Enabling the business-based
Internet of Things and Services

(FP7 257852)

D7.3.1 Technical description of the implementation of Data and Event Management models 1

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: D7.3.1 Implementation of Data and Event Management models 1.doc
Document version: 1.0
Document owner: Matts Ahlsen (CNet)

Work package: WP7 – Event Management and Service Orchestration
Task: T7.3 – Title
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, Peeter Kool, Peter Rosengren (CNet)	2010-12-12	ToC, initial outline
0.2	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2010-12-22	Initial event model suggested
0.3	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2011-01-15	Event architecture and ebbits overall architecture
0.5	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2011-02-02	Coordination with Event Ontology in D7.2.
0.8	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2011-02-22	Event model and Hydra event manager adaptation
0.9	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2011-02-24	Version for internal review
1.0	Matts Ahlsén, Peeter Kool, Peter Rosengren (CNet)	2011-02-28	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Ferry Pramudianto	2011-02-25	Some Comments on the structure and contents.
Maurizio Spirito	2011-02-25	Approved with comments

Index:

1. Executive summary

This deliverable describes a first design and proposed implementation of the event management subset in the ebbits architecture.

The work is performed in the context of Task 7.3 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, 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 ebbits Event architecture framework includes the following subsets,

- The event core ontology. This is a support ontology used by the event management system describing and relating all fundamental event concepts in ebbits. The Event ontology serves as a decision support component, with query and inference capabilities. The Ontology has been proposed in the related deliverable D7.2.
- The event processing model, implements a set of event and data management objects and data types. 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 management components, i.e., a set of software components implementing the event channels and distribution mechanism, for ebbits' publish & subscribe mechanism. The Event manager is based on the Hydra event and network managers.

In this first iteration of the event subsystem design, the emphasis is on the detection and semantic enrichment of low-level events, and the functionality of the event manager components. The 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 first design and implementation for the event management subset in the ebbits architecture.

The work is performed in the context of Task 7.3 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 report is the first version in a series of four successive revisions, aligned with the iterative development plan of the project.

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 Things and Services* (IoTS) 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 IoTS domain.

The *ebbits project* will develop the architecture, technologies and processes, which allow businesses to semantically integrate the IoTS into mainstream enterprise systems and support interoperable real-world, on-line end-to-end business applications. It will provide semantic resolution to the IoTS 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 IoTS. 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 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. Ebbits Architecture

In order to put the event management subset in context, we need an overall ebbits architecture. An initial sketch of an ebbits data fusion architecture is presented below. This architecture is intended as a frame of reference in the further design of event and data management in ebbits. We also note that this is an initial basic architecture resulting from the initial analysis and state-of-the-art in WP7, and which is subject to further design.

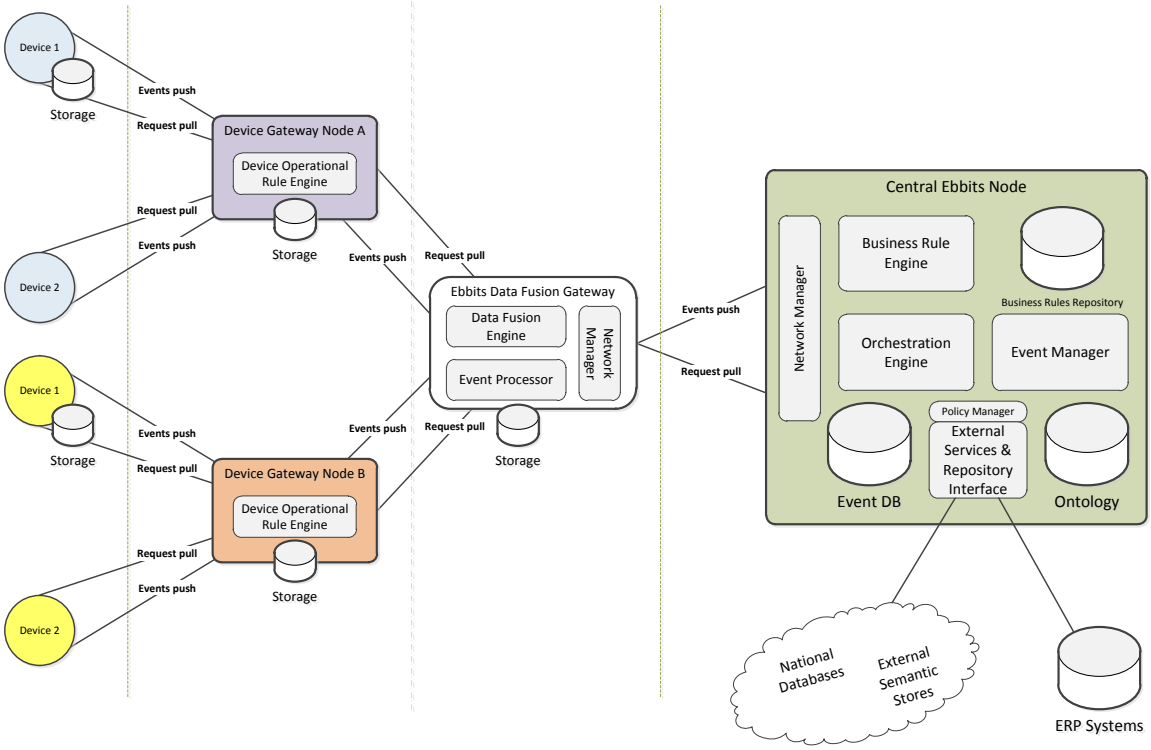


Figure 1: ebbits data fusion architecture

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).

The approach to data fusion in ebbits relies on model reuse, i.e., that both the data fusion architecture model as well as the layered distributed vs. centralized model should be able to overlap and hence be deployable in both food tracability and manufacturing domains.

