



BEMO-COFRA

Brazil-Europe MOnitoring and COntrol FRAMeworks

(Project No. 288133)

D8.4 Training Package

Published by the BEMO-COFRA Consortium

Dissemination Level: Public



European Commission
Information Society and Media

Project co-funded by the European Commission within the 7th Framework Programme
and
Conselho Nacional de Desenvolvimento Científico e Tecnológico
Objective ICT-2011-EU-Brazil

Document control page

Document file: D8.4 Training Package
Document version: 1.0
Document owner: Trine F. Sørensen (IN-JET)

Work package: WP8 – Dissemination and Exploitation
Task: T8.4 – Training
Deliverable type: R

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	Trine F. Sørensen	2011-11-14	ToC
0.2	Trine F. Sørensen	2011-11-15	Content
0.3	Trine F. Sørensen, Ferry Pramudianto	2011-11-17	Exe Sum and introduction added. Edited content.
0.4	Markus Taumberger	2011-11-29	POBICOS chapter added
0.5	Peter Rosengren, Peeter Kool	2011-11-20	Added chapter on extending ontology model
0.6	Peter Rosengren	2011-11-22	Added chapter on extending ontology model
0.7	Peter Rosengren, Peeter Kool	2011-11-23	Added chapter on how to create Basic LinkSmart Application
0.8	Peter Rosengren, Peeter Kool	2011-11-23	Added chapter on how to create Advanced LinkSmart Application
0.9	Ferry Pramudianto, Trine F. Sørensen	2011-11-29	Picture and content edited following Claudio Pastrone's comments. Very minor corrections following review comments
1.0	Ferry Pramudianto, Trine F. Sørensen	2011-11-30	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Claudio Pastrone	2011-11-26	Comments and suggestions to content and language. Approved with comments.

Legal Notice

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

The Members of the BEMO-COFRA 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 BEMO-COFRA 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.

Index:

1. Executive summary	6
2. Introduction	7
2.1 Purpose and context of this deliverable	7
2.2 Background	7
2.3 Scope	7
3. LinkSmart Middleware Overview	9
3.1 Wireless Devices and Networks	9
3.2 Trust and Security	9
4. LinkSmart Architecture.....	10
4.1 Device Classification	11
4.2 Applications and Devices	12
4.3 Applications	12
4.4 LinkSmart Devices / Device Proxies	12
5. LinkSmart Technologies	13
5.1 Java.....	13
5.2 OSGI	13
5.3 .NET	13
5.4 Network	13
5.4.1 P2P / JXTA	13
5.4.2 SOAP	14
5.5 Security Technologies	14
5.5.1 XACML.....	14
5.5.2 XML Security and Web Service Security	14
6. LinkSmart Components	15
6.1 Network Manager	15
6.1.1 HIDs and their management in Network Manager	16
6.1.2 Other Network Manager Functionalities.....	16
6.2 Event Manager	17
6.3 Trust Manager.....	17
6.4 Crypto Manager.....	18
6.5 Access Control Policy Framework	18
6.5.1 XACML.....	20
6.6 Obligation Framework	21
6.7 Discovery Manager	22
6.8 Security Library.....	23
6.9 Security Design	23
7. Installing the LinkSmart Middleware.....	25
7.1 Prerequisites	25
7.2 Required Bundles	25
7.2.1 Core required bundles	25
7.3 Crypto Manager Setup.....	26
7.4 LinkSmart bundles.....	26
7.5 VM Arguments	28
7.6 Running the framework	29

8. Software Development Kit.....	32
8.1 LinkSmart Commons.....	32
8.1.1 LinkSmart Middleware API	32
8.1.2 LinkSmart Middleware Clients.....	32
8.1.3 LinkSmart Configurator.....	33
8.2 Network Manager	35
8.2.1 LinkSmart Definition of Device To Device Communication.....	35
8.2.2 The Peer-to-Peer Network Architecture.....	36
8.2.3 Purpose	36
8.2.4 Main Functionalities	37
8.2.5 LinkSmart Web Service Provider	37
8.2.6 Crypto HIDs.....	39
8.3 Device Application Catalogue	42
8.3.1 The Graphical Browser.....	42
8.4 Discovery Manager (Framework)	49
8.4.1 Physical Discovery.....	50
8.4.1 Network discovery based on UPnP.....	50
8.4.2 External Discovery	51
8.4.3 Semantic Discovery.....	51
8.5 Event Manager	53
8.6 Access Control Policy Framework	55
8.6.1 Policy Enforcement Point	55
8.6.2 Policy Decision Point.....	56
8.6.3 Policy Administration Point	56
8.6.4 Policy Information Point	57
9. Creating a Basic Linksmart Application.....	58
9.1 Creating a Linksmart application from a template	58
9.2 Initiating the Network Manager	59
9.3 Initiating the Application Device Manager	60
9.4 Working with devices	60
9.5 Applications Bindings	61
10. Creating an Advanced LinkSmart Application	64
10.1 Initiate Application	64
10.2 Searching and finding for devices.....	65
10.3 Invoking Device Services	66
10.4 Understanding the LinkSmart Device XML.....	67
10.5 Extending the LinkSmart Device XML	70
11. Device Developer Kit .net	72
11.1 Using Intel Service Author for UPnP Technologies	72
11.2 Using Linksmart .Net DDK tool	74
12. Extending the LinkSmart Ontology model.....	81
13. POBICOS Overview	84

13.1	System inspection protocol	84
13.1.1	General Architecture	84
13.1.2	Services	85
13.1.3	Communication.....	85
14.	Glossary	86
	References	89

1. Executive summary

The BEMO-COFRA project will be using the middleware LinkSmart which was developed in the Hydra IP project and it is therefore necessary to provide BEMO-COFRA software developers (not already familiar with LinkSmart) with a thorough description and tutorial of the considered middleware. This deliverable aims to provide the required knowledge by firstly presenting high level architectural and subsystem level descriptions of LinkSmart and secondly presenting the lower level implementation, functionality and how-to-configure details useful to build applications based on the LinkSmart Middleware platform.

This deliverable will therefore supplement the internal training workshops planned for the BEMO-COFRA project: a first training workshop focusing particularly on LinkSmart will be held during the first week of December 2011 and a second workshop is planned for February 2012.

The LinkSmart middleware is an intelligent software layer that is placed in between the operating system and the application layer. The first part of the present deliverable provides the reader with an understanding the software architecture of LinkSmart, applications and devices. The main building blocks of the LinkSmart middleware are the different "LinkSmart Managers". A LinkSmart Manager encapsulates a set of operations and data that realise a specific functionality. LinkSmart Managers include: Network Manager, Event Manager, Trust Manager, Crypto Manager, and Discovery Manager. These managers, their functionalities, purpose, and their implementation are described in further detail.

The LinkSmart middleware offers a large collection of reusable core software components to experienced developers. Based on these software components, programming abstractions allow for programming with well-known concepts from the field of pervasive and ambient computing through reducing the details of the underlying implementation. LinkSmart applications are built by programming networked ambient intelligent devices. Devices are made programmable by the LinkSmart middleware thru proxies as well as by embedded components. LinkSmart uses different technologies such as Java and OSGI, also integrating network functionalities (e.g. JXTA), and security related technologies (e.g. XACML).

The second part of this deliverable concerns the installation of LinkSmart; what is required and how to install and set-up the different managers. It is complete with illustrations (screen shots). In addition, tutorials on the Software Development Kit (SDK) and the Device Developer Kit (DDK) are presented. The SDK tutorial concentrates on the software interfaces of each LinkSmart component / manager / tool, and how to use them. The DDK tutorial focuses on the process of LinkSmart-enabling a device and is aimed at developers who want to use network embedded devices and build applications on top of the layer that communicates with the same devices. Moreover, how to creating LinkSmart Applications is described.

The project POBICOS, targeted at computing environments which feature collections of objects, equipped with sense-compute-actuate embedded nodes, which differ in their sensor, actuator and computing resource, is presented to give the reader a general overview. A POBICOS training session during the first training workshop in December 2011 entitled "Protocol for monitoring and control of WSANs in unreliable networking environments" will allow the trainees to gain from the results and experiences of POBICOS.

2. Introduction

2.1 Purpose and context of this deliverable

This deliverable D8.4 Training Package is primarily aimed at the software developers in the BEMO-COFRA project. It presents an overview and description of LinkSmart as well as different useful tutorials which will allow BEMO-COFRA's software developers to use, configure and implement the LinkSmart middleware to fit with the objectives of BEMO-COFRA. The LinkSmart middleware was developed in the Hydra IP project and it proved to be a helpful tool for application and device developers to interconnect seamlessly heterogeneous device.

The material in this deliverable will be used to complement the two internal training workshops that will be organised within the first 6 months of the project. The first internal training workshop will take place during the first week of December 2011 and will allow software developers to gain a hands-on experience with the LinkSmart middleware. In addition, knowledge and experiences from the ebbits project and the POBICOS project will be shared in the workshop. Project partners who participated in these projects and who have experience in using LinkSmart in other projects will be providing the training to those partners who are new to LinkSmart. The current deliverable will be available before the first training workshop and will thus allow the trainees to familiarise themselves with LinkSmart before the workshop, just as they can return to the descriptions and tutorials in this deliverable throughout the project as necessary.

2.2 Background

The BEMO-COFRA project will develop an innovative distributed framework allowing networked monitoring and control of large-scale complex systems. Heterogeneous smart objects, legacy devices and sub-systems will be integrated, cooperating to support holistic management and to achieve overall systems' efficiency with respect to energy and raw materials. The BEMO-COFRA project will address both technological aspects and user needs to promote a wider adoption of large-scale networked monitoring and control solutions.

The overall system of the BEMO-COFRA architecture will represent a comprehensive, distributed framework comprising a middleware layer, large WSANs, PLC and SCADA systems, and not least computers and powerful devices being deployed in manufacturing plants. A platform for dealing with WSANs will be developed to support monitoring and control operations in manufacturing environment. To provide IP-based services for WSANs the existing LinkSmart middleware (EU LINKSMART-Project) is going to be utilized, i.e. LinkSmart proxies for WSANs will be developed to offer WSANs based functionality to the outside world. In addition, LinkSmart proxies for PLC and SCADA systems will be defined.

BEMO-COFRA reuses the results of the well-reputed Hydra IP and Pobicos STREP and the recently started ebbits IP featuring a Service Oriented Architecture (SOA) and a middleware able to expose smart objects, legacy devices and sub-systems' capabilities by means of web services.

2.3 Scope

Following the Executive Summary in Chapter One and the Introduction in Chapter Two, an overview of the LinkSmart middleware is given in Chapter Three.

Chapters Four to Six provide a high level architectural and sub-system level description of the LinkSmart Middleware.

More specifically, Chapter Four focuses on the LinkSmart Architecture by describing how it consists of "LinkSmart Managers", each encapsulating a set of operations and data that realise a specific functionality. This chapter also deals with devices and high level applications.

Chapter Five summarises the technologies adopted in LinkSmart to realise its goals and Chapter Six gives an overview of the LinkSmart components, their functionality, purpose, and which technologies are used and why for their implementation.

Chapters Seven to Twelve build on Chapters Four to Six by going into more detail and present the lower level implementation, functionalities and how-to-configure details as used to build applications based on the LinkSmart Middleware Platform.

More specifically, Chapter Seven concerns the installation of the LinkSmart middleware, providing a detailed tutorial and screen shots to illustrate.

Chapter Eight introduces the Software Development Kit (SDK) and describes the software interfaces and gives a complete tutorial on how to use the different LinkSmart components, managers and tools.

Chapter Nine describes how to create basic LinkSmart Applications while Chapter Ten moves on to deal with creating advanced LinkSmart Applications.

Chapter Eleven introduces the Device Developer Kit (DDK).

Chapter Twelve describes how to extend the default LinkSmart Ontology model.

Chapter Thirteen gives an overview of the POBICOS project.

Finally, Chapter Thirteen provides a glossary of important terms used in the LinkSmart "language".

3. LinkSmart Middleware Overview

The LinkSmart middleware provides a development platform for Internet of Things applications (IoT). It can be seen as an intelligent software layer that is placed in between the operating system and the applications. LinkSmart contains several software components or managers, which have been carefully designed to handle the various tasks needed to support the cost-effective development of intelligent applications for networked embedded systems.

LinkSmart can be incorporated in both new and existing networks of distributed devices, which operate with limited resources in terms of computing power, energy and memory usage. LinkSmart allows developers to incorporate heterogeneous physical devices into their Internet of Things applications by providing easy-to-use web service interfaces for controlling any type of physical device irrespective of its network interface technology. Thus the device becomes part of the Internet of Things and can be accessed and controlled using standard IoT technology.

LinkSmart is based on a semantic Model Driven Architecture for easy programming and incorporates solutions for device and service discovery, peer-to-peer communication and diagnostics. LinkSmart-enabled devices also offer secure and trustworthy communication through distributed security and social trust middleware components.

The Software Development Kit (SDK) allows developers to define innovative Internet of Things applications with embedded ambient intelligence computing using the middleware, while the Device Development Kit (DDK) allows device developers to enable their devices to participate in a Internet of Things network.

3.1 Wireless Devices and Networks

Middleware for wireless objects should be able to hide the complexity of the underlying infrastructure while providing open interfaces to third parties for application development and ease of use for end-users. In LinkSmart, the communication layer is not part of the middleware, which is transparent to it. LinkSmart provides:

- Dynamic resource discovery and management
- Tools for advanced control making the solutions reactive to the physical world
- Objects definition and querying for data using semantic technologies
- Unique network adapters in order to avoid specific networking technology.

The LinkSmart middleware leverages on available network technologies to change user-environments and discover available, location based information resources.

3.2 Trust and Security

In order to solve the rapidly growing challenges of privacy, identity theft and trust, the LinkSmart solution adopts two main approaches in tandem:

The first approach targets the secure design and prototyping of mobile ID management for context-aware services relying on heterogeneous mobile and wireless service and networks. Security goals such as confidentiality, authenticity, and non-repudiation are addressed by the particularly trustworthy design and implementation of open-source and web service-based mechanisms, enriched by ontologies and semantic resolution techniques.

The second approach reflects a coherent no-trust context-locked separation through virtualisation of devices and people.

LinkSmart has refined a security and trust model. Multiple different virtualisation models with different security models are assumed and mapping of interfaces will be carried out and considered in the overall architectural model.

4. LinkSmart Architecture

The software architecture described is an abstract representation of the LinkSmart middleware. The architecture is a partitioning scheme, describing components and their interaction with each other. Figure 1 gives a structural overview of the LinkSmart middleware and explains how the elements are logically grouped together. "LinkSmart managers" constitute the major building blocks that make up the LinkSmart middleware. A LinkSmart manager encapsulates a set of operations and data that realise a specific functionality.

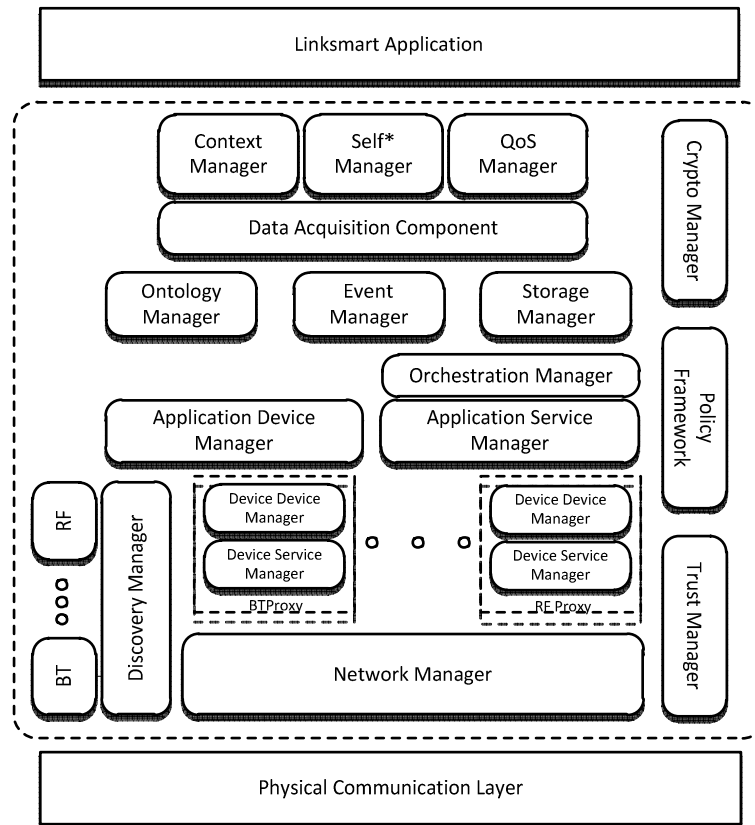


Figure 1: Structural Overview on the LinkSmart Managers

The LinkSmart middleware offers a large collection of reusable core software components to experienced developers. Based on these software components, programming abstractions allow for programming with well-known concepts from the field of pervasive and ambient computing through reducing the details of the underlying implementation. From the bottom to the top of Figure 1 the LinkSmart middleware provides more and more programming abstraction and functionalities for the developers:

- The Network Manager implements Web Services over JXTA as the Peer-to-Peer model for device-to-device communication.
- The Device and Device Service Manager in a bundle implement a service interface for a physical device, handle several service requests and manage the responses.
- The Application Device Manager provide programming interfaces and information for the different devices to the software developers.
- The Discovery Manager automates and facilitates the discovery of devices in a LinkSmart network.
- The Ontology Manager is used by the Application Device Manager to get meta-information about devices and also semantically resolves what type of device has been discovered.
- The Event Manager provides a topic based publish-subscribe service in LinkSmart.

- The Crypto, Trust and Policy Manager take care for cryptographic operations, the evaluation of trust in different tokens and the enforcement of access control security policies.
- The Data Acquisition Component retrieves the data delivered by the LinkSmart devices and sensors (via push or pull mode).
- The Quality-of-Service (QoS) Manager in LinkSmart is a component that accesses and particularly processes all non-functional properties-data for services/components, devices, and networks.
- The Self* Manager provides support for automating application management.
- The Context Manager allows for the definition of an application-dependent context model.
- The Storage Architecture realises the persistent storage of information in LinkSmart middleware.

4.1 Device Classification

The LinkSmart middleware is designed to handle all types of devices, with varying capabilities. The figure below, demonstrates how devices are classified into different categories, based on what technologies they can support, which determines how the device can become "IoT-enabled".

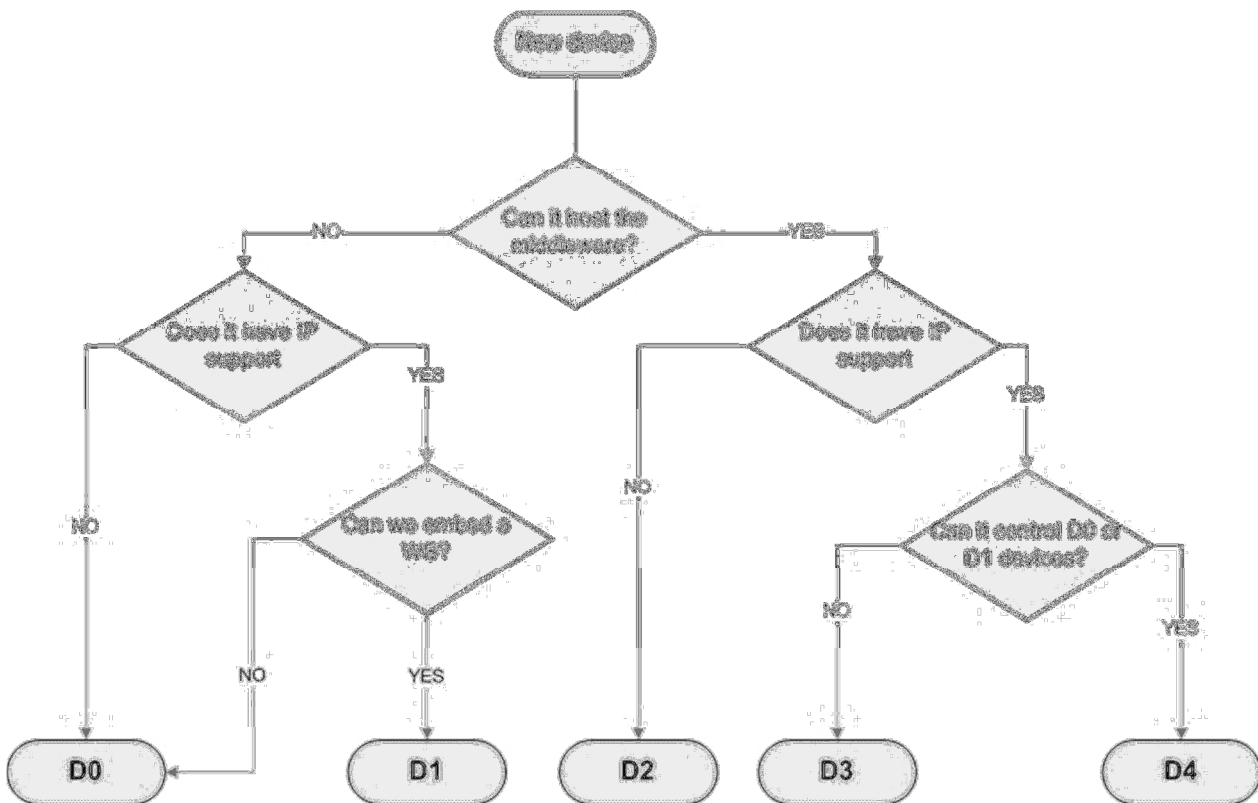


Figure 2: Flowchart for the device classification process

The significance of the D0--D4 categories is that devices within each category are handled in the same way in relation to the LinkSmart middleware and the enabling process. For further details on LinkSmart terminologies please refer to the Glossary section in chapter 10.

Category-D0 devices are used with a proxy, that is, they can only be reached through a proxy-service residing on a Category-D4 device. The proxy service must implement the communication with the D0 device.

Category-D1 devices can host a web service, and the intention is that such embedded web services are created with the Limbo tool.

Category-D2--D4 devices are said to be LinkSmart enabled. LinkSmart enabled devices host the network manager and all other managers needed for that device, but differ in their networking capabilities.

4.2 Applications and Devices

LinkSmart applications are built by controlling and using networked ambient intelligent devices. Devices are made programmable by the LinkSmart middleware thru proxies as well as by embedded components. Whatever the method, it is transparent to application developers, as they access all devices based on a pure service and event based programming model. The API of this programming model is manifested by the LinkSmart SDK, for application development.

4.3 Applications

An application in LinkSmart is built around a DAC (a Device Application Catalogue) which functions as a kind of device registry, holding references to the set of devices which have been discovered and are available to the application.

LinkSmart provides different levels of configuration, depending on the application requirements. A minimal configuration for a LinkSmart application consists of an Application Device Manager, and a Network Manager running on a LinkSmart gateway device (aka D4 device), to which one or more other devices are connected or can be connected.

The minimal configuration can be extended by an Ontology Manager, which will add semantic discovery capability to the system. Additional functionality for context management and security can be obtained by the corresponding managers.

4.4 LinkSmart Devices / Device Proxies

A basic idea in LinkSmart is to differentiate between the physical devices and the application's view of the device, in terms of so called IoT Devices (Internet of Things). A IoT Device is the software representation of a physical device. This representation is either implemented by a proxy running on a gateway device, or, by embedded LinkSmart managers on the actual device. A IoT Device is said to IoT-enable a physical device.

There are five categories of Web Services generated for a IoT Device,

- **A Generic IoT web service**, exposing metadata and management functions common to all LinkSmart Devices.
- **An Energy web service**, providing a set of functions for the monitoring and control of energy consumption of devices.
- **A Memory Service** which allows logging and storing of device internal data such as state variables and energy consumption data.
- **A Location Service** which can be used to query the device about its location and position.
- **A device type specific web service**, exposing the device type specific functions.

5. LinkSmart Technologies

This section summarises the technologies that LinkSmart utilises.

5.1 Java

Java is the core programming language of the LinkSmart Middleware, as it offers “write once, run anywhere” benefits. Using Java, the Network Manager can run on any Java enabled operative system. The Java J2SE can be run on almost any OS; therefore, LinkSmart can be deployed on these OS also. The problem of Java is the resources (memory and processor) that it consumes. It is not a big problem in powerful devices such as PC or Laptops, but in smaller devices, like PDAs or mobile phones it makes it almost impossible to run the LinkSmart middleware on them.

5.2 OSGI

Most of the core components in the LinkSmart Middleware are implemented as OSGi plug-ins [1] because OSGi offers a general-purpose, secure and managed Java framework that supports the deployment of extensible applications (bundles), bundle life cycle management, service oriented architecture and easy and quick deployment. Due to the use of OSGi in LinkSmart, deployment of all managers and proxies is simple and quick and its life cycle can be easily managed, even remotely. OSGIs modularity offers the developers a clean workspace where the managers can be dynamically deployed and un-deployed. Also, the service-oriented foundations are suited for the software architecture that LinkSmart was aiming for. LinkSmart also uses Remote-OSGi (R-OSGI) [2] which is an additional library that allows remote access to OSGi bundles. It uses less effort than using Web Services when component is based on OSGi and “full access” is intended. It is used mainly to connect the LinkSmart IDE to remote bundles for configuration and provides easy remote access to LinkSmart components.

5.3 .NET

Although most components of the middleware are implemented in Java, some important ones such as the Discovery Managers and the Application Device Manager are also implemented using the .NET framework. .NET provides good XML support and good development tools including cross process debugging. It has an Extendable XSLT processor and fast managers with small memory foot print. Furthermore, LINQ [3] is a set of extensions to the .NET Framework that extends C# with native language syntax for queries and provides class libraries to take advantage of these capabilities [4].

There is also an extensive set of Device Managers available for .net, for instance for medical devices, wireless sensor networks and home automation

5.4 Network

LinkSmart is based on various network technologies and concepts which are briefly described below.

5.4.1 P2P / JXTA

A P2P architecture was selected for carrying out the communications between LinkSmart components (Inside LinkSmart Communications). The main benefits of a P2P architecture versus an old-fashioned client-server one are its adaptability to very extensible networks, the responsibilities are distributed among peers; it provides high availability and fault tolerance and enables the full usages of the bandwidth.

Among all the available P2P technologies, JXTA was selected as the most suited for LinkSmart. The reasons that led to the selection of JXTA are:

- **Interoperability:** Enables communication between peers independently of network addressing and physical protocols.
- **Platform independence:** JXTA does not depend on the programming language, network transport protocols and deployment platforms, giving freedom of choice. Java SE and Java ME implementations have been selected for LinkSmart.

- **Ubiquity:** JXTA is designed to be deployed on any device, not just PCs.
- **Security:** for security means regarding authentication, authorisation, and integrity can be implemented based on JXTA. Attacks on the level of the protocol cannot be addressed, as that would require changing the JXTA protocol.
- **Community support:** JXTA is supported by a wide community of developers and the different specifications are fully documented.
- **Wide range of services:** Most of the P2P models studied have been designed exclusively for providing file-sharing services. Instead, in JXTA, thanks to its abstract architecture based on six protocols, it is possible and feasible to create a wide range of interoperable services and applications.

Using P2P, LinkSmart solves the problems of traditional client-server architectures, providing communication even when the different middleware instances are deployed behind firewalls and NATs. Therefore, managers, devices and services can intercommunicate and interoperate.

Also, WS has been combined with P2P communications through a SOAP Tunnelling in order to enable access to services and resources in a transparent way for developers. Using this SOAP tunnel, applications and devices can interoperate transparently, even when they are located in different networks, isolated from each other.

5.4.2 SOAP

Axis 1.4 for SOAP and Web Services provides a well working WSDL parser and generator and it can be easily transformed into an OSGi bundle. In addition, it is compatible with the Limbo tool (see chapter 6.3; Axis 2.0 is not fully compatible). Axis 1.4 provides interoperability for service provision and consumption. However, the main problem with Axis 1.4 is the resources it consumes.

5.5 Security Technologies

Description of used security technologies used in LinkSmart are listed and described below.

5.5.1 XACML

XACML [6] is an OASIS [7] standard that establishes an XML-represented language for access control policies, as well as access requests and responses. It defines a processing model, as discussed in the introduction to this section. It has standard extension points for defining new functionalities, making it rather flexible and extensible, and as such is perfect for integration into the LinkSmart middleware.

5.5.2 XML Security and Web Service Security

The Web Services Security specification (WS-Security) [8] enables Web Services developers to secure SOAP message exchanges by providing them a set of mechanisms. WS-Security enhances existing SOAP messaging which provides quality of protection. This is done by applying integrity and confidentiality to messages and authentication to SOAP messages. Furthermore, WS-Security also provides a general mechanism which helps in associating security tokens with messages. The good thing is that WS-Security doesn't require a specific type of security token. WSS is extensible and supports a variety of authentication and authorization mechanisms.

6. LinkSmart Components

This section summarises all LinkSmart components by providing an overview regarding the functionality and purpose of each component. Furthermore, the implementation of each component is described in detail focussing on the technologies used and the reasons for choosing these technologies. The value each component adds to the overall LinkSmart middleware is also discussed.

6.1 Network Manager

The Network Manager is responsible for network management. It is in charge of providing a transparent view of the nodes in the application and to route the data to the appropriate node (high-level view shown in Figure 3). Network communications are traced with a session mechanism. Furthermore, it takes advantage of the peer-to-peer architecture to create a LinkSmart overlay network and to allow the discovery of other IoT-enabled devices. It also handles the HIDs (LinkSmart IDs) of the nodes in the application. Finally, it is in charge of synchronizing the nodes in the network with referential time. In this way the Network Manager creates the "local" Internet of Things upon which the application can operate.

The Network Manager is the bottom layer of the LinkSmart middleware deployed in gateways and in LinkSmart-enabled devices. It is the entry and exit point of information of the LinkSmart middleware. There is only one Network Manager per device where the middleware is deployed. The Network Manager provides a web service interface which is the information entry point for the middleware. Data transferred between LinkSmart-enabled devices and gateways should always pass through the Network Manager.

Therefore, the main functions of the Network Manager are to provide a unique entry point for the network communications, support session mechanisms, create a peer-to-peer based overlay network, provide HID to nodes at application level, handle a list of the network members, allow discovery of other Network Managers and synchronise network with referential time.

