RUBICON
Robotic UBUquitous COgnitive Network

Project No.: 269914

D1.1 – Functional Design & Specification & Mockup Layer

Editor: | |
---|---|---
Editor: | **Claudio Gennaro** | **ISTI-CNR**
| **Mathias Broxvall** | **ORU**
| **Claudio Vairo** | **ISTI-CNR**
Contributor(s): | **Giuseppe Amato** | **ISTI-CNR**
| **Mauro Dragone** | **NUID UCD**
| **Alessio Micheli** | **UNIPI**
| **Stefano Chessa** | **UNIPI**
| **Davide Bacciu** | **UNIPI**
| **Claudio Gallicchio** | **UNIPI**
| **Alessandro Saffiotti** | **ORU**

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)

© Copyright RUBICON - All Rights Reserved

Issue Date	30/09/2011 (M9, MS1)
Deliverable Number	D1.1
WP	WP1 - Communication Layer
Status	☐Draft ☐Working ☒Released ☐Delivered to EC ☐Approved by EC

Document history			
V	Date	Author	Description
0.1	01/09/2011	Claudio Gennaro	First very draft
0.1	05/09/2011	Claudio Vairo	Added section 4.2.1
0.2	09/09/2011	Claudio Gennaro	Added Requirements
0.3	16/09/2011	Mathias Broxvall	Added PEIS description in Section 3. Updated Sections 4.2 & 5
0.4	23/09/2011	Claudio Gennaro	Added specification and description of the join island + figure.
0.5	27/9/2011	Claudio Gennaro	Updates to the implementation issues.
0.6	28/9/2011	Mathias Broxvall	Added section about proxies in the implementation issues.
0.7	30/9/2011	Mathias Broxvall	Rewrote section , and split into separate section.
0.8	03/10/2011	Claudio Gennaro	Final release

Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The document reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

Executive Summary

This report describes project activities related to Task 1.1 – "Specification and design of the network", which analyzes the state of the art solutions for networking in WSANs already available in the consortium and out of the consortium, in order to assess a baseline on which we will build our solutions for supporting the RUBICON communications.

Contents

1. INTRODUCTION 8
1.1 OVERVIEW OF THE RUBICON COMMUNICATION LAYER 8
2. REQUIREMENTS 10
2.1 PROVIDED INTERFACE REQUIREMENTS 10
2.2 NON-FUNCTIONAL REQUIREMENTS 12
3. BACKGROUND TOOLS AND TECHNIQUES 13
3.1 THE PEIS-MIDDLEWARE 13
3.1.1 Introduction 13
3.1.2 High-level PEIS-kernel design 14
3.1.3 API reference 15
3.1.4 Internal organisation of the PEIS-kernel 17
3.1.5 The Linklayer 18
3.1.6 The P2P layer 20
3.1.7 The Tuple Layer 24
3.2 PROXIED OBJECTS 27
3.2.1 Introduction to proxied objects 27
3.2.2 Design pattern 28
3.3 MAD-WISE STREAM SYSTEM LAYER 30
3.3.1 Introduction 30
3.3.2 The Stream System 31
3.3.3 Implementation Details 34
3.3.4 Conclusions 38
3.4 SMEPP LIGHT 39
3.4.1 Introduction 39
3.4.2 Requirements 39
3.4.3 Specification 41
3.4.4 Energy Efficiency 45
4. HIGH-LEVEL DESIGN 48
4.1 DESIGN OVERVIEW 48
4.1.1 Gateways and Proxies 49
4.2 IMPLEMENTATION ISSUES 53
4.2.1 High level architecture overview 53
4.2.2 Synaptic channels 55
4.2.3 Multitasking 55
4.2.4 Discovery 56
4.2.5 Implementation of proxied objects and processes 56
5. INTERFACE SPECIFICATION 58
5.1 SYSTEM OVERVIEW 58
5.2 TRANSPORT SUBLAYER AND ADDRESSING 59
5.3 SPECIFICATION OF THE MOCKUP LAYER 59

5.3.1 Specification of the functions of the Synaptic_Channels component *59*
5.3.2 S Specification of the functions of the Streams component *61*
5.3.3 Specification of the functions of the Connectionless component *63*
5.3.4 Component Management *64*
5.4 INTERFACE SPECIFICATION FOR PEIS-WSN GATEWAY 64