Components overview:

- **Device 1, 2**

  These devices can be sensors, actuators or even subsystems spread over the ebbits physical environment. For example, it can be a temperature sensor, a RFID reader or some sensor measuring mechanic movement (e.g. pig feeder or roller rotate). Devices generate physical events such as a new temperature value (*stimuli*). Ebbits device storage occurs as close as possible to the device itself. This is in order to ensure full traceability capability in the ebbits architecture

- **Device Gateway Node A & B**

  The ebbits device gateway node can be based on a PDA or laptop. It enables rule assessment derived from its associated network devices as well as performs rule execution on the gateway node level. Each gateway node can be extended with its own storage. This storage can be used for caching events, etc. in the case of network failures and thereby enabling store and forward. Device gateway nodes combine physical stimuli into device events such as "light is turned on".
  Device Operational Rule Engine

  Each gateway will have a rule engine intended to run a set of device specific rules, execute it locally. For example, to not allow the temperature on a device 1 to exceed 5 degrees.

- **Ebbits Data Fusion Gateway**

  The ebbits data fusion gateway has the aim to fuse data that are gathered via a number of ebbits environment's device gateway nodes. Alternatively, the device gateway nodes and the data fusion gateway can be implemented within the same operating platform and thereby enhance the local processing performance. The data fusion gateway combines events from one or several devices and creates application events.

  Data Fusion Engine
  The data fusion engine processes (e.g. aggregates and filters) data from sensors and devices, taking time into consideration. The fused data is further sent to the event processor component.

  Event Processor
  The event processor consumes events and data and creates new application events according to its event management logic. The event processor will dispatch events to the ebbits central node.

  Network Manager
  The Network Manager implements P2P network functionality. It provides a virtualization of service endpoints for devices and applications, enabling event dispatching and service invocation across network boundaries.

- **Central Ebbits Node**

  The central ebbits node is intended to run on high performance machines in the ebbits architecture. By so it will be able to offer the computer power needed to support more intelligent components. This central node can be modelled and positioned as more centrally within an ebbits environment, e.g. at the ebbits service provider. It requires a stable communication with external resources and information providers.

  Business Rule Engine

  This rule engine process a set of business rules defined by the ebbits platform user and describes the intended work flow organisation of the specific domain. Business rules are mapped to services via the orchestration engine. The rule engine uses its own repository. The business rule engine combines one or more application events into a business event which is forwarded to external business system.

  Business Rule Repository
  Here the business rule engine stores and retrieves the set of rules.

Orchestration Engine

The orchestration engine performs tasks on the request pulls over the ebbits network. This can partly be configured by the business rule engine and partly manually through an interface.

Ontology Database

Here the device ontology is stored for use within the ebbits central node.

Event Manager

The ebbits event manager handles events that are broadcasted throughout the network architecture. It deals with events processed on all levels in the ebbits platform and provides event management to external parties.

Event Database

Provides a chronological and/or source-based log for all events, intended to support data mining and traceability analysis.

Network Manager

Similar to the Data Fusion Gateway (above), an instance of the Network Manager will also be required to run on the central ebbits node.

[Policy Manager] + External Services & Repository Interface

The policy manager provides access controls by handling communication to external resources. It guarantees security by restricting access to the different services/repositories involved in the ebbits global system.

ERP Systems

For example, SAP, COMAU, etc.

National Databases

For example, regulatory databases for national agriculture control.

External Semantic Stores

For example, a semantic store in a different ebbits node such as an Event database

4. Ebbits Event Management Framework

4.1 Architecture

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

- 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 event processing model, implements a set of event and data management objects and data types. 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 management components, i.e., a set of software components implementing the event channels and distribution mechanism, for ebbits' publish & subscribe mechanism. The Event manager is based on the Hydra event manager (Hydra 2008).

4.2 Event core ontology

Based on an initial requirements analysis and state-of-the-art (Deliverable D7.2), a generic event model for ebbits was proposed with a conceptual foundation in the SSN ontology from the W3C[1].