The Network Manager is based on P2P and SOAP tunnelling combined with Web Services. The Network Manager builds an overlay P2P network as can be seen in Figure 4:

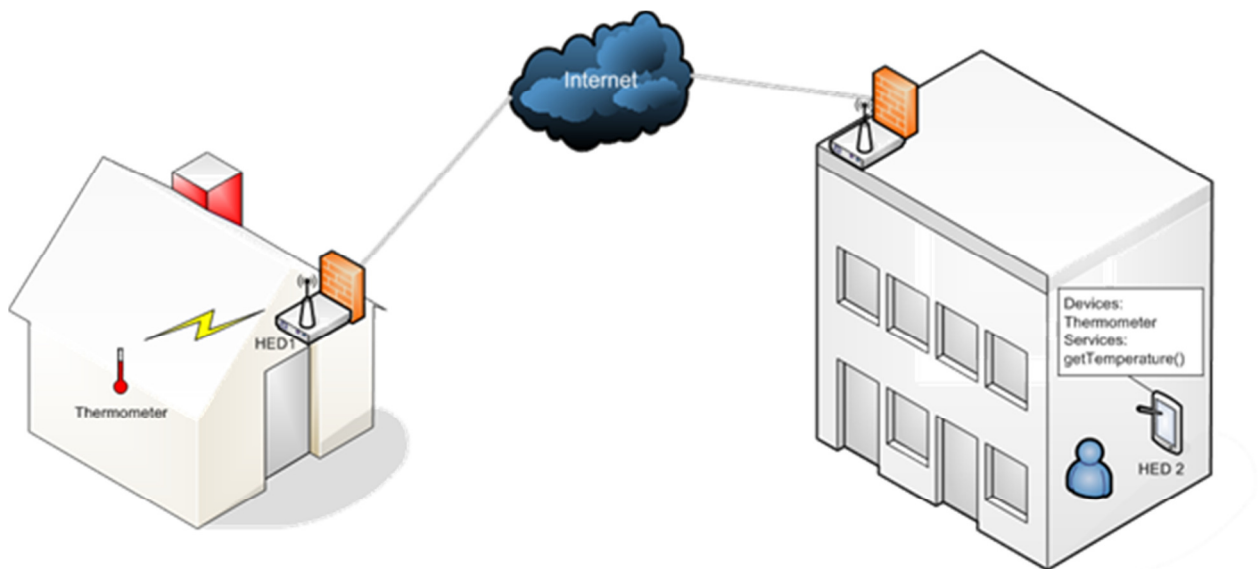


Figure 3: Network Manager overlay P2P network

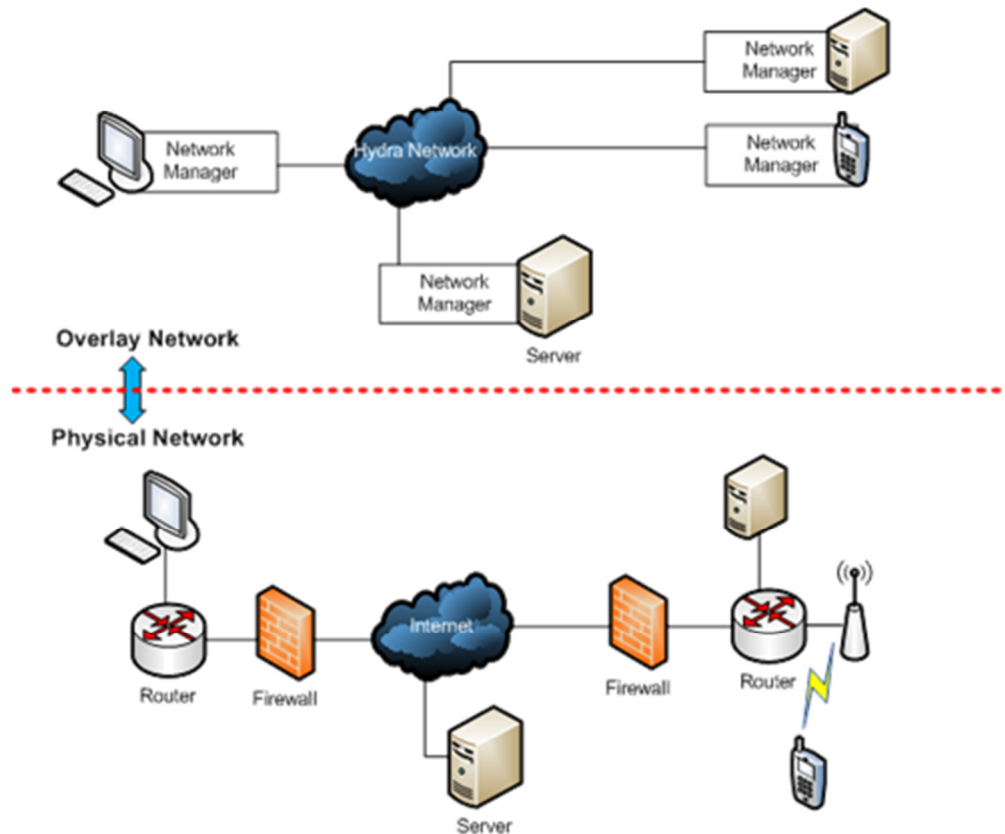


Figure 4: Basic LinkSmart network compared with physical network

6.1.1 HIDs and their management in Network Manager

The network manager allows services to be registered and made available through the LinkSmart Network with the use of LinkSmart IDs (HIDs). Each service from a device is registered in the local Network Manager and a unique context-dependant LinkSmart identifier (HID) is assigned to it. This does not depend on routing or network infrastructure, it can be kept even if the endpoint changes (mobility). The Network Manager provides the mechanisms for creating, modifying and deleting HIDs and ensures uniqueness. In order to register a service, the device (or proxy) provides a local endpoint of the service, an optional short description, and an optional Context or super context. The Network Manager maintains a data structure called IDTable with HID Info of services registered on it as well as HID Info (not the endpoint) of remote services registered on other Network Managers in the network.

6.1.2 Other Network Manager Functionalities

A PEP (Policy Enforcement Point) is integrated in the Network Manager in order to enforce policies associated with HIDs (or crypto HIDs). The enforcement is performed before the service invocation. SOAP Tunnelling is done over BT and UDP by implementation of new Network Manager transport mechanisms (BT and UDP) for SOAP message delivery for last-mile communication (gateway<->device). Protocol switching is made possible by changing the endpoint for a service (HID) in order to switch between communication technologies -> TCP / UDP / BT. The Network Manager is an OSGi R4 compliant bundle and a Network Manager Service is exposed both through WS and OSGi service. Furthermore, the Network Manager is UPnP AV compliant and incorporates Inside LinkSmart Security, multimedia content exchange, and a separate communication mechanism based on P2P for multimedia content.

6.2 Event Manager

The LinkSmart Event Manager provides 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). In general, publish/subscribe communication provides an application-level selected multicast that decouples senders and receivers in time, space, and data (i.e., sender and receivers do not need to up at the same time, do not need to know each other's network addresses and do not need to use the same data schema for events they send). The specific variant of publish/subscribe implemented is topic-based publish/subscribe where key/value pairs represent events. With this approach, any subscriber or publisher defines a topic simply by executing the "publish" or "subscribe" actions.

The Event Manager is used in any place where there is a potential many-to-many relationship between senders and receivers and where asynchronous communication is desirable. In particular, the Application Event Manager provides subscription support by allowing clients to subscribe to published events via a topic-based publish/subscribe scheme, publication support by allowing client to publish event on topics. The Event manager routes events to subscribed clients and assists in interfacing to the Network Manager (e.g., broadcast-, multicast-, or gossiping-based dissemination). The Event Core manages persistent subscriptions and publication to subscription matching etc. The Event Manager allows sending and receiving events through the LinkSmart P2P network.

Both the Event Manager and Resource Manager are programmed from scratch within LinkSmart, without the use of third party components beyond the standard Java libraries. All components make use of OSGi. The use of Java technology for all components was chosen because of the experience available in this language. Additionally, there is a wide selection of open source third party libraries in Java. OSGi was chosen because of the modularity possibilities it affords.

6.3 Trust Manager

The Trust Manager can be used to verify if a token offered by an entity is trustworthy. The term "trust" is used in a very technical meaning: "trust" indicates that the likelihood of an entity being the legitimate owner of a key inside a token. The methodology by which this likelihood is determined is called a "trust model". A trust model is an algorithm that takes a token as input and returns a trust value as output. Although this interface appears to be very simple, the process behind can be arbitrarily complex. The most common trust models like Public Key Infrastructure (PKI) or Web of Trust (WoT) may require sophisticated checking of key chains.

Instances of the Trust Manager are used in several places inside the LinkSmart architecture. It is a useful tool for an application developer to verify application layer certificates, but it is also an important component for middleware layer security. Thus a device developer has to integrate the Trust Manager in a device in order to support device certificate management for the secure middleware communication protocols. Alternatively, device software can interact with the Trust Manager as an external certificate verification service if the computational power of the device is insufficient to run a Trust Manager. Therefore the Trust Manager offers its functionality as a web service. In this case, the communication between device and Trust Manager has to be protected using the Core LinkSmart security mechanism or by other device specific means. The Trust Manager acts as an interface to arbitrary trust models, for example Public Key Infrastructures (PKI), Web of Trusts (WoT) or reputation-based trust models. Which specific trust model should be used depends strongly on the application context the device will be used in and cannot be predetermined.

Thus the Trust Manager implements not a specific trust model but rather provides an interface so that different models can be used without requiring any change on the program code. A device developer can use one of the preconfigured trust model, which are currently X.509 PKI, OpenPGP (Web of Trust), or a null-model (trust every certificate, for development). Furthermore developers are free to implement their own trust models and are able to add their model only by changing a configuration file. During runtime, the trust models can be selected and configured via a web service if the device developer wants to open this functionality.

6.4 Crypto Manager

The Crypto Manager is a stand-alone manager providing various cryptographic operations such as encryption, key management and handling of digital signatures. The main functionalities of the Crypto Manager are to protect messages, encrypt, sign, decrypt, verify signatures, wrap data in different message formats, receive protected messages, manage keys, generate public/private key pairs with or without persistent identifiers contained, generate symmetric keys, and store public keys (certificates) as well as symmetric keys.

The Crypto Manager is basically used in two ways. On the one hand, the Crypto Manager is automatically used by an internal part of the LinkSmart middleware – such as the security modules in the Network Manager. Therefore it provides cryptographic operations as an internal part of the middleware. For example, in case a device is about to join a LinkSmart security domain, a secret shared key has to be agreed between the device and the initiator of the domain (the “domain controller”). This functionality is located in the Crypto Manager, so it has to be attached to the Network Manager (either via web service calls or, better, directly as an OSGi bundle). This way, it is very easy to exchange sensitive cryptographic operations at a later time without touching any other components.

On the other hand, the Crypto Manager can be used by application developers via its web service interface as a stand-alone component for key management, and the creation of protected messages, thereby allowing building of secure storage solutions or communication protection at application layer.

The Crypto Manager provides its methods as a Web Service as well as an OSGi service. It is strongly recommended to use the Crypto Manager over a connection that is not prone to eavesdropping and modification. That is, recommended ways to use the Crypto Manager are either locally as an OSGi service or remotely over a channel that has been protected by Core- or Inside LinkSmart mechanisms, for example.

6.5 Access Control Policy Framework

The Access Control Policy Framework, in the LinkSmart Middleware, is an implementation of the XACML (eXtensible Access Control Mark-up Language) processing model [6]. It adds the functionality to be able to protect LinkSmart devices and services from unauthorised access, at the network level. The XACML Processing Model consists of four main components:

- Policy Enforcement Point (PEP)
 - Intercepts the call before it reaches the resource, and formulates an access request using the known information about the involved entities. Sends the request to the PDP. Enforces the decision returned, and may perform any obligations returned.
- Policy Decision Point (PDP)
 - Makes a decision on the access request, against the repository of policies it holds. The decision is returned to the PEP.
- Policy Information Point (PIP)
 - Resolves attributes referenced in a policy, that aren't featured in the access request.
- Policy Administration Point (PAP)
 - Provides the point of administration for authoring, publishing and managing XACML policies on a PDP.

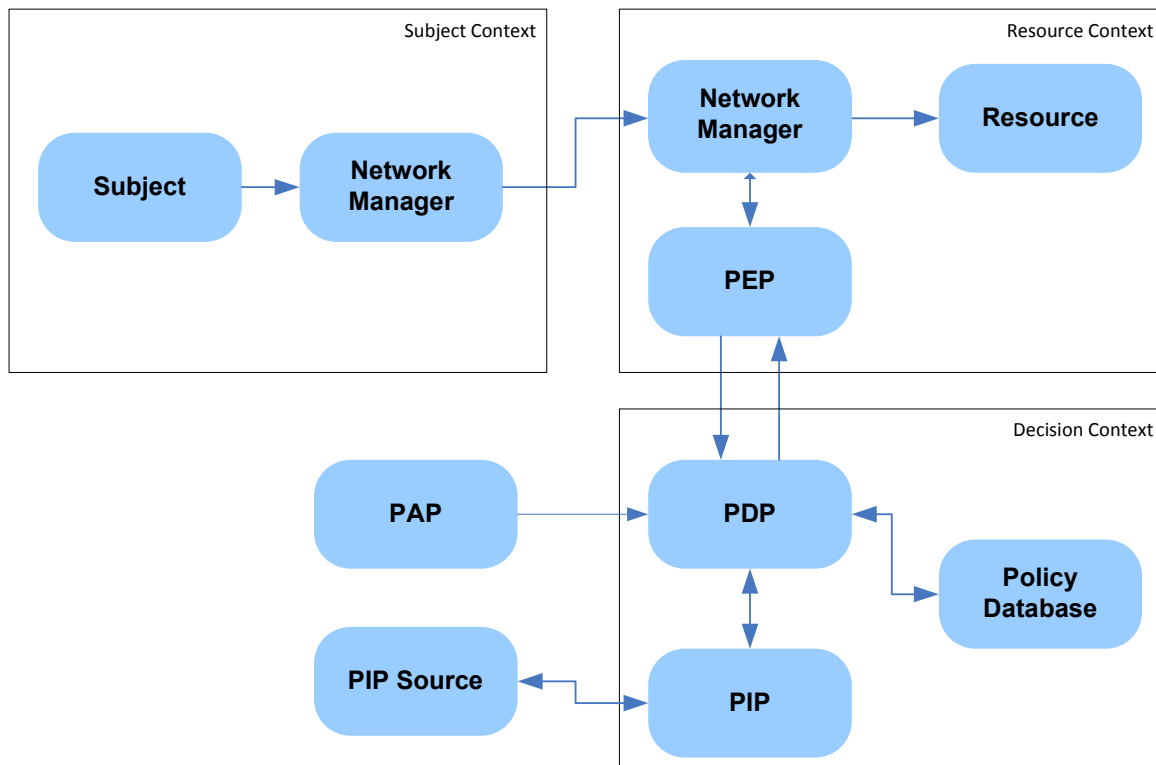


Figure 5: Access Control Policy Framework

Figure 5 above shows the architecture of the Access Control Policy Framework in LinkSmart. As described in 6.1, communication between Subject and Resource is routed through the Network Manager and Soap Tunnelling, with the Network Manager of the Resource Context being that which hosts the Resource service on the LinkSmart network, which can forward the request to the resource. Before doing so, it must request an access decision, passing all credentials of the request to the PEP.

This includes:

- Subject and Resource HIDs
- Method being called
- Session ID
- CryptoHID attributes of Subject and Resource

The PEP formulates this information into an XACML Request Context document, which is then sent to the PDP for a decision to be made. It should be noted that the PDP may or may not be local to the PEP. In the case where the PDP is remote, communication between the PEP and the PDP is again routed through the Network Manager.

Upon receiving the request, the PDP attempts to retrieve any relevant policies from the Policy Repository, which is a local XMLDB. The request is evaluated, which may involve retrieving additional information using a PIP, and a response returned with a decision and possibly a set of Obligations. The possible decisions returned are:

- *Permit* - Access granted
- *Deny* - Access denied
- *Indeterminate* - A decision could not be made, potentially due to an error in evaluation
- *NotApplicable* - No relevant Policy was found for a decision to be made

When the PEP receives the response, it handles any obligations returned, and returns the decision to the Network Manager, as the PEP itself does not have the power to technically enforce the decision itself, but relies upon the Network Manager to perform this role.

The final component of the Access Control Policy Framework is the PAP that provides the interface for authoring, publishing and managing XACML policies on a PDP. In LinkSmart, this is implemented with the Access Control Policy IDE - a component of the LinkSmart IDE.

The PDP uses an XML Database for storing policies published to it, as XACML policies are defined by XML anyway, and the ability to use XPath and XQuery queries to retrieve relevant policy(s) from the database make for a much more efficient system, as opposed to iterating through policies for matches. The database used is eXist XMLDB [9].

6.5.1 XACML

XACML is an OASIS [10] standard that establishes an XML-represented language for access control policies, as well as access requests and responses. It defines a processing model, as discussed in the introduction to this section. It has standard extension points for defining new functionalities, making it rather flexible and extensible, and as such is perfect for integration into the LinkSmart middleware.

The Access Control Policy Framework in LinkSmart is based upon the XACML implementation by Sun Microsystems [10], which is the commonly used XACML implementation in Java. It provides the core components of XACML, with all the core subset of XACML data types, functions and algorithms implemented, and is designed to be as easily extensible as the XACML standard itself. The Sun implementation is based on the XAML 1.x specification, with some components of XACML 2.0. XACML 3.0 is still in the drafting phase.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="ExamplePolicy"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:ordered-permit-overrides">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">MyResource
            </AttributeValue>
          <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="LinkSmart:policy:resource:pid"/>
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">doSomething
            </AttributeValue>
          <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule RuleId="OnlyAllowMySubject" Effect="Permit">
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
        <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="LinkSmart:policy:subject:pid"/>
      </Apply>
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">MySubject
        </AttributeValue>
    </Condition>
  </Rule>
  <Rule RuleId="CatchAllDeny" Effect="Deny"/>
</Policy>
```

Listing 1: A simple XACML Policy

The listing above shows a simple XACML 1.x policy, that protects the "doSomething" method of a resource with a PID of "MyResource", against access from anything but a caller with a PID "MySubject".

As mentioned in the introduction to this section, the Access Control Policy Framework has its main point-of-entry to the LinkSmart Middleware at the Network Manager level, advising the Network Manager on what action to take when it receives a call to one of its hosted services. The other main point of interaction is through the use of PIPs.

The LinkSmart PDP is designed to be extensible, to easily allow for new functionality to the PDP through adding additional PIP components, which includes the ability to resolve certain attributes, add additional functions that can be used in policies, add new data types, and so on.

PIPs, as with all components in the Access Control Policy Framework, are implemented as OSGi bundles, implementing an interface exposed by the PDP such that it is very easy to add new PIPs.

6.6 Obligation Framework

The Obligation Framework in LinkSmart, is a realisation of the event-condition-action (ECA) policies, using some advanced techniques like semantic reasoning, complex event processing (CEP) and enforcement monitors to increase the benefits of the policy framework. The purpose of these policies is manifold: Obligation policies shall help developers in setting up a LinkSmart-based system that automatically adapts its settings and implementations upon context changes (including, but not limited to security settings). Further, they shall complement the access-control policies by adding more expressiveness than simple permit/deny decisions. Last but not least, obligation policies can be defined by end-users (meaning: users of the LinkSmart application) in order to define the behaviour of their devices depending on different situations.

In XACML, it is possible to specify an obligation element for each policy and define whether the obligations contained within must be executed upon a deny or permit decision [8]. So, once the PDP has evaluated an access request, it will send the decision to the PEP, along with the set of obligations of the policy. The PEP must then wait for the PDP's decision, enforce the decision and execute the obligations. The purpose of these XACML obligations is mainly to add some additional actions to plain permit/deny decisions, such as logging all accesses or sending an email to the administrator if access to a certain resource is denied. That is, obligations in XACML can only be sent out as the result of an access request – spontaneously instructing a LinkSmart device to execute an action that has become necessary because of a change of the current situation is not possible, for example. Further, it cannot be defined who shall execute the obligation. In XACML, it is always the PEP that received the access request who has to enforce the obligation. Further, there is no way for the PDP to monitor whether the obligation was actually enforced, so the PEP has to be trusted by the PDP (which is of course always the case, in a typical XACML set-up).

With the Obligation Framework, an obligation can be triggered by any pattern of arbitrary events (e. g., sensor data, middleware events or user interactions). That is, in contrast to XACML, obligation policies are enforced asynchronously. The obligation is then sent to different obligation enforcement points (OEP) that can be located anywhere in the system (as long as they are listening to events from the LinkSmart Event Manager). Furthermore, it is possible to register enforcement monitors in the framework – components to monitor whether an obligation has been executed as requested or not.

So, to summarise, an obligation in XACML is different from obligation policies as supported by LinkSmart. Both have been designed for different purposes and they rather complement each other than they compete. Below, we will also show how both features can be used to integrate the obligation policy framework with the access control framework.

	XACML	LinkSmart
Triggered By	Only access requests	Arbitrary event patterns
Location of enforcement	PEP that intercepted the access request	Arbitrary OEPs
Enforcement monitoring	No	Possible
Granularity	At policy level	Arbitrary. At rule level

Table 1: Differences between XACML and LinkSmart obligations

6.7 Discovery Manager

The Discovery Manager is responsible for low level physical discovery of new (existing) IoT devices using native protocol. It creates a proxy and wraps physical devices with UPnP objects and publishes them onto the network. The Application Device Manager manages all knowledge, metadata and information regarding devices that have been discovered and are active in the LinkSmart network. The Discovery Manager assigns a device type to the device based on Device Ontology, returns service interface for the device, handles device virtualization (semantic devices) and semantic device aggregation, and manages the Device Application Catalogue (DAC;). The Discovery Manager also controls a set of several Discovery Managers that are available for example, the Bluetooth Discovery Manager, the RFSwitch Discovery Manager, the SerialPort Discovery Manager, the External Discovery Manager, the UPnP Discovery Manager etc.

The Discovery Manager also adds LinkSmart Services to a physical device. LinkSmart Services (generic functions) can be in the form of Energy Services, Location Services, Storage Services, Context Services etc. The Device Application Catalogue (DAC) is a catalogue of the currently accessible devices. It provides a graphical DAC browser as a complement to the SDK/API and also provides search interface to DAC. Device XML is the XML structure that encodes all data and meta-data about a device. It can include properties specific to UPnP, the LinkSmart system and/or developer defined properties. A graphical DAC browser is used to look at device XML.

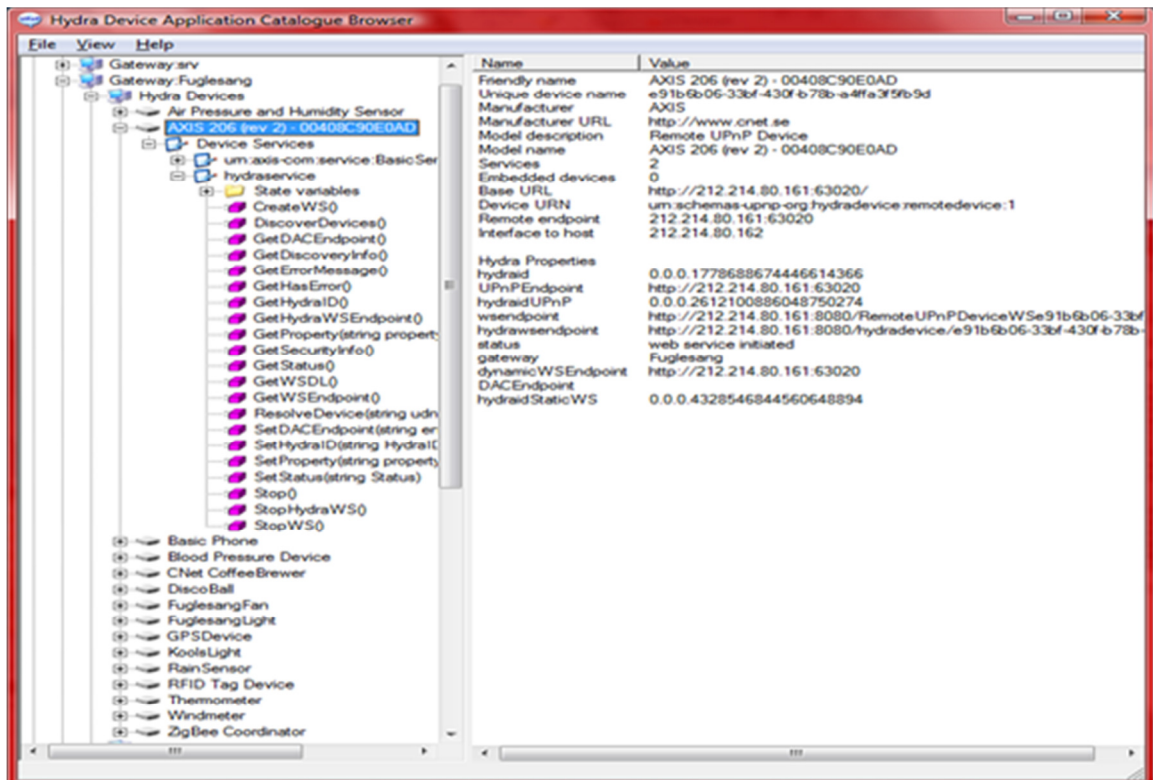


Figure 6: Device Application Catalogue – Graphical View

6.8 Security Library

This chapter deals with the mechanisms to protect messages which are exchanged between LinkSmart Managers. It also describes managers which are used to support the security mechanisms and which also provides functionalities to the LinkSmart developers, who want to use cryptographic functions in their LinkSmart applications.

Core LinkSmart

A single LinkSmart-enabled device can be distributed across many physical machines. In that case, internal parts of this LinkSmart device communicate over a public network, e.g. the internet. To protect this kind of internal communication, the Core LinkSmart security mechanisms are in place. Core LinkSmart security is based on symmetric keys that have to be generated by the device developer and must be deployed to each manager of the LinkSmart device. The implementation of the Core LinkSmart module is integrated into the Network Manager in the form of an Axis handler.

Inside LinkSmart

The Inside LinkSmart approach is based on public and private key pairs. Therefore, other considerations as in Core LinkSmart come into effect. Issues have to be solved for validating certificates, handling tokens, verifying signatures and token creation, storing and deletion. Due to the fact that a HID can change, we consider generating certificates on the fly, which then will be exchanged among the communication partners. The dedicated TrustManager is engaged in the verification of these generated certificates. The deletion of such a certificate is also due to the act that an HID of a counterpart is no longer valid.

6.9 Security Design

To secure messages between Network Managers over the insecure network, we use the Web Service Security (WSS) to secure messages (Strings), similar to Core LinkSmart. The main differences are the handling of security tokens (i.e., X509 certificates or other tokens used by other trust models).

Another aim of this Security Library is also to provide an interface for different security mechanisms. The developer will have the possibility to use any library which supports our envisaged interface; this can also mean that the developer can, e.g., plug in a different security library, which suits his needs.

The integrity and non-repudiation of messages will be achieved by the use of signatures. Due to the fact that HIDs can change and communication Partners (Network Managers) are previously unknown to each other, considerations like token generation and token exchanging must be taken into account.

The dedicated Trust Manager is used to verify the integrity of these tokens by verifying the certificates which are used by the Network Managers. A trust level can be set up by the developer which will be used to evaluate the trustworthiness of a generated and used certificate. Also different trust models can be used, which is also configured by the developer. Trust models can make use of a Public Key Infrastructures, of Web of Trust mechanisms, or even an interaction by the user.

Every time a new communication is initialized the Network Manager generates its own certificate which will be shared via a handshake mechanism, described below, between the different Partners. For the time of communication they will be stored in each involved manager's key store. Certificates can also be deleted after a certain time, i.e. when a HID is dropped or is no longer valid. That means for every HID there will be a generated public/private key pair.

7. Installing the LinkSmart Middleware

This chapter discusses the requirements for installation of the LinkSmart Middleware, using the Equinox (Eclipse) implementation of OSGi. The LinkSmart Middleware will be provided as a stand-alone package which can be run in any generic OSGi framework, not only depending on the eclipse IDE.

7.1 Prerequisites

Some bundles are not provided with Eclipse (Galileo - 3.5 and earlier), and so will need to be downloaded and placed in the plugins folder of your Eclipse installation, if the LinkSmart Middleware is to be launched from within the Eclipse environment, which is not compulsory.

Most significantly this may include

```
org.eclipse.equinox.cm_1.0.100.v20090520-1800
org.eclipse.equinox.ds_1.1.1.R35x_v20090806
```

7.2 Required Bundles

The following sections specify the various OSGi bundles required to launch the LinkSmart Middleware. This includes both core bundles, and the bundles of LinkSmart Managers and components. The configuration provided here is a very basic one, and extra LinkSmart bundles can be added to include their relevant functionalities.

(Note: LinkSmart distribution now includes a dependency management, Apache Ivy, that should take care all of the dependencies needed by Linksmart. It will download them automatically from the Spring Source repository when ant is run).

7.2.1 Core required bundles

These external bundles have to be added in the run configuration:

```
javax.servlet_2.5.0.v200806031605
javax.xml_1.3.4.v200902170245
org.apache.commons.codec_1.3.0.v20080530-1600
org.apache.commons.httpclient_3.1.0.v20080605-1935
org.apache.commons.lang_2.3.0.v200803061910
org.apache.commons.logging_1.0.4.v200904062259
org.apache.log4j_1.2.13.v200903072027
org.apache.xalan_2.7.1.v200905122109
org.apache.xml.serializer_2.7.1.v200902170519
org.eclipse.equinox.cm_1.0.100.v20090520-1800
org.eclipse.equinox.ds_1.1.1.R35x_v20090806
org.eclipse.equinox.http.jetty_2.0.0.v20090520-1800
org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800
org.eclipse.equinox.util_1.0.100.v20090520-1800
org.eclipse.osgi.services_3.2.0.v20090520-1800
org.eclipse.osgi_3.5.1.R35x_v20090827
org.mortbay.jetty.server_6.1.15.v200905151201
org.mortbay.jetty.util_6.1.15.v200905182336
```

7.3 Crypto Manager Setup

The Crypto Manager requires some initial modification of the default Java distribution, in order to provide the functionalities it requires.

How to register the global crypto provider:

1. In order to use the bouncycastle keystore and cryptographic keys longer than 128bit, the "JCE unlimited strength policy files" (Download [Java Cryptography Extension \(JCE\) Unlimited Strength Jurisdiction Policy Files 6](#)) needs to be updated. Copy *local_policy.jar* and *US_export_policy.jar* to **\$JAVA_HOME/jre/lib/security** (overwriting existing files).

(The following step should be optional. You should try it if you get a "KeystoreException: no match")

2. In order to make the bouncycastle crypto provider available for the whole OSGi framework, it needs to be installed as a global java security provider
 - a. Copy *lib/bcprov-jdk14-138.jar* to **\$JAVA_HOME/jre/ext**
 - b. In **\$JAVA_HOME/jre/lib/security/java.security**, add bouncycastle to the available crypto providers¹:

security.provider.5=org.bouncycastle.jce.provider.BouncyCastleProvider

7.4 LinkSmart bundles

Different steps to "install" LinkSmart bundles are described in the following:

Build the middleware using Ant

1. Check out LinkSmart middleware from Source Forge repository at:
<https://linksmart.svn.sourceforge.net/svnroot/linksmart/branches/1.1> .
2. Make sure have Ant installed. If you run the scripts inside your IDE, select the proper Ant distribution by setting ANT_HOME.
3. Run the ant build script located inside the components folder. Ant will now download all dependencies, build the LinkSmart Bundles, and put them in a folder named target_platform.

Set-up target platform

The target platform specified contains the bundles which your code will be compiled against. Having a target platform allows you to compile and run your modified code without having to bring all of the source code into your development workbench. The target platform should be the same platform you are developing for. To set up your target platform:

- From your development workbench select Window-> Preference-> Plug-in development-> Target Platform.
- Select Add... and then Nothing option, hit Next.

¹ Don't set bouncycastle as the first provider. This is a known bug and won't work.

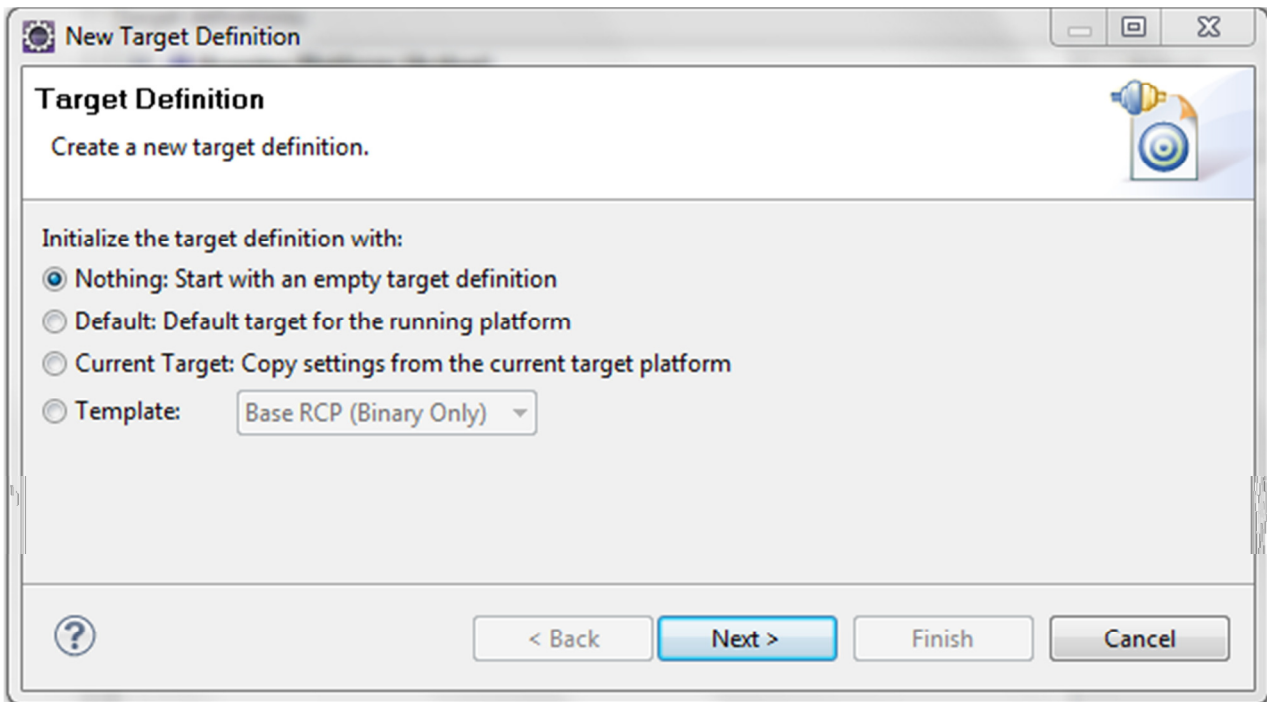


Figure 7: Configure Eclipse Target Platform

Give your target platform a name, e.g. LinkSmart and then add two different locations of the platform you wish to target. For this tutorial select the content from the two following directories (see also screen shot below):

- `${workspace_loc}/opensource/components/target_platform`
- `${workspace_loc}/opensource/components/distribution`

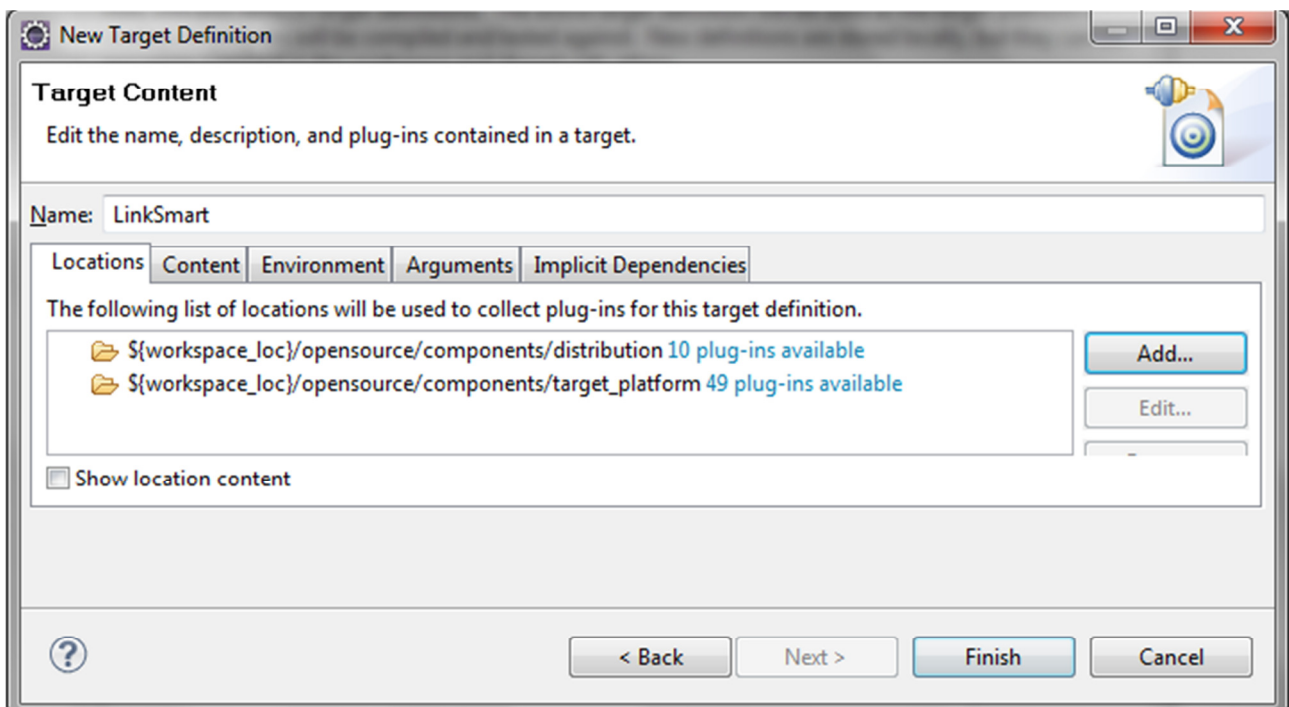


Figure 8: Add New Target Platform Location

You can use the above links to the workspace directory IF you have ticked "Copy projects into workspace" otherwise you have to manually select the proper directories! You can identify if this worked if there is a number of plugins listed behind the directory path!

Hit Finish, and then Apply this new target platform instead of the default set running Eclipse workbench platform.

These LinkSmart bundles have to be added in the run configuration:

```
CryptoManager_1.1.0
LinkSmartManagerConfigurator_1.0.0.qualifier
LinkSmartMiddlewareAPI_1.0.0.qualifier
LinkSmartMiddlewareClients_1.0.0.qualifier
LinkSmartWSPProvider_1.0.0.qualifier
Network_Manager_Bundle_1.7.0.qualifier
```

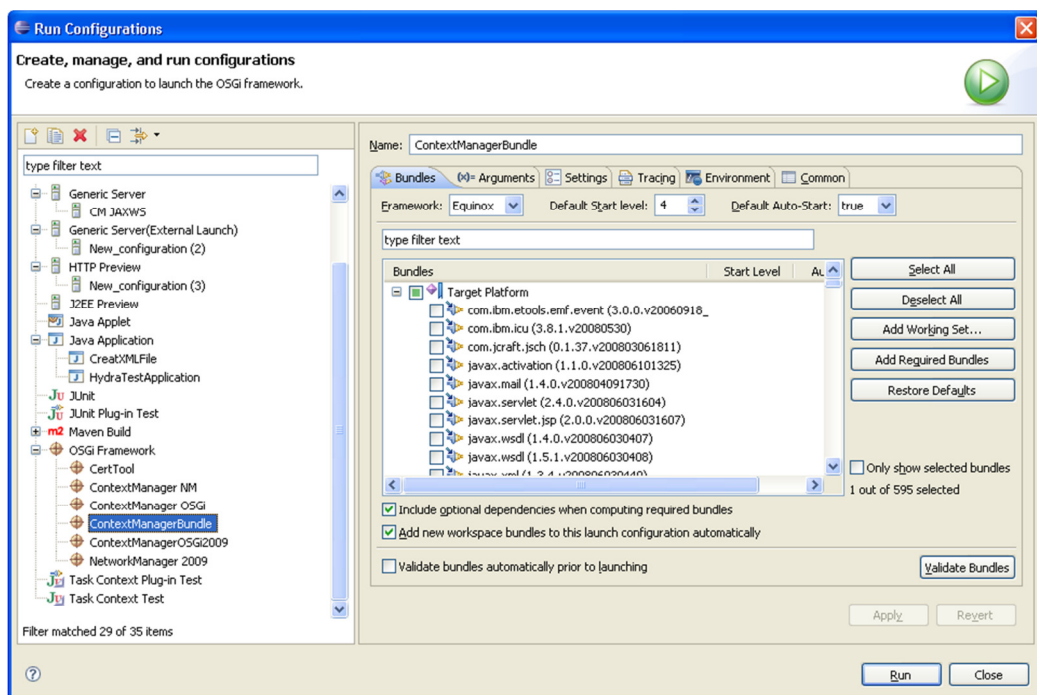


Figure 9: Run Configuration (LinkSmart Bundles)

These bundles provide the basic functionality of the Network Manager with the CryptoManager. Other bundles must be added as required.

7.5 VM Arguments

```
-Declipse.ignoreApp=true -Dosgi.noShutdown=true -Dorg.osgi.service.http.port=8082
```

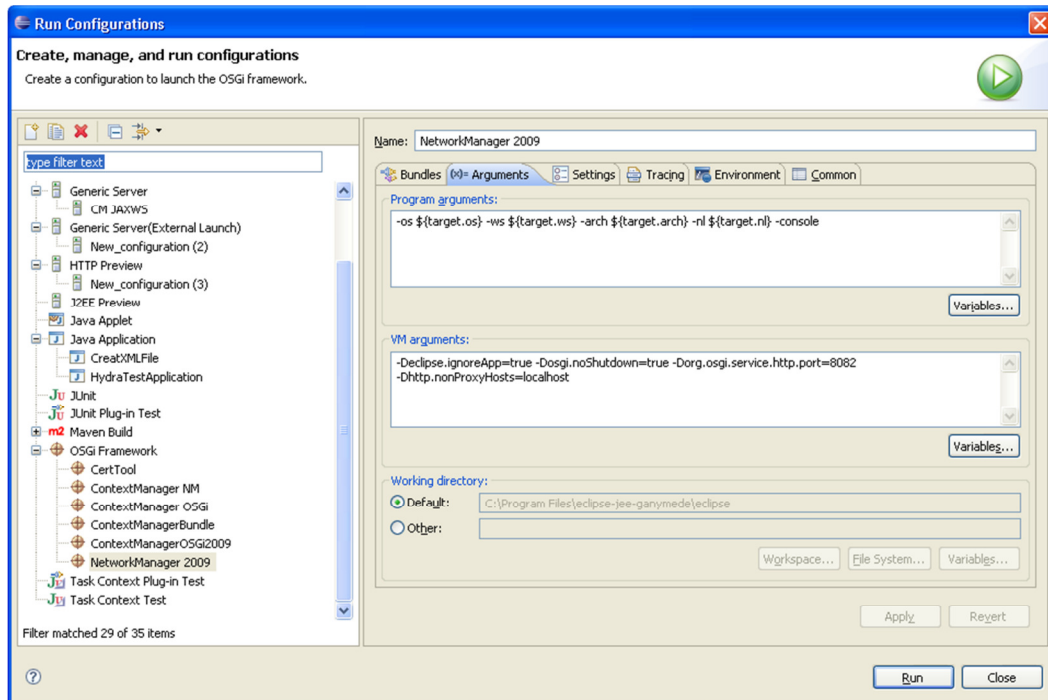


Figure 10: Eclipse Run Configuration (Arguments)

7.6 Running the framework

When the LinkSmart framework is started and 'ss' type in to see the list of installed plugins. It should look like this (id numbers may be different, but the order of the bundles in which they start is vital):

Framework is launched.

id	State	Bundle
0	ACTIVE	org.eclipse.osgi_3.5.1.R35x_v20090827
2	ACTIVE	CryptoManager_1.1.0
3	ACTIVE	org.apache.log4j_1.2.13.v200903072027
4	ACTIVE	javax.servlet_2.5.0.v200806031605
5	ACTIVE	org.eclipse.equinox.util_1.0.100.v20090520-1800
6	ACTIVE	org.apache.xml.serializer_2.7.1.v200902170519
7	ACTIVE	org.eclipse.equinox.http.jetty_2.0.0.v20090520-1800
8	ACTIVE	org.mortbay.jetty.server_6.1.15.v200905151201
9	ACTIVE	LinkSmartMiddlewareAPI_1.0.0.qualifier
11	ACTIVE	org.eclipse.equinox.ds_1.1.1.R35x_v20090806
12	ACTIVE	LinkSmartManagerConfigurator_1.0.0.qualifier
14	ACTIVE	org.eclipse.osgi.services_3.2.0.v20090520-1800
15	ACTIVE	org.eclipse.equinox.cm_1.0.100.v20090520-1800
16	ACTIVE	LinkSmartWSPProvider_1.0.0.qualifier
17	ACTIVE	javax.xml_1.3.4.v200902170245
18	ACTIVE	org.mortbay.jetty.util_6.1.15.v200905182336
19	ACTIVE	org.apache.commons.lang_2.3.0.v200803061910
21	ACTIVE	org.apache.xalan_2.7.1.v200905122109
22	ACTIVE	Network_Manager_Bundle_1.7.0.qualifier
23	ACTIVE	org.apache.commons.logging_1.0.4.v200904062259
24	ACTIVE	org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800
25	ACTIVE	org.apache.commons.httpclient_3.1.0.v20080605-1935
26	ACTIVE	LinkSmartMiddlewareClients_1.0.0.qualifier
27	ACTIVE	org.apache.commons.codec_1.3.0.v20080530-1600

Useful URLs

Go to <http://localhost:8082/NetworkManagerStatus>, It should show something like this:

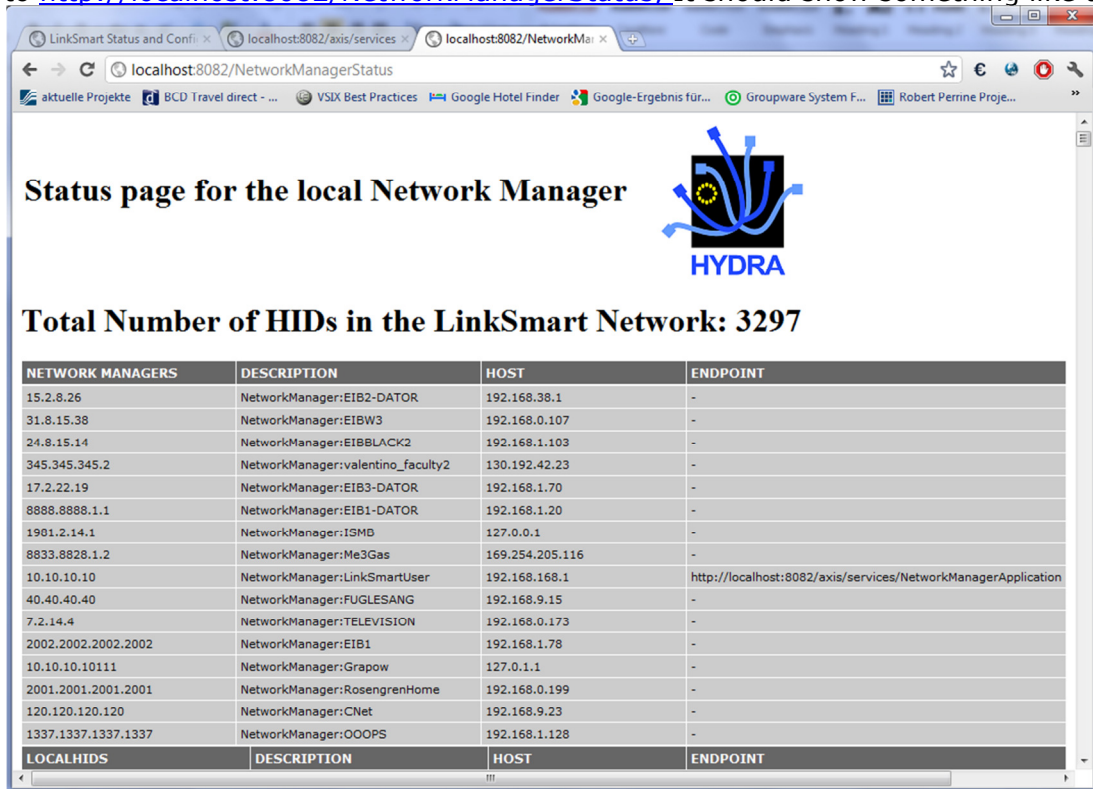


Figure 11: Network Manager Status page in browser

Go to: <http://localhost:8082/LinkSmartStatus>, It should show something like this:

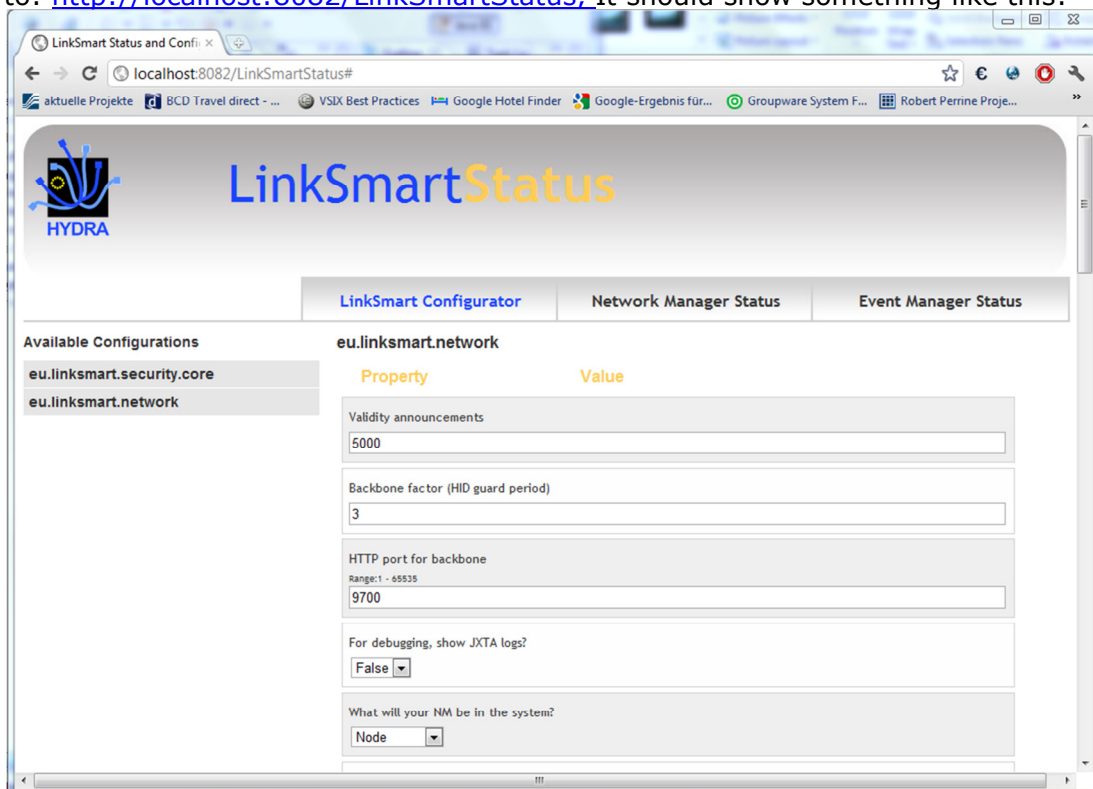
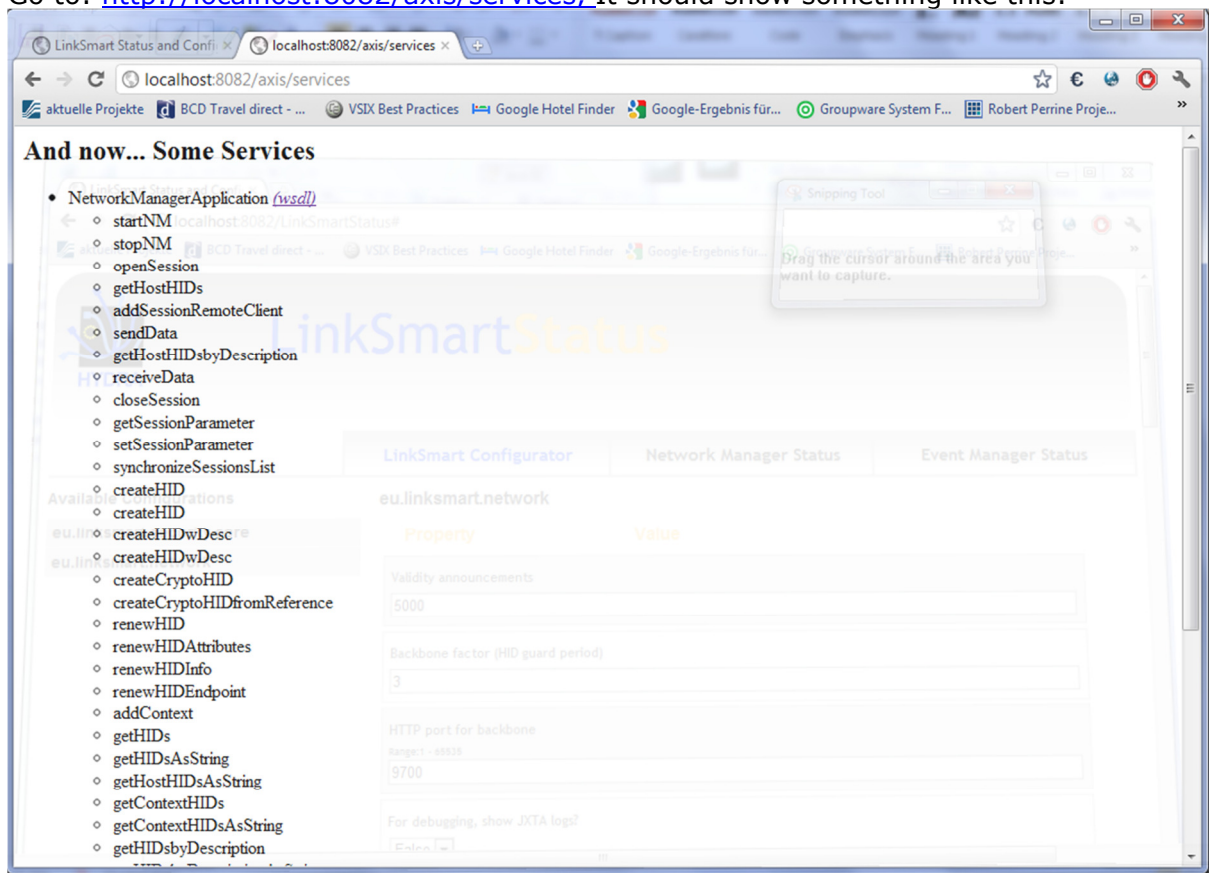


Figure 12: LinkSmart Status page

Go to: <http://localhost:8082/axis/services>, It should show something like this:



The screenshot shows a web browser window with the URL `localhost:8082/axis/services`. The page title is "And now... Some Services". The main content area displays a list of services for the `NetworkManagerApplication` (WSDL). The services listed include:

- startNM
- stopNM
- openSession
- getHostHIDs
- addSessionRemoteClient
- sendData
- getHostHIDsbyDescription
- receiveData
- closeSession
- getSessionParameter
- setSessionParameter
- synchronizeSessionsList
- createHID
- createHIDations
- createHID
- createHIDwDesc
- createHIDwDesc
- createCryptoHID
- createCryptoHIDfromReference
- renewHID
- renewHIDAttributes
- renewHIDInfo
- renewHIDEndpoint
- addContext
- getHIDs
- getHIDsAsString
- getHostHIDsAsString
- getContextHIDs
- getContextHIDsAsString
- getHIDsbyDescription

Below the list, there are three tabs: "LinkSmart Configurator", "Network Manager Status", and "Event Manager Status". The "LinkSmart Configurator" tab is active, showing a configuration form for the `eu.linksmart.network` service. The form includes the following fields:

Property	Value
Validity announcements	5000
Backbone factor (HD guard period)	3
HTTP port for backbone	9700
For debugging, show JXTA logs?	<input type="checkbox"/>

Figure 13: List of services (in browser)

8. Software Development Kit

This chapter provides an introduction to the Software Development Kit (SDK) in LinkSmart, detailing the software interfaces (Web Services etc.) of each LinkSmart component / manager / tool, and tutorials on how to use them.

8.1 LinkSmart Commons

The LinkSmart Commons set of bundles provides the main point of interaction between the developer and the SDK. The commons bundles include:

- Middleware API
- Clients
- Configurator

These bundles make it simpler for the developer to use the managers and components of the LinkSmart Middleware, as well as for the creation of applications.

8.1.1 LinkSmart Middleware API

The LinkSmart Middleware API bundle contains the LinkSmart API, that is, all external interfaces of the LinkSmart managers and the types used in them. In that way, there is one common bundle containing all relevant LinkSmart interfaces, separating them from their implementation, which is necessary for a well-structured integrated middleware. This also includes the classes for the various types that act as parameters for calls to middleware components, as well as a set of utilities to aid with usage.

Use of the Middleware API for particular components is discussed in the sections relevant to each component. Typically, this also involves the LinkSmart Middleware Clients bundle, as described in the following section.

8.1.2 LinkSmart Middleware Clients

The LinkSmart Middleware Clients bundle contains all the Web Service clients for calling the various managers and components of the LinkSmart Middleware. These clients include the generated AXIS files for the creation of Web Service Clients, providing the services as defined in the LinkSmart Middleware API. The middleware clients are located in bundles named as follows:

Using the Event Manager as an example, the developer can generate the Event Manager client in one of two ways. Firstly, by using the generated *Locator* class for each client, as shown below:

```
EventManagerPortServiceLocator locator =
    new EventManagerPortServiceLocator();
locator.setEventManagerPortEndpointAddress(endpoint);
EventManagerPort em = locator.getEventManagerPort();
em.subscribe("ExampleTopic", "0.0.0.235235154145");
```

Here, the locator is configured with an endpoint address. This is the address of the local SOAP Tunnel (exposed by the Network Manager), specifying the *from* and *to* HIDs, as well as the *sessionID*. An example endpoint is given below, with no *sessionID* (0).

<http://localhost:8082/SOAPTunneling/0.0.0.341243145454252/0.0.0.412434465875/0/>

The second method is to use the RemoteWSClientProvider OSGi service that allows retrieval of the relevant manager objects without the need to create specific Web Service Locator objects. This service offers a method, called `getRemoteWSClient`, which interface is:


```
public Object getRemoteWSSClient(String className,
                                String endpoint, boolean coreSecurityConfig);
```

When calling the method, the interface class name of the Web Service, the endpoint to this Web Service and a Boolean value indicating whether we want to call the service with LinkSmart security or not must be provided.

The next lines show an example of calling a method of the Network Manager Web Service. The developer first obtains a RemoteWSSClientProvider element, and via this Web Service Client object finally makes the call to the desired method of the Network Manager (in this case getHIDs).

