and Actuator KPs programming model

Fig.2 shows how:

- sensor KPs interact with the SEB with the *delayed SPARQL update* primitive only
- actuator KPs interact with the SEB with the *SPARQL subscription* primitive only
- aggregator KPs (e.g. services) interact with the SEB with both primitives, *SPARQL subscription* and *delayed SPARQL update*; through the subscriptions they are notified of context changes detected by the SEB; they share their output with other KPs through the SEB thanks to *delayed SPARQL update*.

The delayed SPARQL update primitive has the capability to perform sophisticated and semantics-rich information processing activity, relieving the KPs from such a burden, and simplifying the KP application logic.

Application model and programming style

A very simple, powerful and modular application model is proposed in this tutorial.

According to this model the application consists of the following KPs:

- One KP per sensor (producer KPs) responsible for updating the SEB with up-to-date sensor parameters
- One KP per actuator (consumer KPs) responsible for reacting to appropriate SEB updates with corresponding actuators actions
- One or more KPs responsible for the application specific business logic. They react to SEB updates issued by the sensors (i.e. producer KPs) with SEB updates that will trigger the actuators (i.e. actuator KPs).

Thus sensors, actuators and business logic modules are entirely decoupled, they are not bound to be located on the same platform and each of them may be replaced without any impact on the other components.

Next a simple example is presented to illustrate how to design and build a M3 application based on the application model above depicted.

Let's consider a simple physical world scenario consisting of a lamp (i.e. the actuator) and a presence sensor. They are managed by two legacy adapter KPs. A third KP implements a service (the LampManager) which is responsible of:

- Turning on the lamp when the sensor detects a presence
- Turning off the lamp if the sensor does not detect any presence for a 10 s time interval.

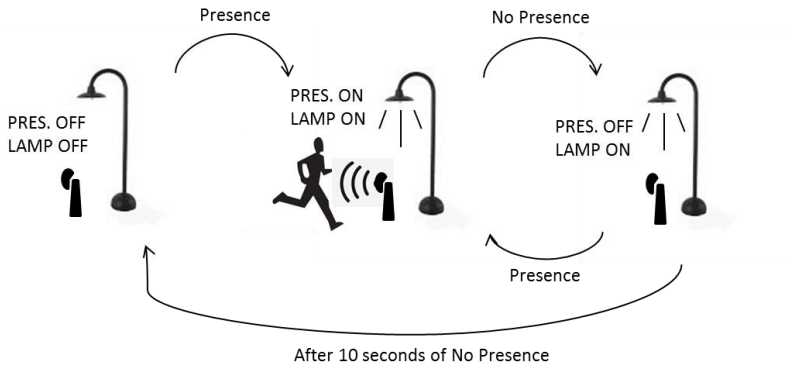

Fig. 3 Application Overview

This simple application shows the programming approach for applications supported by the SEB platform. The Python code for this example is available on the SEB repository on Smart-M3 SourceForge website[1].

The reader is encouraged to draw the finite state machine that describes this application (Fig. 3).

To develop a Smart-M3 application, starting from its requirements and constrains, its software architecture should be designed first. This design work consists of two concurrent steps:

- ontology definition
- application partitioning into KPs and information flow specification

In this application the simple ontology shown in Fig. 4 represents and describes all relevant entities and the relations among them. Specifically Fig.4 shows that we have one instance for each class (*Sensor* and *Actuator* are the classes), while two data properties describe their values and timestamps.

