Enabling the business-based
Internet of Things and Services

(FP7 257852)

D7.3.2 Technical description of the implementation of Data and Event Management models 2

Published by the ebbits Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2009.1.3: Internet of Things and Enterprise environments

Document control page

Document file:	D7.3.2 Implementation of Data and Event Management models 2.docx
Document version:	1.0
Document owner:	Matts Ahlsen (CNet)
Work package:	WP7 – Event Management and Service Orchestration
Task:	T7.2, T7.3, T7.4
Deliverable type:	PU
Document status:	☒ approved by the document owner for internal review ☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Matts Ahlsén	2012-01-13	Document created based on D7.3.1 for revision and update.
0.2	Matts Ahlsén, Peeter Kool, Andreas Persson (CNet)	2012-01-18	Initial event model suggested
0.3	Matts Ahlsén, Peeter Kool, Andreas Persson, Peter Rosengren (CNet)	2012-02-10	Event model and architecture revised
0.5	Matts Ahlsén, Peeter Kool, Andreas Persson, (CNet)	2012-02-14	Event model revised, EPN /EPA components
	Matts Ahlsén, Peeter Kool,(CNet), Peter Kostelnik (TUK)	2012-02-25	Description of event ontology extensions and semantic event processing use cases
0.8	Matts Ahlsén, Peeter Kool, Andreas Persson, (CNet), Peter Kostelnik (TUK)	2012-02-28	Event model and LinkSmart Event Manager adaptations. Event Ontology examples.
0.9	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2012-03-09	Version for internal review
1.0	Matts Ahlsén, Peeter Kool, Andreas Persson, Peter Rosengren (CNet), Peter Kostelnik (TUK)	2012-03-14	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Mark Vinkovitz (FIT)	2012-03-12	Approved with major comments
Claudio Pastrone (ISMB)	2012-03-11	Approved with comments
Mauricio Caceres (ISMB)	2012-03-13	Approved with comments

Index:

1. Executive summary 5

2. Introduction 6

2.1 Purpose, context and scope of this deliverable 6

2.2 Background 6

3. Event processing architecture 8

3.1 Framework 8

3.2 The initial event architecture 8

3.3 The revised event processing architecture 9

3.4 The EPA (Event Processing Agent) 10

3.4.1 Event Receiver 11

3.4.2 Event DB 12

3.4.3 Rule Engine 12

3.4.4 Event Ontology 12

3.4.5 Timer 12

3.4.6 Event Router 12

4. Event structure and semantics 14

4.1 Event properties 14

4.1.1 Event XML structures 14

4.2 Semantic support – Event ontology 16

4.2.1 Examples of event model in use 17

4.2.2 Subset of event ontology 17

4.3 Event lifecycle 20

5. Extensions to the LinkSmart Event Manager 22

5.1 Extension to the LinkSmart Event Manager 22

6. Future development and plans 25

7. References 26

1. Executive summary

This deliverable describes the design and proposed implementation of the event management subset in the ebbits architecture. This report is the second version in a series of four successive revisions, aligned with the iterative development plan of the project. The main updates and additions compared to the initial design (reported in D7.3.1) are the following:

- A generalized architecture for an event processing network (EPN) is introduced
- Event filtering, transformation and aggregation based on event processing agents (EPAs)
- Definition of a generic event data structure with associated meta data
- Revisions of the Event Ontology to represent event meta data
- Revised implementation of the LinkSmart Event Manager

The work is performed in the context of Tasks 7.2-7.4 in work package 7, in which the overall objective is to research, develop and implement the complex Event Management and service orchestration structure of the ebbits platform.

The scope of the work reported in the deliverable includes the detection of events from devices/sensors (proxies), the propagation and semantic enrichment of events, and their mapping to services via business rules. The emphasis is on the detection and semantic enrichment of low-level events, and the functionality of the event manager component.

The report introduces a framework for event management in ebbits and relates this to an initial sketch of the overall ebbits architecture. We then describe the application and adaptation of the underlying baseline technology for networked event management derived from the results of the Hydra project.

The framework for event management in ebbits consists of the following subsets,

- The EPN implements a network of connected event processing components, specialized in event filtering, transformation and aggregations.
- The event processing model, implements a set of event and data management objects and data types, including an XML serialization. This model is partly derived from the event ontology. The model is used by the system to store and communicate event instances with related objects.
- The event core ontology. This is a support ontology used by the event management system for describing and relating all fundamental event concepts in ebbits. The Event ontology serves as a decision support component, which allows specific events being queried and inferred.
- The LinkSmart publish-subscribe mechanism.

In this second iteration of the event subsystem design, the emphasis has been on the adapting the initial publish-subscribe event model into a more flexible event processing network architecture. The components for event management have been further developed with services interfaces and related to the overall ebbits system architecture. The event architecture is subject to further refinement in a series of four successive revisions and prototype validations, aligned with the iterative development plan of the project.

2. Introduction

2.1 Purpose, context and scope of this deliverable

The purpose of this deliverable is to describe a design and implementation for the event management subset in the ebbits architecture. This report is the second version in a series of four successive revisions, aligned with the iterative development plan of the project.. The main updates and additions compared to the initial design (reported in D7.3.1) are the following:

- A generalized architecture in terms of an event processing network (EPN) is introduced.
- Filtering, transformation and aggregation based on components referred to as event processing agents (EPAs)[1].
- Definition of a generic event data model with a corresponding XML serialization.
- Revisions of the Event Ontology to represent event meta data and data, with relationships to other ebbits entities.
- Revised implementation of the LinkSmart LinkSmart Event Manager [3].

The work is performed in the context of Tasks 7.2-7.4 in work package 7. The overall work package objective is to research, develop and implement the complex Event Management and service orchestration structure of the ebbits platform. It will design how physical world events are captured, processed and transformed to application-oriented semantic events that are managed by the ebbits platform. The work package will implement a semantic event and data management server that will provide the needed information management and service orchestration functionality to support the ebbits requirements.

The scope of the work reported in this deliverable includes the detection of events from devices/sensors, the propagation and semantic enrichment of events, and their mapping to services via business rules. The emphasis is on the detection and semantic enrichment of low-level events, and the functionality of the event manager component.

This task builds on the initial results of tasks 7.1 and 7.2 of the work package which both provide a background to and a basis for the further design and implementation of the ebbits event management subsystem. The higher level event processing and business rules execution are topics in other work packages, primarily WP6.

2.2 Background

The *Internet of People Things, People and Services* (IoPTS) is the current vision for an Internet encompassing any IT artefact, information source or service. The ebbits project "Enabling business-based Internet of Things and Services" aims at developing an interoperability platform for a real world populated IoPTS domain.

The *ebbits project* will develop the architecture, technologies and processes, which allow businesses to semantically integrate the IoPTS into mainstream enterprise systems and support interoperable real-world, on-line end-to-end business applications. It will provide semantic resolution to the IoPTS and hence present a new bridge between backend enterprise applications, people, services and the physical world, using information from tags, sensors, and other devices and performing actions on the real-world. The ebbits platform will feature a Service Oriented Architecture (SOA) based on open protocols and middleware, effectively transforming every subsystem or device into a web service with semantic resolution.

Event management is central in this architecture as a conceptual model and execution mechanism for the integrating of physical world events with enterprise systems, pursuing the vision for the IoPTS. The architecture style of ebbits can be characterized as *Event-driven SOA*, integrating intelligent services with advanced semantic event processing and business rules. Thus events can

[1] EPA and EPN are established terms in the area of Complex Event Processing [1][2].

range from lower level atomic signals to higher level semantically enriched message carriers. On a higher abstraction layer ebbits events are mapped to business rules, which can make use of intelligent services, to implement the business logic for reporting, actuation of devices.