```
RemoteWSSClientProvider service = (RemoteWSSClientProvider)
context.getService(context.getServiceReference
(RemoteWSSClientProvider.class.getName()));
NetworkManagerApplication nm = (NetworkManagerApplication)
service.getRemoteWSSClient(NetworkManagerApplication.class.getName(),
endpoint, true);
Vector v = nm.getHIDs();
```

8.1.3 LinkSmart Configurator

In order to achieve high level of integration between the set of managers that conforms the LinkSmart Middleware, a common configuration system for all LinkSmart managers and applications has been implemented.

This common configuration system is based on the use of the configuration admin OSGi service. This service provides a way to update dynamically the configurations, avoiding having to restart the managers for updating them. It also provides persistency for the configurations.

Thus, an OSGi bundle which provides a common interface for the configuration of all LinkSmart managers has been implemented. The bundle is called LinkSmart Manager Configurator. Adapting the configuration of a particular manager is achieved using the Configurator class provided by the LinkSmart API. The Configurator class is a class that implements the ManagedService interface, so that it can receive configurations from the configuration admin OSGi service. The Configurator class provides the methods and attributes for managing the configuration of the manager which instantiates it, and a way for communicating with the configuration admin service in order to apply into this service all changes introduced by the user at the Configurator class, registering itself as a managed service.

The LinkSmart Manager Configurator bundle provides a set of interfaces that allows modifying the configuration of the different LinkSmart managers previously adapted to the new common configuration system. This bundle provides three interfaces for configuring LinkSmart managers:

- A web application called LinkSmart Status
- A Web Service deployed by the LinkSmart Manager Configurator
- An OSGi console command, currently working on Equinox

The LinkSmart Status page (**Figure 14**) is a web application that provides a web interface for configuring the different local LinkSmart managers adapted to the new common configuration system, based on the configuration admin OSGi service. It also provides all the information provided by the well-known Network Manager Status page and Event Manager Status pages.

Regarding the Network Manager information included, the LinkSmart Status page provides information about HIDs, hosts where they are deployed, descriptions and endpoints of all devices detected by the local Network Manager, differentiating between local and remote HIDs (local and remote Network Manager installations).

Regarding the Event Manager information included, the LinkSmart Status page provides information about the topics, the endpoints and the dates of subscription of all the LinkSmart events the local Event Manager is subscribed to.

Regarding the configuration of the managers, the LinkSmart Status page provides a graphical interface for configuring all LinkSmart managers adapted to the new common configuration system in a dynamic way. The available sets of configuration options are loaded dynamically if the manager which uses them is running, identifying themselves by their configuration PID. Clicking over a configuration PID, all its options and their values will be loaded, being possible to modify them and update the modifications done by clicking the Update Configurations button.

Once you update the configuration, the new configuration will be working at the moment, without having to restart the managers. However, if you put `-clean` parameter inside program arguments of your OSGi configuration all updated configurations will be reset to their initial state. The URL of the LinkSmart Status page (a screenshot of the service can be seen in **Figure 14**) is <http://localhost:8082/LinkSmartStatus>, (given that the web server of the LinkSmart installation is running in the 8082 port, which is the default port of the LinkSmart configuration). This functionality is also replicated inside the IDE, as discussed in the relevant section(s).

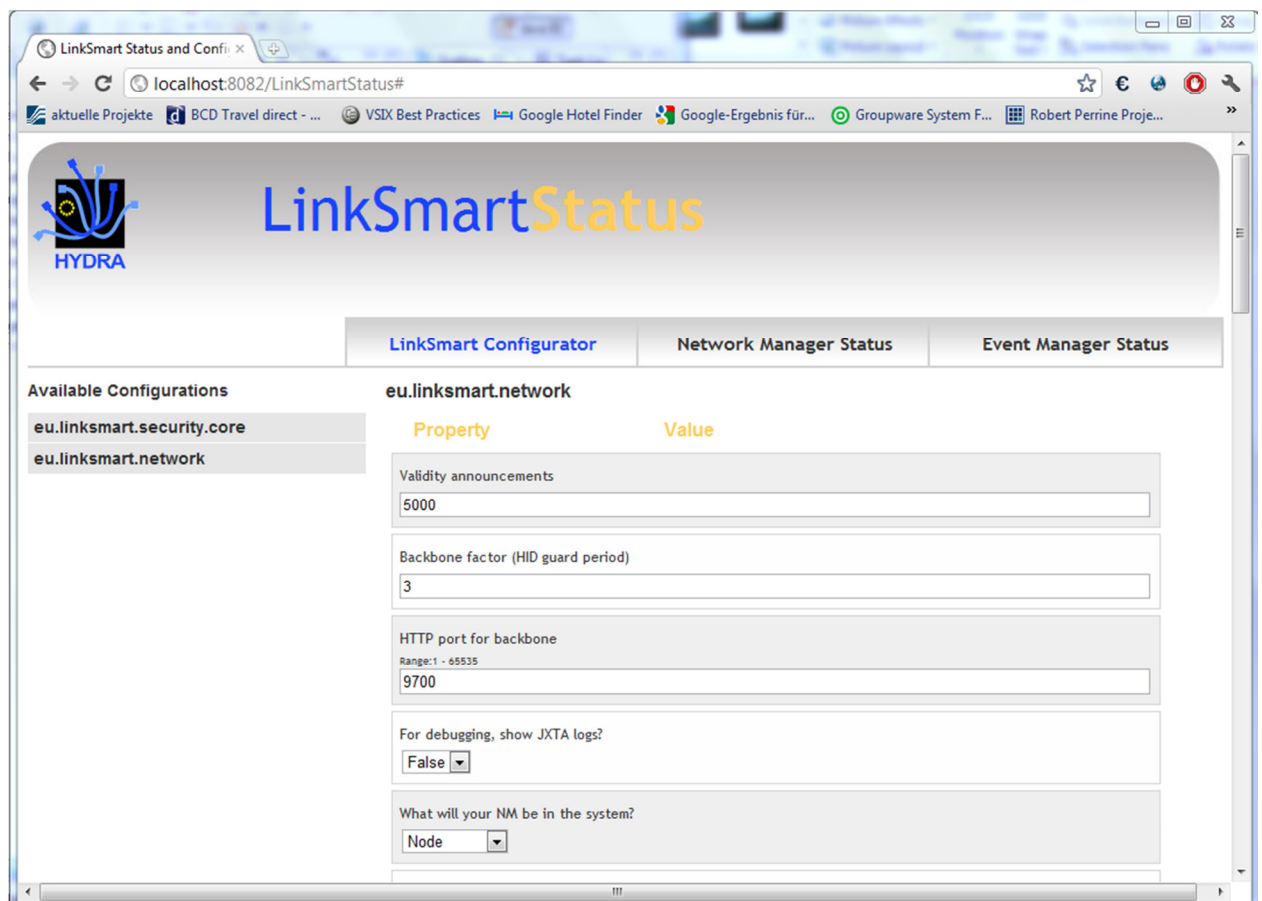


Figure 14: LinkSmart Status page screenshot

Another configuration tool provided is the Web Service, which is deployed by the LinkSmart Manager Configurator bundle, and that provides the following methods:

- `getAvailableConfigurations()`: list the available set of configurations.
- `deleteConfiguration(String configuration_pid)`: delete a concrete configuration from the common configuration system.
- `listConfiguration(String configuration_pid)`: list the options provided by a concrete configuration and their current values.

- `setConfiguration(String configuration_pid, String option_key, String value)`: set the value of a concrete option for a concrete configuration.

Finally, and regarding the configuration tools using Equinox OSGi console, the `configure` command is used, which provides similar options as the ones provided by the Web Service above. These options are the:

- `configure -l`: list the available set of configurations.
- `configure -d <configuration_pid>`: delete a concrete configuration from the common configuration system.
- `configure <configuration_pid>`: list the options provided by a concrete configuration and their current values, e.g. `configuration com.eu.LinkSmart.network` will print current configuration of the Network Manager.
- `configure <configuration_pid> <option_key> <option_value>`: set the value of a concrete option for a concrete configuration.

8.2 Network Manager

The network model complements the runtime platform model regarding the details of the network. In LinkSmart the underlying network is complex and therefore, it needs to be described in a separate (but related) network model. The purpose of the network model is to define what types of network connections will be supported and if there are constraints that have to be adhered to during implementation and network design.

8.2.1 LinkSmart Definition of Device To Device Communication

The Network Manager is the incoming and outgoing point of information in the middleware. Therefore, the main purpose of Device to Device communication will be managing the communication between Network Managers. This means that only LinkSmart-enabled-devices will be involved in this kind of communication.

Devices inside LinkSmart need to communicate in order to exchange information. Each device offers different resources inside the LinkSmart network mechanisms need to be implemented that allow the discovery of such new resources in LinkSmart-enabled devices inside the network. Moreover, in order to consume these resources, LinkSmart devices need the means to establish communication between each other. The following sections will present those aspects.

Addressing

From the middleware point of view, an addressing method based on LinkSmart Identifiers (HID) has been defined for LinkSmart, instead of the usual IP-based one. The Identity Manager is responsible for the management of these HIDs. Its main functionality is providing a unique context-dependant identifier for every device (physical or semantic), resource or service, called HID. It is also responsible for the maintenance of the `idTable`, a data structure dedicated to store the matching between logical and physical identifiers.

However, this addressing is useless if there is not a way to propagate this information to other IoT-enabled devices involved in the LinkSmart Network. The Backbone Manager, a component inside the network manager, is responsible for spreading this information between the different IoT-enabled devices in the network. Thus, every Identity Manager belonging to the LinkSmart Network internally keeps an `idTable` and an updated list of every HID in the network. This process is known as Network Manager Discovery. The LinkSmart middleware will be running in dynamical environments, where new resources are susceptible to constantly appear or disappear. In order to detect new resources inside the LinkSmart network, we need a discovery mechanism.

Inside the LinkSmart network, devices and resources are identified through a LinkSmart ID (HID), which varies depending on the context. In order to contact them, one IoT-enabled device needs to contact the

Network Manager of the IoT-enabled device they belong to. The discovery of Network Managers will be done through Device to Device communication.

Through Device To Device communication, we aim to propose an innovative way to discover Network Managers (and thus, LinkSmart-enabled devices) and also to know more about their features and services provided, in a scalable Wide Area Network. This means that the scope of the LinkSmart network will not be restricted to a Local Area Network.

Communication

As mentioned before, the Network Manager is the incoming and outgoing point of information in the middleware. The LinkSmart network is an "all-IP" network. This means that only devices with IP communication capabilities will be able to communicate directly (through device to device communication) inside the LinkSmart network.

Moreover, the device to device communication will be restricted to the LinkSmart-enabled devices that are able to host the LinkSmart middleware, which in LinkSmart terms means that this communication will be "inside" LinkSmart. Thus, the device to device communication can be defined as the data exchange between devices "inside" LinkSmart network, which are LinkSmart enabled and have IP communication capabilities.

8.2.2 The Peer-to-Peer Network Architecture

There exist multiple objectives regarding device to device communication. First, the LinkSmart middleware needs to offer an efficient way to share resources among the LinkSmart Network, in a scalable, distributed and efficient way. The LinkSmart middleware also needs to prevent system failures when a node is not available. And finally, the LinkSmart Network needs to allow ubiquitous access to the network.

All of these reasons have led us towards a Peer-to-Peer architecture. Several Peer-to-Peer models have been analysed and according to the requirements identified for device to device communication, JXTA P2P communication protocols have been selected as the most suitable mechanism to carry on the communications "inside" LinkSmart. That is, the communication between Network Managers.

The reasons that have led us to select JXTA are:

- **Interoperability:** Enables communication between peers independently of network addressing and physical protocols.
- **Platform independence:** JXTA does not depend on the programming language, network transport protocols and deployment platforms, giving freedom of choice. Java SE and Java ME implementations have been selected for LinkSmart.
- **Ubiquity:** JXTA is designed to be deployed on any device, not just PCs.
- **Security:** for security means regarding authentication, authorisation, and integrity can be implemented based on JXTA. Attacks on the level of the protocol cannot be addressed as that would require changing the JXTA protocol.
- **Community support:** JXTA is supported by a wide community of developers and the different specifications are fully documented.
- **Wide range of services:** Most of the P2P models studied have been designed exclusively for providing file sharing services. Instead, in JXTA, thanks to its abstract architecture based on six protocols, it is possible and feasible to create a wide range of interoperable services and applications.

8.2.3 Purpose

The Network Manager is the bottom layer of the LinkSmart middleware deployed in LinkSmart Gateways and in LinkSmart-enabled devices. It is the entry and exit point of information of the LinkSmart middleware. There is only one Network Manager per device where the middleware is deployed.

The Network Manager provides a Web Service interface (which is the main interface of the Network Manager), which is the information entry point for the middleware. Data transferred between LinkSmart-enabled devices and gateways should always pass through the Network Manager.

8.2.4 Main Functionalities

The Network Manager is responsible of managing the communication between LinkSmart-enabled devices. In order to do this, the Network Manager:

- Creates and overlay P2P network, where all the LinkSmart-enabled devices appear directly interconnected, no matter if they are behind a NAT (Network Address Translator) or Firewall.
- Provides indirection architecture for addressing Web Services hosted by LinkSmart devices using the HID addressing mechanism. Each service is identified in LinkSmart through an HID, which is a global and unique identifier. The Network Manager provides interfaces for other managers, applications and LinkSmart devices for HID creation, modification and deletion. It also offers the possibility to select the transport protocol for the service invocation between TCP, UDP and Bluetooth.
- Provides a transport mechanism over the overlay P2P network for invoking Web Services hosted by LinkSmart devices (SOAP Tunnelling) using the HID addressing mechanism. The SOAP messages addressed to an HID are routed by the Network Manager through the overlay network to the Network Manager hosting the service. Therefore, using the SOAP Tunnelling and the Network Manager any device or application is able to transparently publish and consume services anywhere, anytime, breaking the network interconnectivity barriers and independently of the service endpoint location.
- Provides a transport mechanism over the overlay P2P network for multimedia content exchange between UPnP AV or DLNA devices.
- Provides session management mechanisms between HIDs during service invocations.
- Provides time reference synchronization between different Network Managers.
- Provides a status page for developers, which the developer can use for monitoring dynamic information about the LinkSmart Network and the HIDs available.

Each LinkSmart-enabled device will run one and only one Network Manager. The Network Manager maintains two complex data structures: the LinkSmart ID (HID) and the Session. The following sections provide an overview on these two data structures.

8.2.5 LinkSmart Web Service Provider

First of all, in order to make the deployment of Web Services in the LinkSmart middleware easier, a new OSGi bundle has been created to take the place of the obsolete Axis bundle that the different managers were using since the beginning of the project. This bundle is the LinkSmart WS Provider bundle. The main goal of this component is to provide automatic deployment of Web Services and independence for LinkSmart managers from Axis.

Now it is possible to deploy Web Services, including the LinkSmart manager ones, in an automatic way, without the use of a deployer class or WSDD files.

The LinkSmart WS Provider bundle is still based on the Axis bundle, but it has been adapted to the LinkSmart middleware, providing transparent interfaces to the developers supporting all the characteristics LinkSmart middleware need.

The LinkSmart WS Provider bundle is composed of three packages, as seen in the **Table** .

Package	Definition
<i>com.eu.LinkSmart.security.axis</i>	Provides Core LinkSmart security to the bundle.
<i>com.eu.LinkSmart.wsprovider.impl</i>	The main package, deals with the detection of OSGi services and their deployment as Web Services.
<i>com.eu.LinkSmart.wsprovider.servlet</i>	Deploys a servlet which represents the Axis administration servlet.

Table 1: LinkSmart WS Provider package structure

The main class is the Activator class, which can be found under the *com.eu.LinkSmart.wsprovider.impl* package. It deals with the detection of OSGi services and their deployment as Web Services through a *ServiceTracker* object.

The services to be published as Web Services should have been deployed as an OSGi service. It is also recommended but not mandatory to use OSGi Declarative Services. When a service is to be published as a Web Service using the LinkSmart WS Provider, a set of properties need to be defined:

- *SOAP.service.name*: mandatory property, it defines the name of the service to be deployed. The LinkSmart WS Provider deploys each service using this defined property. Once this property is set, the service will be deployed with this name at *http://localhost:8082/axis/services*.
- *SOAP.service.methods*: optional property, it provides a list of the names of the methods to be implemented. When defined, the LinkSmart WS Provider deploys only the methods indicated. Otherwise, the LinkSmart WS Provider deploys all the methods of the Web Service, i.e. all methods that have an access level of 'public'.
- *LinkSmart.security.config*: optional property, it defines whether the Web Service is to be deployed with or without security. A Boolean value defines this property. All services will be deployed with security by default.

In order to register and deploy a Web Service in the LinkSmart middleware, the developer must register the service in the framework with the *SOAP.service.name* property indicating the name of the service. Programmatically, and in the case of non declarative services, a service is registered as follows:

```
Hashtable props = new Hashtable();
props.put("SOAP.service.name", "EventManagerPort");
context.registerService(EventManager.class.getName(), this, props);
```

By using OSGi declarative services, the services are already registered via the framework, but the properties have to be set in the *OSGI-INF/component.xml* file (at least the mandatory *SOAP.service.name* property), as shown in the Figure 15.

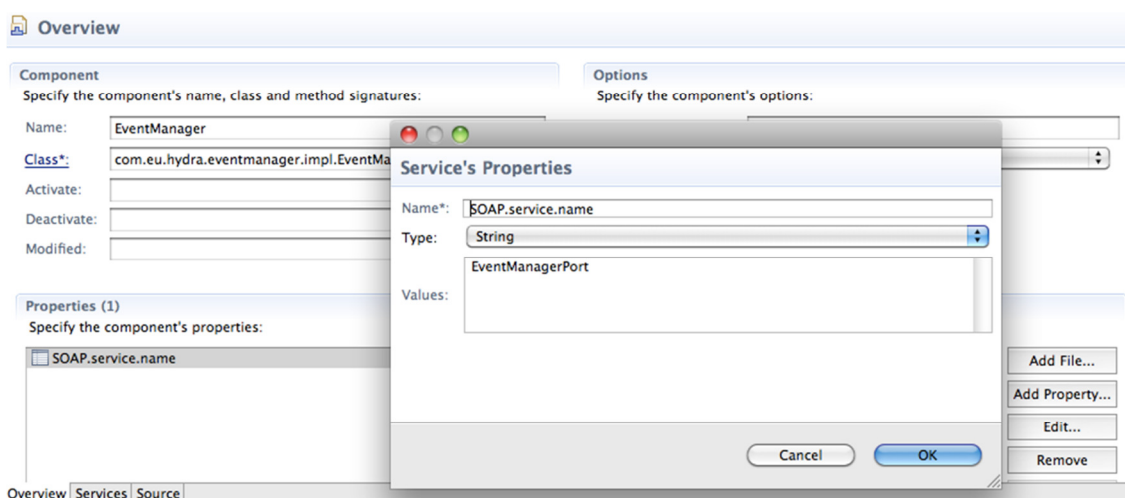


Figure 15: Introduction of a property at the XML file of a declarative service

8.2.6 Crypto HIDs

From the Lessons Learned of the third iteration, we realized that the service addressing mechanisms implemented in LinkSmart, the HIDs, lack of high security features. The HID is an identifier that allows developers and applications to identify each entity evolving in a LinkSmart network. It was designed to identify each service in a given situation (context) but also to dismiss the real identity of the device offering the service.

The main problem with current implementation of HID is that the information related to their description is being exchanged over the network without any encryption between the Network Managers. Thus an attacker to the LinkSmart middleware would be able to identify the identity of HID and the service provided by the owner of the HID by just sniffing the network traffic. Another problem identified with HID, is that the description field associated with them is not enough to unambiguously identify a service, as it is just a String with no fixed format. These problems are not very important for building applications that do not have high security requirements, but when moving to domains that require these high levels of security, like e-Health, the problems become important.

In order to solve these issues, we have extended the HID concept incorporating new security features like certificate linking, HID description through attributes and HID data encryption. These new secure LinkSmart Identifiers are called Crypto HIDs. The main features incorporated to Crypto HIDs are:

Certificate linking

Each HID, when it is created, is associated with a certificate, generated using the Crypto Manager. This certificate is used to encrypt and decrypt all the information sent to and from this HID. Therefore, before sending any information to an HID, the Network Managers perform a certificate exchange process for encrypting the information that is going to be exchanged. This certificate exchange is performed using the Secure Session Protocol, with which certificates will be distributed using a public key exchange protocol.

HID attributes

In order to unambiguously identify a HID in the LinkSmart Network, we have extended the description of HID to attributes. Each HID is created with some attributes, which are securely stored in its certificate. The number of attributes is not fixed, and it is up to the developer to decide which attributes to use. Some examples for attributes would be:

- **PID (Persistent Identifier):** An identifier for the device providing a service (for example, MAC address of the device)
- **SID (Service Identifier):** An identifier for the service provided. It could also be a semantic identifier of the service provided.
- **UserID (User Identifier):** Identifier for the owner of the device providing a service.

These attributes, and any others, are provided during HID generation following the Java Properties class XML schema. An example of attributes for an HID would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="PID">03-43-F3-23-24</entry>
<entry key="SID">ThermometerService</entry>
<entry key="UserID">Peter</entry>
</properties>
```

The attributes are not exchanged between Network Managers during the HID exchange process, this is, Network Managers have all the HIDs in the LinkSmart Network but do not have the information of what does each HID stand for. In order to provide developers and applications the means to know these attributes, we have implemented two mechanisms to retrieve the attributes for a specific HID and to query the network

searching for an HID matching some attributes. These two mechanisms have been designed taking security into account:

- Retrieving attributes for an HID: Using this mechanism, developers and applications are able to retrieve the attributes for a specific HID. As mentioned above, attributes for a HID are stored securely into the certificate linked to it. Therefore, in order to retrieve the attributes for a HID, the Network Manager starts the Secure Domain Protocol, exchanging the certificates of the interested parties. Therefore, nobody without a valid LinkSmart certificate is able to retrieve the attributes for a HID.
- Querying the LinkSmart network for a HID matching some attributes: This is the situation when an application wants to address a specific HID, with some fixed attributes, but without knowing beforehand which is the HID assigned to it. Imagine an application that wants to retrieve the temperature from a specific thermometer. It first needs to know the HID of that thermometer in order to be able to invoke its service.

The process is simple (illustrated in Figure 16 below): a query is generated and sent to all the Network Managers in the network using a multicast channel (step 1 in Figure 16). In the query, the requester has to provide its credentials, this is, its HID and attributes. Each Network Manager receives the query and searches in the local idTable (step 2 in Figure 16) (the table where all the HIDs are stored). If a Network Manager finds a HID that matches the query (step 3 in Figure 16), before answering to the sender, checks with the Policy Manager if there is any policy applied for that HID and provides the sender information (step 4 in Figure 16). The Policy Manager answers the Network Manager if it is allowed or not to send that information to the requester (step 5 in Figure 16). If it is allowed, a query response containing the HID is sent to the sender over a unicast channel (step 6 in Figure 16). If it is denied, no information is sent to the sender. Therefore, in every step of this process, security is ensured.

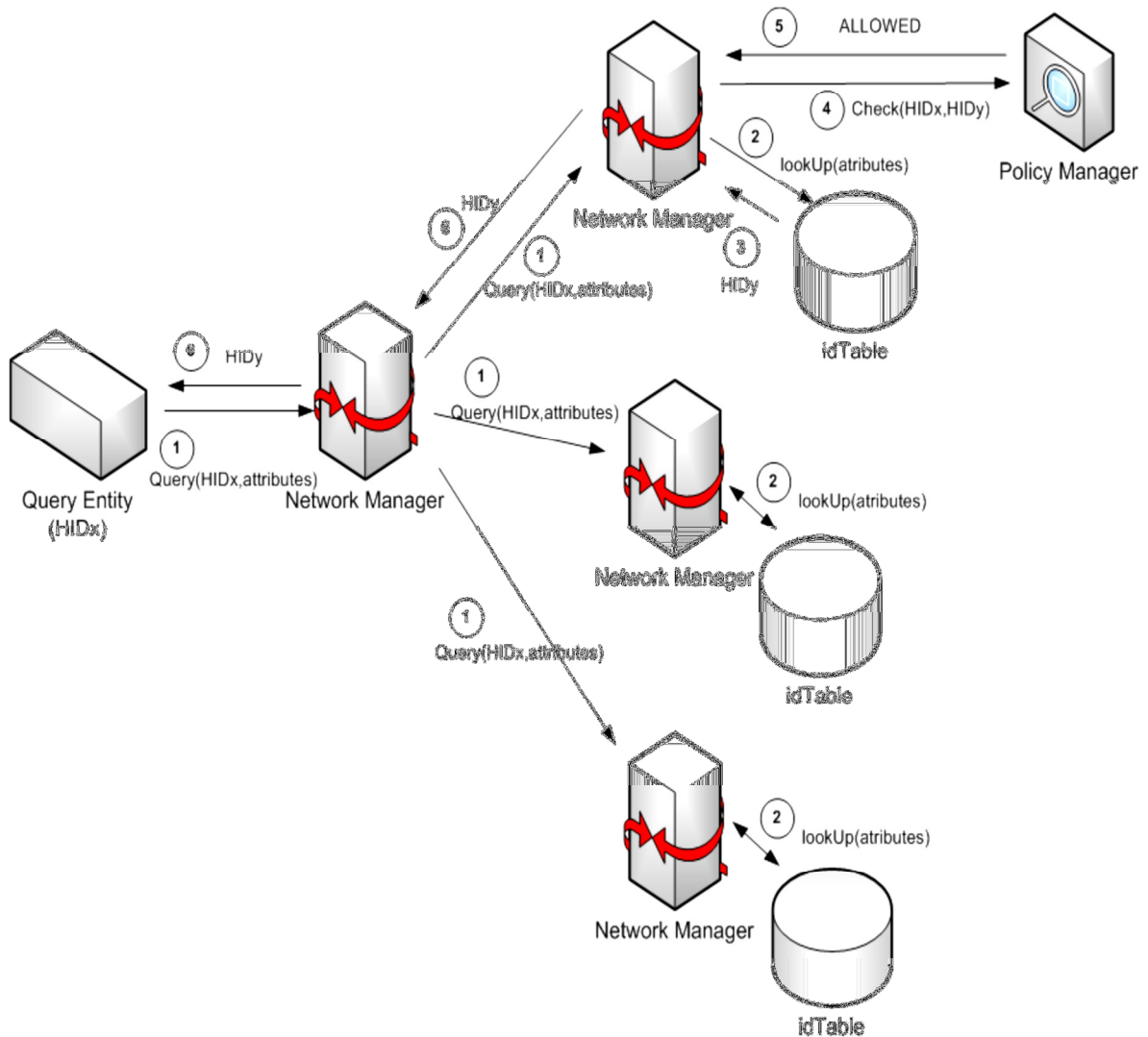


Figure 16: Querying the LinkSmart network for a HID matching some attributes

In order to provide the developers the tools for using these new mechanisms, four new methods have been added to the Network Manager API. The old methods for creating and interacting with HID related information are still maintained, for backwards compatibility reasons, but its usage is discouraged as they have been deprecated.

createCryptoHID

```
CryptoHIDResult createCryptoHID(java.lang.String xmlAttributes,
    java.lang.String endpoint)
    throws java.rmi.RemoteException
```

Operation to create an crypto HID providing the persistent attributes for this HID and the endpoint of the service behind it (for service invocation). The crypto HID is the enhanced version of HIDs, that allow to store persistent information on them (through certificates) and doesn't propagate the information stored on it. In order to exchange the stored information, the Session Domain Protocol is used. It returns a certificate reference that point to the certificate generated. The next time the HID needs to be created, using the same attributes, the certificate reference can be used.

Parameters:

xmlAttributes - The attributes (persistent) associated with this HID. This attributes are stored inside the certificate and follow the Java Properties Xml schema.
endpoint - The endpoint of the service (if there is a service behind).

Returns:

A `com.eu.hydra.network.ws.CryptoHIDResult` containing `String` representation of the HID and the certificate reference (UUID)

Throws:

`java.rmi.RemoteException`

createCryptoHIDfromReference

```
java.lang.String createCryptoHIDfromReference(java.lang.String certRef,  
                                             java.lang.String endpoint)  
throws java.rmi.RemoteException
```

Operation to create an crypto HID providing a certificate reference (from a previously created cryptoHID) and an endpoint The crypto HID is the enhanced version of HIDs, that allow to store persistent information on them (through certificates) and

Parameters:

`certRef` - The certificate reference from a previously generated cryptoHID.
`endpoint` - The endpoint of the service (if there is a service behind).

Returns:

The `String` representation of the HID.

Throws:

`java.rmi.RemoteException`

8.3 Device Application Catalogue

Device Application Catalogue (DAC) is in charge to keep track of all devices and gateways that are available in the Linksmart network. The DAC is deployed in a gateway and communicate with other DAC:s to exchange the list of local devices (devices that are discovered in the local network).

8.3.1 The Graphical Browser

A fundamental part in every LinkSmart-based application is the Device Application Catalogue, which is managed by the Application Device Manager, as was explained in previous chapters. This is a runtime component that keeps track of and manages all devices that are currently active within an application. The LinkSmart Device Application Catalogue serves all LinkSmart middleware managers with the information and metadata they need regarding devices, their services, and their status.

The LinkSmart uses the LinkSmart Device Ontology and models for discovery to recognise new devices when they enter into a LinkSmart network. Based on the discovery model it queries the Device Ontology to deduce what type of device has entered the network. The LinkSmart can be queried by different middleware managers to retrieve a service interface for different devices.

A LinkSmart browser has been developed to allow a user/developer to graphically browse the LinkSmart network and inspect properties and services of devices. The browser tool also allows the user to invoke the different services offered by devices.

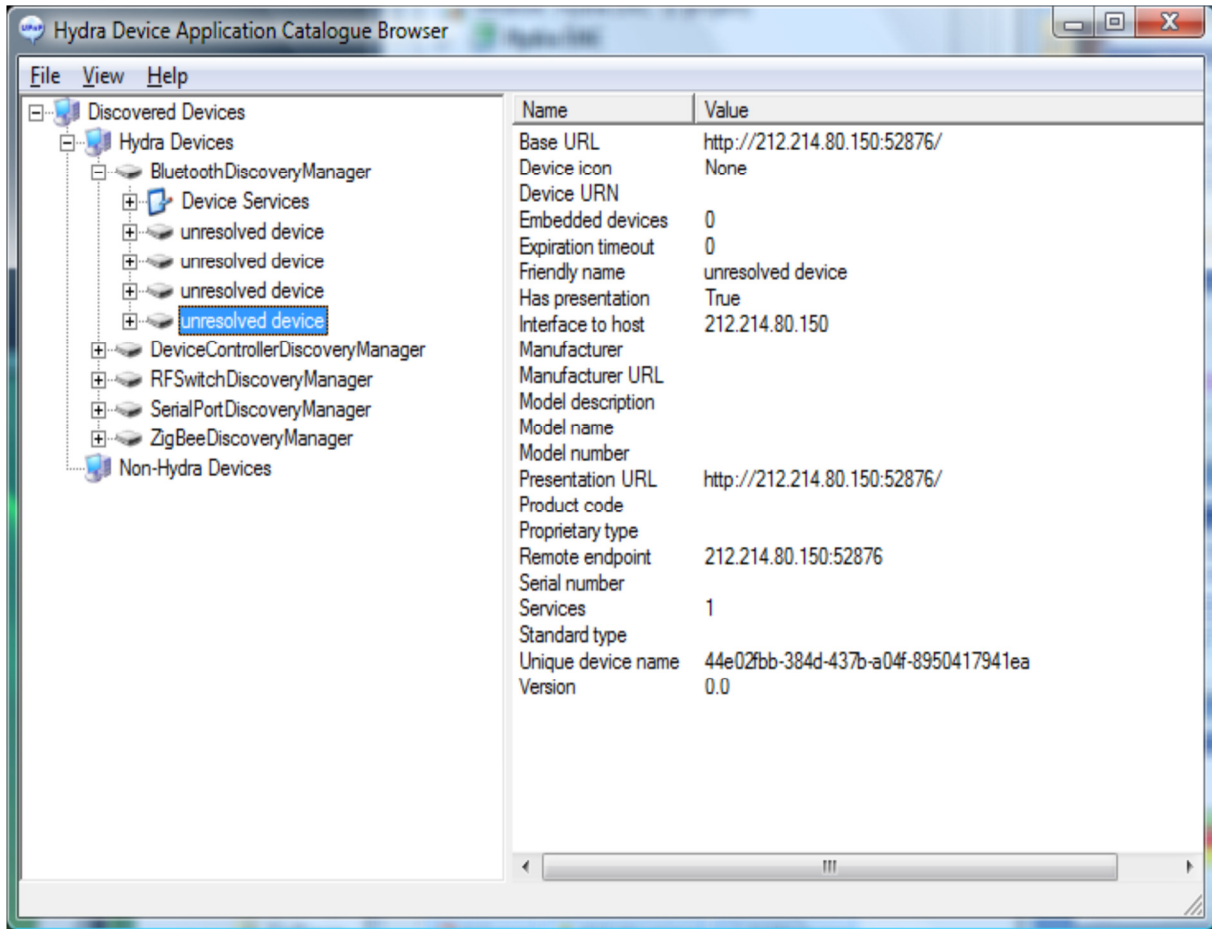


Figure 17: The LinkSmart Browser

By manually invoking the different services, the actual role the Device Application Catalogue plays in the LinkSmart middleware can be illustrated. As can be seen in Figure 17 above, 5 different Discovery Managers are available in the network, each of them is dedicated to discover a certain type of physical device (Bluetooth, RF Switches, ZigBee etc).

Each Discovery Manager keeps track of the device it has discovered and tries to elicit as much information as possible from the device. All this physical discovery information can be accessed by calling the service "Get Device Physical Discovery".