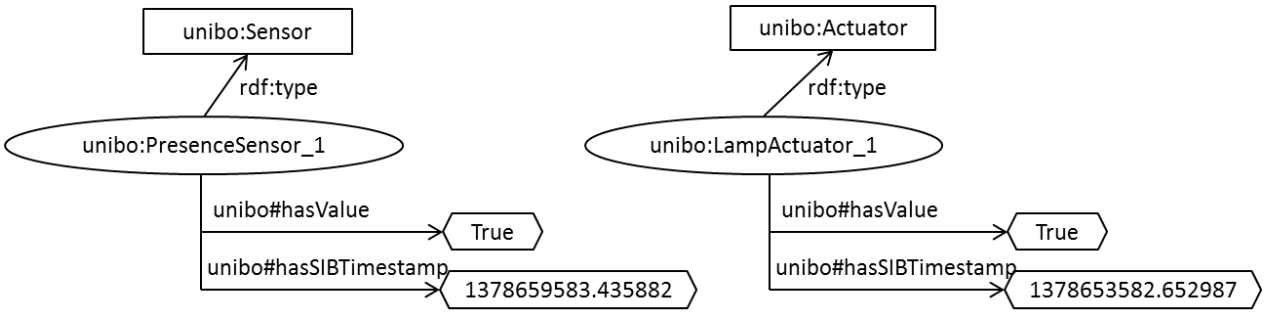

Fig. 4: Information representation for the proposed application example

The KPs definition is simple and intuitive; the following KPs implement the application:

- a Presence Sensor KP
- a LampActuator KP
- a LampManager KP implementing the application business logic.

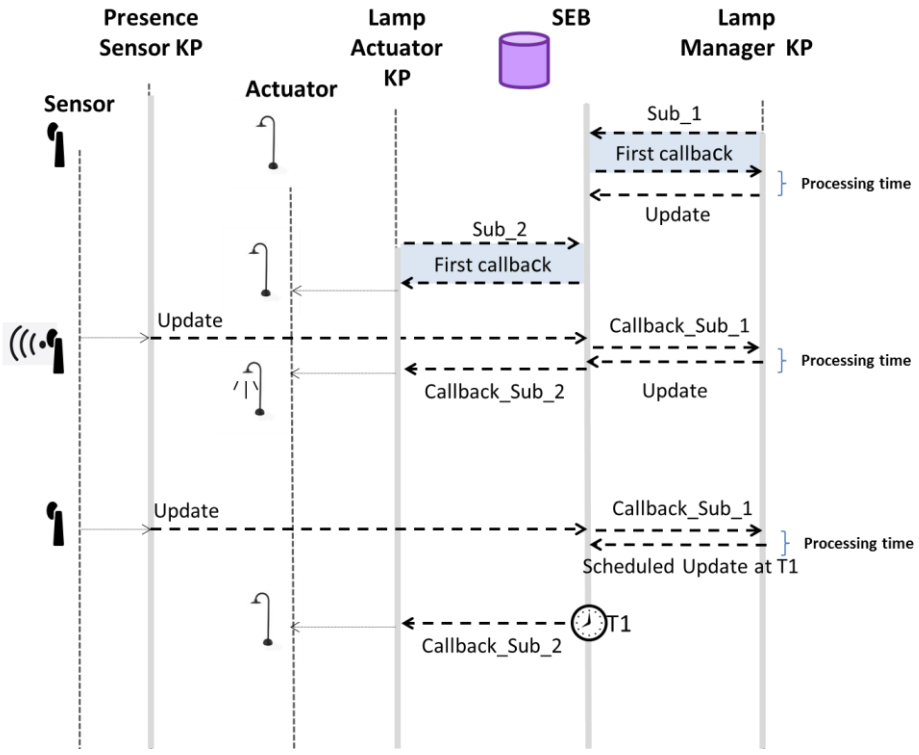

Fig. 5: KPs and associated sequence diagram describing the information flow

Presence Sensor KP

This KP is the interface between the presence sensor and the SEB. It is responsible for sharing with other KPs – through the SEB -the most recent value of the presence sensor. Every time the presence sensor changes the presence value (i.e. *presence/no presence)*, the KP performs a SPARQL update to keep the SEB up to date. The following *SPARQL Update* operation is invoked by the *Presence Sensor KP* when "presence" is detected by the physical sensor handled by the KP.

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

INSERT  {

        unibo:PresenceSensor_1 unibo:HasValue "True" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?real_time .

        }

DELETE  {

        unibo:PresenceSensor_1 unibo:HasValue "False" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?timestamp_sens .