3. Event processing architecture

3.1 Framework

The framework for event management in ebbits consists of the following subsets.

- The EPN implements a network of connected event processing components, specialized in event filtering, transformation and aggregations.
- The event processing model implements a set of event and data management objects and data types, including an XML serialization. This model is partly derived from the event ontology. The model is used by the system to store and communicate event instances with related objects.
- The event core ontology. This is a support ontology used by the event management system for describing and relating all fundamental event concepts in ebbits. The Event ontology serves as a decision support component, which allows specific events being queried and inferred.
- The LinkSmart publish-subscribe mechanism.

3.2 The initial event architecture

An initial sketch of an ebbits event and data fusion architecture was presented in the first iteration of this document (D7.3.1) and is depicted in the figure below. This architecture was intended as a frame of reference in the further design of event and data management in ebbits. We also note that this was an initial basic architecture resulting from the first analysis and state-of-the-art work in WP7, and which was intended for further revision.

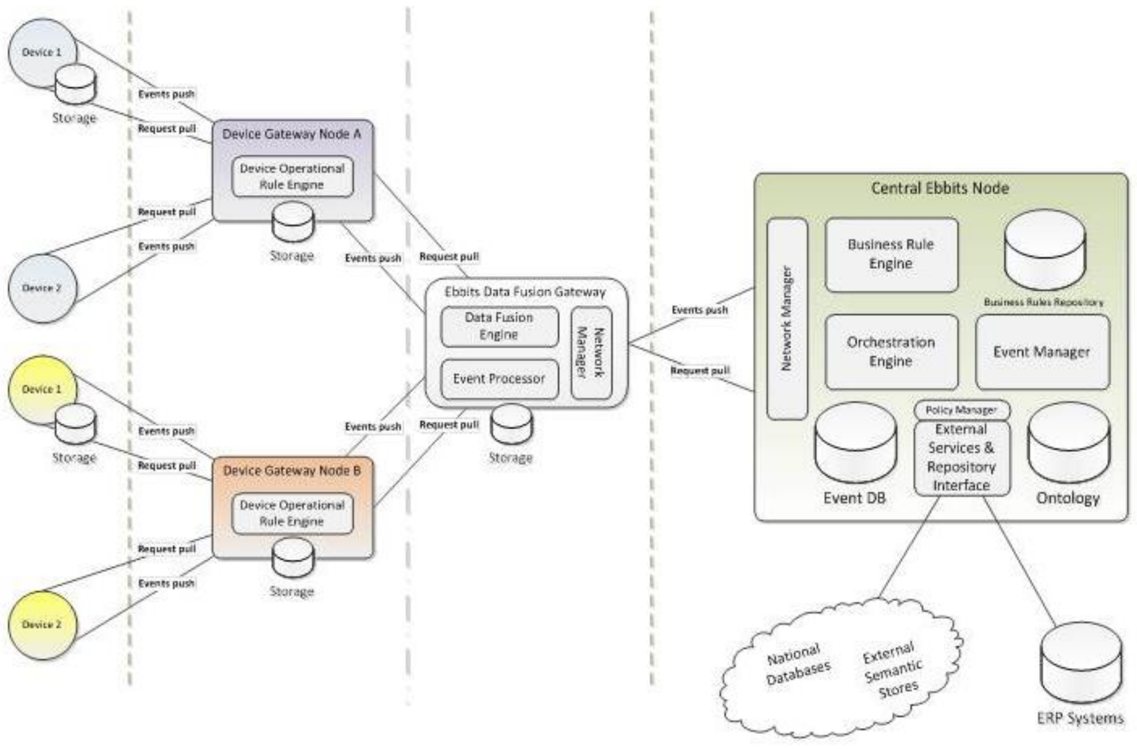

Figure 1: ebbits initial event architecture (from D7.3.1)

The architecture perspective here is a functional component model, emphasizing the flow of data/events and control (it could also be projected as a layered architecture). Device components (layer) at the left (bottom) and the Business rules and services components at the far right (top).

3.3 The revised event processing architecture

The revised ebbits event processing architecture contains a number of event processing agents, EPA, that process and route events. The EPAs form a so called Event Processing Network (EPN) of agents which may provide different forms of event processing capabilities, depending on the needs of applications and the configuration of the ebbits platform.

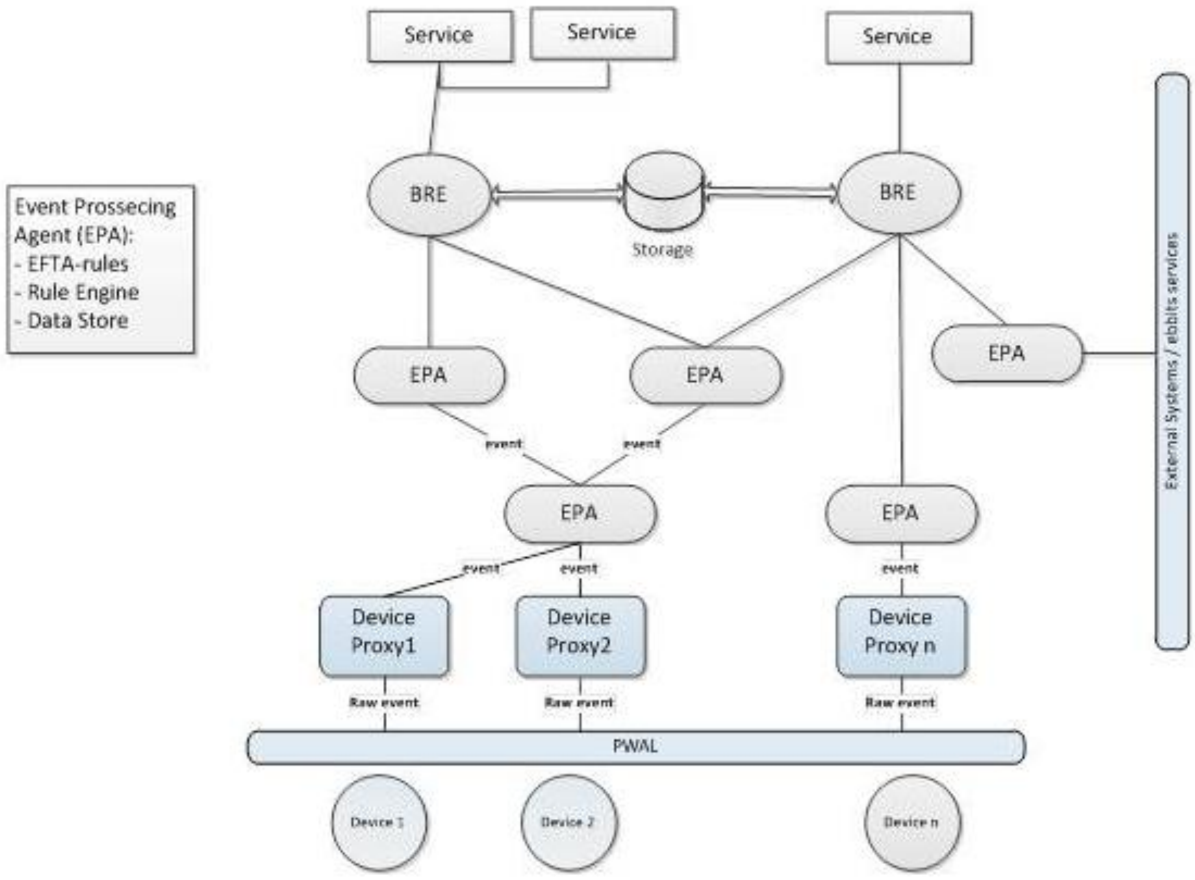

Figure 2: ebbits Event Processing Network (EPN)

Device events are passed from devices through the PWAL (Physical World Adaptation Layer, c.f. WP8) to the Device Proxies, which forward the events into the event processing network. The final consumer of the events are the BREs (Business Rule Engines) which can invoke actions and external services based on rules (which may results in new events being generated).