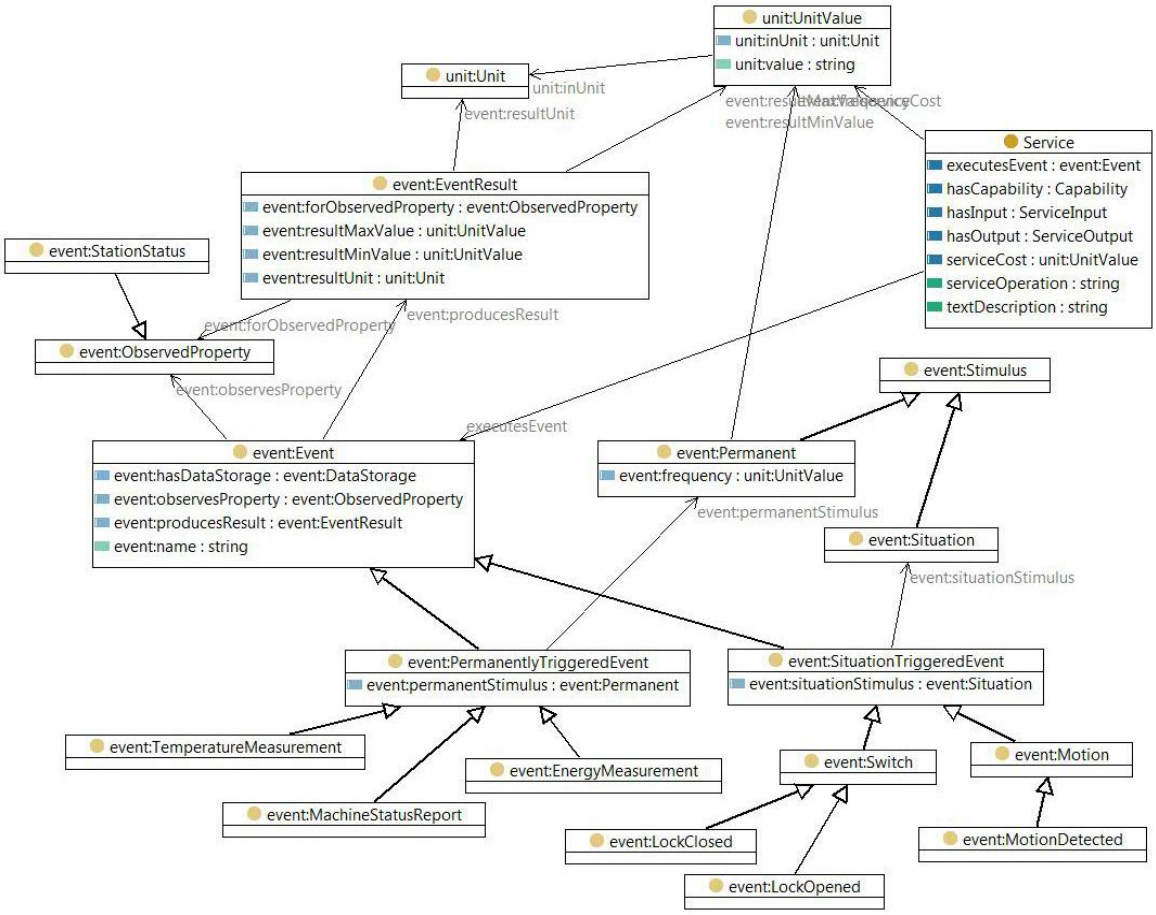


Figure 2: Core ontology for event support in ebbits

[1] http://www.w3.org/2005/Incubator/ssn/wiki/Incubator_Report

4.2.1 The core taxonomy

The core taxonomy of events has two main sub-classes according to the form of the event stimulus, the taxonomy will be further extended in the future according to the devices and events used in the Ebbits:

- *SituationTriggeredEvent* is triggered by the occurrence of situation in the real-world. This type of event has linked the *Situation* class through *situationStimulus* property. The model of situation can be quite complex, as it can also contain the logical relations of certain atomic situations in the world As semantics of situations depend on the use cases, the model for this kind of stimulus is left empty for the prototype and should be specified in the future following the real cases.
- *PermanentlyTriggeredEvent* is not triggered by the occurrence of situation, but continually produces the results in some frequency. This type of event has linked the *PermanentStimulus* class through the *permanentStimulus* property. The model, describing when and how the event is triggered, will be extended in the future according to the real cases.

As in some cases, the functionality provided by the event can be tied with the service, which may execute this functionality using the service call. In the ontology, the *Service* class is associated with the *Event* class using *executesEvent* property.

4.2.2 Observed properties

The observed properties represent properties of the real world, which the device is capable to measure using the event. *Event* class can have multiple associations to the *ObservedProperty* class instances through the *observesProperty* relation. Observed properties serve as the feature of interest for the particular event. Based on the analysis of use cases provided by the real-world data structure state of the art, it seems to be enough to provide the suitable representation of the observations in the form of static taxonomy describing, what is observed with respect to the application domain. This taxonomy can be further extended to satisfy the evolving requirements to the knowledge needed for proper decision making inside of the Ebbits.

4.2.3 Observation result

The result of an event is composed of a set of *EventResult* instances produced by a sensor and linked to the *Event* class through the *producesResult* property. The instances of *EventResult* class may have an association to the *ObservedProperty* instances attached to *Event* through the *forObservedProperty* relation. Using this relation it is possible to tie the produced results to the particular observed properties of the real-world. In addition, each event result specifies the unit of the result and may also specify the maximum and minimum expected value, each also tied to the unit of the result. The unit specified for the result, maximum and minimum value must be the same to keep the interpretation of the resulting value consistent.

A detailed background and description of the event ontology is given in deliverable D7.2.

4.3 Event Process Model

The first implementation of the ebbits event management subset will employ a data model derived from a subset of the event core ontology.

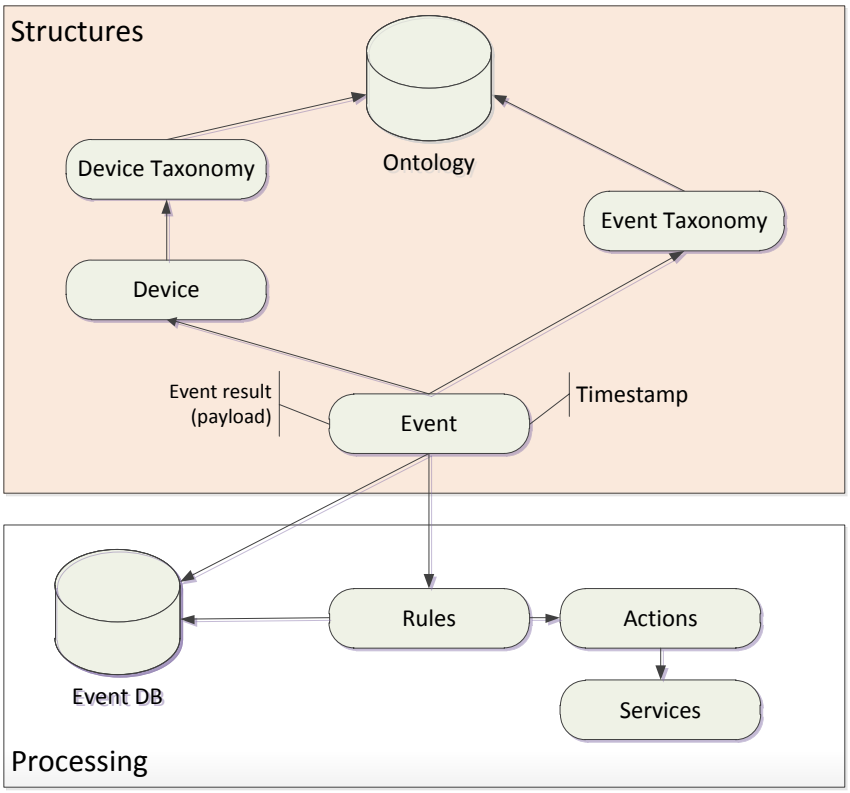


Figure 3: Event Data model

In Ebbits, a semantically Enriched Events (SEE) contains references to the event taxonomy which gives the possibility to reference the information in the event ontology. This will provide the consumers of the events (i.e. rule engines) access to all the semantic descriptions regarding the event. In addition there is also a reference to the device that is the originator of the event, where this is possible. This will provide the consumer possibilities to access the device ontology for the semantic descriptions concerning the device as such.