        }

WHERE   {

        unibo:PresenceSensor_1 unibo:HasValue "False" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?timestamp_sens .

        BIND (get_sib_time() AS ?real_time)

        }
```

Updating a sensor value with its time stamp: the BIND clause retrieves the current time value from the SEB with the *get_sib_time()* function

In this example the WHERE pattern check is made to verify the following condition inside the SEB: "the PresenceSensor_1 value is not 'presence detected' and the timestamp of this sensor instance has a value". In addition to such condition checking, the current UNIX time is bound to the variable named *?real_time* inside the WHERE pattern. The time is retrieved by the SEB using the *get_sib_time()* function and bound to the variable by using the BIND pattern.

If the condition is met the DELETE and the INSERT operations are executed. The variables in the DELETE and INSERT graphs are replaced with solutions obtained in the WHERE pattern matching. I.e. with the DELETE sentence the off value and the old timestamp are removed; while with the INSERT sentence the presence value is set to "True" and the timestamp is updated with the value of the *?real_time* variable. The same approach is taken to change the PresenceSensor_1 value in the SEB when *no-presence* is detected by the physical sensor (see next text box).

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

INSERT  {

        unibo:PresenceSensor_1 unibo:HasValue "False" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?real_time .

        }

DELETE  {

        unibo:PresenceSensor_1 unibo:HasValue "True" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?timestamp_sens .

        }

WHERE   {

        unibo:PresenceSensor_1 unibo:HasValue "True" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?timestamp_sens .

        BIND (get_sib_time() AS ?real_time)

        }
```

Updating a sensor value with its time stamp: the BIND clause retrieves the current time value from the SEB with the *get_sib_time()* function

LampActuator KP

This KP is the interface between the SEB and the actuator reacting to events. The actuator KP just needs to subscribe to its context of interest. In this example the KP subscribes to the variations of the lamp properties in the SEB (both the value and the timestamp). This is done with a *SPARQL subscription*. Once the subscription is performed the SEB will notify the KP of all changes. Based on the notifications the KP will then interact with the physical world via its legacy interface (i.e. the lamp can be properly switched on or off according to the notification).

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

SELECT ?value ?timestamp_lamp

WHERE   {

        unibo:LampActuator_1 unibo:HasValue ?value .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?timestamp_lamp .

    }
```

Subscribing to two variables: ?value and ?timestamp_lamp

LampManager KP

This KP is an aggregator which means that it just modifies information in the SEB (i.e. it does not have any direct interaction with the physical world). It reacts to the sensors changes, applies some intelligence (taking decisions), and manages the lamp property values by publishing new information into the SEB. The programming model reflects these functionalities. The KP needs to subscribe to the context of interest and to manage the corresponding notifications in the appropriate code section (called callback), eventually updating the SEB semantic store with new information. In this example, the LampManager needs to react as follows:

- performing *action_1* when the presence sensor detects a presence (*event_1*)
- performing *action_2* when the presence is not detected anymore (*event_2*)

Two subscriptions may be issued in order to have separate callback sections for *event_1* and *event_2*. In this way the requested actions can be managed separately.

The following *SPARQL subscription* specifies *event_1* context.

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

SELECT ?value

WHERE  {

        unibo:PresenceSensor_1 rdf:type unibo:Sensor .

        unibo:PresenceSensor_1 unibo:HasValue ?value .

        FILTER ( ?value  = "True")

    }
```

SPARQL Subscription to detect when PresenceSensor_1 changes to "True" (event_1)

Whenever Presence Sensor_1 value is changed in the SEB to share the information that a presence was detected (Presence Sensor_1,HasValue, "True") the KP is notified and in the callback the lamp can be turned on (*action_1*) with the *delayed SPARQL update* underneath.

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

INSERT  {

        unibo:LampActuator_1 unibo:HasValue "True" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?time_turning_on .

        }

DELETE  {

        unibo:LampActuator_1 unibo:HasValue "False" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?timestamp_lamp .

        }

WHERE   {

        unibo:LampActuator_1 unibo:HasValue "False" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?timestamp_lamp .