The EPAs mission is to aggregate, filter and transform events depending on rules etc. In principle, there can be an arbitrary number of levels of EPAs connected to each other, and using the routing mechanisms (which will be explained in the EPA section below) we can have EPAs which specialize in certain types of events and with different processing capabilities. This will allow for a more scalable architecture where we can add EPAs as necessary. Because of the routing mechanism we can add additional EPAs wherever needed in the deployment, i.e., if we get a bottle neck at the top-level we can add EPA nodes there. This means that the Event Processing architecture does not need to be pyramid shaped but it can be of any shape (taking into account the restrictions of EPA independence).

An important restriction for the EPA is that they should be considered to be more or less independent of each other and the only flow in-between EPAs are the events that are passed through the event processing network. This implies that rules that rely on historical events can only work on events that have passed the EPA; all other events are unknown to it. This will of course restrict how the EPN can be deployed for an application but we believe that the autonomy of EPAs will give much better scalability, compared to a more tightly coupled approach.

In fact, from an event processing architecture point of view there could be number of EPAs specialized for or associated with specific ebbits functionality, such as the context manager which can be seen as an EPA specialized on managing context.

3.4 The EPA (Event Processing Agent)

The Event Processing Agent is a fundamental architectural component in the Event Processing Network of ebbits. Its primary functionalities include:

- Receiving events (from different protocols).
- Event rule processing, for instance implementing the filtering, transformation and aggregation and of events.
- Semantically enhancing events, e.g., by static knowledge in the rules or based upon previous events, or, based on semantics derived from the event ontology.
- Providing event persistency and storage, for logging and future analysis.
- Routing events to the next level in the Event Processing Network.
- Triggering scheduled rules.

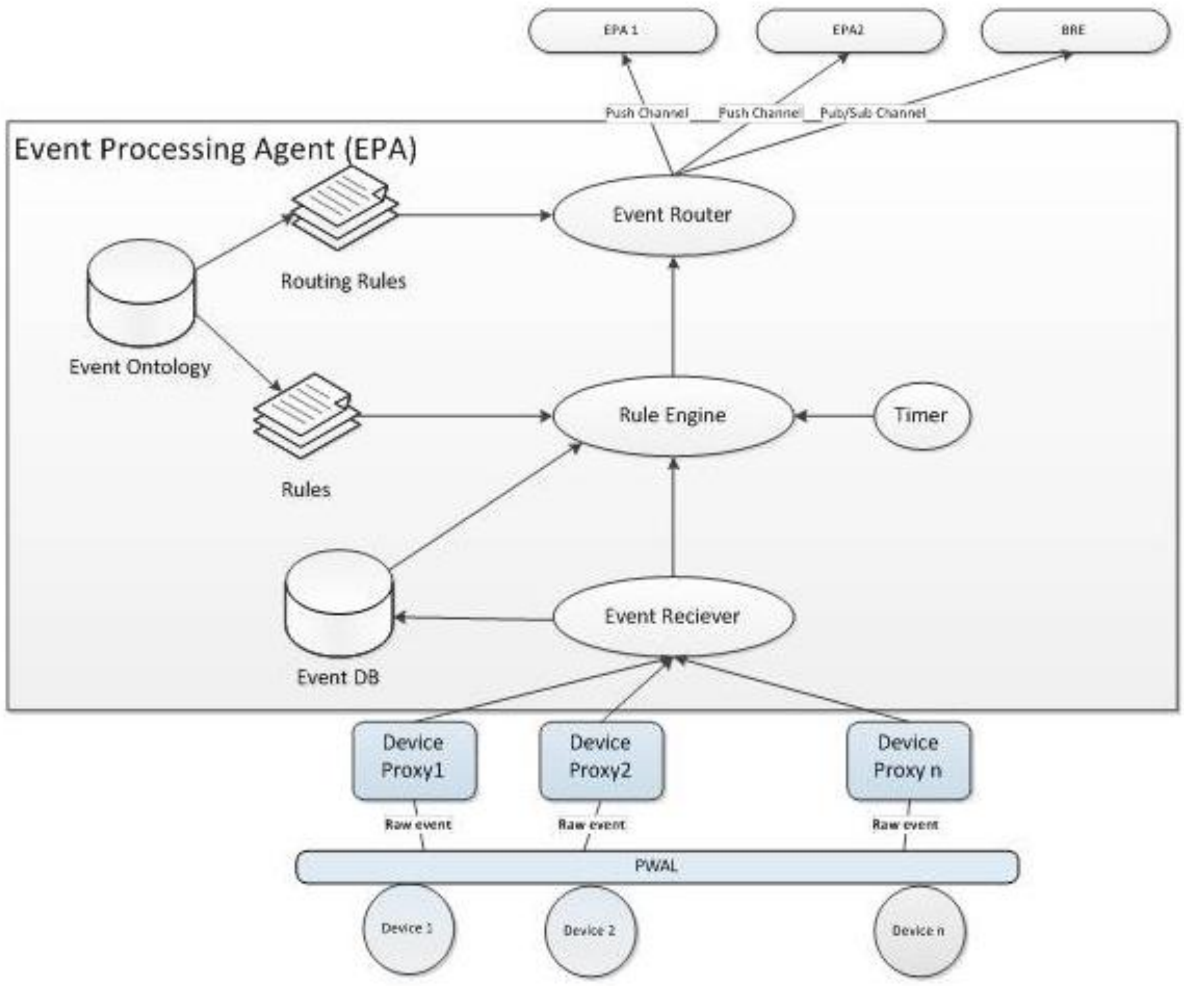

Figure 3: Internal components of the Event Processing Agent

The internal components of an EPA are an *event receiver*, a *rule engine* and an *event router*. Associated with these components are an event database where events are stored, a timer that can trigger reoccurring events, a set of rules to apply to the events and a set of routing rules to coordinate the distribution of events.

In many cases the EPA needs to ensure that the generated and processed events are delivered (leveraging on the ebbits networking layer), and are delivered to the correct receiver, such as an EPA instance. Therefore the EPA can use static (known) addressing instead of a publish-subscribe method, since the latter cannot guarantee that a subscriber actually has received an event. However, in cases where a more loosely coupled distribution of events is suitable, the publish-subscribe method can be used. Thus both methods are available in the architecture.

Below we will describe the internal components of an EPA.

3.4.1 Event Receiver

The Event Receiver is responsible for managing incoming events from outside sources. The component will support the necessary protocols which at least includes the standard EPA interface used in the Event Processing Network as well as the standard LinkSmart Event Manager events. Other protocols that could be supported are for instance Storm[2] , WS Events[3] etc.

[2] http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html

The Event Receiver will forward all events to the Event DB as well as to the rule engine.

3.4.2 Event DB

The primary use of the Event DB (database) is to keep a "sliding window" of historical events. The events can be referred to rules that operate on historical events. The size of the sliding window is configurable and will depend on the rules' need to look at historical events. Different rules can have different requirements with respect to the size and scope of the sliding window.

3.4.3 Rule Engine

The rule engine is responsible for the actual rule evaluation and the processing of incoming events. Typically it will operate on three levels:

- *Filtering*, discards events that are not needed in the further event processing path for the application.
- *Transformation*, transforms the incoming event to a new event or by semantic enhancement of events by adding metadata, possibly derived from the event ontology.
- *Aggregation*, combining multiple incoming events into one or more outbound events, fusing additional data with the event (data) payload.