The SEE contains a time stamp that represents the time of the event creation. This will in most cases refer to the time that the "physical" event was received from a sensor by its device gateway and not the time when the SEE was created in the Data Fusion Gateway. This means that the actual sensors/devices do not need to synchronise the clock since the device gateway will set the the timestamp. For events that are not originating from a sensor, such as events created by evaluation of a business rule, this time will refer the actual entry of the SEE.

The actual event result (payload) is also carried in the SEE. This event result information is also described in the event ontology.

The event management system will not prescribe or assume any specific structure of the event results/payloads. This structure will depend on the application area and on the types of sensors.

Below is an example of an event instance from the Food Traceability application

animal/medication
========
MedicineID=Synophon
DoseGiven=10
PerformedBy=Mr Vet
AnimalID=00000000000004139217
Location=18
Timestamp=27-01-11 01:45:45
Note=after full feed intake

Another example of payload is the event/alarm data exchange format used in farming systems. This format is a proposed extension to the ISO 11788 by the Danish Data standard project (detailed in deliverable D7.2).

906011		Event Notification	
O	DD	Name	Type
K	906001	Device Entity ID	Int
K	990001	Location Entity	String
M	906051	Event Vendor ID	String
M	906052	Event ID	Int
	906053	Event Description	String
	906131	Creation Time	Time
	907132	Updated Time	Time

4.3.1 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 intermediate processing steps of the events until they eventually trigger some higher level business rule.

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

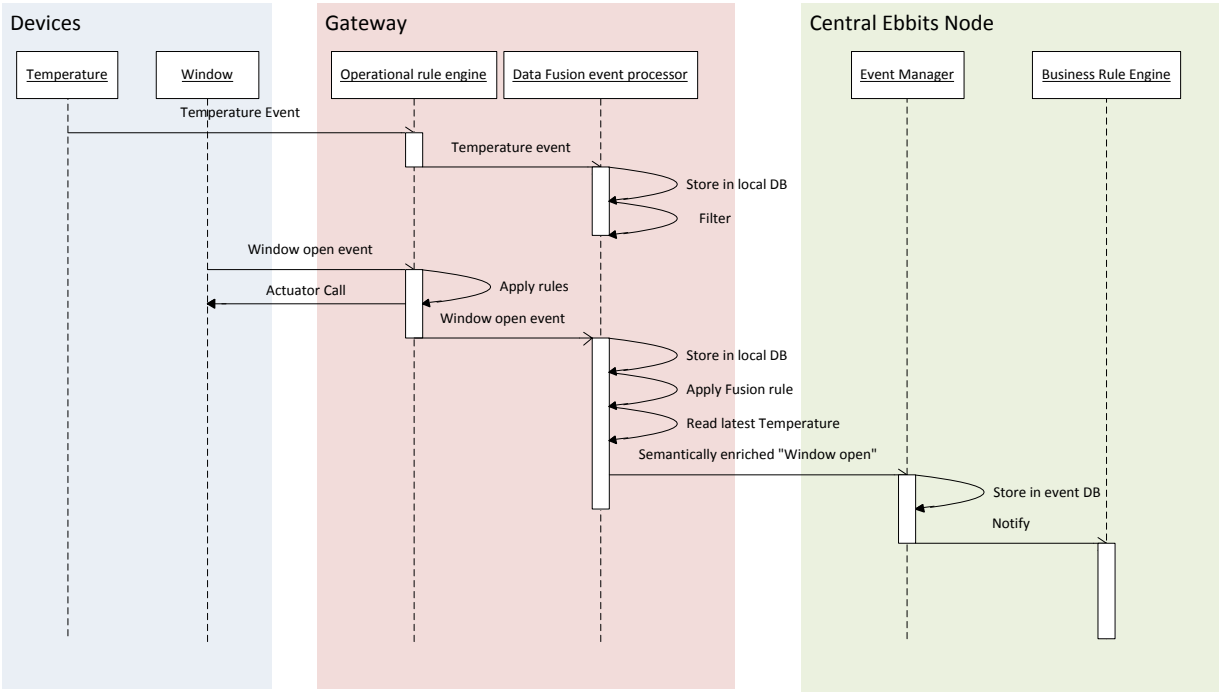


Figure 4: Event life cycle example

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

1. The event is initially received at a Gateway, to which the device is connected. This is a Device Gateway responsible for communicating with one or more devices, which might include both sensors and actuators.

2. The event is then caught by a rule engine which may be configured to perform a basic form of first level event processing depending on event types and device types. For the two event examples the processing is as follows:
    a. The temperature event when caught is simply propagated further.
    b. The second event triggers a rule which results in a call to an actuator device, before sending a (possibly modified) event further.
3. The next level of event processing is done in the context of a Data Fusion Gateway, where an Event Processor provides for semantic enrichment of events. The processing can use the Event core ontology for possible event analysis. At this level also more complex data and event fusion rules can be applied.
4. After processing a, possibly enriched, event may be sent.
5. On the next level (enriched) events are caught by a central Event Manager for further processing.
6. The central Event Manager will store the event in the event database. It will also notify the business rule engine.

4.4 The ebbits Event Management Components

The event management is implemented by a set of components distributed across the nodes in the ebbits overall architecture. Included here are adaptations of the Network and Event managers from the Hydra project.

4.4.1 Device Operational Rule engine

This is a basic rule-based event processing agent, implementing initial simple event filtering. This is assumed not to require too complex processing logic or resources. The idea is to allow the system to provide a first line response and take action for events that require immediate action without the need for further processing up the event processing sequence (we refer to this type of event response as "reflex actions").

The events caught at this level can be based on data streams reporting actual values from devices and sensors, or be device system events, representing error states or other exceptional conditions of devices.

4.4.2 Data fusion Event Processor

This component produces semantically enriched events based on event inputs from ebbits devices. It also provides rule-based processing.

Functions include,

- Aggregation and filtering of events.
- Reference to historic events.
- Contextualization of events, e.g., by fusion of data from different sources.
- Storing events for logging and future analysis purposes.
- Fusing event payloads when aggregating events

The component forwards semantically enriched events to the central ebbits node.

4.4.3 Ebbits Event Manager

This is the main event management component, residing on the central ebbits node. The context here is a central node in an ebbits architecture, where the other main functional components run. The event manager interacts with a Business Rule Engine, rules thus triggered by events may result

orchestration and invocation of services. The Event core ontology may be used for event analysis. The Event manager also maintains an Event database.

