RUBICON
RUBICON Robotic UBiquitous COgnitive Network
Project No.: 269914

D1.4 – Proxy-specific Software Suite

Editor: **Mauro Dragone** **UCD**

Contributor(s): **Claudio Gennaro** **CNR**
Alessandro Saffiotti **ORU**
Claudio Vairo **CNR**

Dissemination level	
X	**PU** = Public
	PP = Restricted to other programme participants (including the Commission Services)
	RE = Restricted to a group specified by the consortium (including the Commission Services)
	CO = Confidential, only for members of the consortium (including the Commission Services)

Issue Date	19/05/2013
Deliverable Number	D1.4
WP	WP 1 - Communication Layer
Status	☐ Draft ☐ Working ☒ Released ☐ Delivered to EC ☐ Approved by EC

Document history			
V	Date	Author	Description
0.1	13/02/2012	Mathias Broxvall	Creation of LaTeX template
0.2	11/03/2013	Claudio Gennaro	First draft
0.3	20/03/2013	Mauro Dragone	WEB-based programming tool
0.4	10/04/2013	Claudio Gennaro	Appendix
0.5	28/04/2013	Mauro Dragone	Final version
0.6	05/05/2013	Maurizio di Rocco	Internal Review
0.1	10/05/2013	Kylie O'Brien	QA Review
0.1	16/05/2013	Mauro Dragone	Final corrections

Executive Summary

This deliverable takes place at the end of (M24) of the RUBICON project WP1 building the RUBICON Communication Layer (CML). The CML provides communication and integration mechanisms built upon middleware for wireless sensor and actuator networks (WSANs) and robotic ecologies. The objective of this deliverable is to describe the output of the work carried out in Task 1.4: *Adaptation of existing data-sharing middle-ware and proxy solutions for WSN/robot integration*. This task involves the identification of common component description for automatic discovery of hardware and processing capabilities and the provision of standard mechanisms that the higher layers of the RUBICON system can use to interact with the underlying sensing and acting infrastructure, including computational constrained devices. Such a set of functionalities is accomplished by the RUBICON component Proxy for WSAN, which extends the functionalities of the Gateway, i.e. the bridge between the Peis middleware and the WSAN described in D1.3.1. The extension of the Gateway includes the generic mechanisms for representation of motes sensor and actuation capabilities, the routing between multiple islands, and the integration of heterogeneous networks. This document refines the initial proxy design presented in D1.3.1. It discusses the motivations behind the design choices and the methods used to ground that initial prototype in the final design and the implementation of the CML in line with the communication stack presented in D1.3.2. The results of this deliverable will be an input and the basis for the integration work to be carried out in WP5 in both the transport and the Ambient Assisted Living application scenarios.

Contents

Figures

Abbreviations

API	Application Programming Interface
CML	Communication Layer
MAC	Media Access Control
PEIS	Physically Embedded Intelligent System
ROS	Robotic Operating System
RUBICON	Robotic UBIquitous COgnitive Network
SPP	Serial Port Profile
SW	Software
UML	Unified Modeling Language
USB	Universal Serial Bus
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actuator Network

Chapter 1

Introduction

The software described in this deliverable builds upon the newer version of the PEIS-kernel, generation G6, described in D1.3.1 and the final communication stack, presented in D1.3.2, to provide the gateway and proxy functionalities of the final release of the RUBICON CML. Deliverable D1.3.1 described the prototype version of the gateway and proxy component used in the first version of the CML. Specifically, the gateway/proxy prototype described in D1.3.1 consists of a unix C program that communicates with a WSAN sink node connected to a local USB port. For each island controlled by a machine, a local PEIS-init component is created by instantiating the gateway/proxy software with the specific hardware address of the USB port it uses to communicate with its WSAN island. This component is configured to start automatically upon boot-up time of the computer. The RUBICON gateway/proxy software uses the sink-mote to provide access to the raw sensor readings from the WSAN nodes reachable by the sink-mote.