6. ACKNOWLEDGEMENTS 66

7. REFERENCES 67

Abbreviations

RUBICON	Robotic UBIquitous COgnitive Network
WSN	Wireless Sensor Network
MaD-WiSe	Management of Data in Wireless Sensor networks
Mote	A wireless sensor network that is capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network.
PEIS	Physically Embedded Intelligent Systems

Figures

Figure 1 - Role of the RUBICON Communication Layer 8

Figure 2 - Relationship of Communication Layer with the devices and the other Layers 9

Figure 3 - The PEIS-kernel stack illustrating the internal modules and layers of the PEIS-kernel (yellow), PEIS-components of a special nature essential for the middleware (blue) and application components (red). 15

Figure 4: Example of a proxy using RFID as interface channel. Proxy manager receives signatures from all interfaces, looks up and creates the proxy for the detected object. The robot communicates with the proxy like for any other device in the ecology. 30

Figure 5: Stream types. T = transducer, W = writing operator, R = reading operator. 31

Figure 6: Command/event sequence when opening a remote stream from sensor A to sensor B (App = Application, SS = Stream System, Net = Network). 36

Figure 7: Command/event sequence when data is written to a remote stream (App = Application, SS = Stream System, Net = Network) 37

Figure 8: Stream System Module interaction diagram. Full arrow lines indicate commands while dashed arrow lines indicate events. 38

Figure 9: Components in SMEPP Light architecture 43

Figure 10: Creating a group 44

Figure 11: Duty cycles and subscriptions 46

Figure 12: Energy consumption (in mA-hr) 47

Figure 13 - Type of communications and devices involved. 48

Figure 14 - Topology of the RUBICON Ecology 49

Figure 15 - Communication between two Motes (A-to-B) in the same island. 50

Figure 16 - Communication between two Motes (A-to-B) in remote islands 51

Figure 17 - Communication between a mote and a remote PC/ Robot. 52

Figure 18 - The communication messages between a mote and a PC in details 53

Figure 19 - The communication messages between two foreign motes in details. 54

Figure 20 - Sequence diagram of the communication between two motes of the Control Layer 54

Figure 21 - Software architecture for the Communication Layer: logical sublayers are separated by a dashed line, software components are small rectangles. 58

Tables

Table 1: Interface of SMEPP Light 40

Table 2 - Summary of the communication relationships.. 52

1. Introduction

1.1 Overview of the RUBICON Communication Layer

Figure 1 depicts the conceptual, layered high-level architecture of the RUBICON system, outlining the main responsibilities of each layer and their main interactions and emphasising the Communication Layer.

