Enabling the business-based
Internet of Things and Services

(FP7 257852)

D5.2.2 Architecture for intelligence integration

Published by the ebbits Consortium

Dissemination Level: PU = 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 version: 1.0 Submission date: Date

Document control page

Document file: D5.2.2 Architecture for intelligence integration
Document version: 1.0
Document owner: Ferry Pramudianto (Fraunhofer FIT)

Work package: WP5 – Architecture for intelligence integration
Task: T5.1 – Architectural analysis and description of the centralized and distributed intelligent service structured
Deliverable type: R/~~P/O~~

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	Ferry Pramudianto (FIT)	14-11-2011	Table of Content
0.2	Boehm-Peters, Antje (SAP)	08-12-2011	Requirements validated and use case diagram SAP
0.3	Mauricio Caceres (ISMB)	10-01-2012	First draft of Software architecture for context management in the device level
0.4	Mark Vinkovits (FIT)	12-01-2012	Network Communication
0.5	Ferry Pramudianto (FIT)	10-02-2012	Context Awareness definition Functional View Logical view Information view
0.6	Peter Kostelnik (TUK)	10-02-2012	Developing Context-aware Applications
0.7	Ferry Pramudianto (FIT)	27-12-2012	Improvement according to the reviewers comments
1.0	Ferry Pramudianto (FIT)	28-02-2012	Final version to be submitted to the EC

Internal review history:

Reviewed by	Date	Summary of comments
Mauricio Caceres (ISMB)	2012-02-24	Minor typos and grammar errors.
Peeter Kool (CNET)	2012-02-24	Accepted with minor comments on the structure.

Legal Notice

The information in this document is subject to change without notice.

The Members of the ebbits Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the ebbits Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Possible inaccuracies of information are under the responsibility of the project. This report reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained therein.

Table of Content:

1. Executive summary ... 4
2. Introduction ... 5
2.1 Purpose, context and scope of this deliverable ... 5
2.2 Deliverable Organization ... 5
3. Context Awareness ... 6
3.1 Context Definition ... 6
3.1.1 Context awareness definition in ebbits ... 6
3.2 Overall architecture proposal ... 7
4. Self-* Manager and Monitoring Device Context ... 9
4.1 Related Works ... 9
4.2 Overall Architecture Refinement ... 10
4.3 Functional View ... 11
4.3.1 Self-* ... 11
4.3.2 Context Manager ... 13
4.4 Logical view ... 14
4.4.1 Domain Analysis ... 16
4.5 Information view ... 18
4.5.1 Sensing ... 18
4.5.2 Control ... 20
4.6 Deployment view ... 21
5. Developing Context-aware Applications ... 23
6. Network Communication ... 25
6.1 The original Network Manager internal structure ... 25
7. Summary and Conclusion ... 29
8. References ... 30

1. Executive summary

This deliverable presents a report on context-awareness in the ebbits platform, as an update of the previous deliverables – D5.2.1 Architecture for intelligence integration and D5.3.1 Specification of Multisensory fusion and context awareness services. As with the two prior deliverables, the idea of defining a middleware for context management is to factor out commonly used functions in this regard from the context-aware applications and services.

The previous two reports defined the possible architecture of context-awareness within ebbits, This deliverable expands on the concepts presented in its previous iterations, updating the architectural elements of the Context-awareness in ebbits, with a greater focus on describing a more detailed functional and information views. The architecture description is elaborated with a use case example for the self-* capabilities in the device proxies.
The self-* properties in ebbits is intended to define a set of functionalities that allow the proxy to recover from the most common faults. These functionalities can be extended by the device developers according to the capabilities of the physical devices.

2. Introduction

As described in D5.3.1 Specification of sensor fusion and context awareness services 1, we identified two areas where context awareness can be applied. The first area aims at improving the reliability the internal ebbits components in a dynamic environment. The second use case is to innovate the development of monitoring application in shop floor as well as for traceability purposes.

2.1 Purpose, context and scope of this deliverable

The purpose of this deliverable is to provide refinement of the previous WP5 architecture deliverable (D5.2.1). The architecture provides the formalization of a system design within work package 5 based on the related requirements that we have gathered in work package 2. The architecture will describe in more details the main block components in work package 5. The architecture is presented in UML 2.2 standard that illustrate the functional view, information view, and the deployment view of the context awareness. In the next iterations, more architectural views will be explained in detail.

2.2 Deliverable Organization

This deliverable is organized as follows:

- Chapter 3 describes the definition of the context awareness and review of the previous proposal of the context awareness architecture in ebbits.
- Chapter 4 describes the refinement of the architecture especially for the Self-* Manager and Monitoring Device Context. The architecture is presented in detail that covers the logical view deployment architecture. This chapter also shows which requirements of the ebbits platform it addresses.
- Chapter 5 describes a workflow how developers use ebbits context awareness for developing Context-aware Applications.
- Chapter 6 provides an overview of the network manager's new architecture.

3. Context Awareness

3.1 Context Definition

There have been many definitions of context awareness and context for the past 20 years. However, there was still different assumption and opinion on context in the consortium. Therefore, we believe it is essential to discuss the definitions of context awareness and establish a common understanding within the scope of ebbits.

The previous definitions of context awareness came from the ubiquitous computing community. The classic definition of context only considered the interaction among human as a user of computing entities with the environment of the user. The following are examples of classic definitions of context:

> Schilit and Theimer '94 refer to context as "location, identities of nearby people and objects, and changes to those objects." (Schilit and Theimer 1994).
>
> Schilit, Adams and Want '94 "Where you are, who you are with and what resources are nearby" (Schilit, Adams et al. 1994).
>
> Brown, Bovey et al. '97 define context "as location, identities of the people around the user, the time of day, season, temperature, etc."(Brown, Bovey et al. 1997).
>
> Ryan, Pascoe et al. '98 define context "as the user's location, environment, identity, and time." (Ryan, Pascoe et al. 1998).
>
> Dey '98 enumerates context as the "user's emotional state, focus of attention, location and orientation, date and time, objects, and people in the user's environment"(Dey 1998).

As context awareness evolves to broader applications, the previous definitions needed to be extended.

> Context Aware: "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task" (Abowd, Dey et al. 1999).
>
> "Something is context because of the way it is used in interpretation, not due to its inherent properties. The voltage on the power lines is a context if there is some action by the user and/or computer whose interpretation is dependent on it, but otherwise is just part of the environment" (Winograd 2001).

3.1.1 Context awareness definition in ebbits

The modern definition of context evolves from the classical narrow definition such as location and id into a broader range of information that is relevant for the tasks that the users do. Since ebbits aim at developing an intelligent platform in which autonomy of the system is desired, the definition of the users has become broader than what has been defined. Users in ebbits include:

- Human users who are the end-users who work with applications that runs on ebbits, and the developers that develop the applications. Definition of context in this scope mimics the existing definitions mentioned by Dey & Abowd which implies any information that is relevant for performing users' tasks such as the location, roles, background knowledge.
- Autonomous components that use other components and run with limited human interference. E.g.: a device proxy of a Linksmart uses network and computing resources of the node. The context in this sense would be any information of the hardware and network that is related to the device proxy's task which is providing a web service interface representing the physical devices.

Considering the available definitions of context awareness could raises ambiguousness in ebbits, we come into conclusion that the definition of context awareness in ebbits is:

> "An application that is able to adapt its behaviour according to any relevant situation that happens in its environment. Situations are defined by several context cues that meet certain conditions. The conditions are relative to the entities that interact and do their task within the application domain."

As a concrete example of this definition is an automatic cooling system of a welding gun should prevent the temperature gets too hot, the situation "too hot" must be defined by the properties of the welding gun, e.g., the material of the welding tip, the raw material that is to be welded, environment temperature. We define these properties as context cues of the welding gun that could define the situation "too_hot". Such as for welding gun XYZ "too_hot" is when the temperature above 300° F. whereas 300°F could still be normal temperature if the raw material and the material of the welding tip are more solid steel.

3.2 Overall architecture proposal

This section summarizes the previous architecture proposal defined in D5.3.1. it is included again in this deliverable to give the context of the architecture that is described in the next sections.

The previously proposed overall architecture of sensor fusion and context-awareness subsystem of ebbits is illustrated at Figure 1.

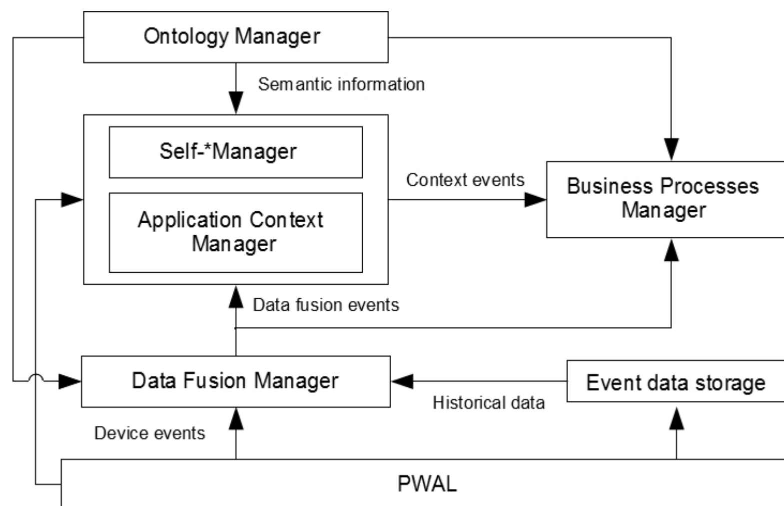

Figure 1. The previous proposal of sensor fusion and context-awareness subsystem.