5. Baseline Event Technology for ebbits

5.1 Support for networking and events

The ebbits Event Manager is deployed in an ebbits architecture as a service in close cooperation with other components, among them the Network Manager, providing publish/subscribe functionality, i.e., the ability for publishers to send a notification to multiple subscribers while being decoupled from them (in terms of, e.g., not holding direct references to subscribers).

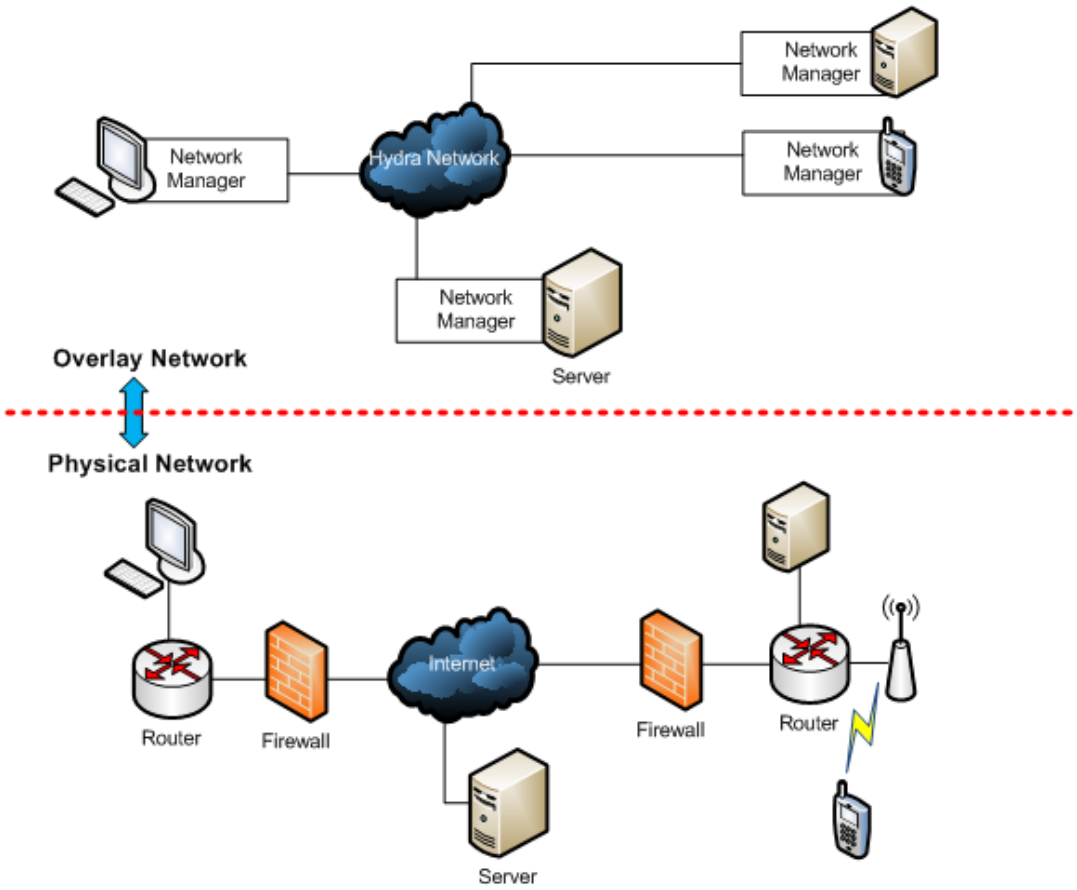
The ebbits event manager is based on an adaptation of the Network and Event Managers from the Hydra project (Hydra 2008) . In order to understand how the Hydra event manager works it is necessary to understand the how the underlying P2P network is created and maintained by the network manager.

5.1.1 Network Manager: Building a P2P overlay network

The main objective of the Network Manager is to interconnect different Hydra Enabled Devices (HEDs) through the network. The main problem of this task is that most of the HEDs may be hidden in Local Area Networks, behind firewalls, routers and Network Addressing Translators (NATs), so it would be difficult to interconnect the HEDs.

However, the Network Manager solves this problem by building an overlay network, independently of the network addressing and protocols.

The Network Manager, relies on JXTA P2P[2] platform in order to build the overlay network. JXTA is a set of open, generalised P2P protocols enabling any connected device on the network to communicate and collaborate. Using the JXTA protocols, HEDs are directly connected even if they are connected in different networks separated by firewalls or NATs.



[2] http://en.wikipedia.org/wiki/JXTA

Figure 5: Overlay Network

The figure above shows an example of how the different HEDs are interconnected in the Hydra Network.

5.1.2 The HID addressing scheme

The addressing scheme used in the Hydra Network is based on Hydra Identifier (HID). An HID represents a service endpoint, for instance a WebService endpoint. Each service or device has to create an HID in order to be visible inside the Hydra Network. For simple services the Network Manager provides one interface for registration:

```
String HID createHIDwDesc(String description, String endpoint)
```

Any application, or software component, in the system can register its services or devices in the Network Manager. A specific interface (createHIDwDesc) provides a mechanism for registering HIDs using a description and the local endpoint where the service will be placed.

The description is provided by the application or component itself, and it provides a way for identifying the service in other Hydra-enabled devices. In this way, an application running on other Hydra-enabled device is able to get all the HIDs matching an specific description through the following Network Manager interface:

```
String[] getHIDsbyDescription(String description)
```

The endpoint allows the Network Manager to know where to deliver the data received for an HID, because otherwise it would be impossible to determine which component or application is responsible of managing the resource registered with that HID.

5.1.3 SOAP Tunnelling Approach for Device Communication

As the Hydra, and ebbits, architecture is service-oriented, where web-services (WS) is the technology used to implement it, the communication between applications running in different Hydra-enabled devices will be based on SOAP messages. Usually, SOAP messages are forwarded through TCP connections to the destination. The destination address corresponds to the endpoint contained in the message.