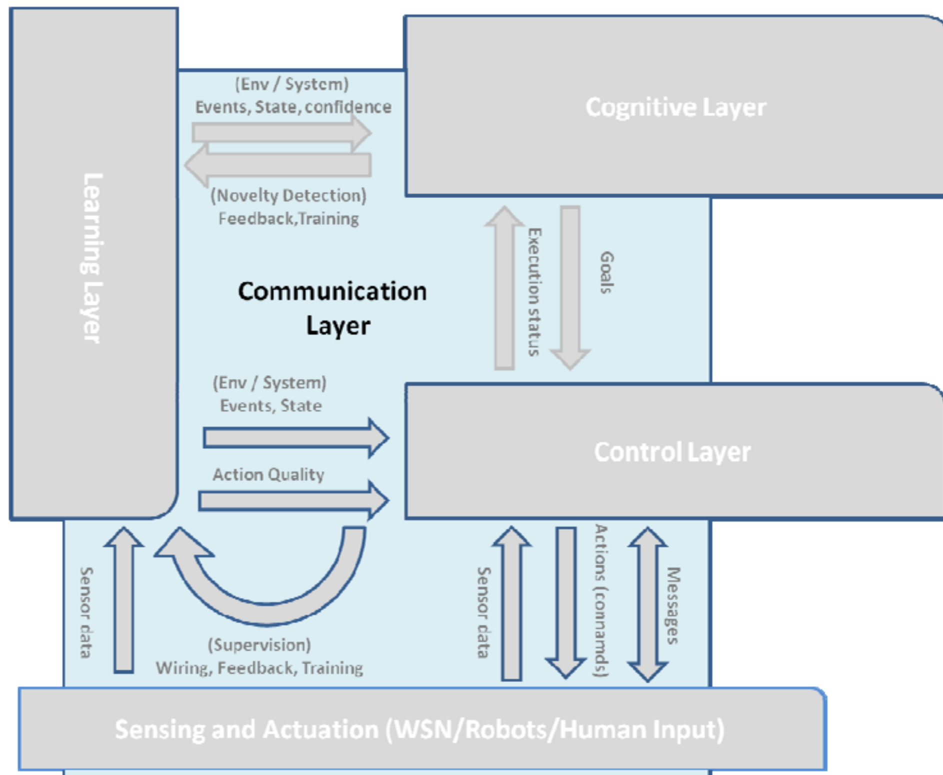

Figure 1 - Role of the RUBICON Communication Layer

The aim of the Communication Layer is to provide a reliable communication infrastructure for the RUBICON framework. The implementation of communications and hardware abstraction services are driven by the requirements dictated by WP2, WP3, and WP4, in order to enable the sharing of data and functionality required by the RUBICON. To meet these requirements the communication layer will be mainly based on the two existing Software: the StreamSystem Middleware by ISTI-CNR and the PEIS Ecology middleware by ORU.

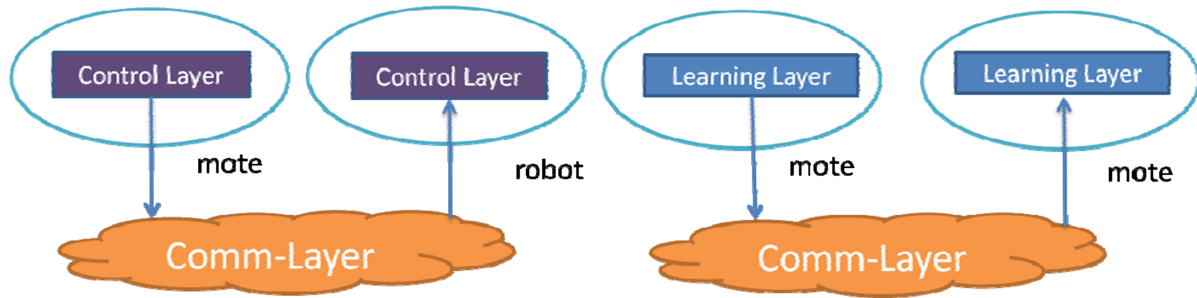

Figure 2 - Relationship of Communication Layer with the devices and the other Layers

The main objective of this layer is to provide different type of communicating mechanisms for exchanging information between Layers running on remote devices (robot, pc, motes, etc). See Figure 2. In particular, the Communication Layer will make available different paradigms of communication on the basis of the type of hardware involved in the communication. This aspect is well described after the analysis of requirements provided in the next Section 4.

2. Requirements

In order to organize the overall presentation of the requirements across D1.1, D2.1, D3.1 and D4.1, this section starts by reporting the functional requirements associated to interfaces that this layer must provide to other layers of the RUBICON architecture, as discussed in D2.1 and D4.1. These requirements, and the use cases collected in D0.1 are then examined in order to infer other functional requirements describing the desired behaviour of the Control Layer, as well as non-functional requirements concerning aspects such as openness (the ability of handling nodes entering and leaving the system), fault-tolerance, etc. Finally, this section examines the requirements requested to the other layers of the architecture.

Requirements are named *RX.Symbol*, where *X* is the first number of the deliverable (matching the WP, e.g. 1 for WP1, 2 for WP3...) and *Symbol* is a short name reminding of what is the flow of data, control, or functionality associated to the requirement, as in the high-level architecture diagram of Figure 1.

2.1 Provided Interface Requirements

This section briefly reports the requirements associated to functionalities/data/control that this layer must provide to interface with the other layers of the RUBICON architecture. These requirements are extensively discussed in the deliverable of the layers requesting them. However, in this section the same requirements are reported and examined from the perspective of the Control Layer in order to ease the analysis of other requirements.

NAME	DESCRIPTION
R2->1.INPUTSTREAM From functional requirement R2.PREDICTION	The Learning Layer requires the timely delivery of input data collected by the sensors' transducers and other available off-the-shelf software components in order to progress with the computation of the predictions of the Learning Network. This requirement is needed in order to satisfy the functional requirement R2.PREDICTION as, in order to provide the requested output prediction at each RUBICON clock tick, the LN needs to have all the inputs readily available at its input interface.
R2->1.COMMUNICATION From DoW, "Work Package Description" for WP2 and the Section 1.3 "S/T Methodology and associated work plan"	The Learning Layer requires the Communication layer to set up synchronous communication channels among the learning components residing on different nodes of the RUBICON ecology . These are used to transmit configuration and control information (e.g. wiring data, components of the learning modules) as well as input data. The goal of the Learning Layer, as described in the Section 1.3 of the Dow, is to "deliver a distributed, adaptive, and self-organizing memory comprising independent learning neurons residing on multiple nodes of the ecology. These neurons will interact and cooperate through the underlying communication channels provided by WP1". Therefore, the Learning Layer needs the Communication Layer to setup appropriate