The responsibility of the particular components can be summarized as follows:

1. PWAL – generates the low-level physical events from the real-world environment. These events are consumed and processed by the ebbits platform.
2. Event data storage – stores all event data generated by the PWAL. This data are available as historical data for further processing, for example for computing the specific information within the defined time windows (e. g. the trend of water consumption of pigs during the time period, which can be further used by the data fusion and context components to infer the possibility of sickness, if the water drinking trend has anomalous characteristics).
3. Data fusion manager – combines the sensor data in real-time and produces the events with the computed fused information. This information is consumed by the context-awareness components. Another responsibility of the data fusion component is to handle the event historical data. Historical data can be used to perform the computations combining the various sensor data, such as aggregated values (e. g. average temperature) or several predefined statistics (e. g. deviation of water consumption trend) during the period of time.

Computed values can provided by request or can be passed to the environment in the form of events.

4. Self-* manager – the rule engine responsible for processing the information related to the system failures. The new states of the system malfunctions are derived by the rules in the form of events (e. g. the sensor is broken).
5. Application Context Manager – the rule engine responsible for processing the information related to the application states. The new states of the application are derived by the rules in the form of events (e. g. the welding has started).
6. Business processes manager – the rule engine responsible for processing the information related to the business logic. The result of the business rules application can be the new events passed to the environment (e. g. need to send invoice), but also the action to be taken (e. g. send sms to the maintenance crew).
7. Ontology manager – provides the additional semantic information supporting the all decision processes provided by the above mentioned components.

The logical decomposition of the architecture elements arises one important problem which has to be focused: the time synchronization. The particular components consume and generate the events and it is also important to take into account the time, when events were generated. As the components will, in run-time most likely, run at the different hardware machines, the time synchronization is crucial. The most probable solution to this problem is the introduction of the central time-synchronization component, which will handle the ebbits specific time for all components. But this solution is still being further investigated for the upcoming D6.4.

Self-*, application context and business processes managers are assumed to be implemented as rule engines, using some conventional implementation (such as Drools [Drools]) or possibly semantic DL reasoners. Also in the case, when there will be selected the same implementation for all rule engines, the logical decomposition of the rule engines is required, because:

- there is assumed a big amount of events in the real-world environment and merging all rule-based computations into the one rule engine would most probably lead to the insufficient run-time performance and scalability issues,
- logical decomposition of the rules enables the more effective knowledge readability and maintenance
- each manager handling the specific rules has its own responsibilities and provides the specific services

4. Self-* Manager and Monitoring Device Context

4.1 Related Works

The autonomic computing initiative, started by IBM in 2001 envisions interconnected computer systems that are capable to make decision on their own guided by high level policies.

IBM has defined four functional areas of self-managing autonomous system that includes:

- Self-Configuration: Automatic configuration of components including deployment of new components and removal of components.
- Self-Healing: Automatic discovery, and correction of faults.
- Self-Optimization: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements.
- Self-Protection: Proactive identification and protection from arbitrary attacks.

The underlying approach that are applied for self-* is closed control loops / feedback loops. Closed control loops typically involves 4 activities (collect, analyze, decide, act). The implementation of a self-adaptive system becomes complicated when inferring the faults of the system based on historical data or several parameters are involved. After the fault of the system is inferred, the system must derive an appropriate plan to act upon the error and execute the plan. These steps go on until the state of the system is in an acceptable interval.

Kepart and Chess have identified a high level architecture of self-managing components. IBM researchers (Computing 2006) have defined self-managing as resources that manage their own behavior in response to higher-level goals, and interact with other resources to provide or consume computational services. Resources should be aware of what they are (e.g.: database, software components, gateways, and servers), which version they are, and service agreements they have with other components.

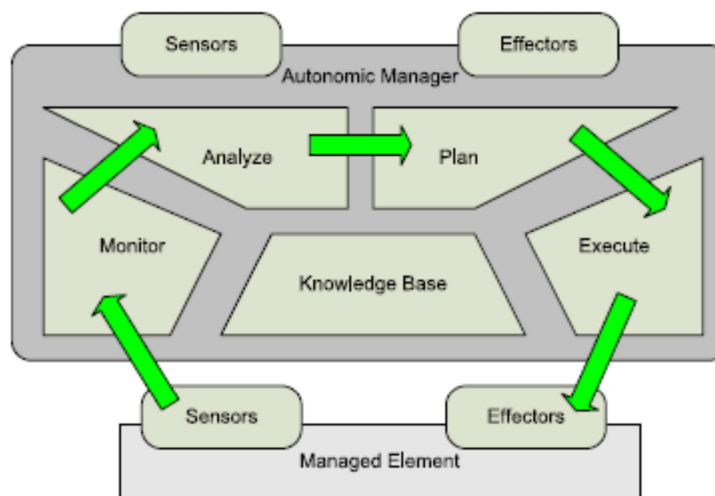

Figure 2. IBM Autonomic System Architecture

Enabling self-managing, each component must be able to collect the information it needs from the system automatically. As shown in figure 1 the monitor function collects and aggregates data from a sensor system. Sensor system is any input that could be delivered from hardware as well as from user input. The analyze function contains mechanism to correlate the acquired data with model and algorithm to infer the state of the system. The plan function construct step by step actions that are needed to achieve the goals and objectives defined in the policies. The execute function performs the action.

4.2 Overall Architecture Refinement

In ebbits, self-* properties are urgently needed to ensure that ebbits components are resilient to dynamic environment, as outlined by our users (requirement #47). Ensuring that every part of ebbits platform is resilient, we need to have a system whose components in different levels are able to adapt to changes in the environment so that ebbits will have a strong foundation from the leave nodes to the core components. However in this iteration, we focus firstly in self-* capabilities for the leaf nodes of the ebbits which consist of Linksmart devices and device proxies as the requirements #140 points out that this should be prioritized.

The device proxies aim at providing Linksmart functionalities for the devices that cannot host the Linksmart middleware inside them such as web service interface and Universal Plug and Play (UPnP) discovery. As a reminder of Linksmart device classification has been documented in D5.2.1 section 5.2.1. The device proxies are initiated when physical devices are discovered by the Device Discovery Manager (DDM) as independent processes that run on the operating system. This design decision was taken to decouple the runtime dependency of proxies and DDM in order to avoid that failed components affect the healthy ones.

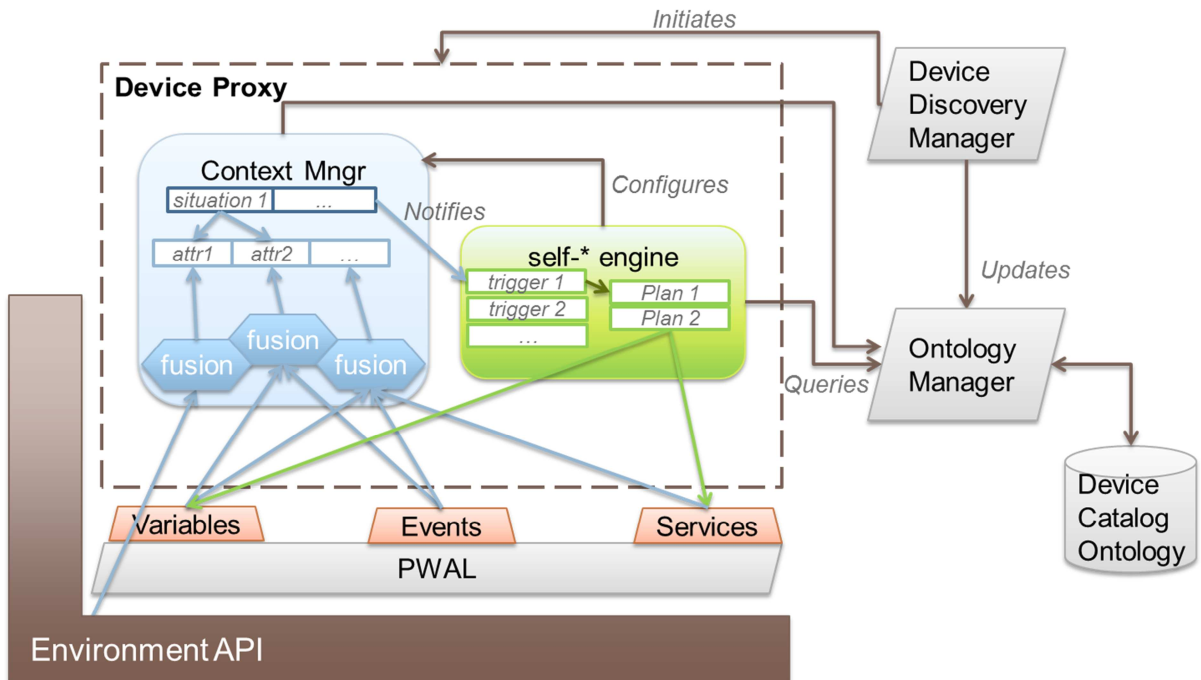

Figure 3. Interaction among ebbits components

This architecture refines the specification described in section 4.2. The high level overview of the architecture is depicted in Figure 3. The Context Manager and self-* engine are part of the device proxy and they interact with external sub-components. This architecture follows to the architecture suggested by IBM's Autonomic System Architecture. We use the Context Manager to monitor and analyze the situations and the self-* engine to develop and execute the plans according to high level policies. The policies describe the goal of the system without defining the detail activities that the system must do. This design decouples the policies with actual implementation of the capabilities of the devices.