The prototype implementation described in D1.3.1 successfully supported the preliminary experimentation activities carried out in the Transport and Ambient Assisted Living RUBICON test-beds. Results of experimentation in the tesbeds has allowed us to refine our initial design and to develop the final version, described in this deliverable, in order to support the requirements dictated by the actual applications.

The work carried out in Task 1.4 addressed a number of limitations of the gateway/proxy prototype implementation, namely:

- It used an ad-hoc description of sensing and acting resources, which was hard to extend to include new types of sensor and actuators.
- It did not support actuation-type instructions to be used to alter the state of actuators, such as relays used to activate/de-activate appliances under the control of the RUBICON.
- It worked exclusively for WSAN networks based on TinyOS and it did not support multiple and potentially heterogeneous network types.
- It did not support the distribution and the management of networks as distinct islands of a multi-island RUBICON system (as discussed in D1.3.1 and D1.3.2).
- It depended explicitly on the Active Message (AM-message) protocol of TinyOS and of all its application-specific message type instances, thus limiting the extensibility of the resulting system.

- It demanded exclusive access to the serial connection used to communicate with the sink node used as base station of the corresponding WSAN island, thus limiting its interoperability within multiple applications.

In order to address the above limitations and prepare the final version of the gateway/proxy component, in Task 1.4 we have re-factored the design to fit the multi-layered communication stack presented in D1.3.2, and implemented the final, fully functional release of the software by availing of the Java language. The result, described in the remaining sections of this document, is a highly modular, network agnostic gateway/proxy component with support for automatic discovery of sensors and actuators, extensibility, scalability, heterogeneous networks and multi-island distribution.

Chapter 2

Architecture

The new RUBICON Proxy implements the proxy design pattern advocated within the PEIS middleware (described in D1.1 and D1.3.1) to incorporate computationally constrained sensing and acting resources within a robotic ecology. In addition, the Proxy supports the cluster-based organization of the RUBICON, discussed in details in D1.3.1 and D1.3.2. Specifically, the proxy supports the following mechanisms:

- Introspection and resource discovery mechanisms to provide an index of available sensing and acting devices available in the RUBICON island controlled by the Proxy, in terms of available sensor signals and accepted actuator signals. The proxy is used to access a single island of the RUBICON, where each island is uniquely identified with the peis-id of the Proxy component. However, each proxy can manage multiple networks (each part of the same island), including 802.15.4/ZigBee networks and KNX [1] home automation infrastructures. Devices in each network may join or leave the system at run-time without affecting the operations of the Communication Layer.
- Proxied sensing and actuation by accepting subscriptions to sensors and/or sensing and actuation instructions to be transmitted to the underlying actuators. The Proxy will interact with the underlying networks, to make sure that the desired sensor data is sampled and published in the Peis tuplespace, and using the proper network protocol to configure specific sensors or send actuation instructions (e.g. set-points) to specific actuators.

Figure 2.1 shows the layered architecture of the Proxy component. In the **top layer**, the proxy interface acts as a single access point over the PEIS tuplespace for all sensors and actuators in the Proxy's island - no matter which hardware and communication protocol are in use. Components in the RUBICON systems will not communicate directly with the actual devices - they will have to direct their requests to the Proxy, which will route them to the devices. The simple reason for this design is that the Proxy manages a single, protocol-independent API, so that the other components in RUBICON do not need to account for low-level, network-specific details.

The **lower layer** is populated by a number of network components, each providing a means of integrating with a particular network protocol. For the purpose of the actual application scenarios tackled in RUBICON, we have

[1] KNX is approved as an International Standard (ISO/IEC 14543-3) as well as an European Standard (CENELEC EN 50090 and CEN EN 13321-1) and Chinese Standard (GB/Z 20965).