The rules can operate on historical events using the Event DB which keeps a "sliding window" of the previous events.

3.4.4 Event Ontology

The main role of the event ontology with respect to the EPA is to describe the structure of and relationships between event types and other ebbits entities. This includes taxonomies for event types and topics, and how different event types relate to other ebbits entities such as devices and sensors. This information can be used for the design and execution of both event and routing rules.

We do not believe that each EPA will normally access the Event Ontology in run-time, but rather that the rules have been created based on knowledge derived from the ontology. So the main part will be used in design time by the developer when creating rules as they define the possible values and events.

3.4.5 Timer

In order to support scheduled rule execution there is a need for a Timer component which can be used to trigger rules at a certain interval. For instance an "average temperature the last 5 minutes" event can be created by using a rule that triggers every five minutes. This rule would only need to operate on historical data as maintained by the sliding window in the event DB), and would be triggered by the timer rather than an incoming event from a device.

3.4.6 Event Router

The main purpose of the event router is to use routing rules to determine where to send a specific event. The component thus forwards (possibly semantically enriched) events to other EPAs or to the BRE.

The default event routing will use static connections to other EPAs and business rule engines, i.e., it will not be based on a publish-subscribe model but the Event Router will know at start up which external interfaces it will submit events to. The reason for this solution is to make the Event Processing Network more robust with regards to connectivity problems. Always using publish-subscribe could lead into difficulties when a subscriber does not manage to make its subscription in due time and could lead to loss events.

[3] http://www.w3.org/TR/2010/WD-ws-event-descriptions-20100209/

Even though the default routing could be considered static it can be changed in runtime by changing the rules. This will be useful when dealing with the situation where an EPA is overloaded and we want to create an additional EPA for handling some of the event traffic of the overloaded EPA.

The event routing is based on rules that operate on the actual event metadata and even the actual event (payload) content. For example, all events of type "Alarm" might be sent further to an EPA that specialises in handling alarms but it could also be sent to an EPA that specialises on energy consumption because it is sent on the "Energy" topic.

The Event Router will also deal with the resending events that could not be delivered for some reason and this will allow for more robust events propagation.

The event router could also publish events in the LinkSmart Event Manager publish-subscribe method if configured so. The primary usage for this type of event distribution will most likely be in debug/development time where we would like to passively monitor the event traffic without really affecting the real flow of events.

4. Event structure and semantics

4.1 Event properties

The Ebbits event structure is made up of event metadata and a payload container in order to be able to semantically annotate the events. The event metadata consists of required as well as optional elements to describe the event, its context and its origin. The design of the event is basically an envelope of metadata encapsulating the actual event content.

The conceptual event model defines the information which potentially can be conveyed by an event instance.

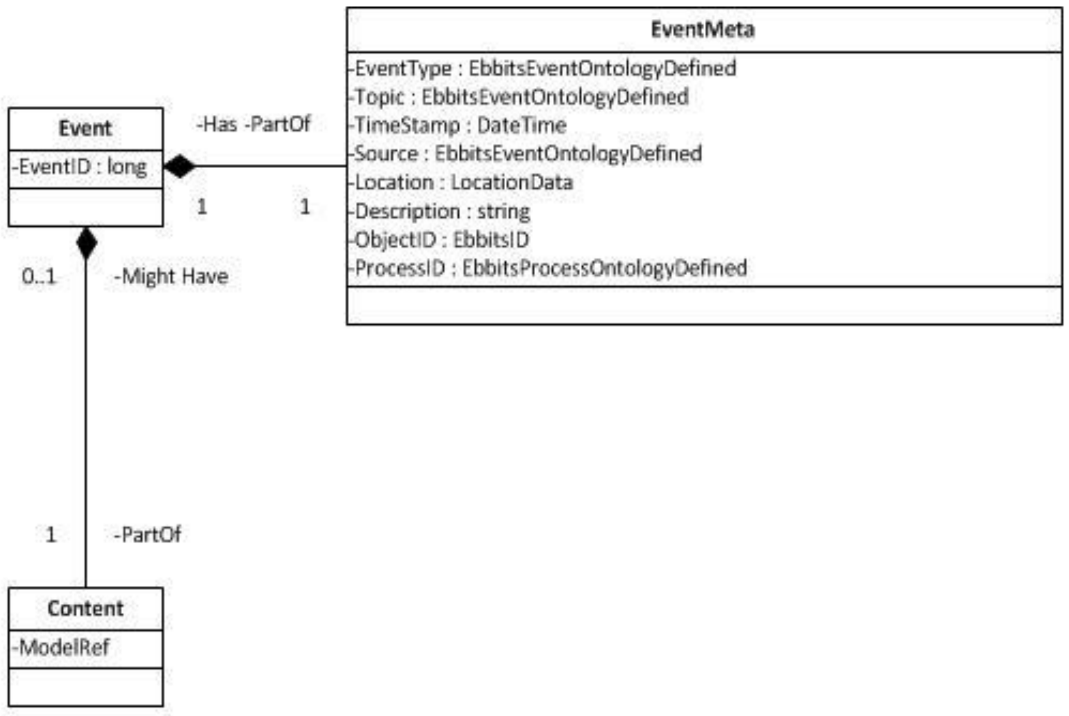

Figure 4: Conceptual event model

An important aspect of the metadata section is that it will allow both the event rule engine and event router to work without having knowledge of the actual event contents, i.e. rules expressed using the metadata parts will be general and reusable independent of the actual contents of the event. Of course it will be possible to express rules that look into the actual content as well.

4.1.1 Event XML structures

The conceptual event model has a corresponding XML-serialization.

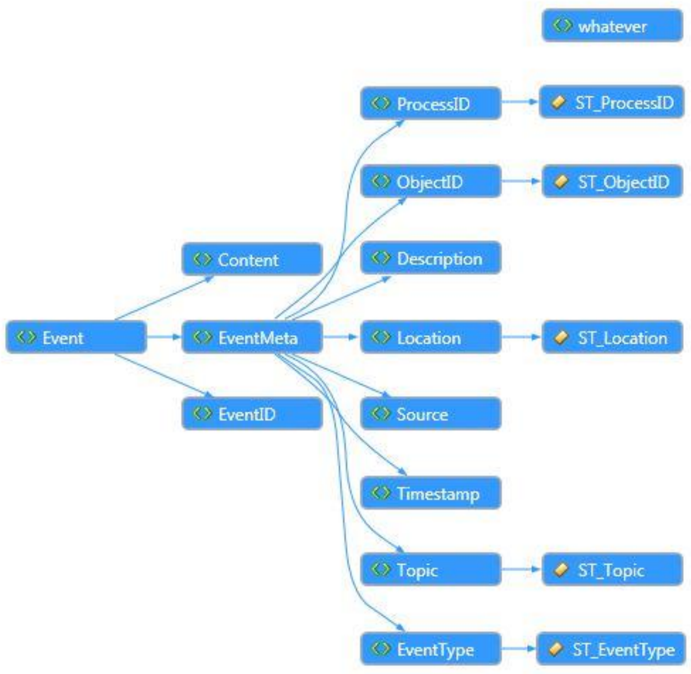

Figure 5: XML schema for the event model

Below is an example of an XML event with all elements instantiated.

```
<?xml version="1.0" encoding="utf-8"?>
<Event xmlns="http://events.ebbits.org/Event"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://events.ebbits.org/Event ebbitsEvent.xsd">
<EventID>1</EventID>
<EventMeta>
      <EventType modelRef="EbbitsEventOntology">Alarm/*</EventType>
            <Topic modelRef="OntologyID">Energy/*</Topic>
         <!-- Optional Topic (channel or #tag) -->
         <Timestamp>2012-02-02T13:30:31.089</Timestamp>
            <Source>EPA</Source>
            <!--The creator of the event-->
            <Location modelRef="SWEREF 99">6121020,743013</Location>
            <!--Location (if available)-->
            <Description>Consumption over threshold</Description>
            <!--Description (if available)-->
            <ObjectID modelRef="ebbits.id">F69023A8-DA07-4339-8BA3-
4E113AE29CAF</ObjectID>
            <!-- The entity that the event concerns (if available)-->
<ProcessID modelRef="http://www.omg.org/spec/BPMN/20100524/MODEL">
    sid-70ec3d5d-6c91-4dc3-86d5-d6729bc6e15e</ProcessID>
     <!-- The process that the event concerns (if available)-->
</EventMeta>
<Content modelRef="ISO-11788">
<!--modelRef examples: ISO ADED, ISA-95 manufacturing standard, EBBITS system events --
>
</Content>
</Event>
```