	communication channels between the distributed learning modules.
R3->1.SENSING From R3.STATE, R3.RELIABILITY, R3.EXECUTION & CONFIGURATION	In order to build and maintain an up-to-date picture of the state of the robotic ecology and its environment (R3.STATE), and to enable collaboration between members of the robotic ecology (e.g. communication of localization data from the ceiling camera to the robot, in the AAL scenario, see R3.EXECUTION & CONFIGURATION), the Control Layer must be able to receive data and periodic status updates from every sensor and actuator it wishes for. In order to support R3.RELIABILITY, the Control Layer should also be able to specify the desired update rate and to be informed of the maximum latency to be expected by the resulting updates. The Control Layer may tolerate the loss of some of these updates but all data must be time stamped in order to be able to ignore old updates.
R3->1.ACTUATION From R3.EXECUTION & CONFIGURATION, R3.RELIABILITY	The Control layer must be able to send control instructions (e.g. new set points, new output values) to every actuator it wishes for. For this type of transmission, the Control Layer does not require the ability to communicate periodic updates of control instructions. However, in order to support R3.RELIABILITY, transmission of control instructions should be reliable (acknowledged). In addition, the Control layer needs to be informed of the maximum expected latency.
R3,4->1.DATA SHARING From R3.KNOWLEDGE SHARING, R3.DISTRIBUTION, R3.EXECUTION & CONFIGURATION,	In order to support R3.KNOWLEDGE SHARING, R3.DISTRIBUTION and R3.EXECUTION & CONFIGURATION, the Control Layer must be able to (asynchronously) share it sensor data, actuator status and other information among distributed nodes (multiple robots, WSN nodes and other devices).
R3,4->1.MESSAGES From R2->3,4.CONTROL R3.DISTRIBUTION, R3.EXECUTION & CONFIGURATION,	In order to support R2->3,4.CONTROL, R3.DISTRIBUTION, and R3.EXECUTION & CONFIGURATION and co-ordinate its operation across distributed nodes, the Control Layer must be able to send reliable and synchronous control messages to all the nodes it wishes for.
R3,4->1.DISCOVERY & TOPOLOGY From R3.OPENNESS, R3.RESOURCES, R3.STATE	In order to support R3.OPENNESS, R3.RESOURCES and R3.STATE, the Control Layer needs an updated picture of all the components available in the system, including all the WSN nodes currently active. Every component should have a unique ID and the Control Layer should be informed whenever any robotic device or WSN nodes join (as they become operative and connect to the network), or leave the system (as they get disconnected, breaks, they battery get depleted or simply move out of network range

2.2 Non-Functional Requirements

NAME	DESCRIPTION
R1.DEBUGMODE	All the modules implementing the Communication Layer must be able to run in debug mode. In this modality, the software generates log files containing information about the messages exchanged. You should not be running in debug mode except when you are trying to isolate a problem because it causes the size of the log files to grow quickly.
R1.MULTITASKING	The Communication Layer must be able to support the communication of more than one process running on the same Mote.
R1.ROUTING	The Communication Layer running in a mote knows which island it belongs to and the corresponding *basestations* that must be contacted for routing messages.
R1.SCALABILITY	The topology and the routing mechanisms of the Communication Layer must be designed to support a growing number of devices without degradation of performance in communications.
R1.DISTRIBUTION	The Communication Layer must be distributed across the Motes, PCs, and Robots in order to minimize network bandwidth, latency, and energy consumption.

3. Background tools and techniques

Within the RUBICON project, we will rely on two main background technologies, the PEIS-middleware and the MaD-WiSe Stream System, for the implementation of the communication layer in WP1. These two background technologies implements partly overlapping services but with important differences in hardware requirements and with services targeted towards applications for robotic devices and for distributed wireless-sensor-networks, respectively. This allows for efficient WSN networks while at the same time allowing for advanced planning and dynamic reconfiguration of robotic tasks at the level of computationally more capable devices.