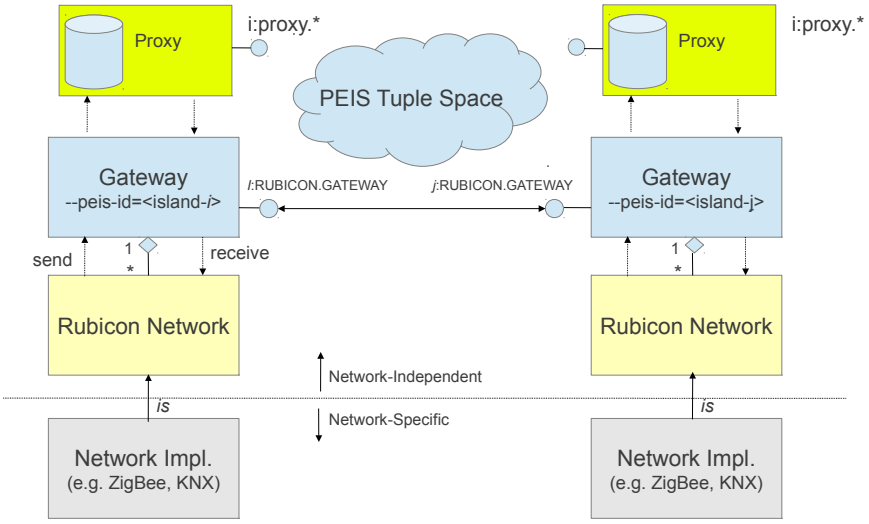

Figure 2.1: Multi-layered architecture of the gateway/proxy component.

provided network implementations to interact with WSANs and with the KNX home automation systems used in the Tecnalia HomeLab testbed.

Finally, in the **middle layer**, a gateway component is used to bridge the network layer and the Proxy interface. Specifically, the gateway component uses the Peis publish-subscribe mechanism to decouple the Proxy interface from the underlying networks, while allowing them to communicate both service-type and application-specific messages. In addition, each Proxy extends the publish-subscribe mechanism across the Peis peer-to-peer network, by allowing devices in the network layer to exchange messages with devices located in different islands. Island to island communication is achieved by managing a special tuple, which is used by a gateway component in each Proxy to receive messages addressed to the devices under its control. Notably, the communication between different gateways is not a real point-to-point communication because it re-uses the Peis distributed tuplespace so that any Peis component can simply send a message to the i-th Proxy without having to know its IP address. The link between two proxies (at the gateway middle level in Figure 2.1) represents a logic link rather than a point-to-point physical link between proxies.

Chapter 3

Application Programming Interface

The proxy software released at M24 offer a number of programming interfaces, programming abstractions and communication protocols to ease the integration of sensing and actuating resources within a RUBICON system.

The main API of the Proxy can be accessed through the Peis middleware. From the point of view of the RUBICON Control Layer and all other participating components in the RUBICON ecology, the proxy provides an homogeneous mechanism for **introspection**, **data collection** and **configuration** for all different types of sensors and actuators used in the ecology. Notably, all details related to device addressing schemes and network and device configuration protocols are hidden behind the Proxy.

Thanks to the Proxy introspection mechanism, every Peis component (i.e. either an architectural layer or functional component in RUBICON) can be aware of what are the sensing and acting capabilities of the ecology, no matter how they are integrated in the system (i.e. with what protocol). Introspection is a run-time functionality, as the Proxy provides an up-to-date picture of these resources in order to reflect newly connected and/or newly disconnected devices. These resources are described in a homogeneous format, so that, for instance, similar sensors that are connected to different networks (e.g. ZigBee and KNX), appear exactly the same in the Peis tuplespace.

The Proxy data collection mechanism makes available the data gathered by the sensors in the underlying networks to higher layers (e.g. control and learning components), which do not need to account for specific network protocols.

Finally, thanks to the Proxy data collection mechanism, components in the higher layers can simply post configuration instructions, for both sensors and actuators, and see those instructions automatically translated by the Proxy in the specific network protocol and routed to their intended device.

Through the Peis tuplespace and the functionalities offered by the Peis kernel, the proxy cooperates with these higher levels of the robotic ecology architecture to support the implementation of control and learning strategies for dynamic, peer-to-peer networks.

The Proxy services are performed by collaborating with multiple network implementations installed in the proxy. This software has been released with support for both 802.15.4/ZigBee wireless networks and for KNX home automation networks, and with simple mechanisms that system developers can use to support different network protocols.