Listing 1: XML event instance

The following is the definition of the different metadata entities inside the ebbits Event:

- *EventType*: Compulsory entity that contains the type of event according to the taxonomy in the Ebbits event ontology. Note that theoretically any taxonomy could be used as the modelRef attribute would contain a URI for the used taxonomy.
- *EventID*: Compulsory entity that contains the events unique ID.
- *Topic*: Optional entity that contains the topic of the event. The topic is a taxonomy that is orthogonal to the EventType taxonomy. The topic taxonomy is also stored in an ontology and the modelRef attribute points to its URI.
- *Timestamp*: Compulsory entity containing the time when the event was created.
- *Source*: Compulsory entity containing the ID of the event creator.
- *Location*: Optional entity that contains the location of the event. The location has a modelRef attribute that defines which location system is used. In the example above it is expressed in the Swedish reference grid 1999 and points to a place close to Malmö. The location could also be expressed in terms of other location systems, in a factory system the location might something like "Building:1,Line:2,Station:3,Robot:5". This will allow the Ebbits system to handle location at a general level as well as in application specific manner.
- *Description*: an optional textual explanation of the event.
- *ObjectID*: Optional entity that contains the subject of the event. For instance if a pig passes an RFID-reader the ObjectID would contain the pigs ID. ObjectID has also a modelRef attribute that can be used for handling different ID schemes.
- *ProcessID*: Optional entity that contains where in the process the event occurred. The contents of this entity depends on which model is used for modelling processes. In the example event this contains an ID that refers to a model expressed in BPMN (Business Process Modelling Notation).

For the actual contents of the event we have the entity *Content* that also has a modelRef attribute indicating if it follows a specific model such as ISA-95 or if it a Ebbits system event. This will allow for handling very different events in the same framework.

4.2 Semantic support – Event ontology

This section aims to describe the extension of the ebbits event model as represented in the event ontology. The extension of the basic event semantic model designed within the deliverable D7.2 Event and data structures, taxonomies and ontologies is based on the requirements derived from the examples of expected usage of the event ontology. The extension aims mainly to map the event model to the context entities contained in the domain ontologies designed for specific applications. The design of the application domain ontologies will be delivered in D4.7.1 and D4.7.2 – *Use case driven semantic models 1,2*.

One of the basic usage patterns of the semantic models in ebbits is the sharing of common vocabularies across the rule systems on several architecture levels. This includes the context, business and also the event rules. As all these rules must share common symbols in the assumption and consequence parts, the need for a common language is essential. The main idea of the inter-rule semantic interoperability is the design and development of unified information structures shared by all ebbits components. These are realized as the semantic models instantiated for specific applications. These models include mainly the events-related information and their mapping to the application domain. The semantic models are used in two phases:

- *Design phase*: The models of all events and their mappings to the application domain are created in advance. The models of events are annotated to the meta-data and information, which will be needed for the logic of the specific application. Once, the models are designed, they are used as the semantic support for design of the event processing and application logic rules. The ontologies are used to guide the whole process of the rules development by offering the possibilities, what information may be used for specifying the pre-requisites and effects of rules. The common vocabulary (in the ontology) thus ensures that the rules will use the same data structures, interpreted in the same way.

- *Run-time phase*: The rules of the application logic may require to retrieve more information related to the data structures to be able to make more sophisticated decisions. This can be realized by dynamic on the fly querying of the ontology containing the actual instantiation of the semantic models.

The conceptual design of the semantic interoperability implementation between the ebbits components is described in D4.5 *Analysis and design of semantic interoperability mechanisms*.

4.2.1 Examples of event model in use

The design of the event ontology extension was driven by the expected examples of its usage. This section will outline three real examples from the traceability scenario and list the several aspects required for the event model. These small examples are derived from the first activities involved in the process of transportation of animals from a farm to slaughter house by truck. The focus is on event detection and event object identification.

Example 1:

Detection/initiation of transport. The scenario starts as an RFID tag at the loading bay door detects the truck arriving at the bay. Tag triggers the event, which must contain the identifier of the truck and the location, when this event happened – the concrete pen.

Example 2:

Detection/identification of event objects for tracing. An RFID tag at the door of the truck generates an event for each pig being loaded. The event must contain the identifier of the loaded pig and the location information – the concrete track. For more, the event must contain the information about the farm, from which the pig is being sent.

Example 3:

Dispatch event to next (external) actor in the process. As the truck leaves the farm, the farm management system generates the event representing, that the batch of pigs is leaving the farm. The event must contain the identifier of the batch of pigs (to be able to track it), the identifier of the truck, location. Additional data is also to be provided e.g., the identifier of the farm, the number of animals and the set of identifiers of animals being transported. This will typically be encoded by event content (payload).

To support this, the event must be able to hold the information about the object, in which it is concerned, e.g. the concrete pig, truck or pen. The event must have associated the location information, e.g. the concrete pen, vehicle or farm. The event must be able to contain the additional information, that can be added by the particular source, which created the event, e.g. the number of the pigs. A reference to some model or resource for the interpretation of this data should given.

4.2.2 Subset of event ontology

The mapping to the domain models enables to extend the events metadata by information useful for the application logic. The extension model is illustrated on the Figure 6: The RDFS schema of the extended event model.. The illustrated RDFS schema contains only the extensions of the basic event ontology.