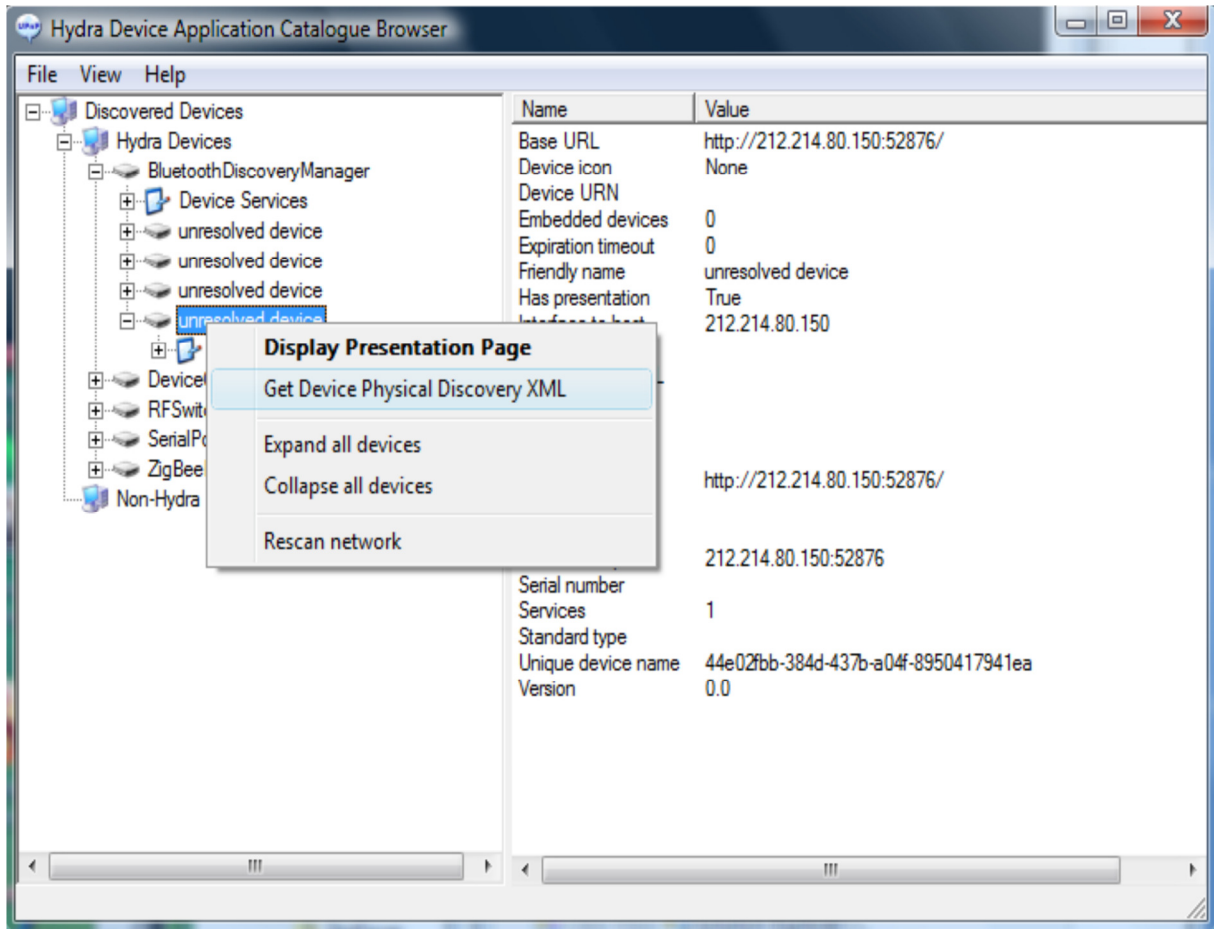


Figure 18: Retrieving discovery information from the physical device

This discovery information is returned as an XML document, which can be seen in the figure below:

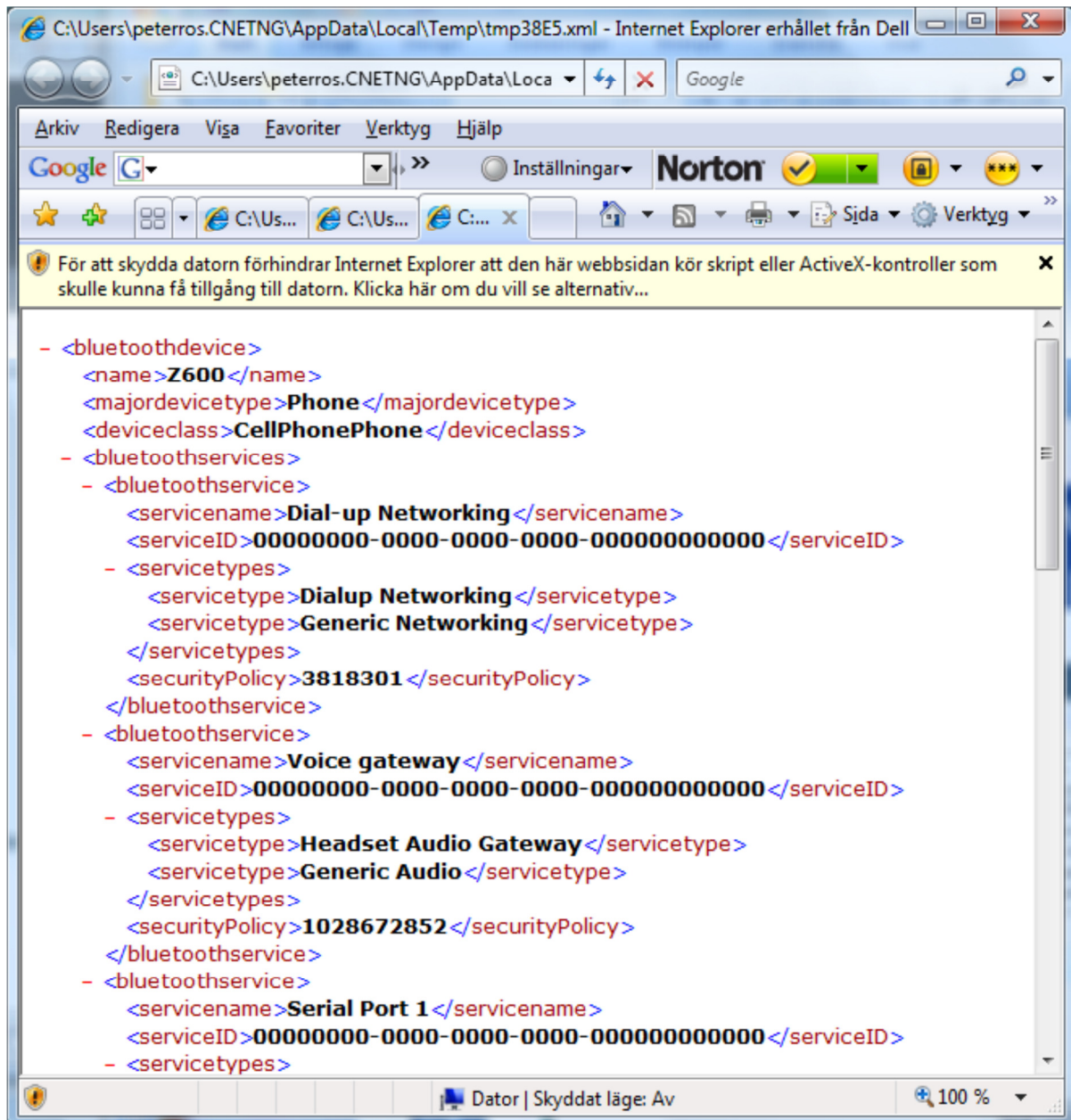


Figure 19: Discovery information from a Bluetooth Device

In Figure 19 we can see that it is a Bluetooth Device that has been discovered, it has the Bluetooth Major DeviceType "Phone" and Minor DeviceType "CellPhonePhone" (Major DeviceType and Minor DeviceType are part of the Bluetooth standard).

The Bluetooth Discovery Manager has also managed to extract the different Bluetooth services offered by the device. This discovery information can now be used to reason about what type of device has been discovered. The physical discovery XML is given to the Device Ontology which deduces that this device corresponds to a "Basic Phone" in the LinkSmart Device Ontology.

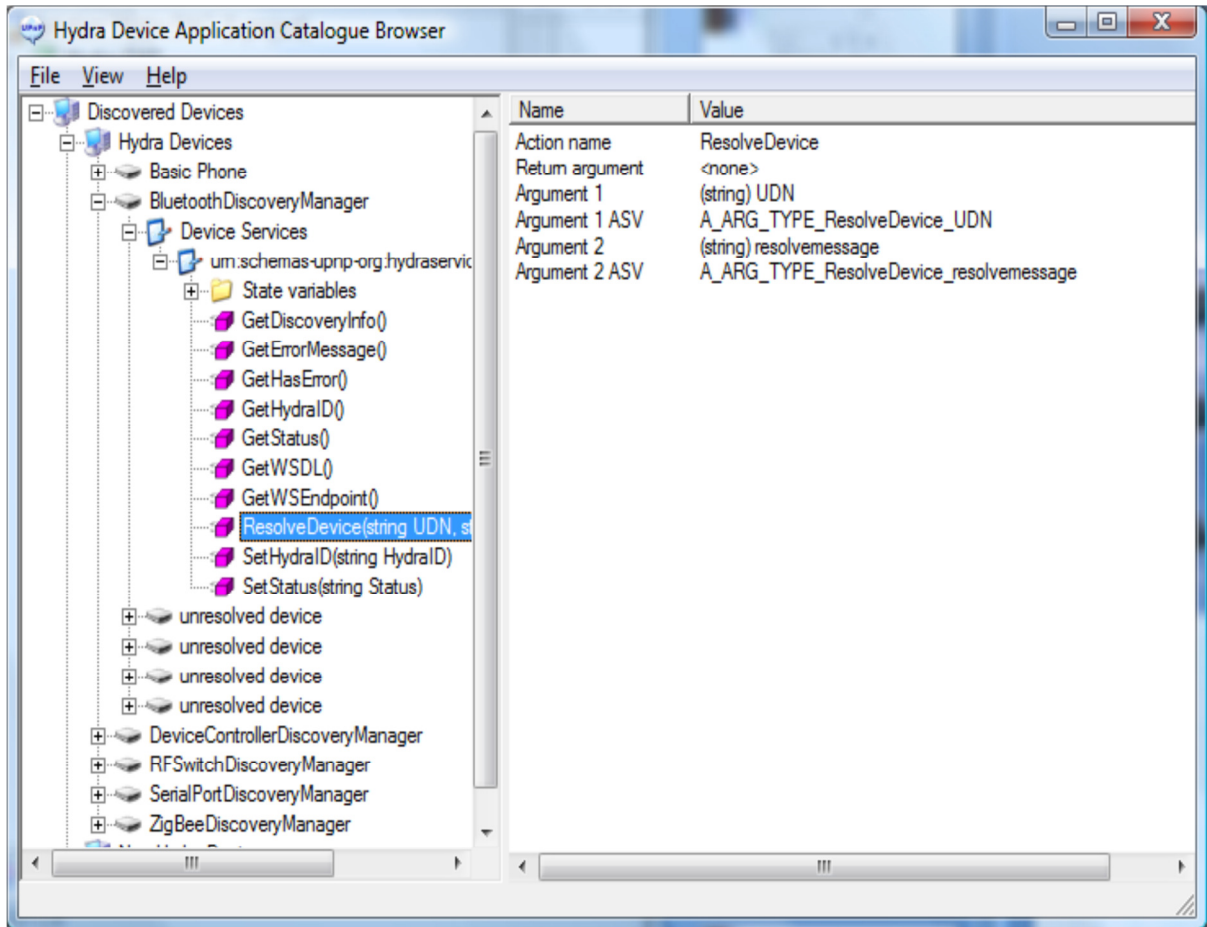


Figure 20: Resolving a physical device into a LinkSmart Device.

By invoking the service “Resolve Device” the Bluetooth Discovery Manager can be told that this is a “Basic Phone”. The idea is of course to do this programmatically, but here it is done manually for illustration purposes.

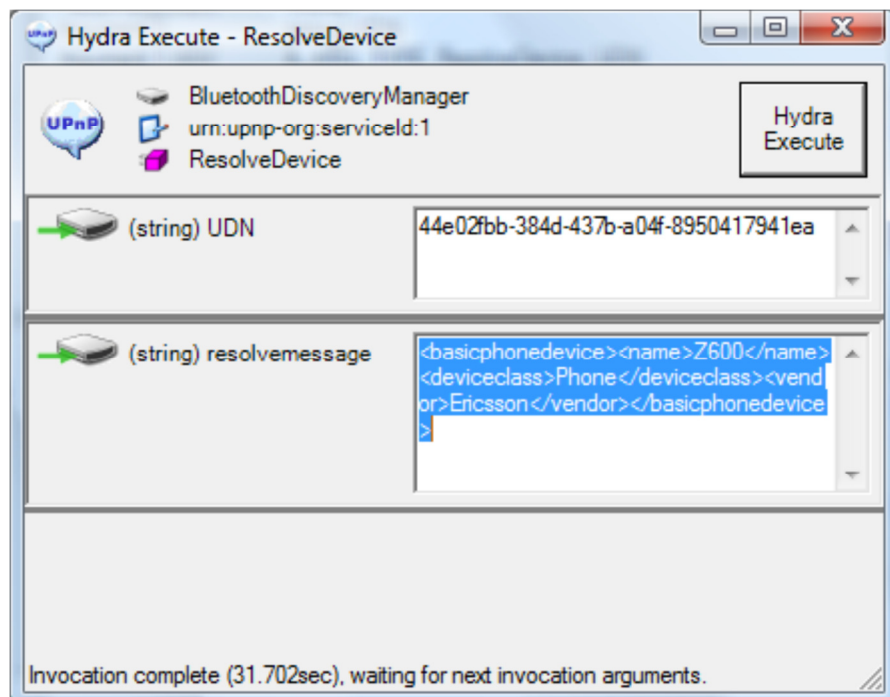


Figure 21: Resolve information is sent as an XML structure to the Discovery Manager

The Discovery Manager then creates and publishes the Device to the network as a "Basic Phone" device. The Basic Phone device is now available together with the services offered by a Basic Phone (in this case a set of SMS read/send functions).

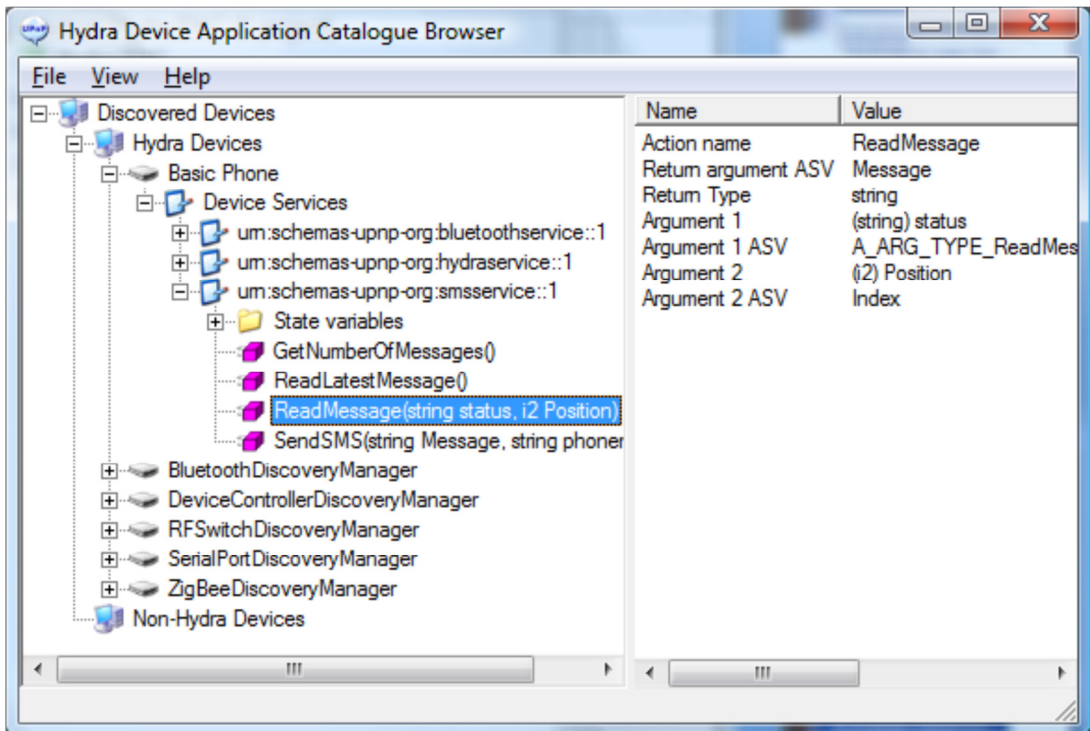


Figure 22: A physical device with unknown functionality has been transformed into Basic Phone Device with services for reading/sending SMS

These services are now directly invocable from the Browser, and for instance, an SMS can be sent.

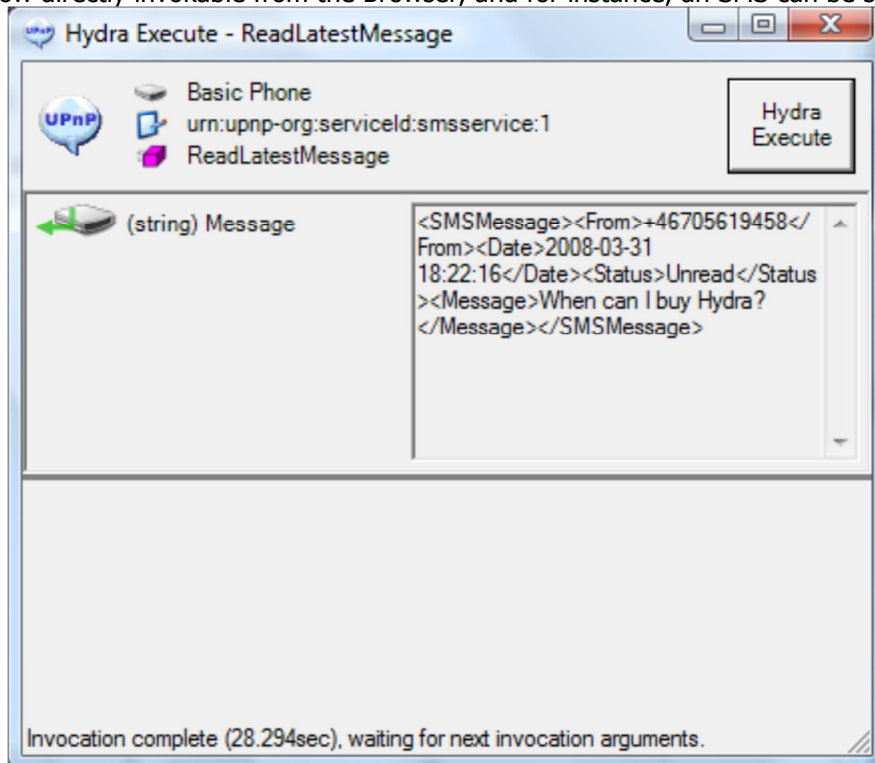


Figure 23: Sending an SMS through the Basic Phone Device

Finally the Browser can be used to retrieve a service description for a web service that allows us to access the device programmatically:

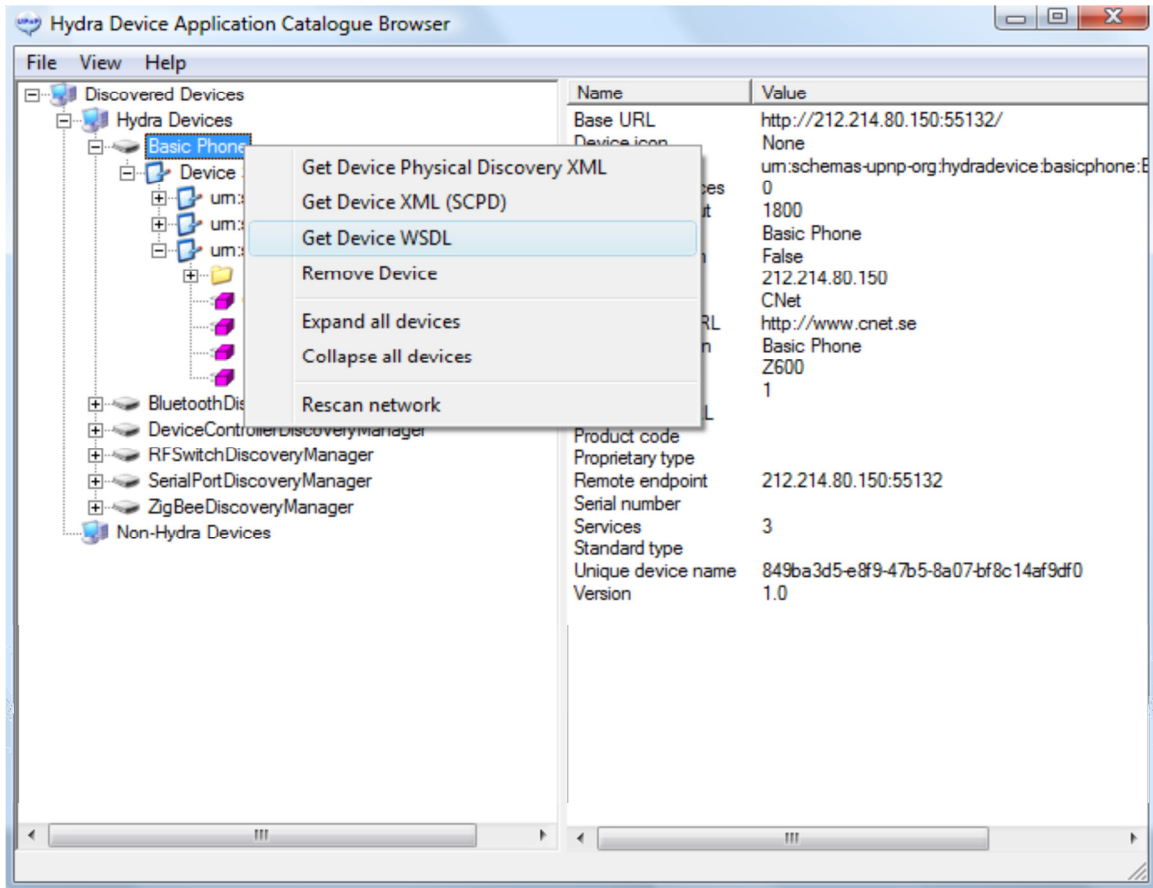


Figure 24: Using the DAC browser to retrieve a WSDL description for the device.

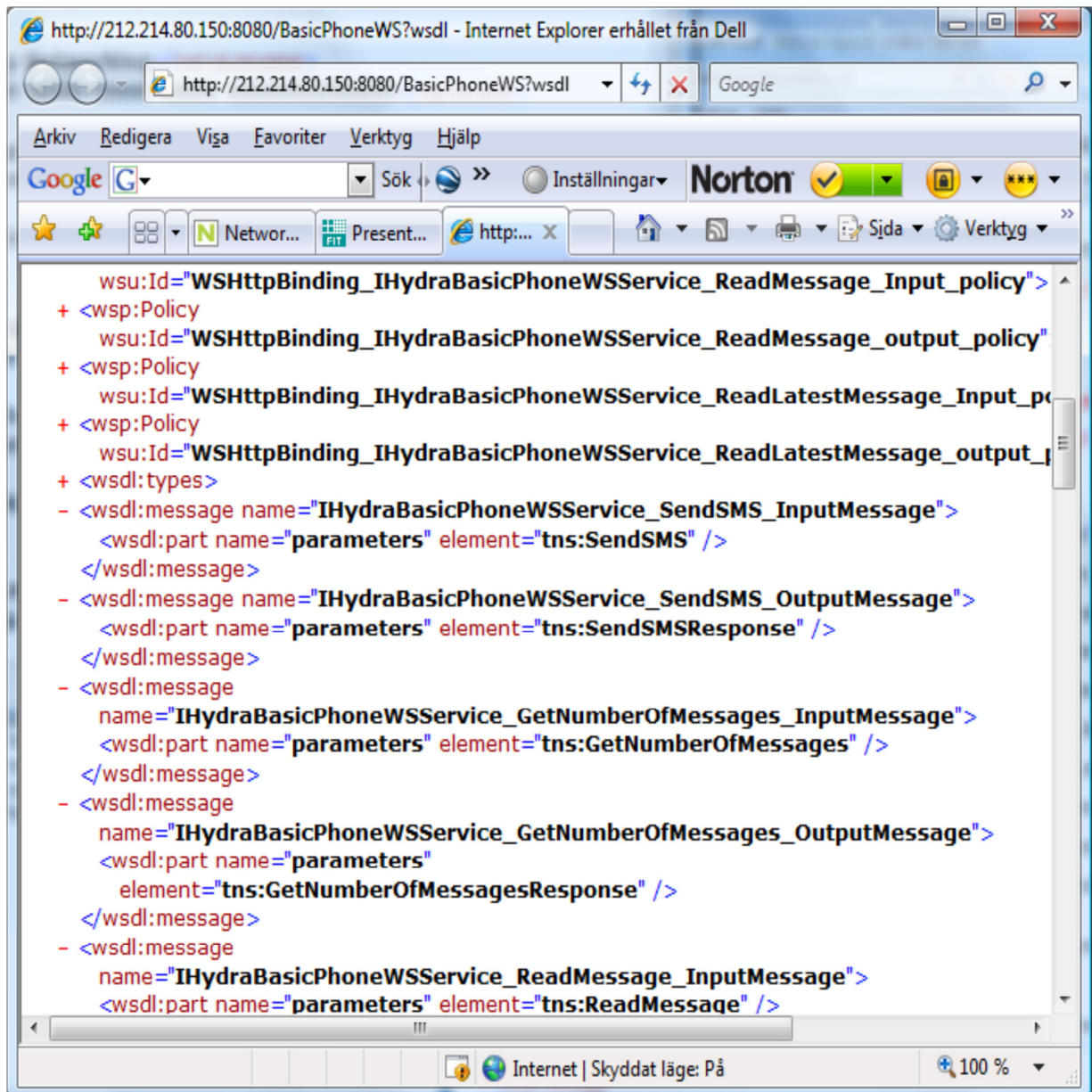


Figure 25: A WSDL (Web Service Description Language) for the device.

8.4 Discovery Manager (Framework)

LinkSmart implements a 3-layered discovery architecture – **physical**, **network** and **semantic discovery**, see the figure below.

In short the 3-layered discovery architecture works this way: First physical devices are discovered using native discovery protocols such as Bluetooth. Then LinkSmart middleware (Discovery Manager) creates a software wrapper that allows further extraction of metadata from the device and makes it available in a LinkSmart network. Finally the device ontology is used to fully resolve what type of device and what kind of functions it has and how the service interface looks like.

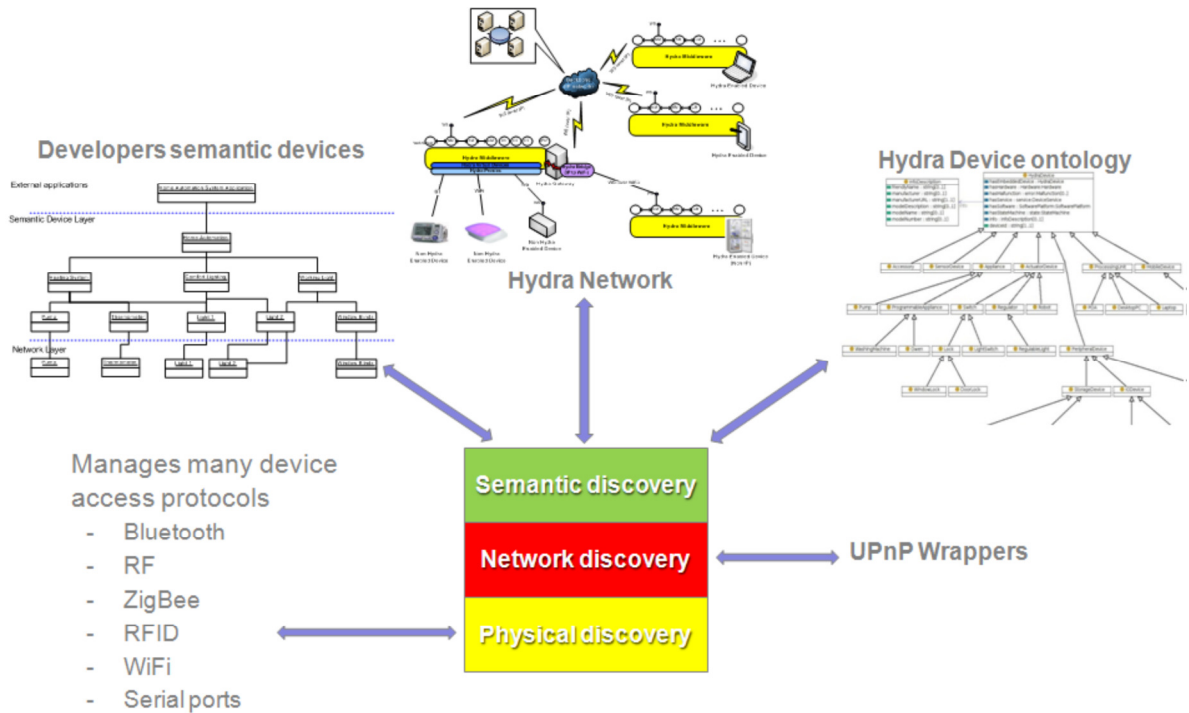


Figure 26: 3-layered discovery architecture in LinkSmart.

8.4.1 Physical Discovery

At the lowest level the LinkSmart project is developing techniques for the discovery at the physical level. This will allow us to discover devices using communication protocols like Bluetooth, ZigBee, WiFi etc. Each of these protocols is handled by a specific Discovery Manager.

The Discovery Manager is part of the implementation (a sub-manager) of the Application Device Manager. This (sub-) manager also implements the base class for all protocol specific discovery managers in LinkSmart. A discovery manager keeps track of the devices it has discovered. As long as the devices are unresolved they are treated as Embedded devices of the Discovery Manager. A discovery manager runs locally on a gateway/PC where it looks for remote devices such as Bluetooth devices.

The following discovery managers exist with interfaces available:

- Bluetooth Discovery Manager
- SerialPort Discovery Manager
- RFSwitch Discovery Manager
- ZigBee Discovery Manager
- UPnP Discovery Manager
- RFID Discovery Manager
- External Discovery Manager

The External Discovery Manager now supports discovery of devices over the P2P architecture.

8.4.1 Network discovery based on UPnP

Once a device has been discovered at the physical level it needs to be discovered at a network level. This is done by creating a UPnP (Universal Plug and Play) wrapper to represent the device on the network. The UPnP wrapper then allows the device to be discovered at a network layer.

The UPnP (Universal Plug and Play) architecture offers pervasive peer-to-peer network connectivity of PCs, intelligent appliances and wireless devices. The UPnP architecture is a distributed, open networking

architecture that uses TCP/IP and HTTP. It enables seamless proximity networking in addition to data transfer between networked devices at home, in the office and everywhere in between.

It enables data communication between any two devices under the command of any control device in the network. UPnP has a number of characteristics:

- Media and device independence. UPnP technology can run on any medium including phone lines, power lines, Ethernet, IR (IrDA), RF (WiFi, Bluetooth), and FireWire. No device drivers are used; common protocols are used instead.
- Common base protocols. Base protocol sets (Device Control Protocols, DCP) are used, on a per device basis.
- Operating system and programming language independence. Any operating system and any programming language can be used to build UPnP products. UPnP does not specify or constrain the design of an API for applications running on control points. OS vendors may create APIs that suit their customer's needs. UPnP enables vendor control over device UI and interaction using the browser as well as conventional application programmatic control.
- Internet-based technologies. UPnP technology is built upon IP, TCP, UDP, HTTP, SOAP and XML, among others.
- Programmatic control. UPnP architecture also enables conventional application programmatic control.
- Extensibility. Each UPnP product can have value-added services layered on top of the basic device architecture by the individual manufacturers.

The UPnP architecture supports zero-configuration, invisible networking and automatic discovery for a breadth of device categories from a wide range of vendors. Devices can dynamically join a network, obtain IP addresses, announce their names, convey their capabilities upon request, and learn about the presence and capabilities of other devices. DHCP and DNS servers are optional. A device can leave a network smoothly and automatically without leaving any unwanted state information behind.

8.4.2 External Discovery

External discovery enables LinkSmart gateways to locally represent all LinkSmart devices in the LinkSmart network even if they reside in a different physical network. This enables the developer to build applications that use devices in exactly the same way independently of their network location.