As a Peis component with peis-id corresponding to the id of the island it supervises, the Proxy, manages a set of

tuples with the following keys:

```
proxy.<networkId>.<deviceId>.property.<propertyId>.value
proxy.<networkId>.<deviceId>.property.<propertyId>.command
proxy.<networkId>.<deviceId>.sensor.<sensorId>.value
proxy.<networkId>.<deviceId>.actuator.<actuatorId>.value
proxy.<networkId>.<deviceId>.actuator.<actuatorId>.command
```

By default, the first part of the key of each tuple is set to proxy to signal that the tuple is managed by the RUBICON Proxy. The other variable elements of the key of each tuple represent, respectively:

- **networkId** - The name of the network to which each device is connected. This starts with the type of the network (e.g. ”wsn”, ”knx”) and it includes a symbolic name used to identify the instance of the network, in the case the same Proxy supervises multiple networks of the same type. By default this is a number automatically generated by the Proxy (e.g. ”wsn-1”, ”wsn-2”) but it can be overridden at installation time, when it is possible to assign a specific symbolic name to each network (e.g. ”knx-floor1”, ”knx-hall”).
- **deviceId** - The unique id of the device. By default this is the address of the device in the underlying network (e.g. the TinyOS device ID, or the KNX device address).
- **propertyId**: The name of a property associated to a particular device or device type. Properties’ tuples represent the value of configuration parameters governing the behaviour of the devices (e.g. ”sampling-rate”, ”duty-cycle”).
- **sensorId** and **actuatorId** - The name of the sensor or the actuator. This starts with the type of the sensor or actuator (e.g. ”temperature”, ”light”, ”switch”) and it includes a symbolic name used to uniquely identify the instance of the sensor to account for cases in which multiple sensors of the same type were attached to the same device. By default this is a number automatically generated by the Proxy (e.g. ”switch-1”, ”switch-2”) but it can be overridden at installation time, when it is possible to assign a specific symbolic name to sensor or actuator (e.g. ”switch-fridge”, ”switch-top-drawer”).

The tuples with leaf sub-key ”value” are used to publish, respectively, the last known sensor reading from each sensor (for sensor-type tuples), the actual state of each actuator (for actuator-type tuples), and the last known value of each device property (for property-type tuples). The tuples with leaf sub-key ”command” are used instead to accept new values with which to set either the property of a device or a status of a specific actuator.

The Proxy creates and publishes these tuples for each known device and utilizes the PEIS-kernel to identify subscribers to specific sensor tuples. This allows initializing sensors only when there exists valid subscribers thus lowering bandwidth usage and energy consumption in the participating devices, which can be disabled until at least one other component in the RUBICON system subscribes to its sensor updates. Finally, the Proxy subscribes to the property and the actuator command tuples, register a Peis callback for each of those tuples so that it is notified of their value changes, and forwards each new value to the corresponding target device. Any Peis component in the RUBICON can thus configure a device or send an actuation instruction (e.g. to activate or de-activate a relay) by simply setting the value of those tuples.

For an example view of the tuples published when the Proxy is running in an ecology with two networks (of type, respectively, KNX and WSN), see Figure 3.1. In this example, a mote is present as mote 1 with accellerometer,