To distinguish between the type of devices that can be reached natively using the PEIS-middleware and the MaD-WiSe Stream System we will refer to the networks as the PEIS-network or PEIS-ecology the wireless-sensor-network or networks (for the case of multiple islands of WSN), respectively. Note that we use the term wireless-sensor-network somewhat loosely here since the nodes under consideration also can contain actuation devices.

We expect this hybrid approach to give a synergetic effect and to allow for the implementation of very heterogeneous devices in the RUBICON ecology.

In this section we will also describe the SMEPP Light middleware, from which we borrow the publishing/subscription mechanism that can offer functionality to develop the auto discovery functionality.

3.1 The PEIS-middleware

Due to the nature of the background PEIS-middleware and the role that it plays both in the communication and in the control layers it is described both in deliverable D1.1 and D3.1 – albeit with a slightly shift in focus on the communication, configuration and control aspects of the middleware, respectively.

We will here describe only the aspects of the PEIS middleware related low level communication, P2P based message routing and the higher level tuple based communication needed for the implementation of proxies. Most notably, we refer the reader to D3.1 for details related to control mechanisms such as dynamic re-configurability.

3.1.1 Introduction

The PEIS kernel and related middleware tools are a suite of software previously developed as part of the *Ecologies of Physically Embedded Intelligent Systems* project in order to enable communication and collaboration between heterogeneous robotic devices [1]. This kernel is a software library written in pure C and with as few library and RAM/processing dependencies as possible making it suitable for a wide range of devices. The original purpose of this library was to enable software programs running on PC (Linux, MacOS, and Windows), PC/104 (Linux/RTAI), Gumstix (uCLinux) to participate as PEIS components in the PEIS Ecology network. This network is composed of an heterogeneous set of mobile robots and networked sensors and actuators and the middleware is used to enable communication and collaboration services between these devices that are in many aspects non-overlapping (and orthogonal) to hardware centric robotic middlewares such as Player/Stage and ROS.

Since there exists bindings for the PEIS middleware to several other programming languages, most notably Java, this middleware is a suitable candidate around which to build the collaboration aspects of the robotic devices in the RUBICON ecology.

Although there exists a TinyOS based version of the PEIS-kernel (TinyPEIS), a design decision was made to not rely on this version for RUBICON. The main reasons for this include the required RAM memory (4kb) and processing requirements that are not compatible with the planned resource utilization of the learning layer (see WP2). For this reason, a hybrid approach is required with another pure communication middleware running on the TinyOS based devices and with dedicated bridge devices that synchronises messages and collaboration tasks between the robotic devices and the Wireless Sensing and Actuation devices.

In the remainder of this section we will describe the communication aspects of the PEIS middleware *as applicable to the RUBICON project*. For the non-communication aspects, we refer the reader instead to Deliverable D3.1. For the TinyOS based communication layer, see Section 3.3, in this deliverable.

3.1.2 High-level PEIS-kernel design

The most important design requirements of the PEIS-kernel have been to provide a decentralized mechanism for collaboration between separate processes running on separate devices that allows for automatic discovery, high-level communication and collaboration through subscription based connections and dynamic self-configuration. These and any additional services should all allow any devices to communicate/collaborate with any other devices as long as there exists any, possibly indirect path of communication between the devices.

For this requirement, a first design choice of the PEIS-kernel was to abstract the notion of communication links, with current implementations for using TCP/IP links, UDP/IP links and Bluetooth devices for the main PEIS ecology.

On top of the basic communication links, a P2P layer implements multi-hop routing and probabilistic broadcasts between nodes as well as acknowledgement, multi-fragment messages and reliability services.

Finally, on top of these communication services a set of higher-level services including eg. a subscription based *tuplespace* mechanism, dynamic-reconfiguration mechanisms is implemented.

From the point of view of WP1 mainly the first two layers, the communication and P2P, are of interest. For the other services and layers, we refer the reader to deliverable D3.1.