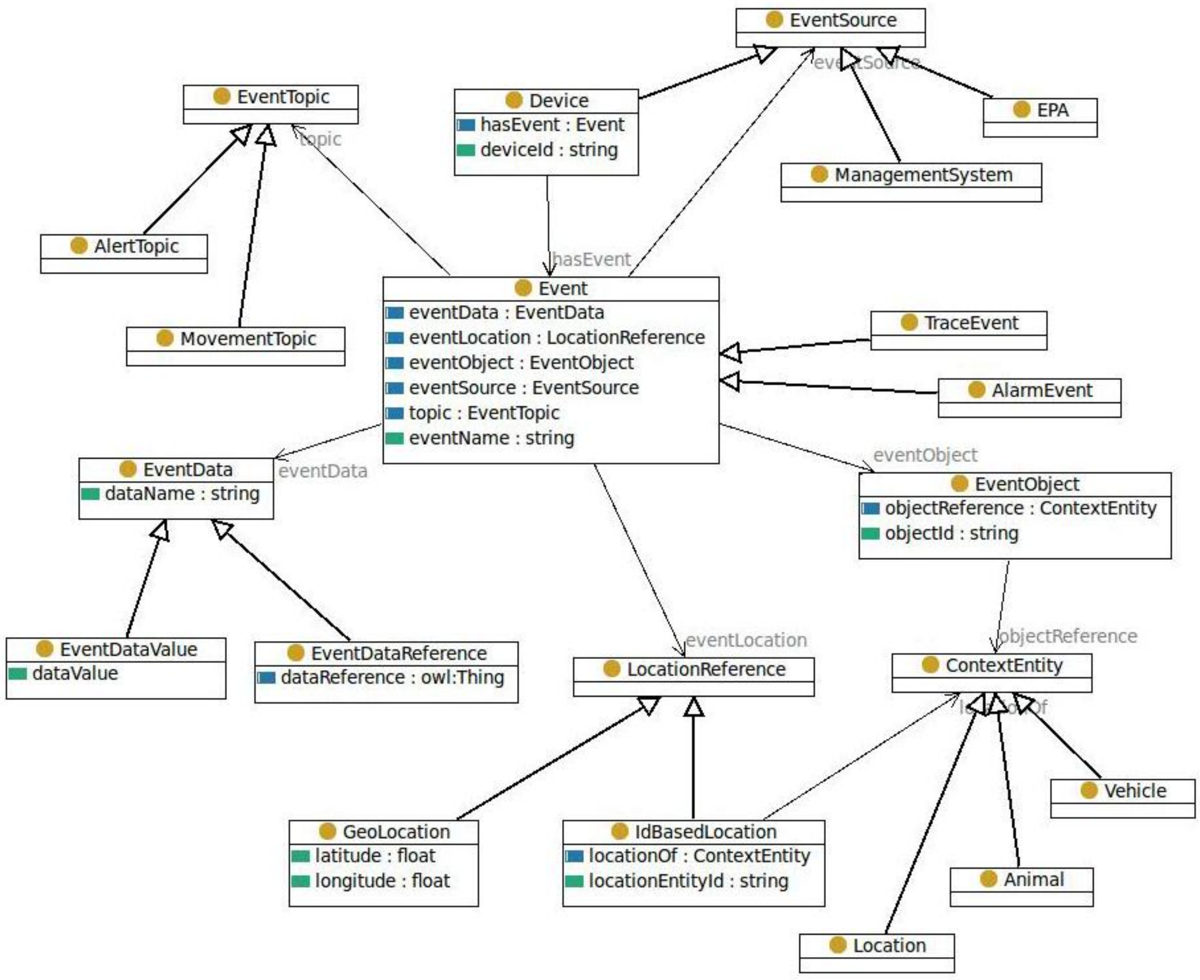

Figure 6: The RDFS schema of the extended event model.

For each event it's necessary to represent the source which generated the event. The basic class *EventSource* is the root of the source taxonomy. The source of the event may be practically anything able to generate the event. For example, it can be the physical *Device*, an *EPA*, but it can be also the context or business rule or the other application logic (e.g. by clicking the button in the user interface). The *Device* subclass of the *EventSource* is the root class of the device ontology.

Each event has a topic, to which the event is published. The *EventTopic* is the root of the taxonomy of topics, which represents the orthogonal classification of events. This taxonomy may be used as the alternative view of the events.

Events may concern specific objects e.g., the RFID reader at the truck reads the id of each pig passing by and generates an event concerning the specific pig, which is then the object of this event. *EventObject* represents the reference to the concrete instance of *ContextEntity* from the application specific domain ontology. The important property of the *EventObjects* is the identifier of the object. The reference represents only the type of the object from the domain ontology, but to be able to track the concrete instances of the objects, the identifier is treated as a variable and must be added by the source, which created the event. In example 2, the *objectReference* will be mapped to the generic instance of the pig and objectId will be filled by the event source by the real identifier of the concrete pig.

Another optional property of the event is the location. The interpretation of the location is very dependent on the concrete application. It can be required to represent the location as the GPS coordinates, it can be reference to the concrete physical location, such as building, but in general, it can be anything. In the example 2, the location of the event generated by RFID tag reading the ids of the pigs is the truck, which is the vehicle. But in this case it must be treated as the location. To be able to represent the location in flexible way, the event is mapped to the *LocationReference* class, which refers to the concrete location type, such as *GeoLocation* or the *IdBasedLocation*, which refers to the application domain entity representing the location. In this case, the design of the *IdBasedLocation* concept is very similar to the *EventObject* reference. *IdBasedLocation* is mapped to the generic context entity, such as truck; and the source of the event must fill the *id* variable, which identifies the concrete instance of the application domain entity (e.g. truck is identified by registration number 'KE-729AZ'). The location property of the event is the same as the location of the source that created the event. The location property of event is optional, as in some cases the source of the event can be virtual, e.g. when event is created by the EPA agent.

Each event may have attached a custom set of additional information, which may represent data required by the application logic. For example, the event created when truck containing the batch of pigs is leaving the farm, may need to contain information about the number of pigs, the list of pig ids, etc. Such additional information is modelled by instances of the *EventData* class. Each event may have attached any list of *EventData* instances. The idea is to represent this additional information as key – value pairs. For This reason, the *EventData name* property represents the key. The value may be a real value represented by some of the basic data types, such as string or number. This concept is represented by the *EventDataValue* class. In other cases, the value may be a reference to a concrete object which can represent any type of data. For example, it can be an entity from the application domain ontology, but also a reference to some external model or standard. This case is modelled by the *EventDataReference* class, which maps the *owl:Thing* as the value, to provide the suitable flexibility.

An example of a concrete instantiation of an event is illustrated in Figure 7: Example of the event generated by the RFID reader at the truck door. The figure represents the example 2. The source of the event is the RFID device. The object of the event refers to the pig instance represented in the domain model, where identifier of the pig will be added for each concrete animal being loaded in the run-time. Event location refers to the instance of the truck, where identifier will be added by the application login in the runtime. For each event it is required to add the additional information representing the farm, from which the pig is sent.