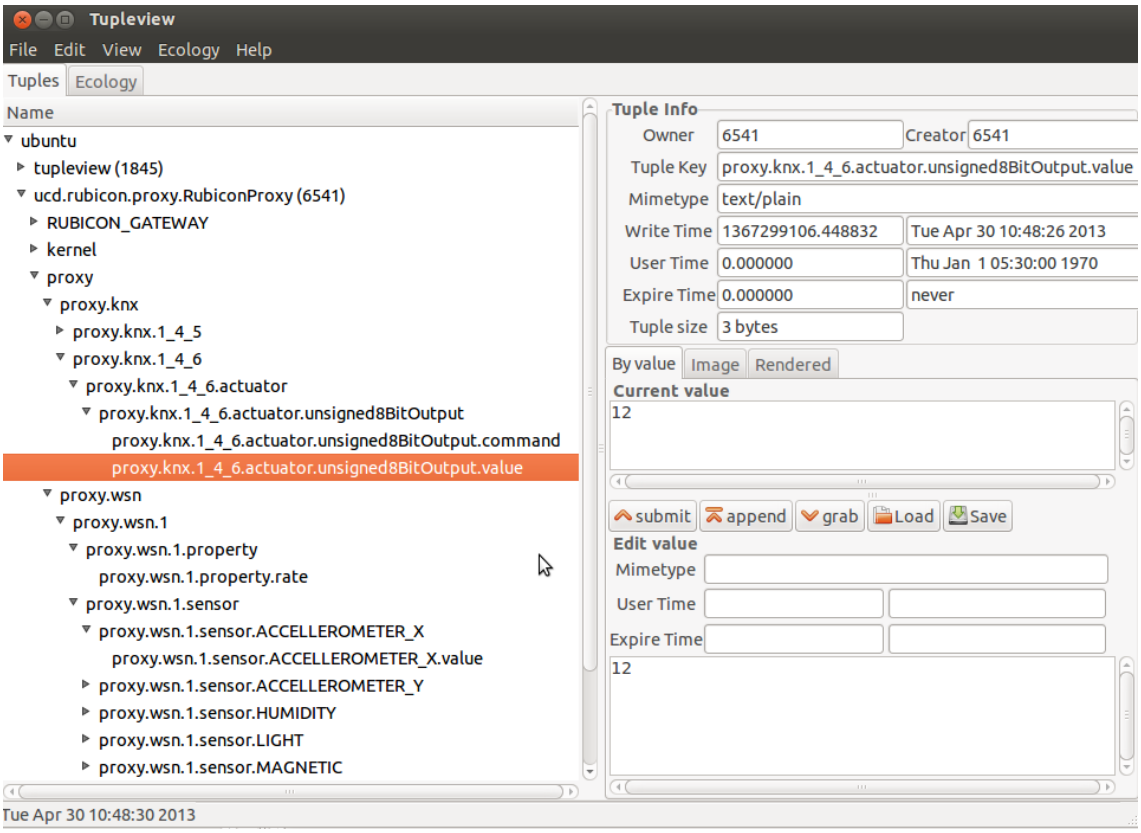

Figure 3.1: Example snapshot of the tuples provided by the Proxy.

light, humidity and magnetic sensors. Figure 3.1 highlights one of two KNX devices also present. Specifically, an unsigned (8 bits) register is used to regulate a light dimmer on KNX address ”1.4.6”.

Chapter 4

Implementation

Figure 4.1 shows a UML class diagram representing the main classes and the interfaces released with the software described in this deliverable.