Traditional WS architectures are based on client-server architectures, where the server is an always-on end system with a well known endpoint address, which should be known by clients beforehand (using either service descriptors or UDDI registries). The SOAP tunnelling approach proposes a way to replace this client-server architecture for a distributed one, using the Network Manager P2P platform (Lardies 2009). In this architecture, all the peers will act as clients and servers at the same time. shows an example of a client-server based architecture and the distributed approach.

Furthermore, actual WS communications require direct connection between the client and the server, making it impossible to consume services across networks.

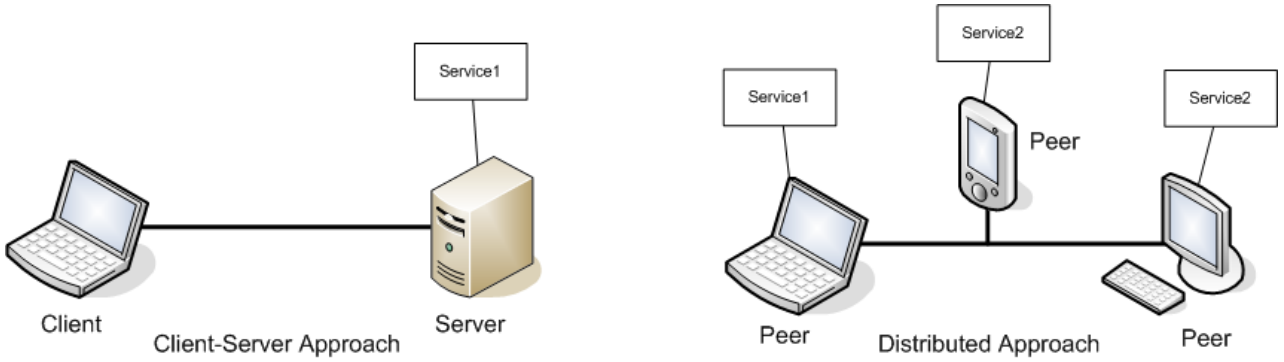


Figure 6: Client-Server vs. Peer-peer

Moreover, in the Hydra middleware, devices are presented as UPnP devices by the Device Manager. But UPnP discovery information is usually restricted to Local Area Networks. Using the SOAP tunnelling the Device Manager will be able to exchange the UPnP information between different Discovery Managers in the Hydra Network. Thus other Device Managers will be able to control UPnP devices located in remote networks using the SOAP technique presented in this section.

Therefore the main objective of the SOAP tunnelling approach is to enable SOAP messages exchange across different networks, making it possible to consume services provided by different Hydra Enabled devices/applications or controlling UPnP devices located in different Local Area Networks**.** The figure below shows an example of the application of SOAP tunnelling. Thanks to the Network Manager and the SOAP tunnelling approach, HED2 is able to discover UPnP devices located at home network (weigh scale and thermometer) and to consume the web services offered by the application running on the HED1.

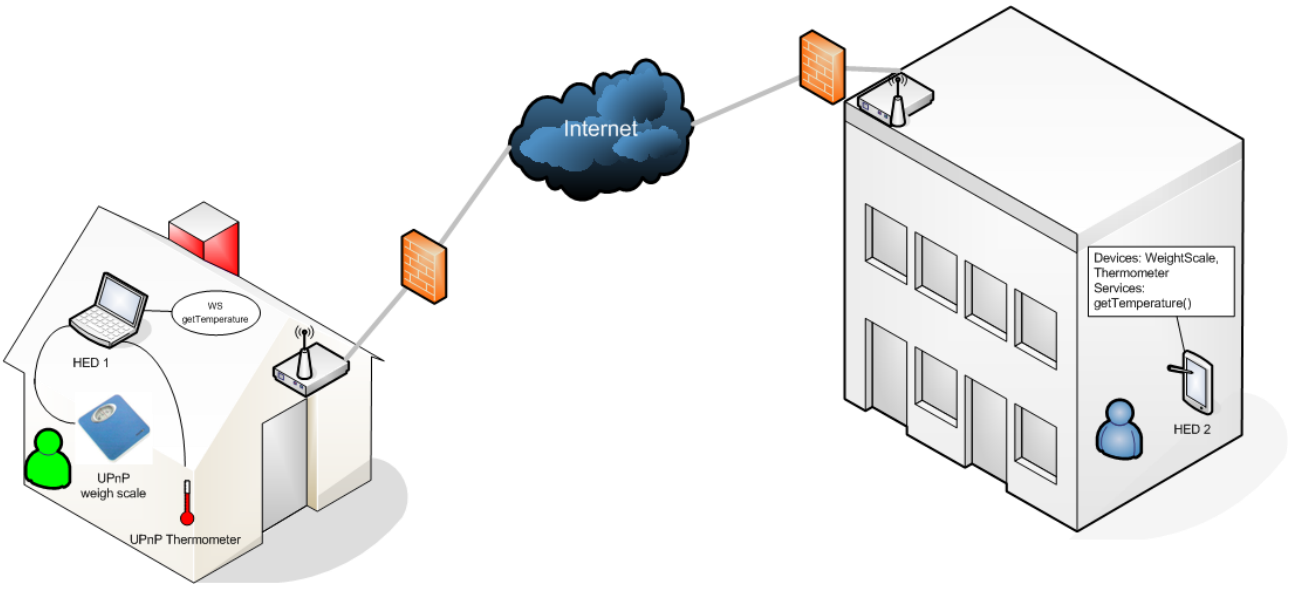


Figure 7: SOAP tunnelling example

5.1.4 SOAP Tunnelling

Thus, the Network Manager enables a way to communicate different Hydra Enabled devices transparently, building an overlay network in which resources (devices, services and contents) are identified by an HID. The main objective of the SOAP tunnelling communication proposed for Hydra is to provide SOAP messages exchange using the P2P transport schemes provided by the Network Manager.

In order to use P2P networking/addressing/transport schemes together with web services and UPnP we need some kind of virtualisation of endpoints that allow us to use P2P networking. For this reason, all endpoints for UPnP and web service calls are grounded in a SOAP sink (ideally locally) which repackages the SOAP message and routes it through the Network Manager, as shown in Figure 7. The Network Manager is responsible of the message transmission and finally calls the SOAP sink that performs a local SOAP call to the intended SOAP endpoint.