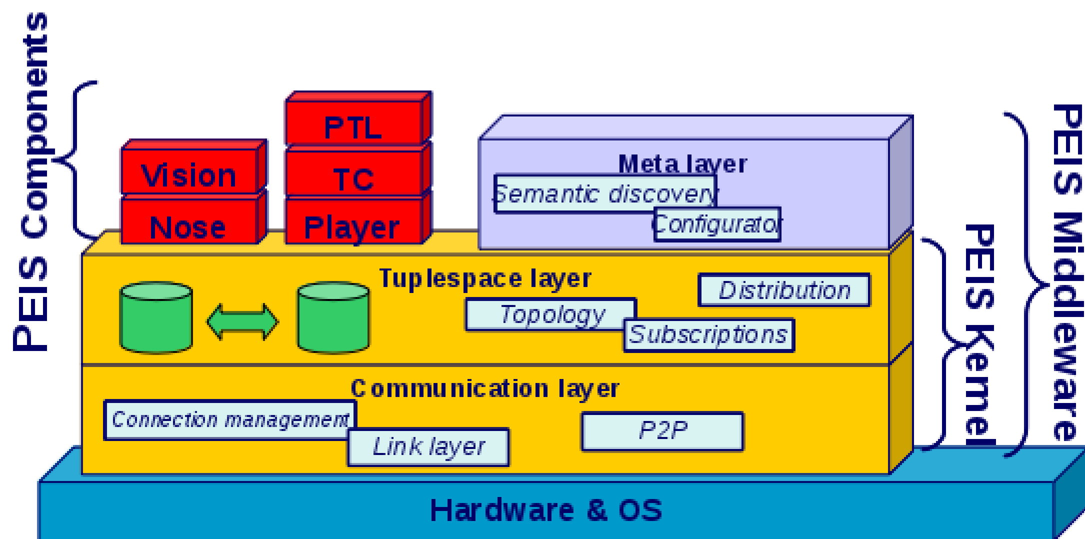

Figure 3 - The PEIS-kernel stack illustrating the internal modules and layers of the PEIS-kernel (yellow), PEIS-components of a special nature essential for the middleware (blue) and application components (red).

In addition to the direct services provided by the PEIS-kernel library (available inside every process linked to the library) there are also a number of essential services that are provided only by a few specialized processes. These services are typically only intended to be run on one process on each host of the network (peisinit) or for debugging or visualization (tupleview) purposes or are of other specialized nature such as action or configuration planners.

Figure 3 illustrates this distinction between what is part of the PEIS-kernel, and what are specialized components or semi-essential services of the PEIS middleware. Again, we refer the reader to deliverable D3.1 for an overview of these other services.

3.1.3 API reference

The following subset of the PEIS-kernel API related to the P2P layer constitutes the main interface to the other communication layers in the RUBICON ecology (ie. to the dedicated WSN/WSAN motes). For all PEIS-kernel functions, the prefix `peisk_` is used in the API functions but are omitted here for simplicity. Furthermore, all internal *datastructures* of the specific PEIS-kernel instance associated with each separate process can be accessed through the globally accessible variable `peiskernel` in the memory space of each such process.

The list of all known hosts in the current ecology can be accessed as a hash table

```
peiskernel.hostInfoHT
```

This provides an interface to the automatic discovery mechanism and allows for a low-level access of every known PEIS device that can be reached from the current network.

Manipulation of this hashtable can be made using the hashtable API functions:

```
hashTableIterator_first,hashTableIterator_next,
hashTableIterator_value
```

The former two provides the mechanism for iterating over all known hosts while the later returns a datastructure of type `hostinfo`, presenting the hostname, unique *peis-id* and magic ID and different known low level addressing mechanisms that can be used to establish direct links to it.

The basic mechanism for sending and receiving low-level messages within the PEIS ecology focus around the notion of *ports* which are an integer describing the targeted service within different *destinations* – specified using the unique *peis-id* identifier that is given to each process participating in the PEIS ecology. These identifiers are part of the host information given above.

To send data to a targeted service within any other PEIS process on the network use the *sendMessage* functionality.

```
sendMessage(port, destination, length, data, flags)
```

This functionally will split the message into fragments that fit within the size limitation of the underlying *linklayer* connections and, depending on the selected flags, will queue the message for re-transmission until it has been verified by an acknowledgement package from the destination (or until the destination is known to be non-routable).

For the point of view of WP1 this mechanism will be used for transmission of message between the gateway devices in the implementation of the dedicated WSN island-to-island routing mechanism.

The basic service for receiving packages is by registering a callback function that is invoked by the kernel whenever a message is received on a specific port:

```
registerHook(port, function-pointer)
```

Where the callback is a pointer to a function that processes the received data package. This allows for a non-polling mechanism for receiving messages.