The basis for the external discovery process is synchronisation of information in-between the Application Device Managers in the network. For each of the found external LinkSmart devices a local device proxy is created using the SCPD of the external device. This will also copy all of the LinkSmart UPnP properties for the device such as the HIDs for the different device services.

The external discovery follows the following procedure:

1. Contact Network Manager to find all Device Application Managers in the network
2. Contact each of the Application Device Managers to retrieve a list of their local devices
3. Contact each device and use the generic LinkSmart Web Service to retrieve the device XML (SCPD)
4. For each device create a local device proxy using the device XML.

8.4.3 Semantic Discovery

Once the device is discovered as part of the network, it needs to be discovered semantically, i.e., the device needs to be related to the LinkSmart Device Ontology so that it is known what kind of device has been discovered.

LinkSmart uses two different XML structures to describe a device and its capabilities. First there is the device description, which contains various metadata regarding the device such as its type, the manufacturer, model etc. An example of device description is shown below:

```
<device>
<deviceType>urn:schemas-upnp-org:device:waterPump:1</deviceType>
<friendlyName>GrundfosPump</friendlyName>
<manufacturer>Grundfos</manufacturer>
<manufacturerURL>http://www.grundfos.com</manufacturerURL>
<modelDescription>Pump</modelDescription>
<modelName>Grundfos Magna</modelName>
<modelName>X1</modelName>
<UDN>uuid:dac824ab-bca1-4d5c-93c5-578a0c697ba1</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnporg:
service:grundfosPumpService:1</serviceType>
<serviceId>urn:upnp-org:serviceId:grundfosPumpService</serviceId>
<SCPDURL>_grundfosPumpService_scpd.xml</SCPDURL>
<controlURL>_grundfosPumpService_control</controlURL>
<eventSubURL>_grundfosPumpService_event</eventSubURL>
</service>
</serviceList>
</device>
```

Secondly, there is the SCPD (Service Control Point Description), which describes the capabilities of the device and how to invoke its different services. An example of service description is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
<specVersion>
<major>1</major>
<minor>0</minor>
</specVersion>
<actionList>
<action>
<name>GetStatus</name>
<argumentList>
<argument>
<name>ResultStatus</name>
<direction>out</direction>
<relatedStateVariable>Status</relatedStateVariable>
</argument>
</argumentList>
</action>
<action>
<name>SetTarget</name>
<argumentList>
<argument>
<name>newTargetValue</name>
<direction>in</direction>
<relatedStateVariable>Target</relatedStateVariable>
</argument>
</argumentList>
</action>
</actionList>
<serviceStateTable>
<stateVariable sendEvents="yes">
<name>Status</name>
<dataType>boolean</dataType>
</stateVariable>
<stateVariable sendEvents="no">
<name>Target</name>
<dataType>boolean</dataType>
</stateVariable>
</serviceStateTable>
</scpd>
```

A final part of the semantic discovery is the service discovery task to find a suitable service provided by specific device (or device type) in accordance to defined requirements. In the context of LinkSmart, the service discovery task defined this way can be used in various cases, for example:

- From a developer user point of view: to find the required service provided by specific device in the process of development of basic communication patterns, such as composed (or orchestrated) services, choreography interfaces or service user interfaces.
- From a system or application point of view: to find the required service provided by specific device when executing the complex process requiring the service orchestration.
- Tools and matchmakers exist supporting the service discovery for both OWL-S and WSMO standards (description of this tools is out of scope of this deliverable), which may be used for particular approach.

In LinkSmart support for SAWSDL annotations is provided. As the SAWSDL approach does not explicitly support service discovery, there are two basic possibilities, which can be used in this case:

- The Service discovery process is realized by searching the SAWSDL according to provided semantic annotations.
- Using the annotations in SAWSDL file, the model of service is annotated in the LinkSmart service ontology and the discovery process is realized by matching the ontology concepts in accordance to specified requirements, similarly as in OWL-S/WSMO approach.

8.5 Event Manager

The LinkSmart Event Manager provides 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 specific variant of publish/subscribe implemented is topic-based publish/subscribe where event are key/value pairs.

The Event Manager is deployed as a service in the LinkSmart network and implements the following interface:

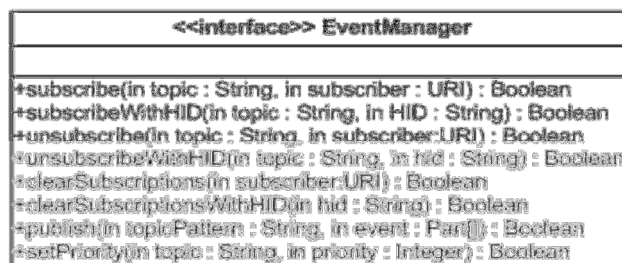


Figure 27: Event Manager Interface

Interaction with the Event Manager can be performed using this service, by creating an Event Manager client handling the calls to the Event Manager. To publish an Event to the Event manager, the *publish* method must be called, passing the *topic* of the Event, as well as an array of key-value *Part* objects, that specify additional data associated with the Event.

```

EventManagerPort em = { Get Event Manager Client };
Part[] parts = { Get Part Array };
em.publish("ExampleTopic", parts);

```

The code snippet above gives an example of using the EventManagerPort interface to publish an Event. An application can subscribe to receive notifications of events by calling the *subscribe* or *subscribeWithHID* methods. These methods take the *topic* of the events being subscribed to, along with callback information, such the the Event Manager can send notifications when the events are published. With the *subscribe* method, this information is provided as a Web Service endpoint address, whereas the *subscribeWithHID* method takes the HID of the subscriber, to then call back through the LinkSmart Network Manager.

```
EventManagerPort em = { Get Event Manager Client };
em.subscribeWithHID("ExampleTopic", <Subscriber HID> );
```

The above code snippet gives an example of a subscriber subscribing to the Event Manager. Furthermore, subscribers must implement the following interface:



The figure below shows the resulting deployment:

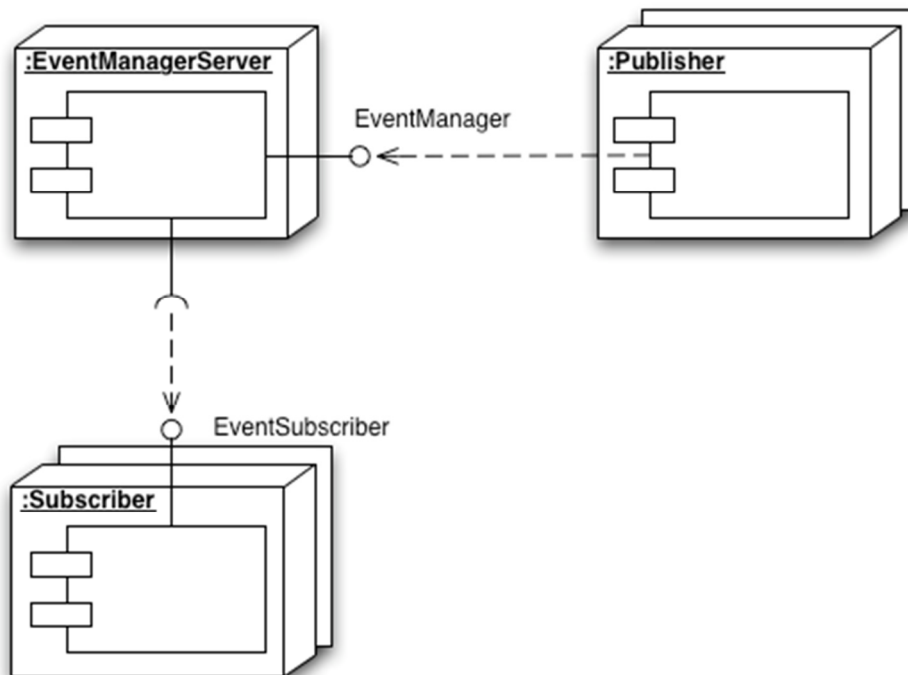


Figure 28: Event Manager Deployment

Given such a deployment, the figure below shows a typical interaction with the EventManagerServer (where address is the address of the Subscriber web service that later should be notified):

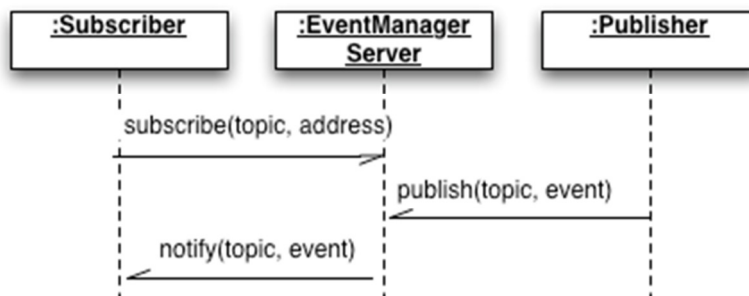


Figure 29: Subscriber Notification

8.6 Access Control Policy Framework

The Policy Framework provides policy-driven, access-control protection for LinkSmart devices and applications. Policies can be utilised to ensure access to devices and applications is limited only to those permitted access, including the ability to restrict the level of discoverability of an end-user's devices and applications.

The Policy Framework, consisting of its various components, provides the functionalities to create, update, and maintain Policies, in addition to its core of evaluating access requests, and enforcing the decisions made. The SDK functionality of the Access Control Policy Framework comes with three distinct interfaces, as well as another interface for extension, these being:

- Policy Enforcement Point
 - Called at the point of interception of a request
 - Formats the credentials of the request in to an XACML RequestCtx object, and calls the PDP for a decision
 - Enforces the returned decision, handling any obligations specified in the Policy
- Policy Decision Point
 - Receives the XACML RequestCtx object from the PEP
 - Analyses the request against the policies stored in its policy repository
 - Returns the determined decision
- Policy Administration Point
 - Interface exposed by the PDP for the administration of XACML policies
 - Active / deactivate XACML policies
- Policy Information Point
 - Extension interface for PDP
 - Adds functions for the PDP to use when they are referred to in XACML policies

8.6.1 Policy Enforcement Point

Typically, in the context of communication in LinkSmart, the Access Control Policy Framework is used to provide access control at the level of the Network Manager, such that access decisions can be made on receiving a request, through the SOAP Tunnel, for a hosted service, before actually forwarding the payload of the request to the endpoint service. The Policy Enforcement Point (PEP), therefore, is utilised by the Network Manager, when it receives a call, forwarding the various credentials it has of the request, to the Policy Enforcement Point.

Although the PEP itself doesn't expose a service to the LinkSmart Network, it does register itself with an HID and certificate, such that it can be identified as being a PEP. The SID of the PEP is as follows:

SID = com.eu.LinkSmart.policy.pep

Configuration of the PEP bundle specifies the following important configurations:

- **Pep.PID** = The PID of the registered PEP service
- **Pep.PdpPID** = The PID of the PDP service that the PEP should use to retrieve an access decision

The PEP exposes a couple of methods also, used by the Network Manager, to pass the credentials of a request. These are:

```
public PepResponse requestAccessDecision(String senderHid,
                                         String senderCert,
                                         String receiverHID,
                                         String receiverCert,
                                         String soapMsg,
                                         String sessionId);
```

```
public PepResponse requestAccessDecisionWMethod(String senderHID,
                                               String senderCert,
                                               String receiverHID,
                                               String receiverCert,
                                               String method,
                                               String sessionId);
```

Both methods request an access decision, but for different contexts. The Network Manager uses the *requestAccessDecision* method, passing on the complete SOAP Message received from which the PEP extracts the credentials of the action to be performed, whereas the *requestAccessDecisionWMethod* method passes the name of the method directly instead. The *senderCert* and *receiverCert* arguments are the encoded CryptoHID certificates for the two entities involved at either end of the request.

8.6.2 Policy Decision Point

The Policy Decision Point (PDP) is a manager on the LinkSmart network that registers two different services, one for the process of access requests, returning a decision, and another for the administration of the XACML policies that the PDP uses in these decision making processes. This administration service is described in the next chapter.

The *PolicyDecisionPoint* interface, of the LinkSmart Middleware API, declares just one single method, for the evaluation of XACML RequestCtx's, as follows:

```
public String evaluate(String requestXml);
```

This method takes the RequestCtx object, encoded as a String, and evaluates it against the set of policies in the policy repository. The ResponseCtx, containing the decision made along with any obligations with the decision, is returned as encoded XML.

The SID of the PDP service is:

SID = com.eu.LinkSmart.policy.pdp

The PDP has minimal configuration, using the configurator, as follows:

- **PdpService.PID** = PID of the PDP
- **Pdp.UseDatabase** = true / false depending on whether XMLDB based storage or file-based storage is to be used for XACML policies

8.6.3 Policy Administration Point

The interface for actually authoring XACML policies is part of the IDE, and discussed in the chapter 7.7. It uses the interface exposed by the PDP, that is distinctly separated from the service performing the decision functionality, as described in the previous chapter. It exposes the following methods as the *PdpAdmin* interface of the Middleware API:

```
public boolean activatePolicy(String policyId);
public boolean deactivatePolicy(String policyId);
public String[] getActivePolicyList();
public String[] getInactivePolicyList();
public boolean publishPolicy(String policyId, String policyXML);
public boolean removePolicy(String policyId);
public String getPolicy(String policyId);
```

The key method involved here, is the *publishPolicy* method that publishes the policy with the given id, *policyId*, with the XML-encoded content provided with the *policyXML* argument.

The SID of the PDP service is:

SID = com.eu.LinkSmart.policy.pap

The PAP has even less configuration than the PDP, using the configurator, as follows:

- **PdpAdminService.PID** = PID of the PDP Administration service

8.6.4 Policy Information Point

The LinkSmart PDP is designed to be extensible, to easily allow for new functionality to the PDP through adding additional Policy Information Point (PIP) components, which includes the ability to resolve certain attributes, add additional functions that can be used in policies, add new data types, and so on.

PIPs are implemented as OSGi bundles that register services, recognised by the PDP, that it uses to extend the functionality, adding new Functions and Attribute Finders to the PDP at runtime. These interfaces are *PipFunction*, and *PipModule*.

PipFunction provides a method that the PDP can use to retrieve the custom XACML Functions (com.sun.xacml.cond.Function) that it then installs to the PDP Function factory, such that they are then immediately available for use. Therefore, the interface is simply:

```
public interface PipFunction {  
  
    public Set<Function> getFunctions();  
}
```

Implemented *PipModule* components define XACML AttributeFinders, that can retrieve attributes that are not available in the request, but are specified in an XACML policy. The *PipModule* itself is essentially just an extension of the AttributeFinderModule defined by the XACML 1.x implementation by Sun, providing a service name unique to the LinkSmart Access Control Policy Framework. Therefore, the *PipModule* interface is:

```
public abstract class PipModule extends AttributeFinderModule {  
  
}
```

9. Creating a Basic Linksmart Application

In this chapter we describe the basic steps to create a Linksmart application in a .net environment, showing how the Linksmart SDK is integrated into the Visual Studio development environment.

9.1 Creating a Linksmart application from a template

In Visual Studio start by selecting "New Project":

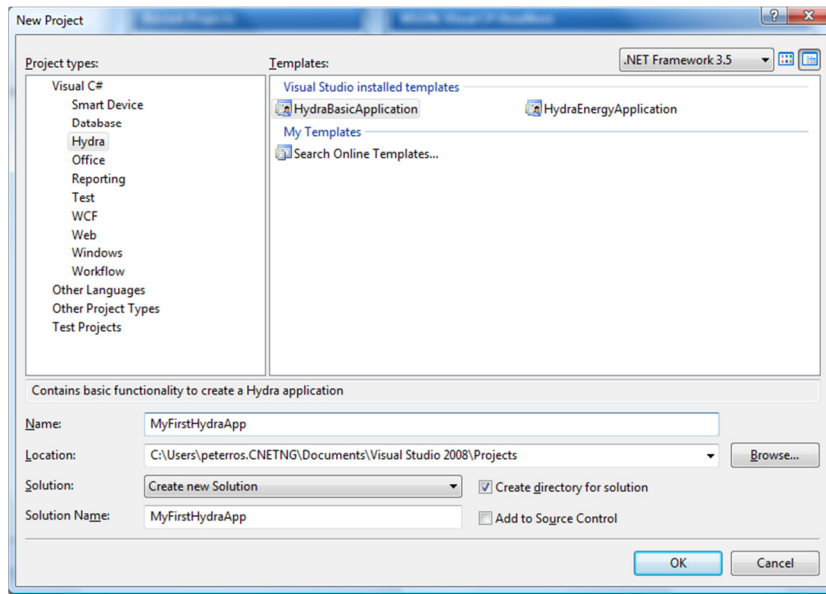


Figure 30: Template view in Visual Studio

Under the Visual C# menu the Linksmart category appears. Select the type of Linksmart application you will develop, for instance a basic Linksmart Application.

Once you have selected the type of application and click OK, Linksmart creates the necessary project files for you:

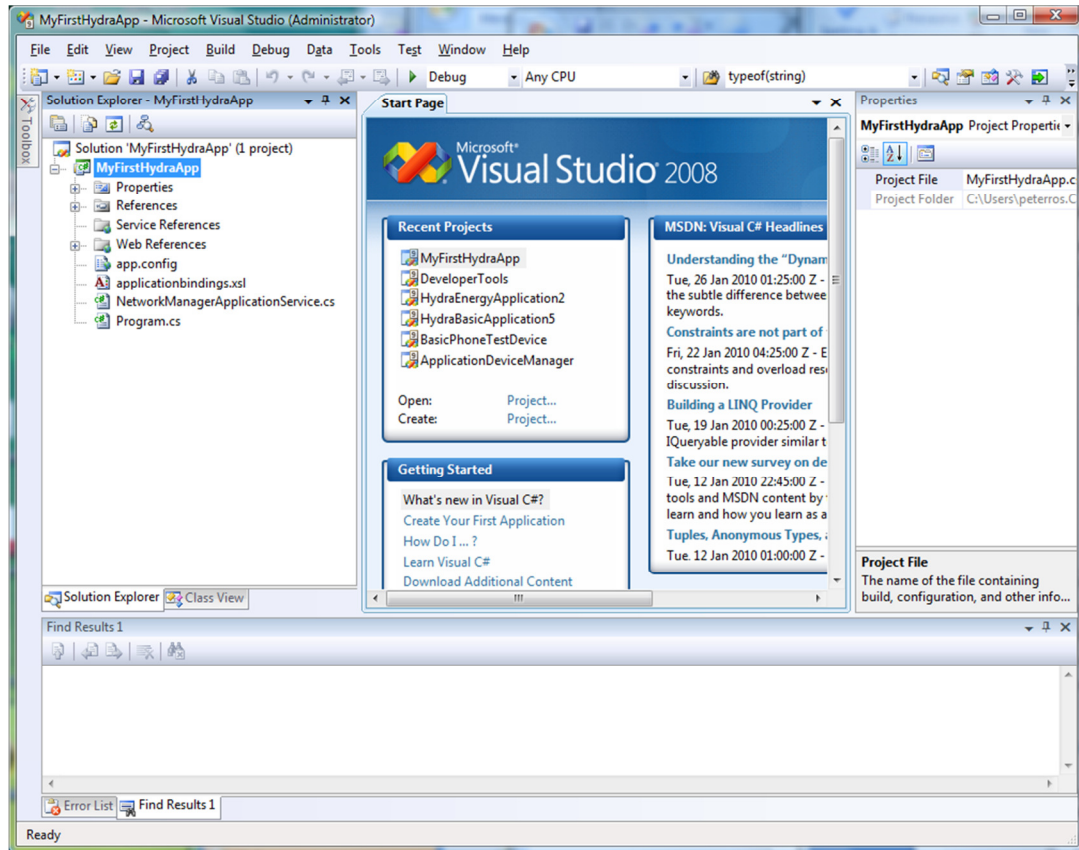


Figure 31: Auto generated files for Basic Linksmart Application

The following files and references are automatically created:

- Your main program file named "program.cs"
- A rule file for binding your devices to identifiers (PIDs). This file is called applicationbindings.xsl
- A Web Reference to the Application Device Manager
- A Web Reference to the Network Manager (in file networkmanagerapplicaitonsservice.cs)
- A Web Reference for creating WS clients for accessing basic IoT devices

9.2 Initiating the Network Manager

The first step in any Linksmart application is to initiate the Network Manager in order to be able to communicate with other Linksmart Managers and devices. This is done in the method

```

SetUpNetworkManager:
void SetUpNetworkManager(string url)
{
    m_networkmanager = new NetworkManagerApplicationService();
    m_networkmanager.Url = url;
    System.Net.ServicePointManager.Expect100Continue = false;
}

```

9.3 Initiating the Application Device Manager

The next step is to initiate the Application Device Manager. There are three things you need to do to initiate the Application Device Manager:

- Retrieve the Linksmart ID for the Application Device Manager from the Network Manager
- Use the HID to create an endpoint URL for the Application Device Manager
- Load your device bindings into the Application Device Manager (if you don't provide a bindings file the Application Device Manager, will use default ways of making bindings instead).

```
void SetUpApplicationDeviceManager(string gateway, string endpoint, string appname)
{
    m_applicationdevicemanager = new ApplicationDeviceManager.ApplicationDeviceManager();

    //Use NM to find HID for Application Device Manager
    string AppDevMgrHID =
m_networkmanager.getHIDsbyDescriptionAsString("ApplicationDeviceManager:" + gateway +
":StaticWS");
    string[] DacHIDs = AppDevMgrHID.Split(' ');
    AppDevMgrHID = DacHIDs[0].Trim();
    m_applicationdevicemanager.Url = endpoint + "/SOAPTunneling/0/" + AppDevMgrHID";
    try
    {
        //check if bindingfile is correct xml before sending to application device manager
        XmlDocument myDoc = new XmlDocument();
        string bindingrules = "";
        myDoc.Load("applicationbindings.xml");
        bindingrules = myDoc.OuterXml;
        m_applicationdevicemanager.AddApplicationBinding(appname, bindingrules);
    }
    catch (Exception e)
    {
    }
}
```

Figure 32: Initiating the Application Device Manager

At this point we have established a connection to a NetworkManager and we have our DAC initiated.

9.4 Working with devices

Once you have initiated Network Manager and Application Device Manager you can start working with devices. To use a IoT Device in your application you start by creating a Web Service client for it. Use your web reference, IoTDevice (found in the Solution Explorer window).

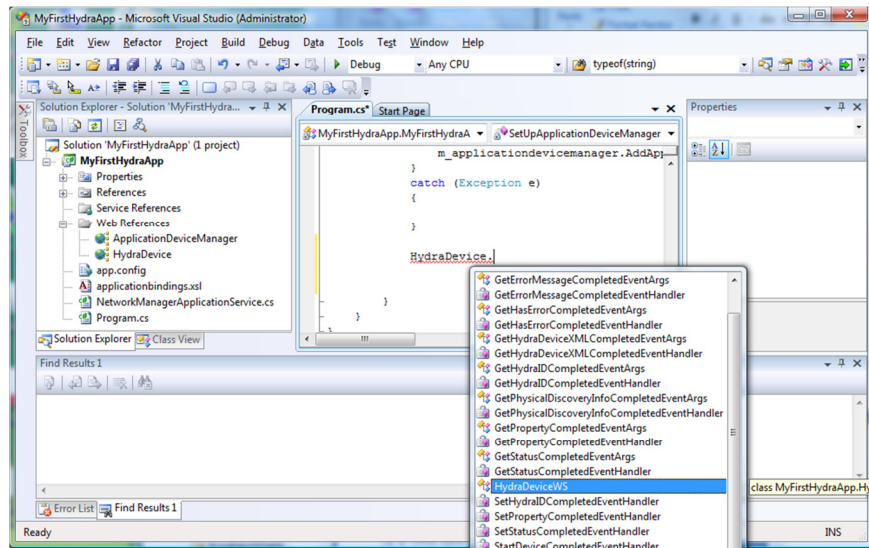


Figure 33: Creating WS clients for device

Now you need a Linksmart identifier to create an endpoint URL for the device (assuming you have the base url in the variable "endpoint") which we assume you have assigned the identifier "PetersPhone" in your application bindings file:

```
IoTDevice.IoTDeviceWS myIoTDevice = new IoTDevice.IoTDeviceWS();

string myhid=m_applicationdevicemanager.GetHID("", "PetersPhone");

if (myhid != "")

    myIoTDevice.Url = endpoint + "/SOAPTunneling/0/" + myhid + "/0/hola";
```

Once you have established a url for the device you can now start consuming its IoT Services. In this example we are only working with devices at a generic level, as IoT Device and therefore only have access to meta data services like "GetDeviceXml":

```
string myXml = myIoTDevice.GetIoTDeviceXML();
```

9.5 Applications Bindings

The application bindings file (*applicationbindings.xml*) is used to assign persistent and context dependent identifiers to devices.

The bindings are expressed as a set of xslt rules over the IoT Device XML (detailed in paragraph 10.4).

```
<binding>
```

```
<xsl:template match="upnp:device">
```

```
.....
```

```
<xsl:if test="upnp:deviceType='urn:schemas-upnp-org:iotdevice:basicswitchdevice:1' or ...
```

```
<xsl:if test="upnp:friendlyName='DiscoBall'">
```

```
<hydraUDN>DiscoBall</hydraUDN>
```

```

    <locationdata>
      <building>CNet Office</building>
      <room>Main</room>
      <position>Table</position>
    </locationdata>
  </xsl:if>

  <xsl:if test="upnp:friendlyName='PetersLight' and hydra:gateway='DELL1'">
    <IoTUDN>DemoLight</IoTUDN >
    <locationdata>
      <building>CNet Office</building>
      <room>Main</room>
      <position>Table</position>
    </locationdata>
  </xsl:if>

  <xsl:if test="IoT:gateway='Casa Domotica'">
    <hydraUDN><xsl:value-of select="upnp:friendlyName"/></hydraUDN>
    <locationdata>
      <building>Casa Domotica</building>
      <room><xsl:value-of select="upnp:friendlyName"/></room>
    </locationdata>
  </xsl:if>
  .....
</binding>

```

Figure 34: Application Bindings XSL

The *IoTUDN* is the IoT Unique Device Name, which can be derived from any of the properties in the Device XML, though normally it is set to the `upnp:friendlyName`. The binding combines the *IoTUDN* with possible location (context) data, into a PID (Persistent Identifier) for the device. Applying the binding rules to the Device XML results in the specific binding being added to the DAC where it can be used by the application code.

The developer can define the application bindings by updating the bindings XML file (an associated XML schema supports the editing). In any case the SDK also provides a default binding of devices, based on the upnp:friendlyName and without context data.

10. Creating an Advanced LinkSmart Application

It is expected that one of the most common uses of the LinkSmart middleware will be for monitoring and controlling the energy consumption of physical devices.

10.1 Initiate Application

To create an Energy Application you follow the same steps as before:

- Select New Project
- Select LinkSmartEnergyApplication template

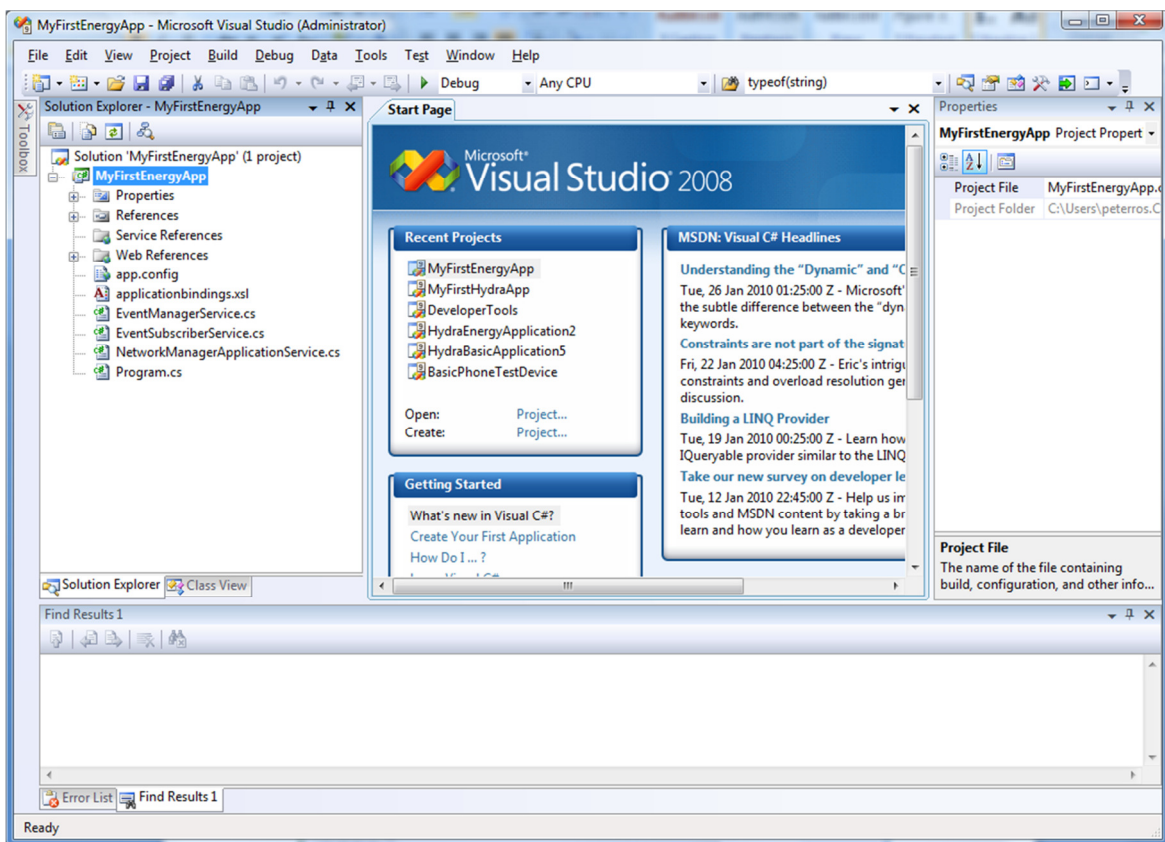


Figure 35: Energy Application Template view

The following files and references are automatically created:

- Your main program file named "program.cs"
- A rule file for binding your devices to identifiers. This file is called applicationbindings.xml
- A Web Reference to the Application Device Manager
- A Web Reference to the Network Manager (in file networkmanagerapplicationservice.cs)
- A Web Reference to the Event Manager (in file eventmanagerservice.cs)
- A Web Reference for creating WS clients for accessing basic IoT devices.
- A Web Reference for creating WS clients for accessing Basic Switch and Enhanced Switch devices.

- Web Reference for accessing the Energy services of IoT devices.

10.2 Searching and finding for devices

Next step is now to access and control some energy consuming devices. As you can see under the "Web References" menu, you now have 2 references to "BasicSwitchDevice" and "EnhancedSwitchDevice".

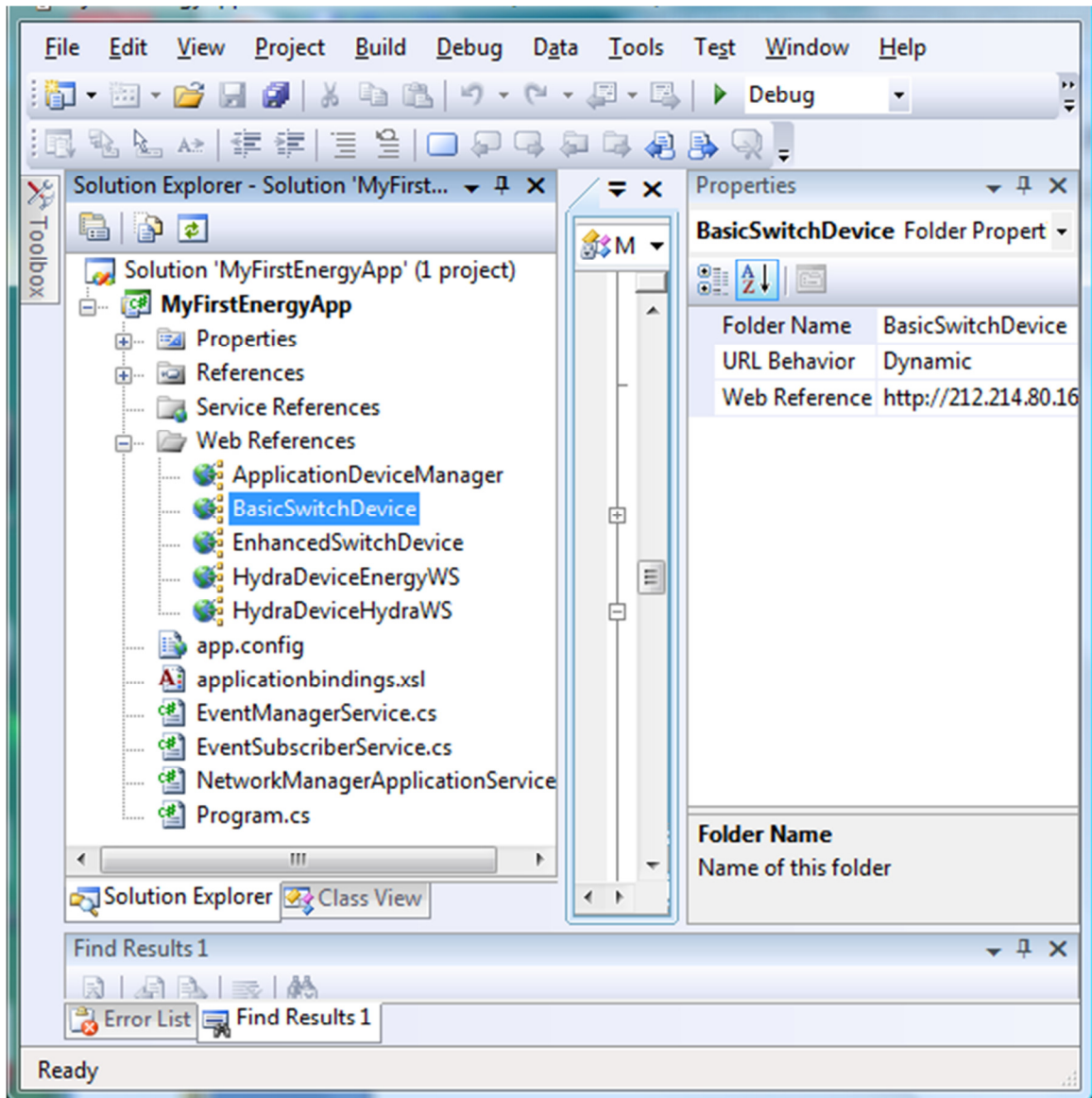


Figure 36: Selecting web references to devices

These can be used to control a particular device, but you need to find the device. You need to setup the Network and Application Device Manager as was described in previous section (the code is already in your program.cs file).

Once this is done, you can query the Application Device Manager to find the devices. To do this you need to be familiar with the LinkSmart Device XML structure and the standard XML query language "XPath".

The following XPath statement will match each device that is of type "basicswitchdevice".