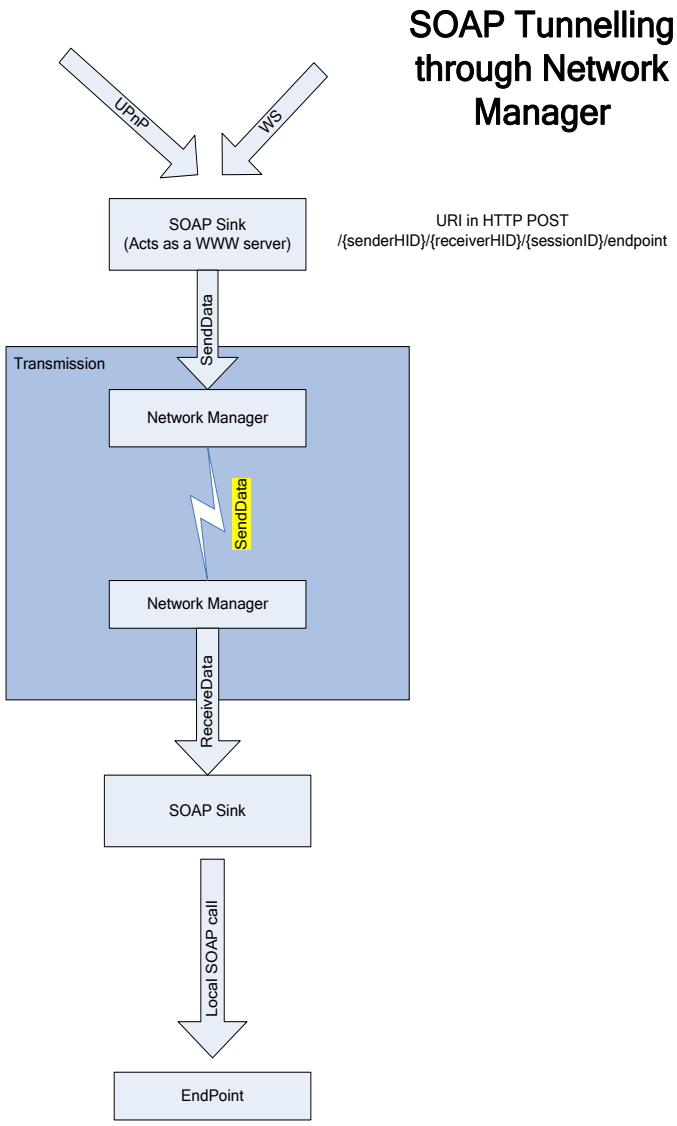


Figure 8: SOAP tunnel

The P2P networking with the SOAP tunnelling technique will facilitate event management, as well as SOA in general, the ebbits architecture. Certain adaptation will however be needed, and are addressed in the sequel.

5.2 Using events

Below we describe the practical usage of events when developing application logic based on the Hydra event manager. Examples are based on the .Net IDE of Hydra, but was also developed the corresponding Java IDE. This approach is subject to adaptations for ebbits platform, as its architecture evolves.

5.2.1 Event structure

Events are a useful tool for several situations in application development. When working with sensors publish/subscribe based events processing is an efficient way of retrieving values, instead of polling sensor values. This way, multiple clients can receive events with the current sensor values.

Events in Hydra are implemented using a standard "publish/subscribe" model and the event itself is built up with a "topic" that is used for defining the event topic and an arbitrary number of key value pairs.

```
<Event>
 <Topic>MyTopic/SubTopic</Topic>
 <Part>
  <Key>ExampleKey</Key>
  <Value>ExampleValue</Value>
 </Part>
 <Part>
  <Key>ExampleKey2</Key>
  <Value>ExampleValue2</Value>
 </Part>
 <Part>
  <Key>ExampleKeyN</Key>
  <Value>ExampleValueN</Value>
 </Part>
</Event>
```

Listing 1: Event topic and events

There are two main parts when working with events to be used in applications:

- Creating events to be consumed elsewhere
- Listening to events

5.2.2 Creating events

In order to create events one only needs to contact the Event Manager. In Hydra projects created using the .net templates the Event manager is available in m_eventmanager. Basically one sets the Topic of the event and then populate the Key/Value pairs.

```
//Create an event with two key/value pairs
global::part[] eventValueKayPairs = new global::part[2];
eventValueKayPairs[0].key = "ExampleKey";
eventValueKayPairs[0].value = "ExampleValue";

eventValueKayPairs[1].key = "ExampleKey2";
eventValueKayPairs[1].value = "ExampleValue2";

 m_eventmanager.publish("ExampleTopic", eventValueKayPairs);
```

Listing 2: Example code creating and publishing an event

5.2.3 Listening to events

Listening to events require that a web service is created that receives the events and it is required that it follows a specific EventSubscriber WSDL. In the projects created with the Hydra .net templates this already done and exists in the EventSubscriberService.cs.

The creation of the Web Service is done using standard .net WCF methods:

```
//Create the ws on port 8123
string address = string.Format("http://{0}:{1}/Service", "localhost", "8123");
Uri[] BaseAddresses = new Uri[]{
            new Uri(address)};
//Turn off 100-continue
System.Net.ServicePointManager.Expect100Continue = false;
//Create the event subscriber
using (ServiceHost serviceHost = new ServiceHost(typeof(Test), BaseAddresses))
```

```
{
    try
    {
        ServiceMetadataBehavior smb;
                        if ((smb =
   serviceHost.Description.Behaviors.Find<ServiceMetadataBehavior>()) == null)
        {
            smb = new ServiceMetadataBehavior();
            smb.HttpGetEnabled = true;
            serviceHost.Description.Behaviors.Add(smb);
        }
           serviceHost.AddServiceEndpoint(typeof(IMetadataExchange),
           MetadataExchangeBindings.CreateMexHttpBinding(), address + "mex");
        serviceHost.AddServiceEndpoint(typeof(EventSubscriber), new
        BasicHttpBinding(BasicHttpSecurityMode.None), "");
        serviceHost.Open();
    }
      catch (Exception e) { Console.WriteLine(e.Message); }
```

Listing 3: Creation of the Web Service for event listening

The next step is to define the function that will receive/implement the Web Service call. In this case the message handling needs to be quick, because the Event Manager will remove the subscriber if the call fails due to time out. If the processing of individual events takes a lot of time one should consider using asynchronous worker threads so that Web Service call can return immediately.