The interaction is started when a new device is discovered by the device discovery manager. It then updates the device catalog in the ontology and initiates an appropriate device proxy. After the device proxy is initiated. Based on the policies that are defined by the administrator, the self-* engine configures the Context Manager to monitor certain situations. The Context Manager monitors context attributes and queries the ontology manager to get the necessary taxonomy to determine which sensors and data sources could deliver the context cues e.g.:

Policy : "WHEN Resource-utilization_too_high REDUCE Cpu_Usage"

Situation: "Resource-utilization_too_high = CPU_Usage > 90% AND duration > 5 Minutes"

The "Resource-utilization_too_high" is a situation that the Context Manager recognizes when the "CPU_Usage" context attribute meets the condition of higher than 90% and the "duration" context attribute meets the condition of higher than 5 minutes. These conditions to determine the situations are application dependent and therefore should be tuned through extensive studies in the application domain.

4.3 Functional View

4.3.1 Self-*

Purpose

The goal of the self-* properties in the leaf nodes can be summarized firstly to do corrective action on unexpected events and faults that happened in the device proxy as well as network connection between physical device and the proxy. Secondly it also aims at doing preventive action on the expected faults before the system goes into faulty state.

Preventive Actions

As we learned during the Hydra project, using hardware resources in parallel without a proper resource management could result in a bottleneck. In Linksmart, several device proxies will be automatically initiated by a discovery manager. This process happens in a gateway that has limited resources. Therefore, a proper resource management needs to be done to avoid the quality of the services drop beyond an acceptable level or even fail completely.

To prevent this problem, we will monitor and analyse computing resources such as CPU and network usage and try to initiate an appropriate action that helps preventing an over utilization. We identified several parameters that could influence the network and hardware utilization of the gateway for instance:

- The number of proxy instances that share the resources.
- Securing the communication through encryption and decryption.
- The number of events generated by the physical devices (sampling frequency).
- The number of events that the proxies must generate to inform the application.
- An instance of proxy that consumes and blocks most of the resources.

These parameters will be adapted according to the resource utilization of the gateway.

Corrective Actions

What we meant by corrective actions are adaptations to the system after it has experienced specific predefined failures. In the device proxy we identified several failures that could occur and use these use cases as an initial self-healing behaviour. These behaviours are a set of generic failures that can be extended by the developers. These failures can be categorized into:

1. Network failures that include the network connection between physical devices and the network connection between the proxy and the application this includes, e.g., high number of packet lost, high latency, and full disconnection.
2. Service execution of the devices can be disrupted by unreliable network connections, over utilization of the physical devices, or erroneous sensor readings (outliers) etc. this could end up in a half way state changes that needs to be rolled back.
3. Internal device proxy states might not be correctly initialised because of the dependencies to other Linksmart managers or distributed components have not yet fulfilled.
4. Unexpected behaviours caused by programming errors such as unforeseen state that causes an endless cycle, forgot to (re-)initialize the variables etc.

Main Functionalities

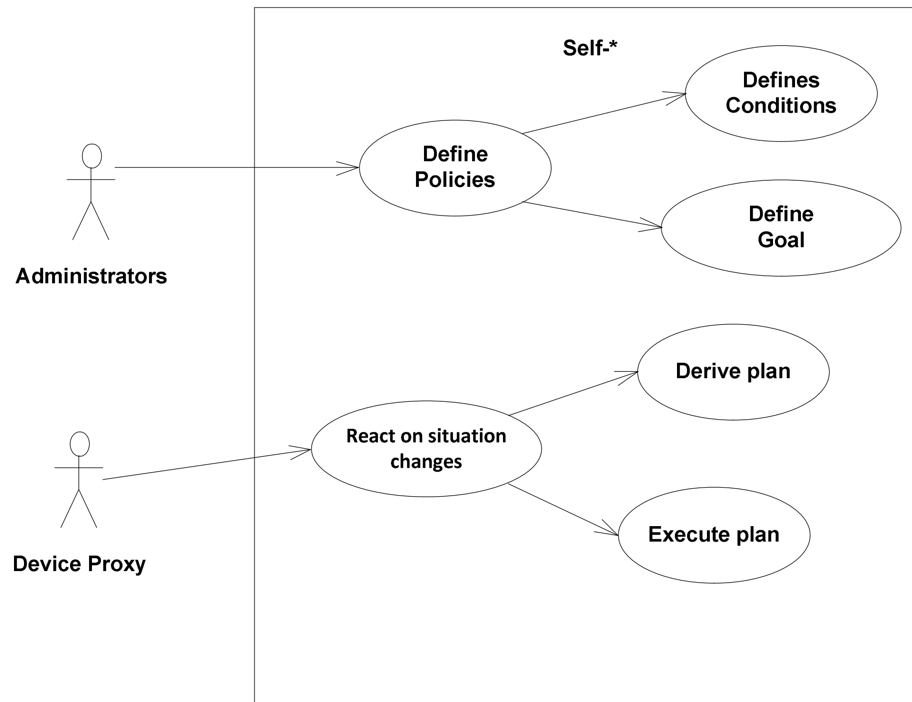

Figure 4. Use case of the self-* engine

Based on the self-* use cases and the Description of Work (DoW), we will make use of the context awareness to infer the situations described in the use cases. This situational contexts are to be used by self-* engine to take decision and perform the adaptation to the environment.

The self-* engine has several functionalities that are depicted in the use case diagram on Figure 4

1. The Self-* Engine can be (re)configured by defining certain policies. The policies contain generic rules for recovering from the common faults (when to re-connect to the physical devices, when to reduce the traffic).
2. The device proxy provides an interface that the developer must implement, i.e., device specific code to build connection or reduce network traffic.
3. It should provide an interface for the developer to extend the self-* policies, if any device has specific features.

Relevant Requirements

#138 The system should support distributed intelligence on embedded system.
#139 Support runtime reconfiguration
#40 The middleware should monitor device's resource usage.
#454 The system must monitor the state of devices and entities.
#141 Report errors in devices (/ebbits system)
#47 Resilience and adaptable to environment condition changes
#454 The system must monitor the state of devices and entities.
#139 Support runtime reconfiguration

4.3.2 Context Manager

Purpose

The key role of context-awareness in ebbits is to enable applications to delegate the process of recognizing and monitoring contextual situations. This is essential to decouple the business logic of applications from the task of sensing context information since these tasks involve specific sensor and actuator implementations, sensor fusion algorithm and communication technology.

The Context Manager provides the capability for the configuration of situational contexts that may collect relevant data together for a purpose, represent semantic entities, and represent data sources. These situations may contain rules that perform the reasoning over the context cues, semantically processing it, to perform context-sensitive actions. These actions may be used to notify context-aware applications of significant situation changes, for the purpose of the application, among other possibilities.

Main Functionalities

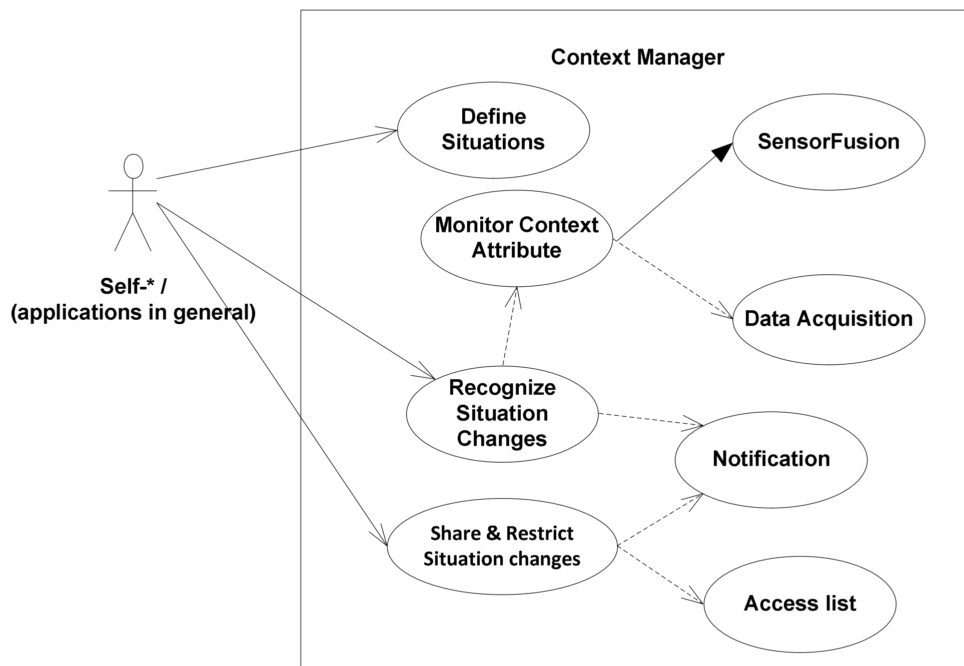

Figure 5. Use case diagram of the Context Manager.

The Context Manager should provide the following functionalities:

1. It allows the applications to define the situations that the applications are interested in. the application configure the Context Manager by describing a situation and its relationship to context cues in a form of a rule.
2. The Context Manager should recognize the situation changes by monitoring context cues in the environment. It uses the rules defined by the application and the semantic model to assess if the situation has changed.
3. The Data Acquisition is responsible to make the communication to the data sources is transparent to the sensor fusion module. It provides data to the component it is used by, reporting it to the subscriber asynchronously, either at a specified frequency (for data "pulled" from a data source), or as events are received ("pushed" from the data source).
4. It should provide a way to pre-process and fuse the raw data coming from the sensors so that insignificant small changes do not overwhelm the system.
5. It should provide a notification to inform application when interesting situations appear.

6. It should allow the application to share their situations to other applications as well as restrict other application to query the current state of the situations.