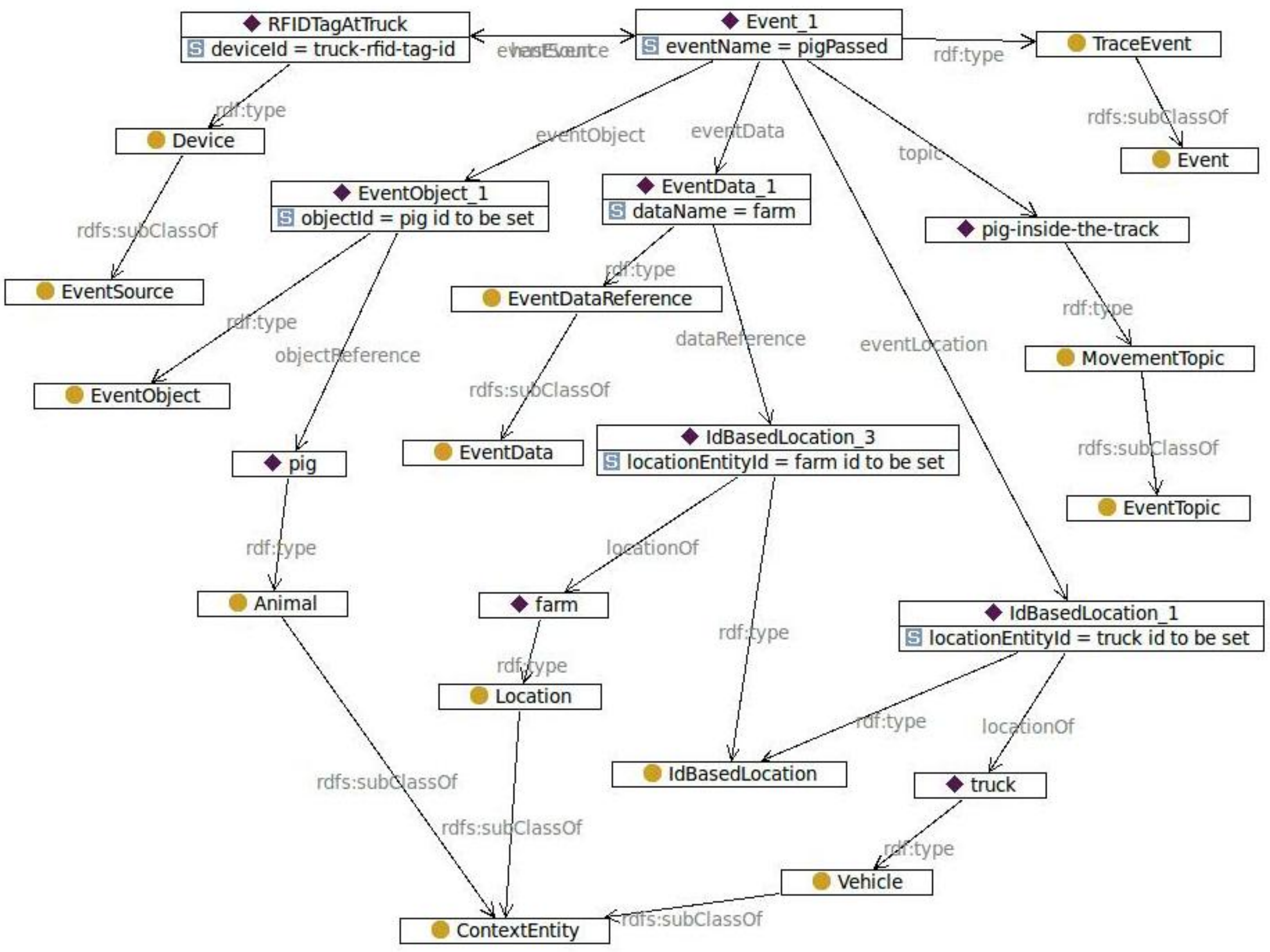

Figure 7: Example of the event generated by the RFID reader at the truck door.

This event model is used in design time to guide the development of the particular related rules which will catch or create this event. If the rule catches the event, the developer is aware of what the event contains, and on which information the decision may be based. When the rule creates the event in the consequence part, the semantic model provides all information for the developer to be aware of what information must be added to the event and which required run-time information specific for each firing of the rule must be assigned (e.g. the identifiers of particular entities, in the concrete context).

4.3 Event lifecycle

The event life cycle is illustrated in the sequence diagram below, with some example event types. The event management system takes its starting point in the events generated from various types of devices, and provides a set of intermediate processing steps of the events until they eventually trigger some higher level business rule.

Devices in the sequence diagram are a temperature sensor and a window lock sensor.

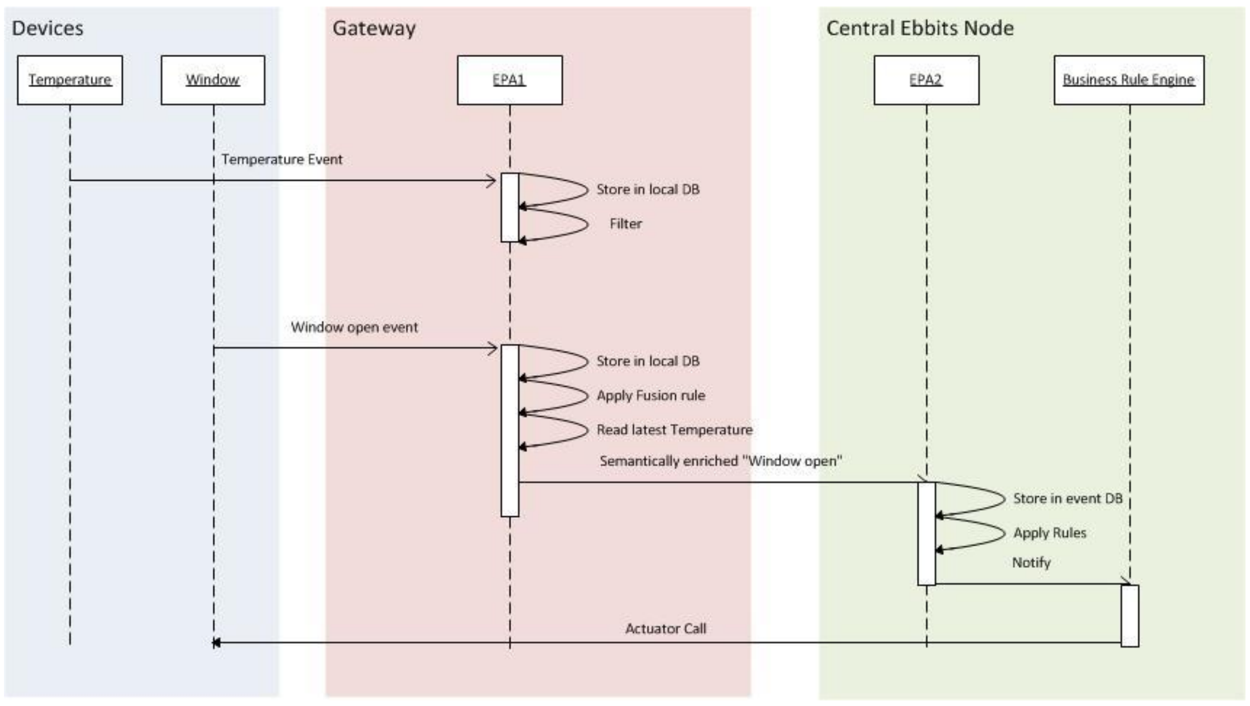

Figure 8: Event life cycle example

Devices generate "raw" events, and their "life-cycle" depends on the subsequent processing on different levels in the event management architecture.

1. Events enter the EPN from device proxies.
2. The event is then caught by an EPA which may be configured to perform a basic form of first level event processing depending on event types. For the two event examples the processing is as follows:
    a. The temperature event when caught is simply stored, and not forwarded
    b. The second event triggers a rule which results in a new refined event (including the latest temperature reading) which is sent to the next EPA.
3. On the next level of event processing an EPA notifies a rule engine (BRE), which takes action by invoking a service representing an actuator call.

5. Extensions to the LinkSmart Event Manager

The LinkSmart Event Manager provides basic publish-subscribe functionality, i.e., the ability for publishers to send event notifications to multiple subscribers while being decoupled from them (in terms of, e.g., not holding direct references to subscribers). A description of the functionality and use of the LinkSmart Event Manager is provided in the previous version of this deliverable D7.3.1. In this deliverable we describe the ebbits extensions to the publish-subscribe mechanism.

5.1 Extension to the LinkSmart Event Manager

The LinkSmart Event Manager will still provide one of the communication facilities within ebbits. The publish-subscribe pattern is useful in many applications and can provide for a low coupling in-between different components. Though it is important to understand that the LinkSmart Event Manager only provides a communication facility and not event processing, i.e. EPAs could use LinkSmart Event Manager for communication but not to provide any event processing capabilities.