In addition to sending messages to a specific host known by its identifier, the PEIS-kernel also allows for broadcasts that will ensure that a given message is received by every PEIS that can be reached in the network.

```
broadcast(port, length, data)
```

This mechanism is used to implement many of the decentralized services that synchronises eg. the tuplespace and the distributed clock on the network.

From the point of view of WP1 this mechanism will be used for detecting the available gateway devices in the implementation of the dedicated WSN island-to-island routing mechanism.

For the communication between gateways and robotic devices, WP1 will utilize the distributed tuplespace of the PEIS-kernel. A complete list of relevant API calls for this is provided in the automatically generated documentation [10], but we present here a reference of the most vital methods. We give first the two convenience methods used to create and insert tuples directly from a given set of data and using default values for all other properties such as timestamps and expiry dates.

```
setTuple(key, len, data, mimetype, encoding)
setRemoteTuple(owner, key, len, data, mimetype, encoding)
```

For general purpose tuples the following functions are used insert tuples into the tuplespace.

```
initTuple(&Tuple) initAbstractTuple(&Tuple)
setTupleName(&Tuple,name)
insertTuple(&Tuple)
appendTupleByAbstract(&Tuple,nlen,added-data)
```

Read of tuples are generally performed with either one of the direct reading methods, by using the associative search based reading, and/or by registering a callback on the tuples.

```
subscribe(owner,key) subscribeByAbstract(owner,key)
getTuple(owner,key,flags) getTuples(owner,key,ResultSet*)
getTupleByAbstract(&Tuple,flags)
getTuplesByAbstract(&Tuple,ResultSet*)
registerTupleCallback(owner,key,userdata,callback-fn)
registerTupleCallbackByAbstract(&Tuple,callback-fn)
```

In addition to these there are a large number of API calls relevant to the manipulation and reading of tuples from the distributed tuplespace. We refer the reader to the doxygen documentation of the PEIS-kernel for this purpose.

For the point of view of WP1 these tuple manipulation API's are of interest in the implementation of the gateway devices tuplespace based communication with robotic devices.

3.1.4 Internal organisation of the PEIS-kernel

Internally, the PEIS kernel consists of a number of modules that implement the different services provided by the kernel. These services are implemented using the notion of periodic functions and event-hooks that can be registered during the initialization of each module (which is done during the initialization of the kernel).

A periodic function is a function that will be attempted to be invoked by the kernel at a fix frequency. Since the kernel is not dependent on multi-threading this is done by an internal scheduling that calls the functions at the appropriate time. This scheduler is run whenever the *peisk_step* function is run which must be done manually by users of the non-threaded PEIS-kernel and is done automatically by the more common multithreaded PEIS-kernel. In the former case, applications should call the scheduler at a minimum of 20 Hz and in both cases should ensure that the scheduler is not blocked longer than 50ms.

The scheduler will call each periodic function with up to the frequency specified for it. In general, it does not call a function multiple times in order to "catch up" if the step function have been delayed. This allows for (1) gradual degradation when CPU usage is 100% and (2) to register *periodics* with a period time of 0.0 in order to be called on every step.

Event-hooks exist not in the generic sense but rather for specific network packages and for specific tuple updates. They are registered by high-level modules (or user programs) and are called when the corresponding network package or tuple is detected by the kernel. Again, since the core kernel is non-multithreaded this is typically done indirectly by the scheduler calling a periodic function, which triggers the corresponding event upon receipt of a network package or the timeout of a package/expiry of a tuple.

The modules *currently* implemented in terms of these two types of functions (and additional API functions) can loosely be organized as a number of layers providing services to higher layers, and in the end the software application. The three most important layers are:

- The Link Layer, which provides basic communication channels between different software instances on the same computer, or distributed on any computers on a Ethernet network, connectible by bluetooth connections or any future additional link mechanisms.
- The P2P Layer, which creates an ad-hoc peer-to-peer network on top of all available network links in order to let any PEIS component in the ecology communicate with any other components.
- The Tuple Layer, which implements the basic tuplespace on top of the P2P layer. Although this layer provides the core of the services used by applications, some of the other library functions are of use in some applications.

For a complete list of all modules in the PEIS-kernel, see the Doxygen page for software modules included in the PEIS-kernel G6 release, available online at [10]. Also note that some of the implementations with periodic functions and sockets have been made more efficient than apparent in this document by using e.g. Unix signal masks instead of polling reads, efficient *datastructures* – these standard computer science techniques have been omitted in this document to simplify the description of the basic concepts.