Relevant Requirements

#138 The system should support distributed intelligence on embedded system.
#380 Monitored/sensed data should be contextualized (timestamp, geotag, type, etc).
#453 The system must be able to assign fused data as a context attribute of an entity
#454 The system must monitor the state of devices and entities.
#82 System should provide access restrictions to sensitive information.
#451 Sensor fusion algorithm must be added during run-time in a modular and extensible way.
#401 ebbits platform should have a publish-subscribe system.
#139 Support runtime reconfiguration
#456 The system should be able to process a large number of sensor events

4.4 Logical view

In this view, we describe the logical clustering of concerns that should be separated. In the Context Manager we identify five building blocks:

1. Situation management is responsible for receiving a set of situations defined in a specific language. It then parses the configuration to retrieve the context cues that it should observe and configure the data acquisition and sensor fusion to observe these context cues. The context cues are semantically modeled using ontology describing which sensors and sensor fusion algorithms to be used.
2. The sensor fusion is pre-defined with several filtering algorithms that can be chosen based on the type of sensors and data. The algorithms are mainly used to increase the confidence of the sensor readings. These algorithms include: weighted moving averaging, Kalman-Filtering, etc.

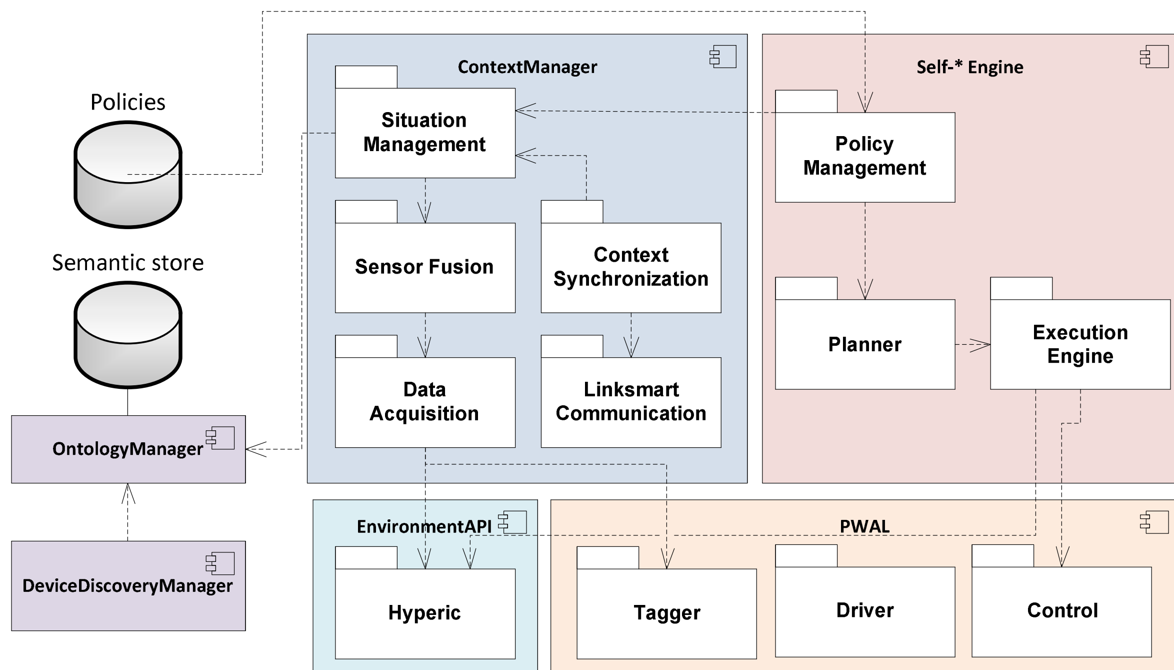

Figure 6. Package view of the Self-* Manager and the Context Manager.

3. Context synchronization is responsible to synchronize the context of entities to other Context Managers in the Linksmart network. To ensure the privacy of the entities and to keep the communication overhead low, synchronization will only happen for the entities that the Context Managers share. The context synchronization will make use of the Event processing agent defined in the "D7.3.2. Technical description of the implementation of Data and Event Management models".
4. The data acquisition consists of classes that encapsulate the process of acquiring data from different sources. Such as depicted in the Figure 6, the data acquisition could retrieve data from the PWAL layer as well as platform environment data such as CPU, network, and memory usage. The data acquisition package will provide an uniform interface that abstracts different underlying technology.

The self-* engine contains 3 main responsibilities that include:

1. Policy management which is responsible to provide interfaces for defining and maintaining policies. These policies comprise high level goals that the system should pursue when certain situations occurs. Policies are used to decouple general reusable rules from small tasks that are tightly coupled with the hardware and software implementation.
2. When adapting the system, these general policies must be mapped onto concrete activities that involve hardware and software implementation. The planner aims at defining these concrete activities based on the policies and available resources. The activities should tell the system specifically to execute a given task with specific resources such as sensors, actors, as well as computing resources such as CPU, memory, and network interface. Moreover, the planner will generate a plan that includes the activities, their priorities, dependencies, and the order of execution as a guide for the execution process.
3. The execution engine executes the plans generated by the planner and monitors the execution process. We have not yet consider how to handle parallel execution in this engine. Parallel execution of the plans could reduce the execution time; however it introduces computational complexity caused by parallelism problems such as race conditions, mutual

exclusion, and synchronization. Therefore we plan to evaluate the performance of adaptation in sequential order first.

4.4.1 Domain Analysis

We conducted a domain analysis from the technical use cases (D2.1 Scenarios for usage of the ebbits platform) and state of the art analysis (D5.1.1 Concepts and technologies in intelligent service structures) and derived more detail logical view of the roles for the Context Manager and the self-* engine.

As depicted in the Figure 5 the policies guide the planner to generate plans that divide responsibility between sensing and controlling. The sensing is done by the Context Manager whereas the controlling is done by the self-* engine.

The Context Manager is however designed to be used not only by the self-* engine but also by any monitoring applications. Therefore, the Context Manager must be able to be easily configured by the applications that currently use it.

After analysing the requirements and DoW, we decided to configure the Context Manager through policies that guide the Context Manager to determine which environmental parameters (context attributes) it has to monitor.

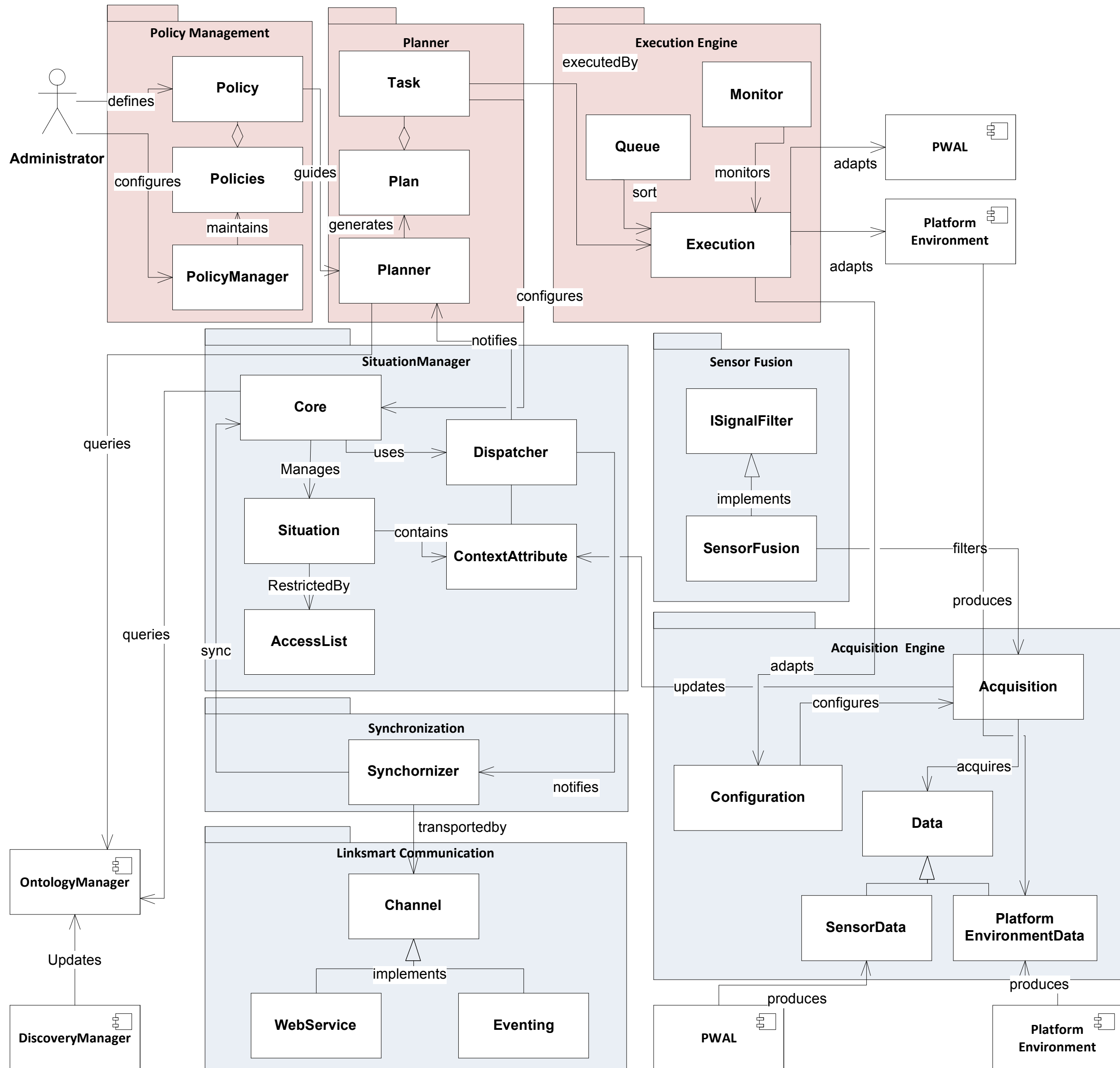

Figure 7. Class Diagram of the Self-* Manager and the Context Manager.