The code below shows an example implementation of the method that receives the events. This implementation only writes the event to the console.

```
#region EventSubscriber Members

//Event call back interface
notifyResponse EventSubscriber.notify(notify request)
{

    string result = request.topic + "\n========\n";

    try
    {
        foreach (part _part in request.@event)
        {
            result += _part.key + "=" + _part.value + "\n";

        }

    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.WriteLine(result);
    return new notifyResponse(true);
}

#endregion
```

Listing 4: Event reception method

The final step is to subscribe to the events that one wants to handle. This involves creating an HID for the event Web Service endpoint and registering the events that will be subscribed using the Event Manager.

```
    //Create HID for Event Subscriber WS
       string myhid = m_networkmanager.createHIDwDesc("eventExample", address);

    //Listen to ExempleEvent
    m_eventmanager.subscribeWithHID("ExampleEvent", myhid);
```

```
//One can listen to multiple events with same interface
m_eventmanager.subscribeWithHID("ExampleEvent2", myhid);
```

It is also important to remove the subscriptions when the process ends. Otherwise one can receive multiple events for one actual event, because we might have multiple subscribers with the same end point. The code below cleans up subscriptions and HID

```
//Unsubscribe to Events
m_eventmanager.subscribeWithHID("ExampleEvent", myhid);
m_eventmanager.subscribeWithHID("ExampleEvent2", myhid);

//Remove HID
m_networkmanager.removeHID(myhid);
```

Listing 5: Subscribe/Unsubscribe to events

5.3 Adapting the Hydra event/network manager

The Hydra event manager and network manager needs to be extended in order to support the proposed Ebbits architecture.

5.3.1 Adding store and forward functionality

The Hydra event manager does not guarantee delivery to the subscribers, if an event fails to be transmitted it will just be dropped. The Ebbits architecture relies on secure delivery of events so this approach will not suffice.

One of the most common approaches is the "Store and forward" technique where the message is not lost if communication fails, instead the message is stored until the communication is working again when it will be forwarded. This approach is widely used in scenarios where real time requirements are low, and is one aspect of Opportunistic P2P Communications, researched in work package 8 in the project.

By adding store and forward functionality to the Hydra event manager will make it suitable for the Ebbits architecture since messages will be guaranteed delivery. Additionally it will be necessary to ad a timestamp to the event that can be used for sequencing events since "store and forward" does not guarantee that events arrive in the same sequence as they were sent.

5.3.2 Adding ontology/taxonomy support

In Ebbits the events are intended to be part of a taxonomy derived from an event ontology. This can be accommodated in the Hydra event structure by annotations in adding key/value pairs that reference the taxonomy and ontology. Though, it is better to introduce a new event type for handling these types of events, making sure that all semantically enriched events contain this information and also removing the risk of spelling errors in the key value pairs. The original type of events should also be allowed to exist in parallel because they are very useful for implementing device drivers etc.

5.3.3 Storage

The store and forward approach implies that the event manager must have storage available. Also the Ebbits central node contains an event database that will be used by the business rule engine for evaluating rules which may depend on previous events. Therefore an event database is needed at all event management nodes in the architecture, although the database implementation might differ depending on where the event manager runs. At the central Ebbits node the database is likely an industry strength database server whilst at the distributed nodes it would be enough with a simple file based solution. Note also that the Event Ontology should be mapped to the ebbits event storage, event instances and data are not stored in the actual event ontology, only their semantic descriptions.

5.3.4 Time synchronisation

There is a need to synchronise time in the distributed Ebbits system architecture because the business rules can express time dependence in the rules, for instance one rule might trigger if event A happened before event B. Since "store and forward" does not guarantee the arrival order of events there is a need to time stamp events in order to be able to order the events in the correct time sequence at the Ebbits central node. Events can also be created in different geographical sites o there is a need to have synchronised clocks in the Ebbits network. Since the network might bridge firewalls etc it might not possible to use a NTP[3] (server for this. Therefore the Network Manager part of the event management architecture should be extended with functionality to synchronise time in-between the different nodes in the Ebbits network.

5.3.5 Stateful event processing and persistency

The event processing foreseen in ebbits, will require implementation of stateful processing i.e., the processing of an event may be influenced by one or more other events. It must also be possible to maintain an event history, and to have access to any results or consequences of the events occurred. The latter is supported by provision of persistent storage on the event processing nodes in the architecture.

[3] Network Time Protocol http://www.ntp.org

6. Future development and plans

In this first iteration of the event subsystem design, the emphasis is on the detection and semantic enrichment of low-level events, and the functionality of the event manager components. The architecture is subject to further refinement in a series four successive revisions, aligned with the iterative development plan of the project.

In a first proof of concept prototype of the event management subsystem, we will perform an analysis and evaluation of the basic Quality Attributes of the architecture:

- Test the scalability and performance of the proposed event architecture.
- Test the fault resilience and delay tolerance with respect to the propagation of events.
- Storage and performance requirements concerning distributed event storage.
- Evaluation of the Event core ontology with respect to its effectiveness in the processing of events.

In the future development we currently foresee that we need refinements in the following areas:

- Development of envelope data structure.
- Database design for event history storage and logging.
- Design and development of domain specific representations for event results/payloads.
- Management of rules sets for event processing.
- Selection/adaptation of rules engines for the different types of event processing rules.
- Refining the mapping of the Event core ontology to the Event process and data model.

We expect that the prototype implementations and evaluations will reveal additional areas for refinements and architecture revisions. These results will be reported in the successive versions of this deliverable.

7. References

Hydra (2008). D3.9 Updated System Architecture Report. Hydra Project Deliverable, IST project 2005-034891.

Lardies, F. M. (2009). Deploying Pervasive Web Services over a P2P Overlay. 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, IEEE Computer Society. **:** 240-245.