```
"/.*[name()='deviceType' and .='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1']"
```

If you use this statement as input to the method `GetIoTURLsFromXPath` of the Application Device Manager you will get a list of [URLs](#) to all discovered and active devices in a LinkSmart Network. Since a device might expose several web services you need to specify which one you are interested in. In this case it is "IoTidStaticWS". This method is a shortcut compared with retrieving the HID and composing the URL yourself as was done in the previous example.

```
public void TurnOnAllSwitchDevices()
{
    string basicswitches =
m_applicationdevicemanager.GetIoTURLsFromXPath("/.*[name()='deviceType' and .='urn:schemas-
upnp-org:IoTdevice:basicswitchdevice:1']", "IoTidStaticWS", "");
}
```

10.3 Invoking Device Services

Now it is time to start controlling the devices. This code shows an example how you turn on all switches:

```
public void TurnOnAllSwitchDevices()
{
    string basicswitches =
m_applicationdevicemanager.GetIoTURLsFromXPath("/.*[name()='deviceType' and .='urn:schemas-
upnp-org:IoTdevice:basicswitchdevice:1']", "IoTidStaticWS", "");

    char[] splitchar = new char[1];

    splitchar[0] = ',';

    string[] switches = basicswitches.Split(splitchar);

    foreach (string switchurl in switches)
    {
        BasicSwitchDevice.BasicSwitchWS mySwitch = new BasicSwitchDevice.BasicSwitchWS();

        mySwitch.Url = switchurl;

        mySwitch.TurnOn();
    }
}
```

Figure 37: Invoking device services

This example shows how to use the EnergyWS web service to calculate the current total effect for all running devices:

```
public int GetTotalCurrentEffect()
{
    int returnvalue = 0;
```

```

        string basicswitches =
            m_applicationdevicemanager.GetIoTURLsFromXpath(".*[name()='deviceType' and
                .='urn:schemas-upnp-org:IoTdevice:basicswitchdevice:1']", "IoTidEnergyWS", "");

    char[] splitchar = new char[1];

    splitchar[0] = ',';

    string[] switches = basicswitches.Split(splitchar);

    foreach (string switchurl in switches)
    {

        IoTDeviceEnergyWS IoTDeviceEnergyWS mySwitch = new
            IoTDeviceEnergyWS.IoTDeviceEnergyWS();

        mySwitch.Url = switchurl;

        string effectstring = mySwitch.GetCurrentEffect();

        if (effectstring != "")
        {
            returnvalue = returnvalue + System.Convert.ToInt32(effectstring);
        }
    }
}

```

Figure 38: Energy WS call

10.4 Understanding the LinkSmart Device XML

Since all metadata and the state of a device is communicated using an XML structure it is fundamental to understand this structure and how it can be used. Below is an example of the LinkSmart Device XML for a device. The LinkSmart Device XML is an extension of the UPnP SCPD XML (Service Control Point Document) vocabulary. Elements with the namespace "IoT" are the Linksmart-specific extensions.

```

<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <device>
    <deviceType>urn:schemas-upnp-org:IoTdevice:enhancedswitchdevice:1</deviceType>

    <IoTidDynamicWS xmlns="hydra">0.0.0.6189708676876140718</IoTidDynamicWS>

    <energywsendpoint xmlns="hydra">http://212.214.80.144:8080/IoTdevice/8619ff3a-af98-44a9-85da-
      7f5f18f7e562/energy</energywsendpoint>

    <IoTidStaticWS xmlns="hydra">0.0.0.6592261886889156134</IoTidStaticWS>

    <discoveryinfo
      xmlns="hydra"><tellstickdevice><name>PetersLight2</name><vendor>Nexa</vendor><deviceid>2</
      deviceid></tellstickdevice></discoveryinfo>

```

```

<IoTidUPnPService_urn_schemas-upnp-org_memoryservice_1
  xmlns="hydra">0.0.0.4695383175879738995</IoTidUPnPService_urn_schemas-upnp-
  org_memoryservice_1>

<networkmanager
  xmlns="hydra">http://localhost:8082/services/NetworkManagerApplication</networkmanager>

<IoTUDN xmlns="hydra">PetersLight2</IoTUDN>

<standbytime xmlns="hydra">60</standbytime>

<status xmlns="hydra">web service initiated</status>

<IoTidStaticWSDescription xmlns="hydra">PetersLight2:StaticWS</IoTidStaticWSDescription>

<IoTidUPnPService_urn_schemas-upnp-org_locationservice_1
  xmlns="hydra">0.0.0.8817877591614169464</IoTidUPnPService_urn_schemas-upnp-
  org_locationservice_1>

<IoTidUPnPService_urn_schemas-upnp-org_energyservice_1
  xmlns="hydra">0.0.0.410334127518851262</IoTidUPnPService_urn_schemas-upnp-
  org_energyservice_1>

<IoTWSEndpoint xmlns="hydra">http://212.214.80.144:8080/IoTdevice/8619ff3a-af98-44a9-85da-
  7f5f18f7e562</IoTWSEndpoint>

<UPnPEndpoint xmlns="hydra">http://212.214.80.144:64277/</UPnPEndpoint>

<IoTidUPnPService_urn_upnp-org_serviceld_switchservice_1
  xmlns="IoT">0.0.0.7715272012937744631</IoTidUPnPService_urn_upnp-
  org_serviceld_switchservice_1>

<dynamicWSEndpoint xmlns="hydra">http://212.214.80.144:64277/</dynamicWSEndpoint>

<wsendpoint xmlns="hydra">http://212.214.80.144:8080/0/EnhancedSwitchWS</wsendpoint>

<IoTidHydraWS xmlns="hydra">0.0.0.713272519360667694</IoTidHydraWS>

<DACEndpoint xmlns="hydra">http://212.214.80.144:8080/ApplicationDeviceManager</DACEndpoint>

<IoTidUPnPDescription xmlns="hydra">PetersLight2:UPnP</IoTidUPnPDescription>

<IoTidHydraWSDescription xmlns="hydra">PetersLight2:HydraWS</IoTidHydraWSDescription>

<securityinfo xmlns="hydra"><securityInfo xmlns="hydra"><property
  name="tellstick.api.version"><value>2.1</value></property><property
  name="switch.mode"><value>2</value></property><property
  name="EncryptionProtocol"><value>None</value></property></securityInfo></securityinfo>

<IoTidUPnPService_urn_upnp-org_serviceld_1
  xmlns="hydra">0.0.0.6339391984478104269</IoTidUPnPService_urn_upnp-org_serviceld_1>

```

```

<IoTidEnergyWSDescription xmlns="hydra">PetersLight2:EnergyWS</IoTidEnergyWSDescription>
<gateway xmlns="hydra">BLONDIE</gateway>
<IoTidUPnP xmlns="hydra">0.0.0.3263501067198386232</IoTidUPnP>
<IoTidEnergyWS xmlns="hydra">0.0.0.3952190387415366563</IoTidEnergyWS>
<friendlyName>PetersLight2</friendlyName>
<manufacturer>Telldus</manufacturer>
<manufacturerURL>http://www.telldus.se</manufacturerURL>
<modelDescription>Remote switch</modelDescription>
<modelName>Tellstick</modelName>
<modelName>X1</modelName>
<UDN>uuid:8619ff3a-af98-44a9-85da-7f5f18f7e562</UDN>
</device>
</root>

```

Figure 39: Device XML

The following element is an example of a standard UPnP element. It specifies the device type:

```
<deviceType>urn:schemas-upnp-org:IoTdevice:enhancedswitchdevice:1</deviceType>
```

This element is an example of a Hydra-specific extension. It specifies the gateway where the device is running:

```
<gateway xmlns="hydra">BLONDIE</gateway>
```

There are a number of methods that allows for searching of devices in the network. These require an XPath expression as parameter. This Xpath expression is evaluated against the LinkSmart Device XML for each device to decide if the match the search criteria or not.

The various elements can be grouped into categories:

PID

The IoTUDN element represents the PID (Persistent ID) that has been assigned to this particular devie.

```
<IoTUDN xmlns="hydra">PetersLight2</IoTUDN>
```

IoTids

The following elements represent the different IoT IDs (HID) for different device services. The IoTidStaticWS is the normal HID to be used, while IoTidHydraWS is the HID to access the generic IoT services of the device.

Note the element `IoTidUPnPService`, for each UPnP service a HID is created with the format `IoTidUPnPService_serviceid` (where in the service id : has been replaced with _ as in "IoTidUPnPService_urn_schemas-upnp-org_energyservice_1")

```
IoTidStaticWS
IoTidDynamicWS
IoTidHydraWS
IoTidEnergyWS
IoTidUPnPService_
```

Endpoints

The endpoint elements represent the endpoint to the device service. Normally this should not be used. Use the corresponding HID instead.

```
energyserviceendpoint
wserviceendpoint
dynamicwserviceendpoint
UPnPServiceendpoint
```

Other LinkSmart elements

The `DACEndpoint` element represents the DAC that has discovered and created the IoT Device. It "owns" the device

```
<DACEndpoint xmlns="hydra">http://212.214.80.144:8080/ApplicationDeviceManager</DACEndpoint>
```

The `gateway` element represents the gateway where the device is running:

```
<gateway xmlns="hydra">BLONDIE</gateway>
```

UPnP elements

The following elements are standard UPnP elements

```
<deviceType>urn:schemas-upnp-org:hydradevice:enhancedswitchdevice:1</deviceType>
<friendlyName>PetersLight2</friendlyName>
<manufacturer>Tellidus</manufacturer>
<manufacturerURL>http://www.tellidus.se</manufacturerURL>
<modelDescription>Remote switch</modelDescription>
<modelName>Tellstick</modelName>
<modelName>X1</modelName>
<UDN>uuid:8619ff3a-af98-44a9-85da-7f5f18f7e562</UDN>
```

10.5 Extending the LinkSmart Device XML

It is possible to extend the LinkSmart Device XML to incorporate your own metadata and state information. Simply call the method `SetProperty` in the IoT WS, then you can add properties to the device which will be available in the LinkSmart Device XML and can be used as part of your search expressions.

Calling `myDevice.SetProperty("myproperty", "value1")`, will create the following element in your LinkSmart Device XML:

```
<myproperty xmlns="hydra">value1</myproperty>
```

You can then easily select devices in the network that has myproperty="value1".

For instance the following call will get an Hydra encoded URL to the Energy WS for the all devices that has myproperty="value1".

```
m_applicationdevicemanager.GetIoTURLsFromXpath("//*[name()='myPropety' and .='value1']" ,  
"IoTidEnergyWS", "");
```

11. Device Developer Kit .net

There are two main tools for creating device code for .Net in Linksmart:

- Intel Service Author for UPnP Technologies
- Linksmart .Net DDK tool

The example device that we will create in this tutorial is an OBEX device for a smart phone.

11.1 Using Intel Service Author for UPnP Technologies

This tool is used for creating the service methods and producing an SCPD that will be used as input for the final code generation.

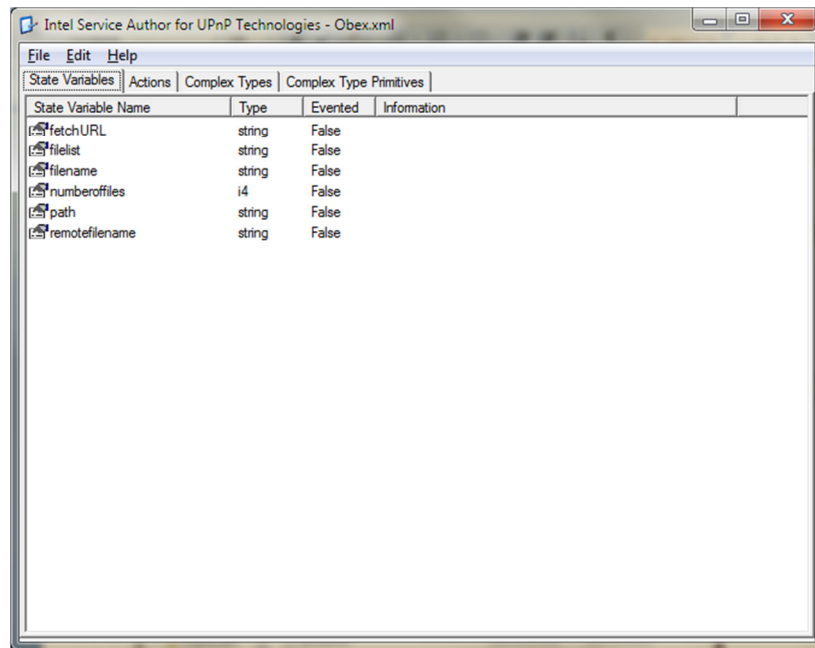


Figure 40: State Variables in Service Author

The first step is to define the state variables that will be used by the service. State variables have to be defined for all Input/output parameters used in the service. In this case we have a number of state variables defined with their respective types.

The next step is to define actions, i.e. the methods that this service should support. This is done in the "Actions" tab.

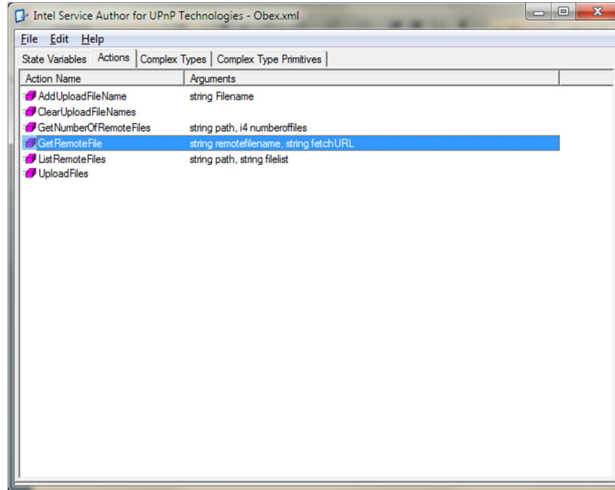


Figure 41: Actions in Service Author

Here we have defined a number of methods with their corresponding arguments. The methods are added using the "Action Editor" which allows for adding arguments and defining in which direction it is used, see below.

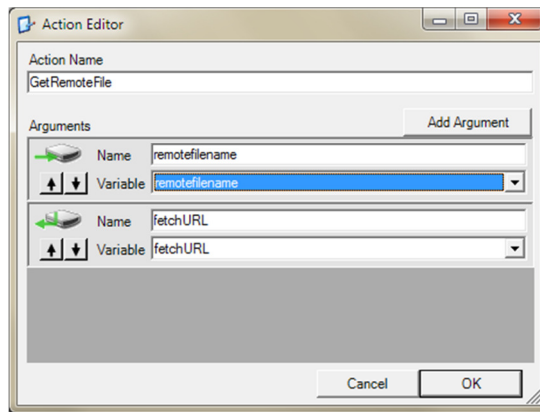


Figure 42: Action Editor

When one is finished it is time to save the SCPD to file for later processing in the Hydra .net DDK tool.

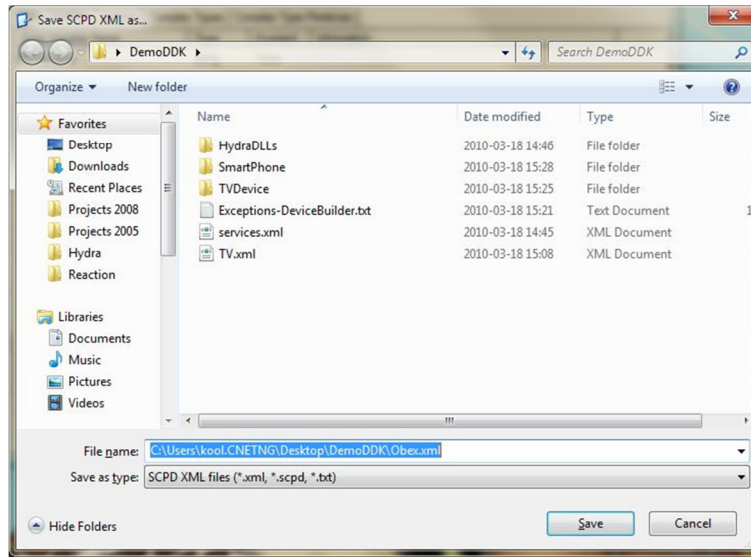


Figure 43: Saving scpd file

11.2 Using Linksmart .Net DDK tool

The actual code generation is done in the Linksmart .Net DDK tool. It is also where the actual configuration of device type and other settings are done.

The first step is to "Add Device" by right clicking in the tools left pane.

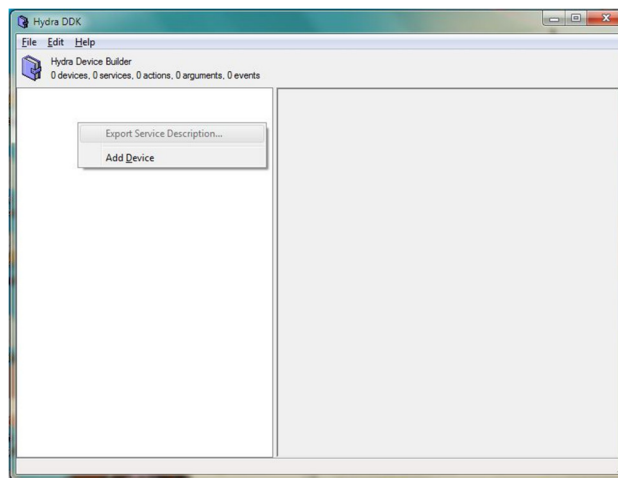


Figure 44: Create IoT Device in DDK tool

The next step is to edit the meta data for the device, i.e., device name, type, description etc.

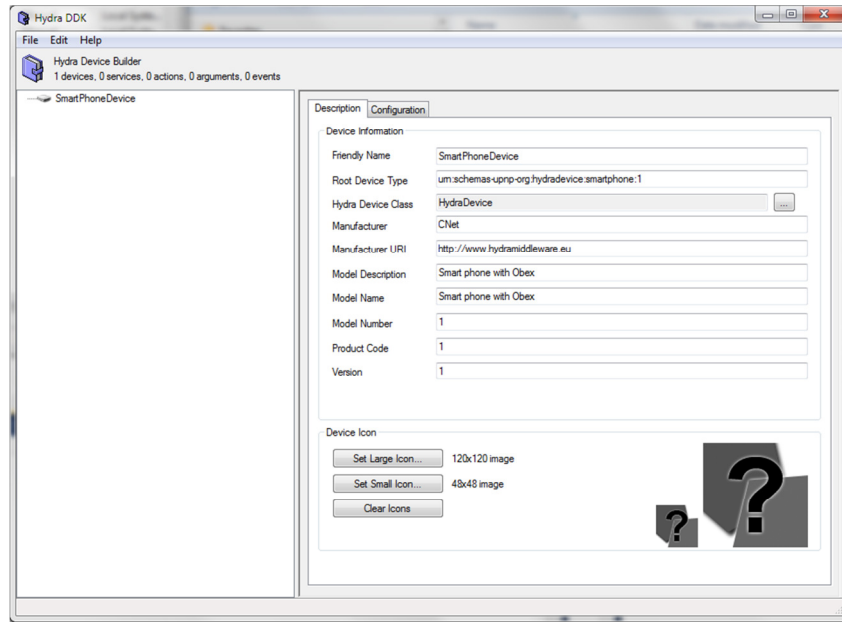


Figure 45: Properties of Hydra Device

Then one adds the service created in the previous section by right clicking on the device in the left pane.

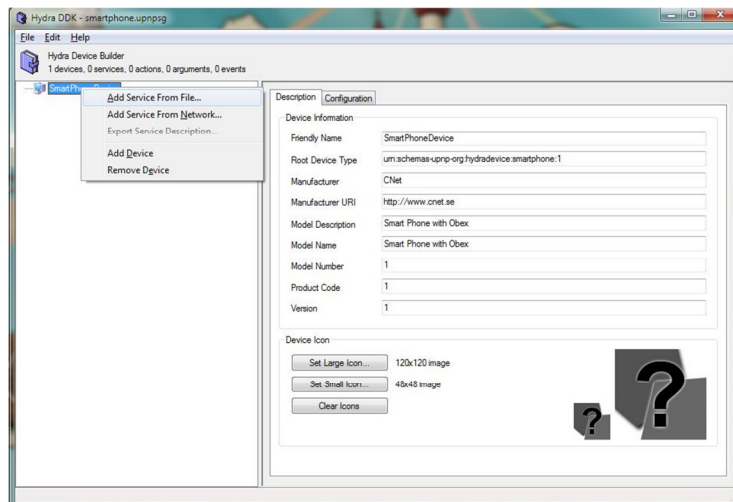


Figure 46: Adding service to Device

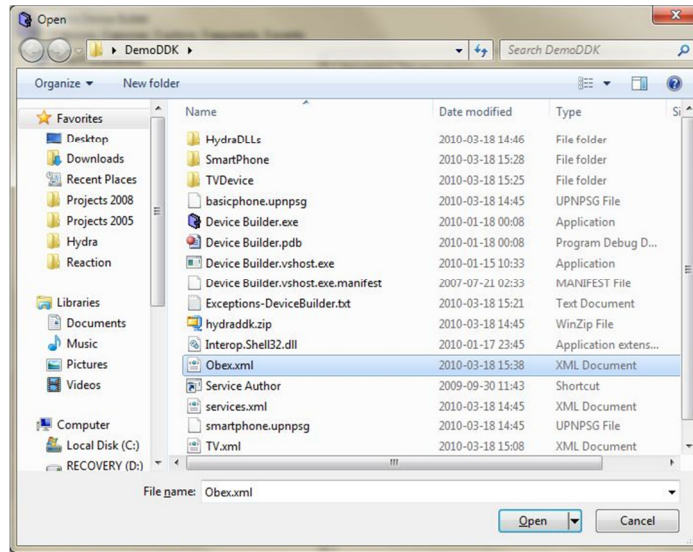


Figure 47: Selecting the SCPD file

Now we have added the OBEX service and we can see all the methods in that service.

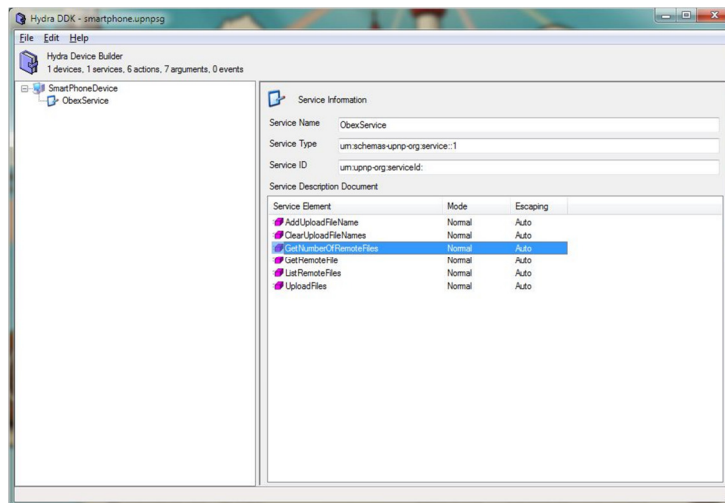


Figure 48: Device with services

The next step is to select the device type of the device using the ontology class browser. For this device we select MobilePhone as the ontology class. This information is used when the device template is entered in to the ontology.

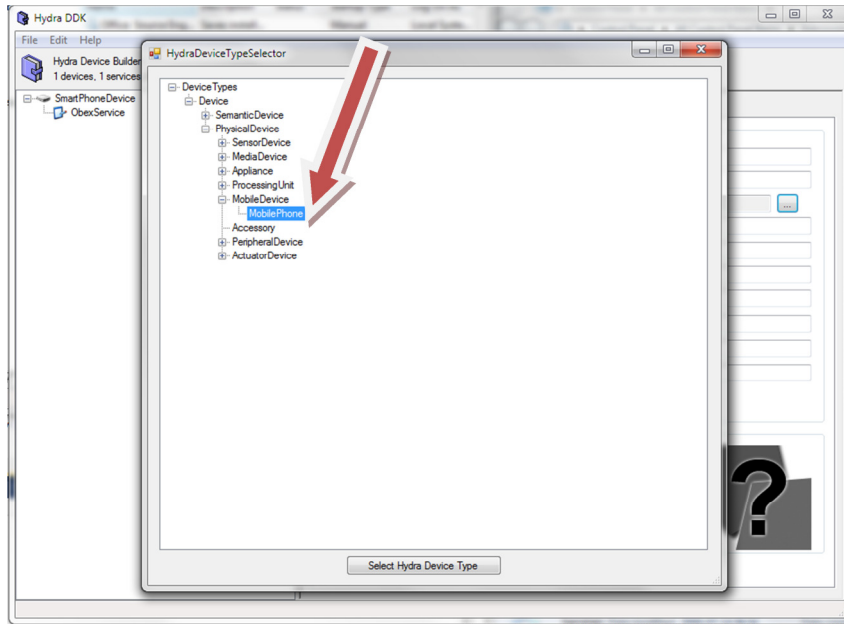


Figure 49: Adding Hydra device class for the Device

Finally we have arrived at the stage where it is time to generate the code for the Hydra device. Select the "File" menu and choose "Generate Hydra Device".

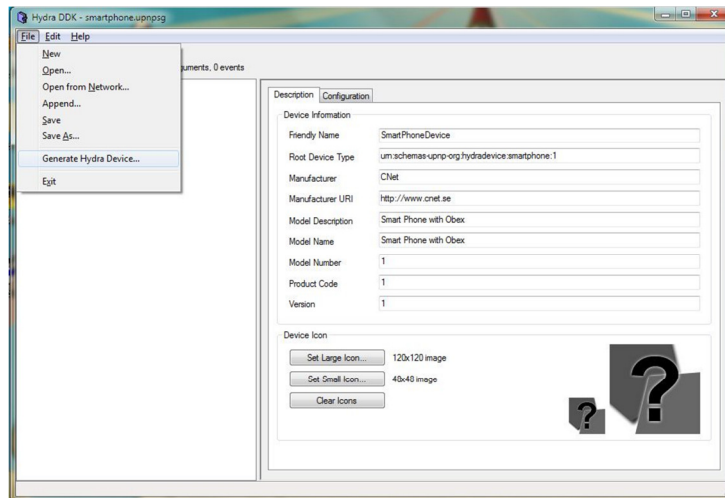


Figure 50: Generate the Hydra device code

In the code generation dialogue one has to decide the project name and optional Namespace for the generated code. In the Hydra settings tab we can select that we want to add the Device as a template device in the ontology.

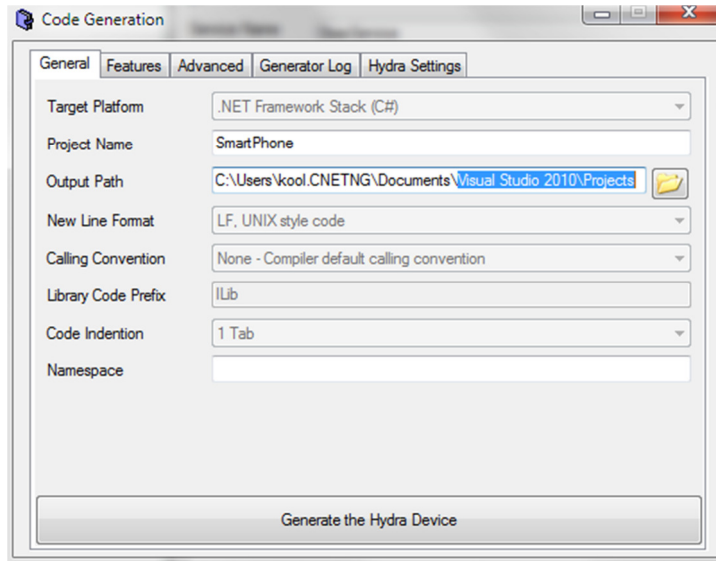


Figure 51: General settings for the code generation

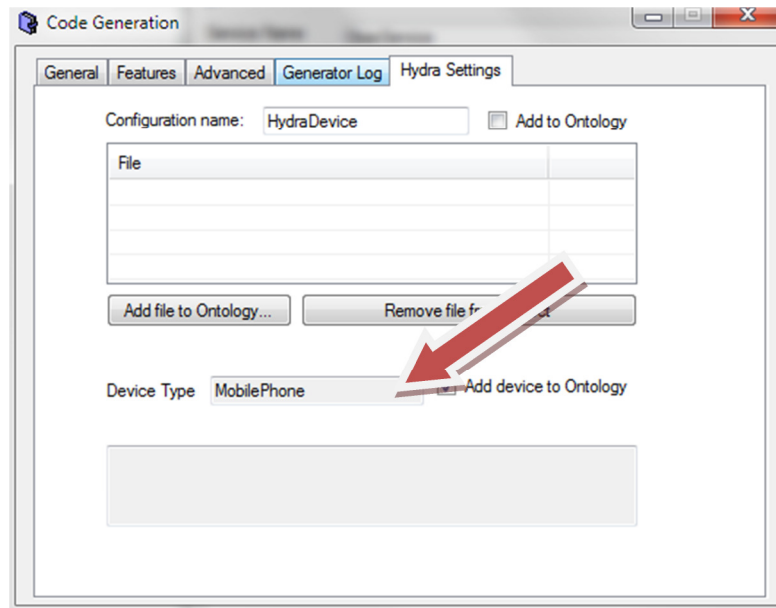


Figure 52: Ontology options in the code generation

A complete Visual Studio project is created with the necessary Hydra references and the device template is added to the ontology.

In the ontology manager tool we can see the information that was added to the ontology by the DDK for the device template.

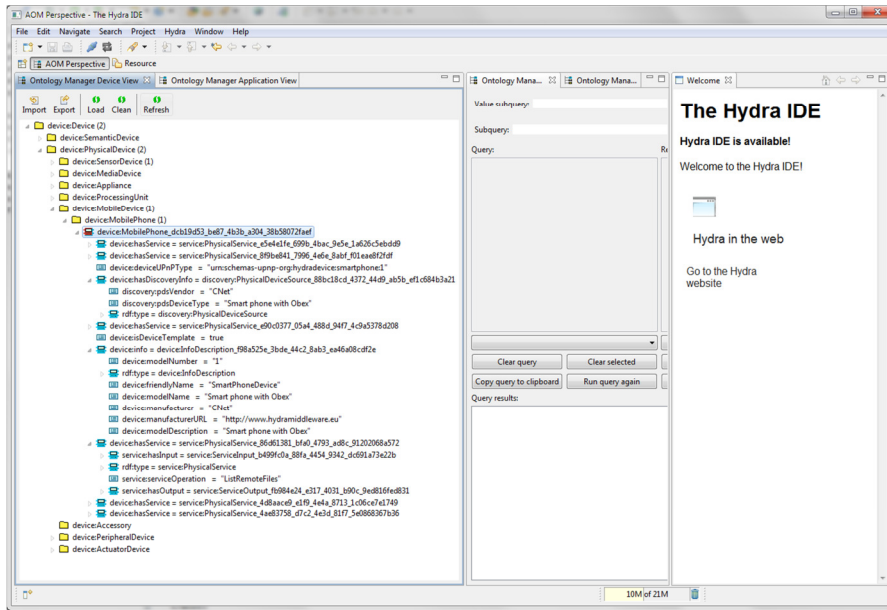


Figure 53: Ontology browser with the created device template

The next step is to open the Visual Studio project.

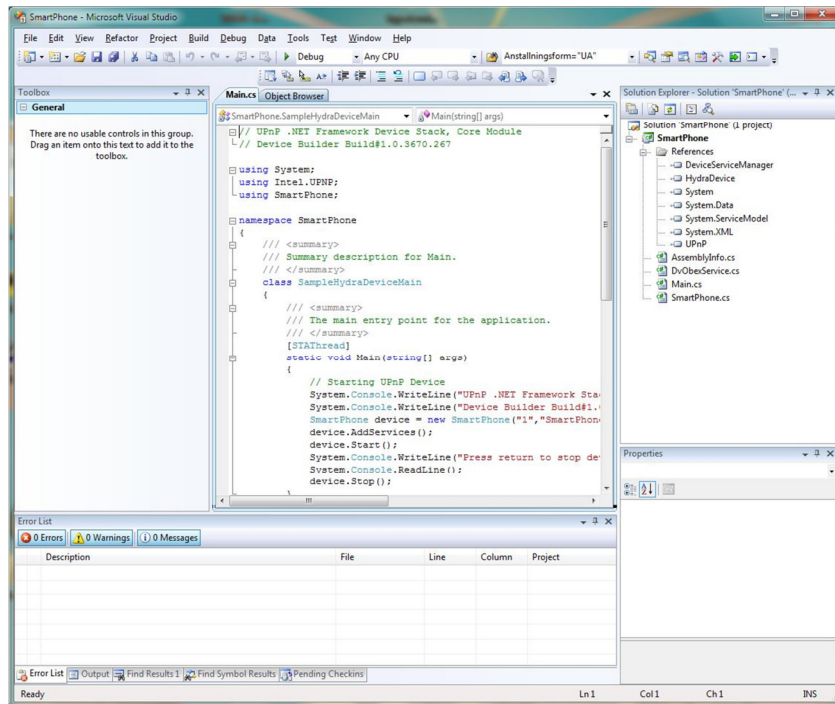


Figure 54: Generated Visual Studio project

The device code is already executable since all methods are stubbed. Normally one would then change the code in the stubs to do the actual device communication. The location of the stubs to be changed is in "Device name".cs, i.e. SmartPhone.cs in this project. But in this case we will start the device by opening the "Debug" menu and selecting "Start debugging".

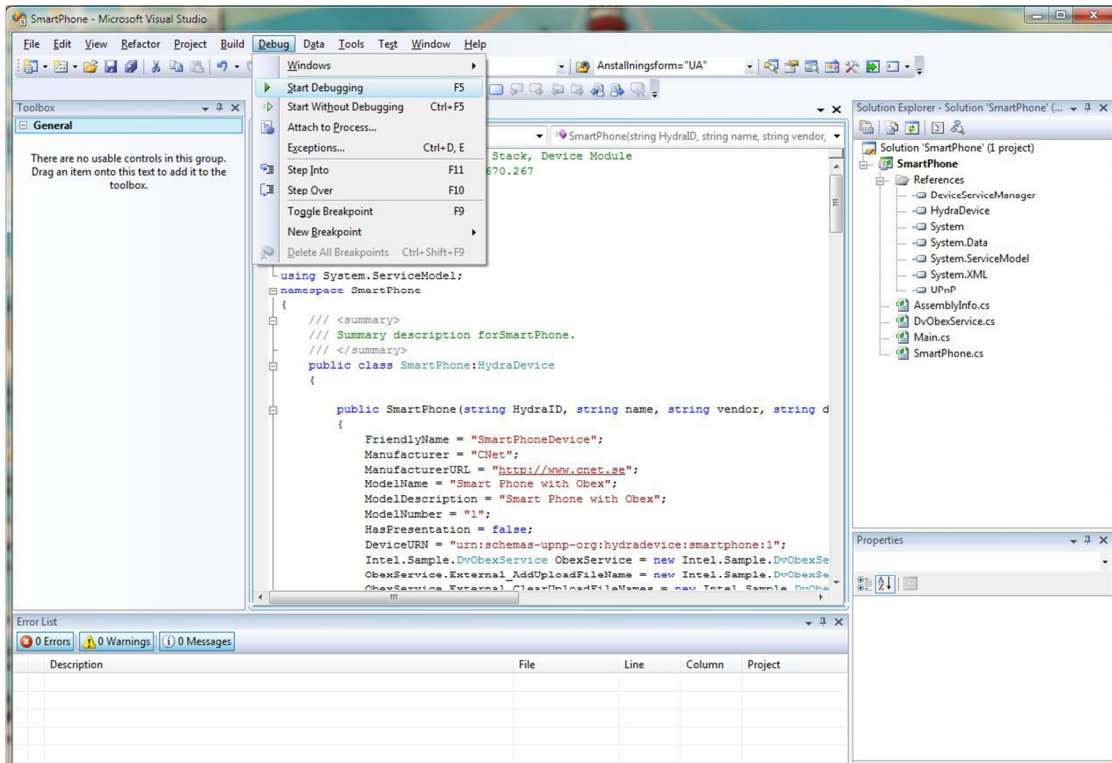


Figure 55: Start Debug

If we run a Linksmart DAC tool we will find our newly created device with all its services. Note that we have automatically received the relevant Hydra services: HydraService, EnergyService, LocationService and Memory service. We can also see that the Device is properly discovered and has all the Hydra properties such as HID.

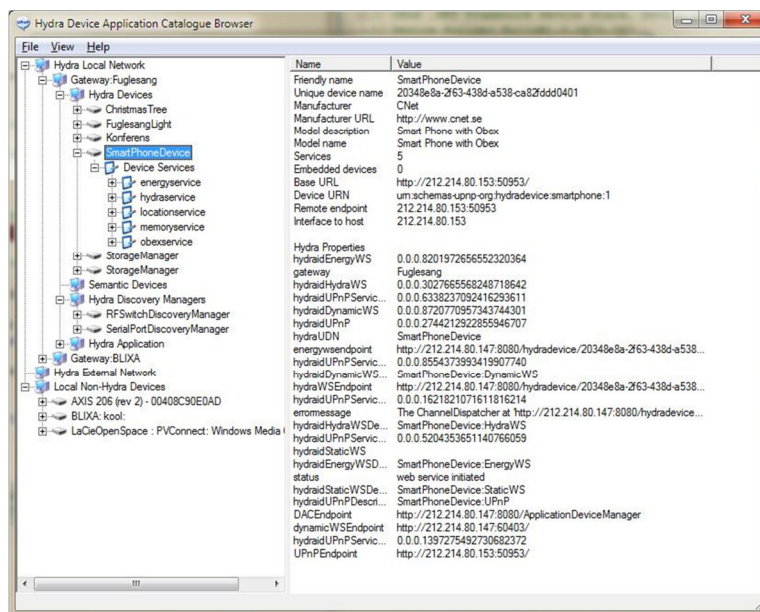


Figure 56: Graphical DAC browser with the newly created device

12. Extending the LinkSmart Ontology model

It is possible to extend the default LinkSmart Ontology Model. Figure 57 shows how new additional information has been added to the running instance of an Ontology Manager. The tool used is part of the LinkSmart IDE² and allows ontologies to be exported to OWL files and opened in an ontology editor e.g. Protégé. Figure 58 consist of a piece of the exported OWL file showing a Thermometer that has been created.

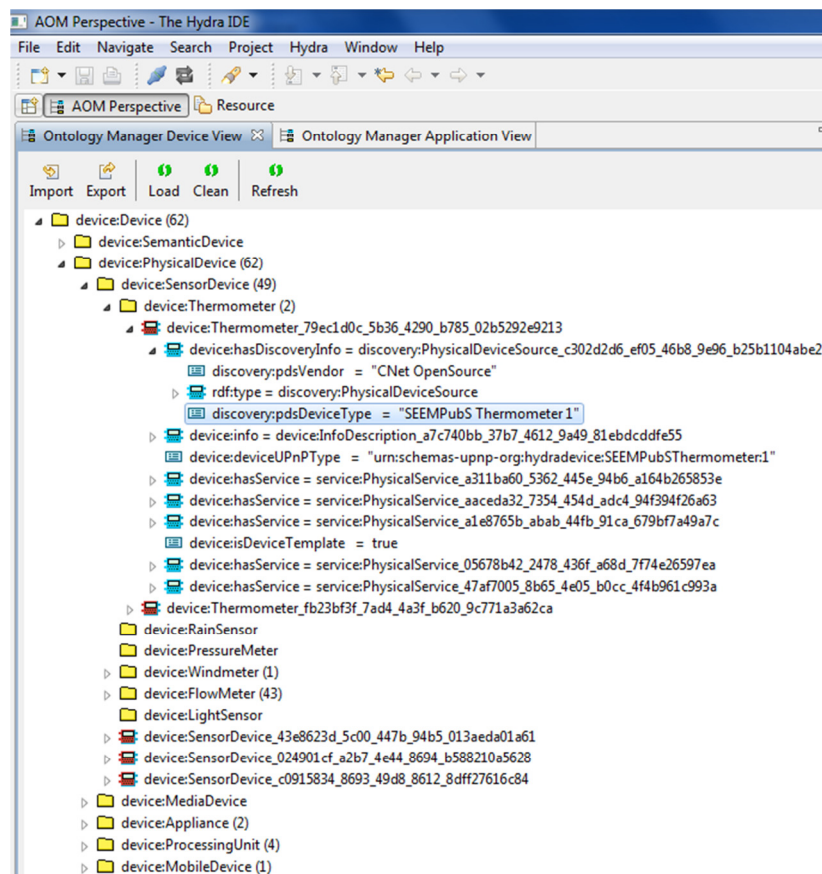


Figure 57: The IoT device in the LinkSmart Ontology Manager IDE

² The LinkSmart IDE (Integrated Development Environment) is implemented as a plugin to the Eclipse IDE.