The domain model starts with the policies that are defined by the administrator. The collections of policies are saved in the system that is administered by **the Policy Manager**. The user (administrator) configures the policy manager to activate one of the policies.

The Planner has a core planner that generates plans and task based on the policy. The tasks are used by **the Situation Manager** and execution engine packages as means to configure their behaviour.

Inside **the Situation Manager** there is a core class that manage the flow of information. **The Core** keeps track a set of situations. These situations consist of context attributes that have values. The situation package also has a dispatcher that is responsible to notify other components when the desired situation is fulfilled.

To handle the local sensors, the acquisitions engine has a controller class that is named **Acquisition**. It uses the representation data class to encapsulate heterogeneous sensor data it acquires from different sources. **The Acquisition Engine** has also a configuration that could be influenced by **the Self-* Engine**. It also uses the sensor fusion to determine filter the raw data and infer the confidence of the data.

As ebbits is meant to address the Internet of Things (IoT) where everything could be distributed over the internet, the data sources might be as well dispersed in the internet. Therefore means to synchronize and secure the information through the network must be provided. Thus, situations are restricted by an access list that allows and forbid other Context Manager in the internet to access the information. The Context Manager also has a synchronizer that is responsible to sync the incoming and outgoing context. It uses the communication channel that abstract different communication means such as polling web service and listening to network events.

The execution engine uses a queue to schedule the execution of the tasks it gets from the planner. This allows the execution is ordered according to priority. The execution is also monitored and logged in a log file.

4.5 Information view

4.5.1 Sensing

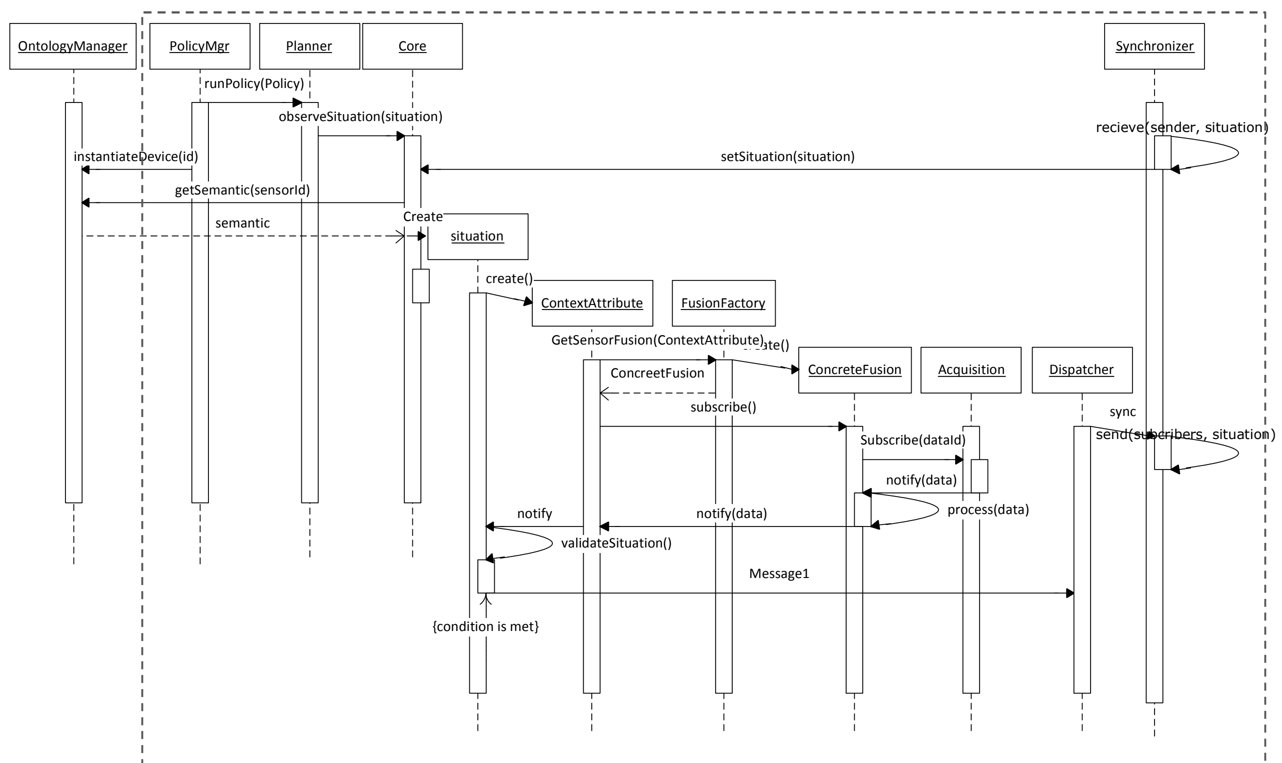

Figure 8. Interaction diagram of the Context Manager.

Configuration of the Context Manager is started when monitoring applications, such as policy manager inside the self-* engine, extracts the situations to be recognized from the policy and sends these situations to the core in the situation manager package. The core will extract the entity involved in the situations and their context attributes. The core needs to monitor these context attribute based on the input from some data sources such as sensors. The core asks the ontology which data sources are able to provide the values of the context attributes.

The automatic discovery of sensors and data sources is possible when the relationships among situations, entities, and context and data providers are modelled in the ontology. In Section 5 an example of context model in the ontology will be presented.

This ontology model permits the core to know which data source is able to provide the value of the context attribute. The core then delegates the process of acquiring the data to the acquisition component. The acquisition component loads the appropriate library to retrieve the data. The data could be provided by a physical device which can be accessed through the PWAL. Secondly, which is

more relevant to the self-* case, the data could also be provided by the platform on which ebbits runs which can be accessed through the platform environment component.

The type of the data also determines what kind of sensor fusion that should be used by the acquisition process. The sensor fusion algorithms can be implemented as a plug in that could be added and loaded during runtime. The simplest data fusion algorithm is to stabilize a noise data that use a simple averaging over time. There are several improvements to this algorithm such as Kalman filter that uses the weights and averaging the values with its predicted values. The weighted moving average analyzes the variance of the data and generates the confidence value of a data.

The fusion in this level is meant to deal fuse only discrete-time signals in order to reduce noise in the data. As we assumes that all data contains noise from the environment or caused by the imperfection of the sensors. Fusion among several sensors and other data sources is addressed by the data fusion component that is being developed in the work package 7 under the T7.2 Event management. In this task, combining various events from devices will be discussed.

Using semantic models and simple policies to change the behaviour of the application give a great flexibility when new policies and devices are introduced into the system. However, this degree of freedom raises the complexity of the deployment for instance how details the semantic model and policy must be defined. Therefore we aim to achieve an acceptable balance between the complexity of the set up and reconfiguration.

Remote Context Synchronization

Inferring situation, a Context Manager might need to monitor context attribute of an entity that is connected to another Context Manager in the network. Therefore synchronization among Context Managers must be provided. However, in order to reduce the amount of traffic we only synchronize the context attribute if there is an application that is interested in it. The sequence of syncing context attributes is described as the following.

Firstly, the local Context Manager tries to find the context providers locally by contacting the Acquisition Engine. If the Acquisition Engine does not have the necessary sensors, the Core sends out an asynchronous broadcast to the other Context Managers through the Network Manager. When the other Context Managers receive the broadcast, they will parse the request and try to find the suitable context providers and check whether if the sensor has started monitoring by contacting its data acquisition engine. If it finds the suitable sensor and the sensor is not restricted to the sender, the core will inspect if the condition requested by the sender is met. When it is met, it notifies the sender. We apply the event processing locally to reduce the communication overhead and network traffic in ebbits. Next, the sender will receive this value through the synchronizer. The synchronizer will then compare it with the last value to ensure that the value is new. If it is newer than the last value, the synchronizer will forward it to the core so that it could update the relevant context attribute, otherwise the value will be ignored.