        BIND (get_sib_time() AS ?time_turning_on)

        }
```

SPARQL update to perform action_1

The above *delayed SPARQL update* updates the lamp sub-graph: it sets to on the lamp actuator instance in the SEB and updates its timestamp; the function *get_sib_time()* within the BIND sentence synchronizes *action 1* with the SEB time.

The following *SPARQL subscription* triggers a notification when the sensor instance in the SEB changes from a presence to a no-presence value (*event_2*):

```
PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

SELECT ?value ?timestamp

WHERE   {

        unibo:PresenceSensor_1 rdf:type unibo:Sensor .

        unibo:PresenceSensor_1 unibo:HasValue ?value .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp ?timestamp .

    FILTER ( ?value  = "False")

    }
```

SPARQL Subscribe to detect when PresenceSensor_1 changes to “False” (event_2)

In the callback associated to *event_2* the lamp manager logic requires to switch off the lamp after 10 seconds if no presence is detected during these 10 seconds (*action_2*). The subscription notification delivers the time of the last presence detection (?timestamp) so that the update can be scheduled at the time + 10. For example if the SEB returns the last timestamp with value 1378659583.435882, the *delayed SPARQL update* will be scheduled at 1378659593.435882.

```
PREFIX scheduled_time:< 1378659593.435882>

PREFIX unibo:<http://www.UniboExample/LampExampleOntology.owl#>

INSERT  {

        unibo:LampActuator_1 unibo:HasValue "False" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?time_turning_off .

        }

DELETE  {

        unibo:LampActuator_1 unibo:HasValue "True" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?timestamp_lamp .

        }

WHERE   {

        unibo:PresenceSensor_1 unibo:HasValue "False" .

        unibo:PresenceSensor_1 unibo:HasSIBTimestamp "1378659583.435882" .


        unibo:LampActuator_1 unibo:HasValue "True" .

        unibo:LampActuator_1 unibo:HasSIBTimestamp ?timestamp_lamp .

        BIND (get_sib_time() AS ?time_turning_off)

        }
```

Delayed SPARQL update to perform action_2

In order to ensure consistency to the Delayed SPARQL update results, the WHERE pattern checks that:

- neither the presence sensor timestamp nor its value were changed in the time window between the update delivery time and the update scheduled time (first two clauses of the WHERE pattern)
- the lamp is on at the scheduled time (third clause of the WHERE pattern)

Hands on

Downloading, installing and running the Semantic Event Broker

Downloading

SMART-M3 SourceForge website is http://sourceforge.net/projects/smart-m3/. Here, in the *files* section, please download *Smart-M3-RedSIB_0.9*.

Installing

On Ubuntu (tested on v.12.04) please execute the *install.sh* script. The script automatically installs the SEB and a customized *Virtuoso* RDF store. During the installation process a user interface asks for a password (and for its confirmation): this password (to be selected by the user) is required only to run the SEB with *Virtuoso*.

To install from the SEB folder please issue the following command from the terminal:

```
$ sh install.sh
```

Running

Once the installation is completed please run the SEB. This requires two terminals.

From the first terminal please enter:

```
$ redsibd
```

From the other terminal please enter:

```
$ sib-tcp
```

To get more information on how to configure the SEB please add *--help* to both commands. As an example the SEB can be run with its RDF store on ram rather than on *Virtuoso* (*Virtuoso* is recommended for a large amount of triples or when a persistent triple store is required). The SEB can also be run by setting a customized port (for the RDF store). On the *readme.txt* file you can find more detail about the current SEB release and about the installation process.

Examples

The folder named *SEB_examples* includes two application examples. Both of them are coded in Python.

The *presence-lamp* folder contains the example enlightened in the previous section. Here the file *UNIBOLampExample.owl* is the simple ontology proposed for the application. It was created with Protégé [5] and it includes the initialization time description of the system actors (as shown in fig. 4).

To run this demo please run from your terminal the following python KPs:

1. *SEB_initilize.py* is the KP responsible to load the shared ontology (*UNIBOLampExample.owl*)
2. *SEP_Gateway_Sensor.py* is the KP that simulates the presence sensor: sensor events (i.e. presence / no-presence detection) are simulated from the keyboard: when “t” or ”f” key is pressed (i.e. true or false) the KP updates the SEB accordingly.

3. *SEP_Gateway_Actuator.py* is the actuator KP(i.e. it is a consumer). It is notified when the lamp has to be turned on or off. During the simulation a message reporting the lamp status is displayed on the terminal.
4. *SEP_Processing_LampManager.py* is the aggregator KP. It consumes (i.e. it is notified when information originated by the sensor is stored in the RDF store) and in turn it produces information to feed the actuator.

References

1. SMART-M3 on SourceForge website, http://sourceforge.net/projects/smart-m3/
2. F. MORANDI, L. ROFFIA, A. D'ELIA, F. VERGARI, T. SALMON CINOTTI. (2012). *RedSib: a Smart-M3 Semantic Information Broker Implementation,* Proceedings of the 12th Conference of Open Innovations Association FRUCT, Eds Sergey Balandin and Andrei Ovchinnikov, Oulu, Finland, November 5-9, 2012,SUAI SAINT-PETERSBURG, (pp. 86-98) ISSN 2305-7254.
3. SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/][]
4. SPARQL 1.1 Update, http://www.w3.org/TR/sparql11-update/
5. Protégé Ontology Editor, http://protege.stanford.edu/
6. L. ROFFIA, A. D'ELIA, F. VERGARI, D. MANZAROLI, S. BARTOLINI, G. ZAMAGNI, T. SALMON CINOTTI, J. HONKOLA. (2010). *A Smart-M3 lab course: approach and design style to support student projects.* Invited paper, 8th Conference of Finnish-Russian University Cooperation in Telecommunications (FRUCT).Lappeenranta (Finlandia). 9-12 Nov 2010. (pp. 142 - 153). ISBN: 978-5-8088-0567-5. SAINT-PETERSBURG: SUAI.
7. E. OVASKA, T. SALMON CINOTTI, A. TONINELLI. (2012). *The Design Principles and Practices of Interoperable Smart Spaces.* In Advanced Design Approaches to Emerging Software Systems: Principles, Methodology and Tools. (pages 18-47) Liu Xiaodong, Li Yang Eds., IGI Global, 2012, doi:10.4018/978-1-60960-735-7.ch002 ISBN: 9781609607357 -http://www.igi-global.com/Bookstore/TitleDetails.aspx?TitleId=49573. Hershey – PA U.S.A.
8. A D'ELIA, J.HONKOLA, D.MANZAROLI, T.SALMON CINOTTI. (2011). *Access Control at Triple Level: Specification and Enforcement of a Simple RDF Model to Support Concurrent Applications in Smart Environments*. In *Smart Spaces and Next Generation Wired/Wireless Networking*. (pp. 63 - 74) a cura di Sergey Balandin, Yevgeni Koucheryavy, Honglin Hu. ISBN: 978-3-642-22874-2. LNCS 6869, presented at ruSMART11 (4th Conference on Smart Spaces, St. Petersburg, August 2011). Heidelberg: Springer

.

SMART-M3 v.0.9:
A semantic event processing engine supporting information level interoperability in ambient intelligence

A quick start tutorial

This tutorial is addressed to all post-graduate students in Electronic Engineering and Information Engineering at the *Scuola di Ingegneria e Architettura* of the University of Bologna attending the following courses: *Laboratory of Interoperability of Embedded Systems, Calcolatori Elettronici M* and *Attività Progettuale di Calcolatori Elettronici M*.

This tutorial is about the Semantic Event Processing infrastructure deployed in the School Lab named LAB2. It includes the guidelines to build distributed applications where clients interact through an *event broker.* Clients may interact also with the physical space and inter-client information interoperability is based on a shared knowledge representation model named *ontology*. This tutorial is focused on client design and on the primitives that provide the means for client-*event broker* interaction.