```
<device:InfoDescription rdf:about="http://localhost/ontologies/Device.owl#InfoDescription_a7c740bb_37b7_4612_9a49_81ebcdcf55">
<device:friendlyName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">SEEMPubS Thermometer</device:friendlyName>
<device:modelDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">SEEMPubS Thermometer</device:modelDescription>
<device:modelNumber rdf:datatype="http://www.w3.org/2001/XMLSchema#string">1</device:modelNumber>
<device:manufacturerURL rdf:datatype="http://www.w3.org/2001/XMLSchema#string">http://www.hydramiddeeware.eu</device:manufacturerURL>
<device:modelName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">SEEMPubS Thermometer 1</device:modelName>
<device:manufacturer rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CNet OpenSource</device:manufacturer>
</device:InfoDescription>
<service:PhysicalService rdf:about="http://localhost/ontologies/Service.owl#PhysicalService_47af7005_8b65_4e05_b0cc_4f4b961c993a">
<service:hasOutput rdf:resource="http://localhost/ontologies/Service.owl#ServiceOutput_39213c02_59f1_4545_ba24_b1ccaa7a62ee"/>
<service:serviceOperation rdf:datatype="http://www.w3.org/2001/XMLSchema#string">GetMaxTemperature</service:serviceOperation>
</service:PhysicalService>
<service:ServiceInput rdf:about="http://localhost/ontologies/Service.owl#ServiceInput_19256246_8b32_4d61_9bb3_5022908dfc1b">
<service:sendEvents rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">>false</service:sendEvents>
<service:parameterName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">minTemperature</service:parameterName>
<service:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">string</service:parameterType>
<service:relatedStateVariable rdf:datatype="http://www.w3.org/2001/XMLSchema#string">currentTemperature</service:relatedStateVariable>
</service:ServiceInput>
<service:PhysicalService rdf:about="http://localhost/ontologies/Service.owl#PhysicalService_aaceda32_7354_454d_adc4_94f394f26a63">
<service:hasOutput rdf:resource="http://localhost/ontologies/Service.owl#ServiceOutput_c874b3e2_5088_4084_affe_bb82faee3ad8"/>
<service:serviceOperation rdf:datatype="http://www.w3.org/2001/XMLSchema#string">GetMinTemperature</service:serviceOperation>
</service:PhysicalService>
<service:ServiceOutput rdf:about="http://localhost/ontologies/Service.owl#ServiceOutput_c874b3e2_5088_4084_affe_bb82faee3ad8">
<service:sendEvents rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">>false</service:sendEvents>
<service:parameterName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">minTemperature</service:parameterName>
<service:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">string</service:parameterType>
<service:relatedStateVariable rdf:datatype="http://www.w3.org/2001/XMLSchema#string">minTemperature</service:relatedStateVariable>
</service:ServiceOutput>
<service:ServiceOutput rdf:about="http://localhost/ontologies/Service.owl#ServiceOutput_39213c02_59f1_4545_ba24_b1ccaa7a62ee">
<service:sendEvents rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">>false</service:sendEvents>
<service:parameterName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">maxTemperature</service:parameterName>
<service:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">string</service:parameterType>
<service:relatedStateVariable rdf:datatype="http://www.w3.org/2001/XMLSchema#string">maxTemperature</service:relatedStateVariable>
</service:ServiceOutput>
<service:PhysicalService rdf:about="http://localhost/ontologies/Service.owl#PhysicalService_a1e8765b_abab_44fb_91ca_679bf7a49a7c">
<service:hasInput rdf:resource="http://localhost/ontologies/Service.owl#ServiceInput_7e5f83c4_f3e0_4091_8fbd_045780deb7b8"/>
<service:serviceOperation rdf:datatype="http://www.w3.org/2001/XMLSchema#string">SetMaxTemperature</service:serviceOperation>
</service:PhysicalService>
<service:PhysicalService rdf:about="http://localhost/ontologies/Service.owl#PhysicalService_a311ba60_5362_445e_94b6_a164b265853e">
<service:hasInput rdf:resource="http://localhost/ontologies/Service.owl#ServiceInput_19256246_8b32_4d61_9bb3_5022908dfc1b"/>
<service:serviceOperation rdf:datatype="http://www.w3.org/2001/XMLSchema#string">SetMinTemperature</service:serviceOperation>
</service:PhysicalService>
<service:ServiceInput rdf:about="http://localhost/ontologies/Service.owl#ServiceInput_7e5f83c4_f3e0_4091_8fbd_045780deb7b8">
<service:sendEvents rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">>false</service:sendEvents>
<service:parameterName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">maxTemperature</service:parameterName>
<service:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">string</service:parameterType>
<service:relatedStateVariable rdf:datatype="http://www.w3.org/2001/XMLSchema#string">maxTemperature</service:relatedStateVariable>
</service:ServiceInput>
```

Figure 58: Piece of the exported OWL file where the Thermometer information is shown

The device information from the default LinkSmart Device Ontology can be integrated with additional device information. This can be used, for instance, to integrate information about Locations and Energy Features that can be connected to LinkSmart Devices.

The Protégé screen dump in the figure below shows ontology information such as *friendly name*, *model name* etc exported as an OWL file from the LinkSmart IDE for the device we just created (Thermometer).

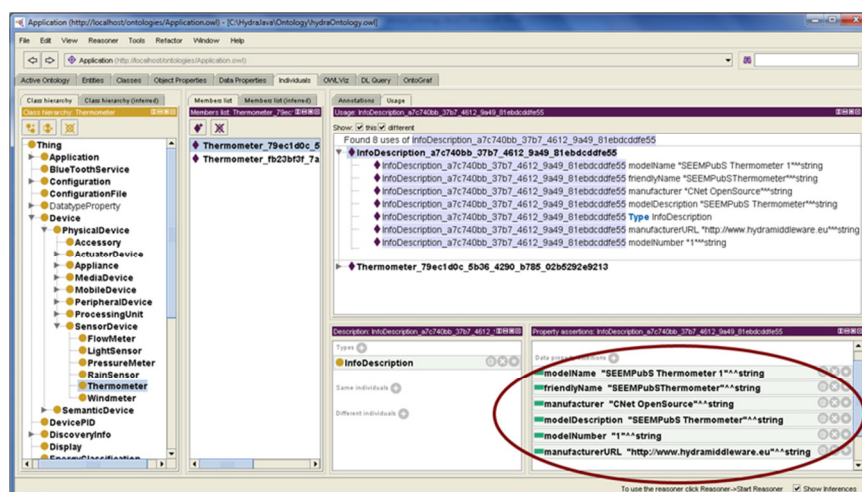


Figure 59: IoT Device information Description viewed in LinkSmart Device ontology using the Protégé editor

Figure 60 shows the OWL representation of the *GetMinTemperature* method previously added in Service Author.

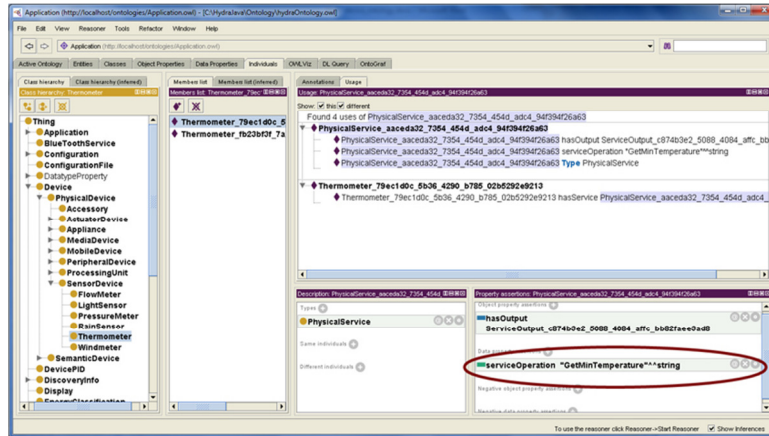


Figure 60: IoT Device *GetMinTemperature* method

13. POBICOS Overview

The POBICOS project targets computing environments which feature collections of objects, equipped with sense-compute-actuate embedded nodes, which differ in their sensor, actuator and computing resources. Moreover, the actual mix of objects, and the resources provided by those objects, which will be available during execution is partly unknown when programming the application(s).

POBICOS aims to design, implement and test a platform that simplifies both the development and the deployment of applications for such heterogeneous and incompletely specified systems. The key challenge is to enable applications to take the best advantage of what-ever "resource opportunities" exist at runtime, provided by the objects that happen to be available. The platform shall make such "opportunistic" behaviour largely transparent to the programmer.

Towards this goal, the main objectives of POBICOS are:

- The design of a programming model and supporting mechanisms for opportunistic pervasive computing;
- An ontology-driven approach for modelling and flexibly accessing resources for a given application domain;
- The implementation of a corresponding middleware on top of embedded wireless sensor/actuator nodes;
- The provision of suitable resource abstraction and domain-based customisation tools as well as application development, simulation and deployment tools;
- The experimental validation of the middleware and tools for a selected application domain in the area of home automation.

The domain of energy efficiency at home is the source of scenarios and requirements. Proof-of-concept applications will be tested in a real setting.

13.1 System inspection protocol

The system inspection is needed for gathering various monitoring information and enabling control actions to the POBICOS system. The monitoring and administration services can be exploited by the developers of the target system, technical experts, and the end-users through the monitoring and control tool. The system inspection protocol for monitoring and control of WSN in unreliable networking environments will be used as starting point for the WSN management protocol and administration tool of BEMO-COFRA.

13.1.1 General Architecture

For the monitoring and administration purposes, the connection between the user and the POBICOS system is provided by means of a so-called gateway node, which is between the POBICOS network and the device hosting the monitoring and control tool. The communication between the gateway node and the user device is performed through UART.

Importantly, the gateway node is a common member of the network, and any node at any time can be chosen as the gateway node. Other nodes recognize the chosen gateway node, and forward their monitoring information to it. Thus, the system inspection is flexible for various environments and user preferences.

The chosen gateway node and the more powerful device share a "base station" functionality. The general view of the POBICOS network is gathered in the computationally powerful device, and so the middleware resources are not wasted. All user-independent, network internal management functionalities are independent of the gateway node, and so their processing logic need not be in the gateway node.

13.1.2 Services

System inspection offers the following services for application, network, node, and system-wide monitoring and control.

System-wide monitoring:

- retrieving a dynamic general view of the system
- logging the event occurrences inside the system

Application monitoring and control:

- monitoring the status of the applications (running/stopped)
- retrieving the application happiness level
- retrieving the application tree (the micro-agents and their parent-child relations)
- retrieving the mapping of the micro-agents in the nodes
- monitoring the execution of the micro-agents
- controlling the application specific configuration settings

Node and network monitoring and control:

- retrieving the type and resources of the nodes and the host objects
- monitoring the communication traffic among the nodes
- monitoring the CPU load in individual nodes
- controlling the object specific configuration settings

13.1.3 Communication

For every monitoring and control action, a corresponding message is constructed. The messages can be broadcast, multicast, or unicast depending on the purpose of the message. For example the request to start online monitoring is broadcast to every node, whereas e.g. the request to change an object setting value needs to be sent only to the corresponding node. Multicast is used for example in the case of application configuration setting change; the new setting value is forwarded only to nodes hosting micro-agents that use the setting.

The messages are classified into requests, responses, spontaneous messages, and network internal management messages. Requests, responses and spontaneous messages are related to online monitoring and control. The network internal management messages are needed whether or not the online monitoring is in use.

Requests are sent to the network when the user wants to gather information from the system or perform a control action. A response is an answer to a request or an acknowledgement to a control message. Spontaneous messages are notifications of the event occurrences in the target system. They are automatically forwarded from the network to the monitoring gateway node and further to the device running the monitoring and control tool. The network internal management messages are used inside the network for example to pass required application settings from node to node.

14. Glossary

This chapter aims at providing a comprehensive understanding of important terms used in and derived from the LinkSmart project. In addition, the terms listed in this chapter try to convey a sense of their application and present the background of the fundamental concepts. Even if some of the subsections seem to be a repetition of things already documented, this chapter can be seen as a central point of access to a description of the LinkSmart terms. The definitions listed here have been agreed on by the LinkSmart Consortium. (The terms are ordered from high-level to low-level)

Physical Device:

A "Physical Device" is a common device that offers some functions that affect the "physical world".

Such functions could for example be providing light, heat, wind, open door, or reports physical properties such as temperature, blood pressure, pulse, movements, etc. LinkSmart constitutes a middleware that enables networking of physical devices.

Appliance:

An "Appliance" represents a physical device that is dedicated to a single purpose. Appliances refer to more complex physical devices and are especially prominent in the field of home automation or home entertainment.

LinkSmart-Enabling a Device:

"LinkSmart-enabling a Device" means the process of making the functions of a LinkSmart-compliant physical device available and controllable for other devices in a LinkSmart network. Depending on its device class, three methods make such a device LinkSmart-enabled:

- Installing (parts of) the LinkSmart middleware on the device
- Using the Limbo tools to embed Web Services on the device and generate a Proxy
- Using a Proxy to represent the device on a Gateway

At the end of this process the functions of this device can be invoked using Web Services, and metadata about the device is provided in the format and protocol required by LinkSmart.

LinkSmart-Enabled Device:

A "LinkSmart-Enabled Device" is a LinkSmart-compliant physical device that has successfully run through the LinkSmart-enabling process. A LinkSmart-enabled device owns a software representation, i.e. a LinkSmart Device, in a LinkSmart network and

- Can be discovered by other devices in a LinkSmart network
- Makes all or a subset of its functions accessible as Web Services
- Offers its Web Services either natively (embedded code) or through a proxy
- Supports UPnP and advertises its entry into a Local Area Network through UPnP broadcasting
- Supports LinkSmart Generic Services and LinkSmart Energy Service.

LinkSmart Device:

A "LinkSmart Device" constitutes the software representation of a LinkSmart-enabled device and its functionalities, in order to enable access and control. The LinkSmart Device can either run as a Proxy for the LinkSmart-compliant physical device on a gateway or it can run embedded in the device. A LinkSmart Device can obtain LinkSmart identifiers for its services (HID) and also application specific identifiers. Furthermore, a LinkSmart Device implements the "LinkSmart Generic Services" and "LinkSmart Energy Services". For one physical device there might exist one or more LinkSmart Devices. A LinkSmart Device might also incorporate services from several physical devices.

Gateway:

A "Gateway" is a physical device with IP capabilities, which manages a set of proxies for controlling LinkSmart devices. A gateway must support Web Services and UPnP and should also be able to run LinkSmart Discovery Managers. In addition, a gateway may also host other components of the LinkSmart middleware.

Proxy:

A "Proxy" is a LinkSmart Device that consists of a software component responsible of communicating with a physical device, understanding the technology used and the format of the data exchanged. It is deployed on a gateway and represents the device to be controlled.

LinkSmart Network:

A "LinkSmart Network" represents a network of LinkSmart Devices and applications that communicate with each other using Web Services and IP communication on top of a Peer-to-Peer overlay.

LinkSmart Middleware:

The "LinkSmart Middleware" is a collection of interrelated components, i.e. LinkSmart Managers, that work together to realise a platform of networked heterogeneous physical devices. The LinkSmart middleware allows such devices to be part of an ambient intelligence environment.

Device Discovery:

The process "Device Discovery" covers several steps where a physical device is discovered, semantically resolved and made accessible as a LinkSmart Device. In order for a device to be discovered in a LinkSmart network, a definition of the device type must exist in the LinkSmart Device Ontology.

LinkSmart Manager:

A "LinkSmart manager" (or short "manager") constitutes the major building blocks that make up the LinkSmart middleware. A LinkSmart manager encapsulates a set of operations and data that realise a specific functionality and is mostly subdivided into several internal components.

LinkSmart Generic Services:

The LinkSmart Generic Services are supported by all LinkSmart Devices and contain a set of meta-data methods that can be used to query the device about its properties.

LinkSmart Energy Services:

The LinkSmart Energy Services are supported by all LinkSmart Devices and provide methods to retrieve information from the energy profile of the device and from the energy policy.

LinkSmart Identifier (HID):

A "LinkSmart identifier" (or simply "LinkSmart ID" or shorter "HID") constitutes a unique identifier for every LinkSmart Device, service or resource within a LinkSmart network. Network Manager generates the HID, is responsible for the matching between logical and physical identifiers and for the propagation of this information to other peers of the LinkSmart network.

CryptoHID:

An application developer has the opportunity to assign his own CryptoHID to a certain LinkSmart Device. This CryptoHID can directly be used throughout the application code and referred to when expressing security, energy and other policies.

Session:

A "Session" traces the communication between elements of a LinkSmart network, in order to keep the communication coherent. Sessions allow the maintenance of the state of each network element as they communicate with each other. The Network Manager comprises a dedicated Session Manager that creates and maintains the lifecycles of the session objects.

Ontology:

An "Ontology" is a representation of the knowledge of a formally defined system of concepts and relations. In addition, an ontology can contain inference to derive new knowledge and integrity rules to assure its validity. Therefore, an ontology forms a network of information and logical relations described through a formal language such as the Web Ontology Language (OWL).

LinkSmart Device Ontology:

The "LinkSmart Device Ontology" is an ontology that contains knowledge about device classes, their properties and services offered.

LinkSmart Peer-to-Peer Architecture:

The "LinkSmart Peer-to-Peer Architecture" allows LinkSmart Devices in different local LinkSmart networks to access and communicate with each other. This means Web Services calls can be executed remotely over a P2P overlay.

References

- [1] <http://www.osgi.org/Links/DeveloperKits>
- [2] The SENSORIA Development Environment, CASE Tool for SOA Development
<http://home.mit.bme.hu/~rath/ppt/SDE.pdf>
- [3] <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>
- [4] <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>
- [5] <http://www.mono-project.com>
- [6] eXtensible Access Control Markup Language (XACML) Version 2.0 (2005)
<http://www.oasis-open.org/committees/xacml/>
- [7] OASIS: <http://www.oasis-open.org>
- [8] IBM, Web Services Security
<http://www.ibm.com/developerworks/library/specification/ws-secure>
- [9] eXist-db Open Source Native XML Database
<http://exist.sourceforge.net>
- [10] Sun XACML Implémentation. <<http://sunxacml.sourceforge.net>>