4.5.2 Control

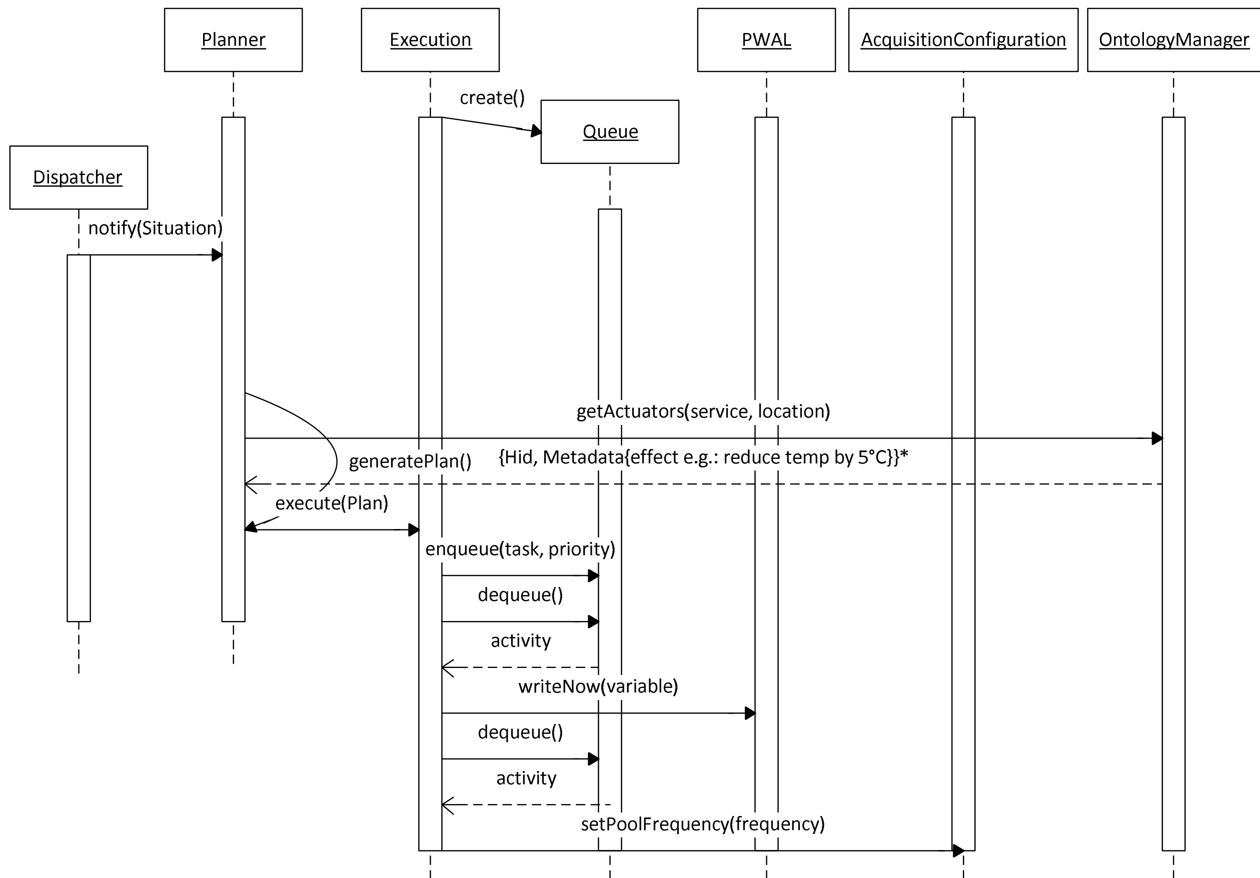

Figure 9. Self-* Engine

As situations have been recognized by the event manager, the self-* engine uses the information to adapt the system to follow the policy defined by the administrator. The interaction inside the self-* engine is started as soon as the planner is notified (by the dispatcher) when a situation of its interest, is recognized. The planner then checks the policy to get the proper goal that it has to achieve. This goal is then used as a guide to choose the available services from the actuator.

In general, the planner requests the information of the respective services from the ontology manager. The ontology manager replies the request with a list of services and their metadata. The metadata contains the description of the services such as capability of the services, the effect of the services. The planner uses the effect of the services to generate a plan that describes which services have to be invoked.

The planner sends the plan to the execution engine. The execution engine parses the task to be done and queue them into its queue stack. The execution object takes the task one by one from the queue and executes them. For instance, self-* engine could reduce the pooling frequency to the physical device to reduce the resource utilization. It could also control the actuators that are connected to the proxy through the PWAL layer.

The monitor is then informed with the status of the execution for instance, which tasks have been executed successfully, and the tasks that are still in the queue.

4.6 Deployment view

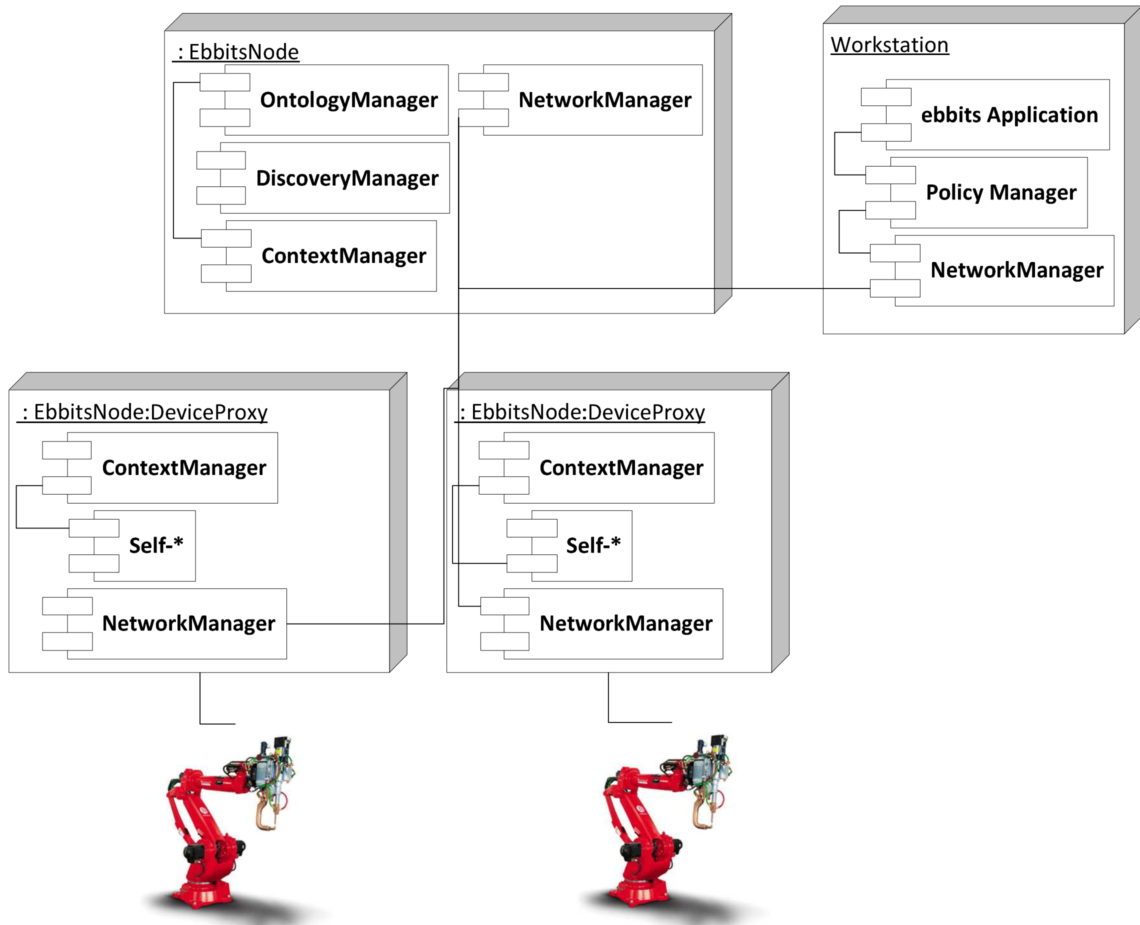

Figure 10. Deployment of the distributed context management.

The deployment is depicted in the Figure 10. The Context Managers are distributed over the ebbits network. The goal of the distributed system is to process the sensor events locally and only report interesting context events to the root Context Manager and finally to the application. Each device proxy has a Context Manager that takes care of the sensing tasks for the self-* engine, as well as sensing tasks that are delegated from the other Context Managers i.e. root Context Managers at the gateways.

The Context Managers could also be deployed separately from the device proxy, e.g., on the ebbits gateways. The network architecture of the Context Manager is peer-to-peer (P2P). However, in the implementation we have to define how the proxy looks like, e.g., a robot can be represented as a proxy that offers services from different sensors and actuators that is attached to it or if the robot changes its tools constantly (welding and grip), it would make sense to represent each tool with another proxy. When many devices are represented by a single proxy, the network architecture forms a hierarchy.

Figure 10 depicts an example where proxies and gateways are deployed in a dedicated hardware. In this case, a network manager must be deployed on each hardware component. The network manager is responsible to make the communication among Context Managers and proxies transparent. These components do not have to deal with physical network addressing. For the Context Managers and applications it seems that they all are in the local hardware, as they only use Linksmart HIDs to address services.

An application can use any of the Context Managers in the ebbits network. A request to monitor certain context is disseminated through all Context Managers, and the Context Manager that has the information will add the application to its subscription list. Having pure peer to peer network

architecture might not be optimal if dealing with a large number of nodes, since the number of broadcast messages increases dramatically and need to be transported through the network manager. A possible solution is to apply clustering to the Context Manager where the cluster heads are the nodes that are responsible for handling the broadcast messages, then they reroute the messages to its members (introducing a delay). Thus, for the first implementation, we would like, firstly, to explore the peer to peer solution without any clustering.

5. Developing Context-aware Applications

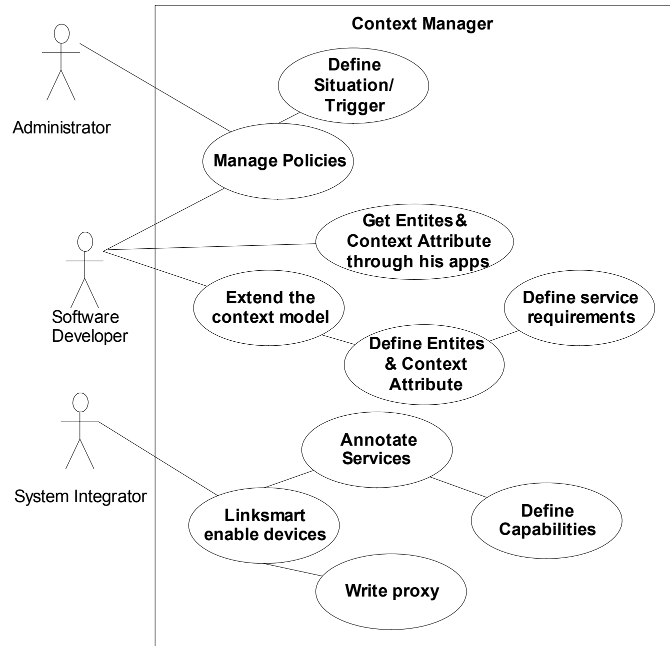

Figure 11. Developer Use Cases.