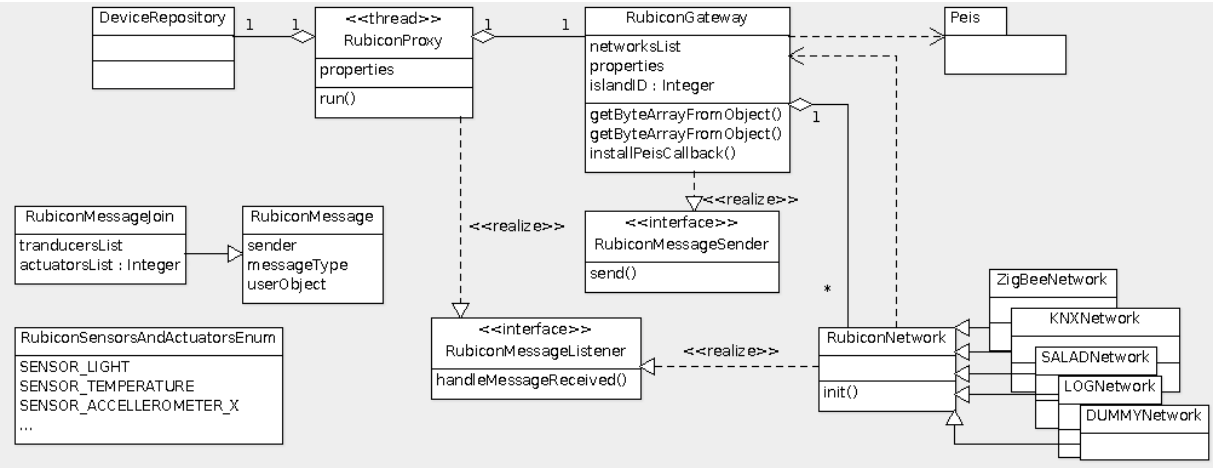

Figure 4.1: Proxy system: Class Diagram

Each new network must subclass the class *RubiconNetwork* in order to access configuration utilities and to communicate with the Proxy. Messages between the network implementation level and the higher levels of the architecture, and between multiple and distributed higher-level components (e.g. multiple proxies and also multiple and distributed instances of the RUBICON Control layer) are encapsulated by object instances of the *RubiconMessage* class, representing the source of each message and its content. Higher level components can define their own specialized version of the RubiconMessage (e.g. to carry application or layer-specific information), and use a simple publish & subscribe API exported by the *RubiconGateway* class to exchange messages with each other.

In order to cross the boundaries between multiple islands, the RubiconGateway class:

- Creates and subscribes to changes of a special tuple, with key 'GATEWAY', which other components in the network can use to post RubiconMessages to any of the components in the island.

- Offers a *send(island, RubiconMessage)* function, which can be used to posts the serialized content of the RubiconMessage class to a remote RubiconGateway.
- Allows clients to add a user object, i.e. a client-specific (but serializible) Java object, to the default RubiconMessage. The user object remains uninterpreted within the Proxy and gets simply passed through the tuplespace together with the RubiconMessage. Together, each RubiconMessage and its user objects are serialized and stored in Peis as binary tuples with mime-type rubicon/application, by availing of the included in Peis generation 6, described in D1.3.1.

In order to support dynamic networks, with devices joining and leaving the system at run-time, for instance, due to device mobility, component failure and power outage, a simple discovery protocol is defined to allow the network layer to signal the presence of new devices, together with the description of the type of the sensors and actuators installed on each device. In the case of WSN/WSAN, the protocol must be initiated at the device level. For WSN/WSAN networks, RUBICON relies on the embedded nesC/TinyOS implementation of the Communication Layer described in D1.3.2, which can be programmed with the description of the specific sensor and actuator types installed on each device, and with the join protocol described in D1.3.2. Specifically, once a new device is first activated, it will initiate the join protocol in order to acquire a valid RUBICON address. After that, the device will be able to communicate with the Proxy instance in charge of supervising the RUBICON island to which the device is connected. Noticeably, the join protocol defined in D1.3.2 is able to resolve cases in which a device can physically communicate with multiple proxies, and also cases in which a device travel among multiple islands. In addition, the devices and the Proxy use a simple keep-alive message protocol to allow the RUBICON to maintain a picture of the available devices. The Proxy listens to keep-alive messages and also to data updates sent on regular intervals by each device. It collects a list of the identifiers of all devices and continuously publishes a list of reachable devices that satisfy the criteria that at least one message have been received from the device during the last period (of configurable duration).

The join message is a pre-defined specialization of the default RubiconMessage (*RubiconMessageJoin*) containing a list of sensors and actuators types. In order to create a unique representation of these types to be used across all network types, the ucd.rubicon.network project defines the *RubiconSensorsAndActuatorsEnum* class to enumerate a wide range of valid types of sensors and actuators. These include the typical sensors installed on the sensor boards of the WSN motes used in RUBICON's application domains, as well as general descriptors for digital/analogic input/outputs, as supported for instance by home automation protocols, such as KNX (e.g. 1 bit outputs used to drive relays, N bytes digital registers used to drive dimmers or more complex actuators). Each sensor or actuator type is described with a 2-byte short number, which will allow the software to be extended to support a maximum number of 65536 types.