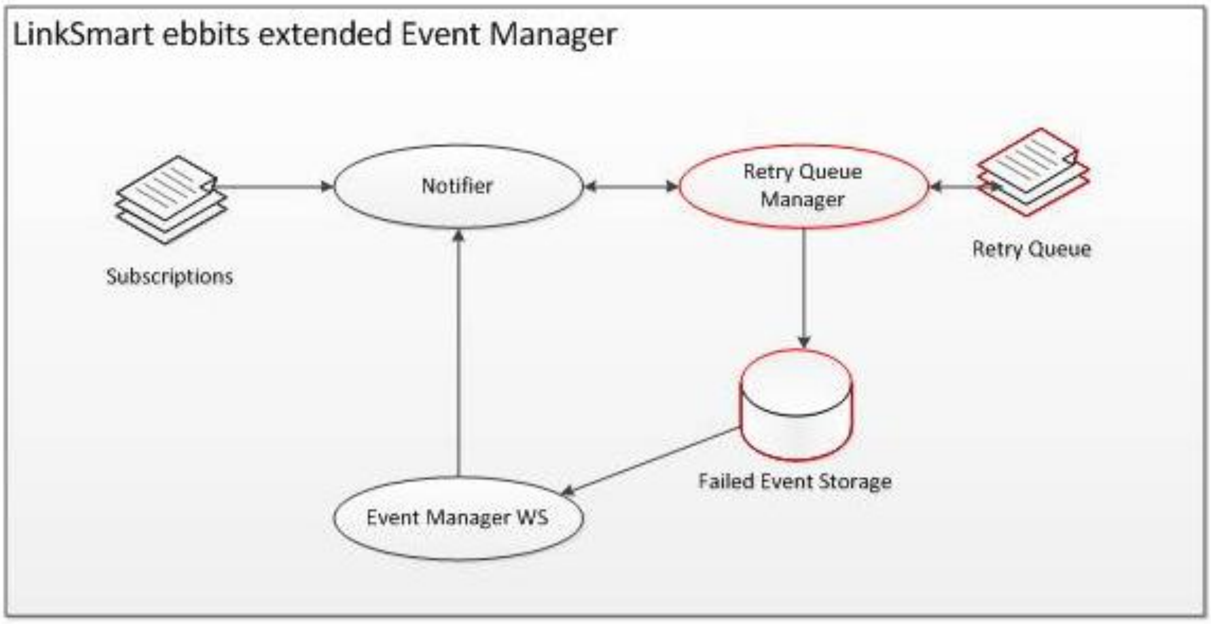

Figure 9: ebbits extended LinkSmart Event Manager (Additions in red)

The original LinkSmart Event Manager has been extended with some basic functionality to improve the reliability of eventing. The original LinkSmart event manager lacked support for dealing with network and connectivity problems. If an event was not possible to forward to a subscriber it would be dropped without giving the subscriber any possibility of knowing that it has missed an event. Finally if the connection to the subscriber failed more times the whole subscription would be deleted.

In order to provide a more reliable infrastructure in ebbits for the parts that use the LinkSmart Event Manager for sending events we have added the following functionality:

- Event retry queuing
- Prioritisation of events and subscribers
- Failed event storage

Event retry queues implement store and forward for subscribers, if an event fails to be published to a subscriber it will be stored in the retry queue for the subscriber. At given intervals the events will be resent to the subscriber, if the transmission succeeds the event will be removed from the retry queue otherwise it will remain in the queue. Depending on the configuration this can be repeated in infinity (which is the default) or the message will be moved to a Failed event storage if a configured limit is exceeded. This can be used for avoiding starvation of resources when the size of the retry queues grows extremely large, but this should be considered to be a last resort and not part of normal operation.

Furthermore a concept of priority has been introduced, both with regards to the events themselves as well as the subscribers. The priority mechanism is simply an integer based property where higher priority means that it will be prioritised higher than those with lower priorities. The meaning of priority depends on if the priority is expressed on the events or on the subscriber. Note that priorities only have meaning when dealing with the event retry queues.

Priorities in events

For the events there is a possibility to set a priority for a certain Topic by using the new method:

```
public bool setPriority(string topic, int priority)
```

The priority for events is used in two ways. In the case that we failed to notify a subscriber of the event we will check if we have an priority for the event. If we have a priority and it is under the configured threshold (Default of 5) we will just drop the message. If it is within the threshold, or lacks priority it will be added to the retry queue. This will allow for managing the situation where there are events which are not historically interesting and do not need to be resent in the case of communication failure.

The second usage is when the LinkSmart Event Manager tries to resend the failed messages. This resending will always occur according to the following ordering to a subscriber: First we order on the priority descending and secondly on the resend count descending. This means that high priority events will always have precedence on those with lower priority. Secondly we always try to send the messages that have waited longest for transmission by using the resend count. Please not that this will not guarantee the ordering of events. A subscriber might still get the events in a different order than the actual produced order even if the priority is the same.

Priorities in subscriber

For subscribers there has been added a priority parameter in the subscription methods of the event manager:

```
public bool subscribe(string topic, string endpoint, int priority)
public bool subscribeWithHID(string topic, string hid, int priority)
```

The priority of subscribers is also used in two ways as the priorities of events. First if we fail to forward an event to a subscriber we will first check if the event is to be dropped or queued (see Priorities in events above) then we will check if the subscriber has a priority. If the subscriber has a priority which is below a configurable threshold (default 5) we will drop the message. If the subscriber has no priority or it is above the threshold. This will enable subscribers to define themselves if they do not want to have resends. The second usage of priorities is that the higher priority the subscriber has, the more often the retry of messages to it will be made.

Failed event storage

The failed event storage provides the final safety net for failed event notifications. Two methods have been added for retrieving the failed notifications. The parameter clearFailes will remove the retrieved events if true.

```
public Event[] failedNotifies(string topic, string endpoint, bool clearFailes)
public Event[] failedNotifiesWithHID(string topic, string hid, bool clearFailes)
```

This will enable subscribers to retrieve any messages that they should have received. But as this should not really be considered normal processing these functions should only be used for restarting after intermittent problems or for debugging. The Failed event storage can also be accessed using SQLite based clients for querying the database. The table definitions is shown below.

```
CREATE TABLE [CallbackAddress] (
[Endpoint] NVARCHAR(3000)  NULL,
[HID] nvARCHAR(3000)  NULL
)

CREATE TABLE [Events] (
[ID] VARCHAR(36)  UNIQUE NOT NULL,
[stored_timeStamp] DATE DEFAULT CURRENT_TIME NULL,
[Topic] NVARCHAR(3000)  NULL,
[Priority] INTEGER  NULL
)

CREATE TABLE [KeyValuePairs] (
[EventId] NVARCHAR(36)  NOT NULL,
[Key] NVARCHAR(3000)  NULL,
[Value] NVARCHAR(3000)  NULL
)
```

Listing 2: Failed event storage table definitions

6. Future development and plans

In this second iteration of the event subsystem design, the emphasis has been on evolving the initial publish-subscribe event model into a more flexible event processing architecture, corresponding to an Event Processing Network (EPN).

The event core ontology has been changed to better support the new event structure and processing.

The components for event management have been further developed with services interfaces and are under implementation for inclusion in the next (M24) ebbits prototype platform.

The architecture is subject to further refinement in a series four successive revisions, aligned with the iterative development plan of the project.

In the first proof of concept of the event management subsystem, the publish-subscribe event model as provided by LinkSmart was evaluated, with respect to scalability and performance. As a results of this, we have made a separation between the event processing functionality and the actual event communication and distribution, i.e.,

- filtering, transformation and aggregation of events are performed within the EPN.
- the LinkSmart Event Manager provides a communication facility and not event processing, and it could be replaced in the EPN by other distribution mechanisms.

In addition, the LinkSmart Event Manager has been extended to provide more reliable distribution.

The future development focus test and validation of the architecture, including,

- the verification of rule engines for the different types of event processing rules as well as for routing.
- the performance and scalability of the EPN.
- testing tools and procedures for designing and maintain rule sets.

As with the previous iteration, we expect that the prototype evaluations will reveal additional areas for refinements and architecture revisions.

7. References

[1] Etzion, O. and Niblett, P. (2011). Event Processing in Action. Stamford, Manning Publications Co.

[2] Chandi, K. M. and Schulte, W. R. (2010). Event Processing: Designing IT Systems for Agile Companies, McGraw-Hill.

[3] LinkSmart. (2012). "LinkSmart Wiki ", from http://sourceforge.net/apps/mediawiki/linksmart/index.php?title=Main_Page

.