We identified several user roles that use the Context Manager in ebbits: the first user is the system integrator that integrates the physical devices of the existing systems as well as new devices into ebbits; software developers who in charge of developing applications that monitor and react to the situations and context of the entities; and the administrator that manages policies over time.

In order to implement a context aware application on top of ebbits, software developers must have the ontology of the devices and their services in the ontology manager. The ontology describes the capabilities of the services. Such capabilities will be then used by the Context Manager to find the suitable sensors that could deliver the value of the context attributes.

Software developers could then easily model his context model in the ontology using ontology tool such as Protégé. The context model is done in the ontology to give the flexibility to change the model and the policy when the business logic changes or new applications that need the same sensing resources are to be introduced. This approach allows modifications that often are needed over time, done without having to re-build the Context Manager.

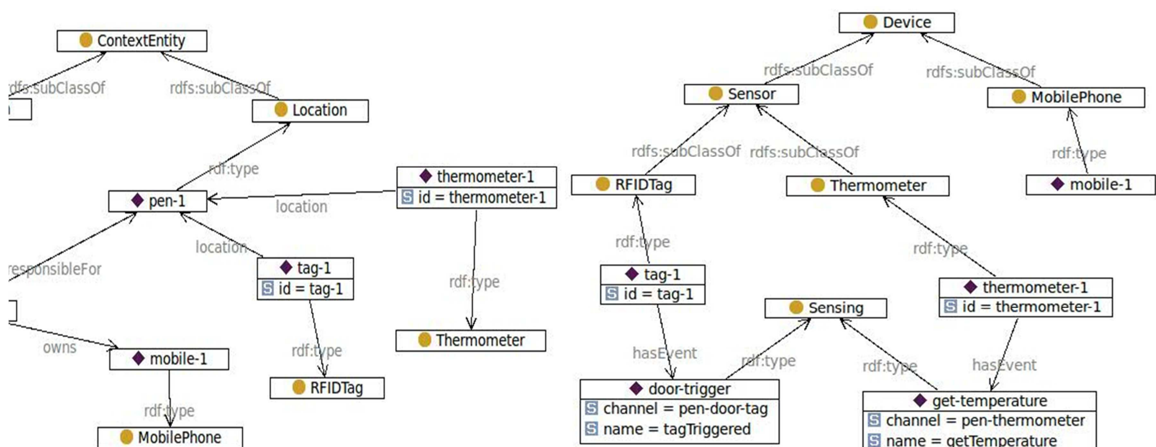

Figure 12. The use-case device ontology and application ontology.

As the software developer defines the entities and the context attributes in their application, they also define the service requirement for their context attribute. For instance, an entity "pen" XYZ has a context attribute "temperature" which in turn has a service requirement of any thermometer that measures the corresponding pen. As the system integrator has previously annotated the thermometers with their capabilities, the ontology manager is able to tell the Context Manager, the most suitable sensors for measuring the thermometer in pen "XYZ" (e.g., as depicted in Figure 12). The unambiguous vocabularies are still to be defined in WP4 where semantic infrastructure for ebbits is researched.

Once the device developer has done modeling the context model in ontology, he must define situations that trigger action of his application in a policy.

```
IF Data.getValue(
            model[entity-retrieve][entity.id],
            model[subentity-retrieve][subentity.channel],
            ) > 25 THEN
            RAISE EVENT {
              channel: 'temperature-too-hot', location: e[location]
            }

ON EVENT e[channel] = 'temperature-too-hot'
       EVALUATE
              model = Ontology.getData(
                            entity-match: {
                                   entity.type: MaintainanceCrew,
                                   entity.responsibleFor: e[location],
                                   entity.owns: subentity
                            },
                            subentity-match: {
                                   subentity.type: MobilePhone
                                   subentity.hasService.type: SMSService
                            },
                            entity-retrieve: {},
                            subentity-retrieve: {subentity.id}
                     );

       IF isDefined(model[subentity-retrieve][subentity.id]) THEN
              ACTION SMSService.sendSMS({
                            deviceId: model[subentity-retrieve][subentity.id],
                            sms: 'Someting devilish happends at e[location]'
                     });
```

Figure 13. Example of a policy with a situation "temperature-too-hot"

Figure 13 shows a simple policy that detects situation "temperature-too-hot". When the policy is triggered, The application having the policy finds the mobile applications that belongs to a maintenance crew on duty, and it notifies him. More details how the ontologies are constructed can be found in the D4.5 Analysis and design of semantic interoperability mechanisms.

6. Network Communication

The network communication is an essential part that allows the distributed nature of the Context Manager exchange and synchronizes the context attributes. The original network manager does not provide a scalable solution that could be effectively used by the Context Manager since it could only be deployed in resourceful devices. Meanwhile the Context Manager is intended to monitor situations using devices equipped with sensors and actuators. This includes resource constrained devices such as mobile devices and wireless sensor network.

6.1 The original Network Manager internal structure

The original Network Manager comprised the following sub-managers (depicted in Figure 14):

- The Routing Manager manages the network interface and protocol inside Linksmart. Up to now, the routing manager decides if the messages are to be sent via Bluetooth or JXTA.
- The Backbone Manager manages and encapsulates specific backbone implementation such as Bluetooth or JXTA.
- The Identity manager generates HIDs that encapsulate specific network addressing scheme.
- Session manager keeps track of the sessions among HIDs.
- Time Manager synchronizes time with a Network Time Server

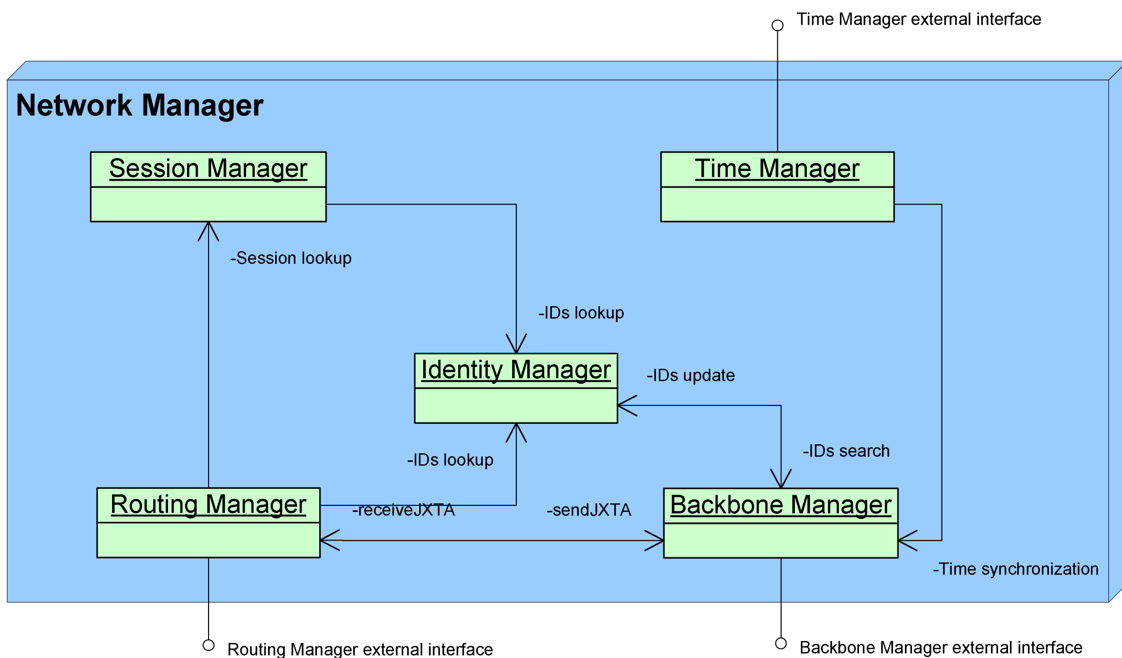

Figure 14. The Original Network Manager Packages

The disadvantage of the original network manager is that the backbone manager is actually a wrapper for JXTA. This makes it closely coupled to JXTA, a protocol which is not suitable for a resource constrain devices.

Addressing ebbits requirements and the recommendation of the reviewers that ebbits should minimize the use of central nodes to avoid single point of failure, we propose a new architecture that decouples the implementation of the network manager from heavy communication technology such as JXTA. The new architecture introduces interfaces for each package that should be implemented by specific technology as plug-ins. It will allow a new network managers being implemented specifically for resource constrain devices as depicted in Figure 14.

Lightweight network managers will only need to implement network protocols that are suitable for the device and the underlying platform (e.g., android mobile phones support UPnP protocol). More

powerful devices in the ebbits network, that could host several backbones, will route the messages for other devices. This allows communication among different backbones to be mediated by the resourceful nodes, and when these resourceful nodes fail, communication among the homogenous backbones could still take place.