The *RubiconMessageJoin* message is a default message that must be directly interpreted by each specific Network implementation. Following the layered design of the RUBICON communication stack presented in D1.3.2, application and/or layer-specific messages can still be explicitly represented by specializations of the RubiconMessage class. However, they are not interpreted by the Network layer but are passed directly to the embedded application installed on the sensing and/or actuating devices. In both cases, the Network implementation must take care of transforming (serializing) each RubiconMessage into the network-specific embedded counter-part.

Five Network classes are provided with the software delivered at month 24, respectively:

- **802.15.4/ZigBee Networks** - whose implementation (available in /1-Comm/ComunicationLayer/Network) is detailed in D1.3.2.

- **Salad Tecnalia Middleware** - a UDP client to the protocol described in D1.3.2, Section 4.2.1, used to access the Tecnalia's SALAD middleware (implemented in 1-Comm/ucd.rubicon.network.salad)
- **KNX** - an interface to native KNX infrastructures (available in ../1-Comm/ucd.rubicon.network.knx) built upon the Calimero Java library.
- **LOG** - a Network implementation that reads sensor data from log files produced with the logger tool described in Deliverable 1.3.2. This LOG Network is provided in order to test the higher layers of the RUBICON architecture (i.e. Learning, Control and Cognitive layers), offline, with real data gathered from the WSAN.
- **DUMMY** - a Dummy Network implementation provided for unit testing purpose.

All the network implementations already supported by the Proxy include a number of built-in messages, which are used by the Proxy to deliver its configuration service. These include the *RubiconMessageCtrlActuateCmd* used to send a new command to an actuator. The 802.15.4/ZigBee implementation also supports the *RubiconMessageCtrlPeriodicCmd* message, which is used to instruct a device to start sending updates for a sub-set of its sensors every N milliseconds (corresponding to a built-in rate property for WSN/WSAN mote-class devices).

Finally, the KNX class allows the RUBICON Proxy to exchange information with KNX bus devices connected via medium twisted pair, radio frequency, power line or IP/Ethernet in environments not covered by the SALAD middleware. Bus devices can be either sensors or actuators and potentially include equipment such as lighting, blinds / shutters, security systems, energy management, heating, ventilation and air-conditioning systems, signalling and monitoring systems, interfaces to service and building control systems, remote control, metering, audio / video control, etc. The Proxy interacts with KNX hardware by way of KNX frames - that are encapsulated in Ethernet frames. To this end, the Proxy uses the library Calimero [2] - a collection of Java APIs that together form a foundation for KNX high level applications.

[2] Available online from http://calimero.sourceforge.net

Chapter 5

Conclusions

We conclude this report with the an overview of the planned tasks achieved in the second year of RUBICON within WP1, Task 1.4: *Adaptation of existing data-sharing middleware and proxy solutions for WSN/robot integration* (M13-24), as documented in the Description of Work, and the performed work as documented in this deliverable and in second interim project report.

Task 1.4 have been performed as planned and finished in March 2013. The result of this task have provided the RUBICON with a modular, network agnostic Proxy component with support for automatic discovery of sensors and actuators, extensibility, and heterogeneous peer-to-peer networks.

Task 1.4 involved the identification of common and extensible component description and mechanisms for automatic discovery of hardware and processing capabilities and the provision of standard mechanisms that the higher layers of the RUBICON system can use to interact with the underlying sensing and acting infrastructure, including computational constrained devices. In order to support heterogeneous networks and operate within the research and application testbeds used in RUBICON, the Proxy component has been implemented as a Peis peer-to-peer component and integrated with the network mechanisms and middleware described in D1.2 and D1.3.2.

5.1 Impact

The software described in this deliverable is accessible in the software repository containing all the code and examples needed for the execution of the Communication Layer, including testing scripts and example classes that can be used as templates to guide the integration between the RUBICON Proxy and other networks besides the ones already supported.

The software is currently present at and in active use by all of the partners. Based on the feedback of the partners, further work is expected in order to validate the performance and tune all the mechanisms presented in this deliverable once they are operating in the final application testbeds.

No significant unexpected developments have occurred during the execution of the second year of WP1.