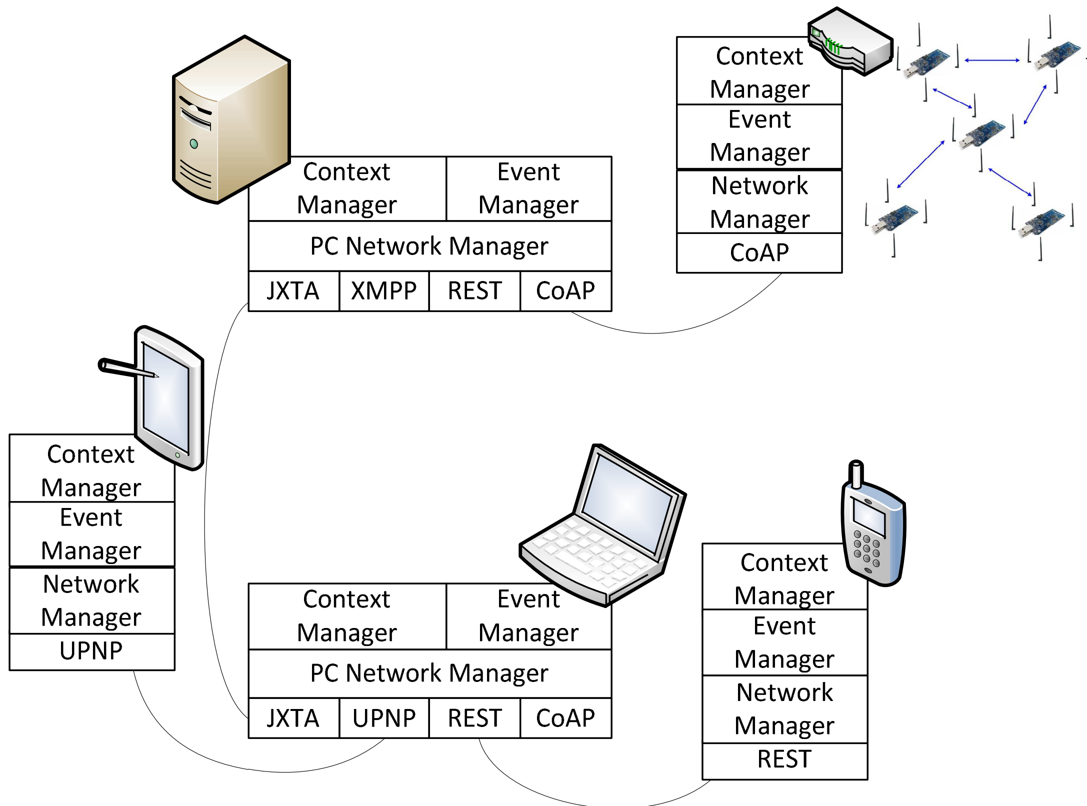

Figure 15. Multiple Backbones Communication

The internal architecture of the network manager will be discussed in detail in WP8 deliverables. In this deliverable, we only present the overview as background information how the Context Managers will communicate among each other.

The network manager mainly has several packages that expose interfaces.

Backbone

Backbone bundles are responsible for handling the physical channel over which messages are sent. The backbone has to handle the synch or asynch nature of the communication, as for the rest of the application, backbone calls should always be handled asynchronously. Broadcasting and multicasting is also handled internally and with a best effort approach. The Backbone holds a routing table to pair HIDs to physical Ids. The physical Ids should never leave the Backbone only for presentation purposes. The Backbone must take care for itself how to get the HIDs from the sent packages. The backbone as depicted in the Figure 15 could be implemented with different technology for instance JXTA is suitable for supporting internet connectivity meanwhile COAP is more suitable for supporting resource constrain devices.

BackboneRouter

BackboneRouter holds references and manages several Backbones. The BackboneRouter does a mapping from HID to Backbone. Also, it offers methods to register and unregister routes automatically, by storing the HID to the Backbone where the message came from. The backbone router is responsible to choose the best backbone for sending the message.

NetworkManagerCore

The Core implements the interfaces of the NetworkManager. It is the connection bundle between the different modules as it forwards requests coming to it to the destination. It has a ConnectionManager which holds references to Connections which process data for sending over the network. This includes security operations, compression, encoding, etc. The NetworkManagerCore should be kept as simple as possible and logic should be put into external modules.

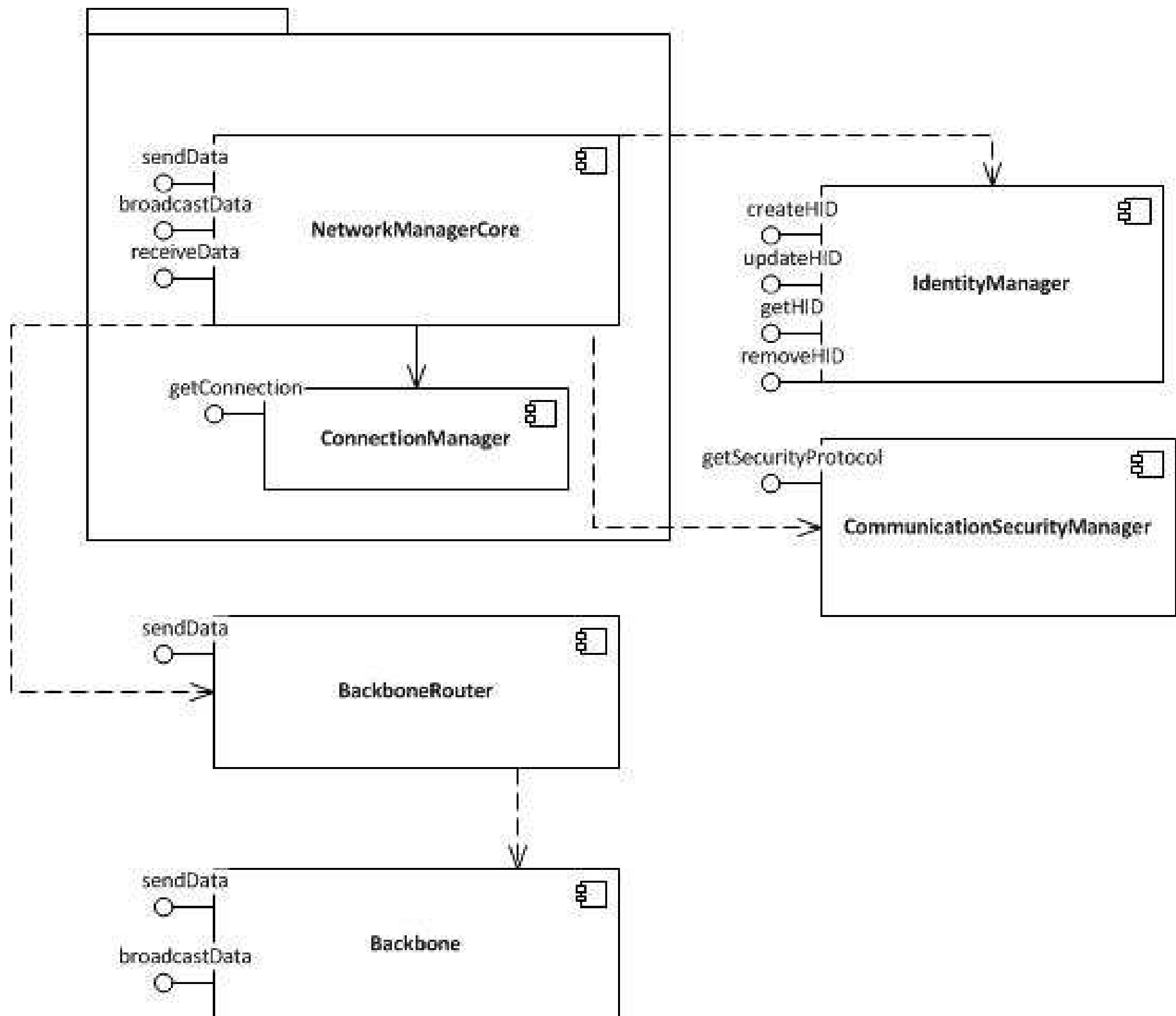

Figure 16. Network Manager Interfaces

Messages

A message object is used for representing a packet received over the network. It holds all information needed to process it as sender, receiver, data and any additional parameters. Messages are created by a Connection object and turned back into serialized data to send by a Connection.

CommunicationSecurityManager

It provides implementations of SecurityProtocol objects which can be used for securing the communication. This is done by assigning a SecurityProtocol object to a Connection object. The CommunicationSecurityManager holds references to required bundles like CryptoManager and TrustManager, and provides a configuration interface. A SecurityProtocol object provides 3 essential methods. The processMessage method is used to process the handshake messages. When the SecurityProtocol is initialized the protect and unprotect methods can be used.

HIDManager

This component is responsible for generating HIDs and to pair HIDs to identities. An identity is independent from the Backbone which uses it. This means that the HIDManagershould only see the

application level HID identity and nothing else. An HID in general is an address and a set of attributes. What these attributes are is a responsibility of the HIDManager implementation.

7. Summary and Conclusion

In this deliverable we presented a refined architecture of the context awareness that we introduced in the D5.2.1 Architecture for intelligence integration. We defined the context awareness in ebbits to achieve a common ground for the activities running in work package 5. Furthermore, in order to ensure the scalability of monitoring entities for manufacturing and traceability use cases, we proposed a distributed architecture for the Context Manager that balances out the workload of the monitoring tasks. We also proposed an architecture that can be easily configured to handle the business logic that changes overtime, by allowing the business logic to be defined in policies. The architecture takes advantage of ontology modeling that ease the application developer to reason upon the taxonomy of the entities, context attributes and devices that correspond to them.

8. References

Abowd, G., A. Dey, et al. (1999). Towards a better understanding of context and context-awareness, Springer.
Brown, P. J., J. D. Bovey, et al. (1997). "Context-aware applications: from the laboratory to the marketplace." Personal Communications, IEEE **4**(5): 58-64.
Computing, A. (2006). "An architectural blueprint for autonomic computing." IBM White Paper.
Dey, A. K. (1998). Context-aware computing: The CyberDesk project.
Ryan, N. S., J. Pascoe, et al. (1998). Enhanced reality fieldwork: the context-aware archaeological assistant, Tempus Reparatum.
Schilit, B., N. Adams, et al. (1994). Context-aware computing applications, IEEE.
Schilit, B. N. and M. M. Theimer (1994). "Disseminating active map information to mobile hosts." Network, IEEE **8**(5): 22-32.
Winograd, T. (2001). "Architectures for context." Human-Computer Interaction **16**(2): 401-419.