For the purpose of RUBICON, we plan to build gateway devices that utilize some of the same mechanisms as for the PEIS-kernel modules, most notably access to primitive peer-to-peer network communication mechanisms. These gateway devices can also be considered a part of PEIS-kernel as well as of the general RUBICON middleware.

3.1.5 The Linklayer

This is the lowermost layer in the overlay network dealing with any connections in the PEIS network (which should not to be confused with the OSI *linklayer*) The API layer of the OSI stack for e.g. TCP/UDP can be considered the *linklayer* for the PEIS-kernel.

The main purpose of the *linklayer* is to provide a notion of primitive connections for the P2P layer. It offers these services to the higher layers:

- Notion of low-level addresses that can be used in establishing connections
- Discovery of possible connections to establish
- Connections that can send 1024 byte large packages to neighbours.

The *linklayer* is built with modules for different link mechanisms. Currently we have the following mechanism, as more esoteric hardware becomes available we may add more mechanisms.

- *TCP/IPv4* (currently the dominant link mechanism) provides bidirectional links to any other PEIS on the same Ethernet network.
- *UDP/IPv4* (currently not used) has a few advantages with regards to speed, overhead and latencies but have to throttling manually.

- *Bluetooth* creates connections between any bluetooth connected PEIS as soon as they enter the same physical space. It allows for a much simpler (even nonexistent) infrastructure, ie. no network configurations are required for the PEIS to find each other.

Some of the modules (ie. the bluetooth module) can be enabled/disabled during compile time depending on the available low-level libraries.

Future developments that are not planned as part of the RUBICON project include adding *linklayer* modules for raw access to WiFi networks and raw access to primitive ARP messages. These should be implemented to remove some of the constraints on the current TCP/IP infrastructure – although this requires elevated privileges to run on most operating systems. Also, a simple Unix-socket or shared-memory space *linklayer*s may be implemented for very high throughputs of components onboard the same physical hardware and is planned to reduce the latency of ROS to ROS communication passing through the PEIS layer, as described in Deliverable D3.1.

Each *linklayer* module exports upon initialization time a list of interfaces that it can use to establish connections. An interface represents different physical devices or logical networks (of the *linklayer*) along which the *linklayer* can communicate. Each *linklayer* module there has an associated lowlevel-address that uniquely names the interface and, when appropriate, can be used for establishing connections.

Each process that runs the PEIS-kernel can have more than one *linklayer* module active at each time, and more than one interface available in each such module. The lowlevel-addresses from each *linklayer* module and interface is given in the *hostinformation* structure together with the global peis-name and *peis-id* of the process.

3.1.5.1 TCP/IPv4 linklayer

This *linklayer* uses TCP connections on top of IPv4. For each physical and logical IPv4 network interface on the machine a corresponding IPv4 *linklayer* interface is established. For the purpose of RUBICON, we plan to rely primarily on this *linklayer* for the communication between robotic devices and between the gateways that connect disparate WSN islands.

Each process that runs the PEIS-kernel allocates a TCP port (default 8000) that it will use to listen for incoming connections, with the same port number running on each interface. Furthermore, each such process opens the multicast port 227.1.3.5:10001 for listening to announcements on the local networks. (This is an alternative to processing ARP requests to simplify portability and required privileges).

During run time, each process that runs the PEIS-kernel sends out periodic announcements on the above multicast port publishing its `hostinfo` structure.

It provides an API function for attempting a connection to a specific other TCP/IPv4 lowlevel-address. Furthermore, it periodically listens to the incoming socket and creates a connection structure for each successful incoming connection.

It also provides a functionality for sending an atomic package along a given connection. Each such package is up to 1024 bytes large.

3.1.5.2 UDP/IPv4 linklayer

This is a *linklayer* interface using UDP over IPv4. For the purpose of the RUBICON project, we plan on not relying on this *linklayer* due to the basic constraints posed by UDP traffic.

3.1.5.3 Bluetooth linklayer

Bluetooth is an alternative *linklayer* that can be used standalone or in conjunction with the TCP *linklayer*. The bluetooth *linklayer* module is compile time optional, and requires that `libbluetooth-dev` is installed to compile.

The bluetooth layer can use one or more bluetooth adapters, these are given as commandline options using `--peis-bluetooth hciX` where hciX is a specific bluetooth adaptor. Use `hcitool dev` to see which bluetooth adaptors are available on your system. It can use multiple adaptors by being given multiple instances of that option. It does not require to use all adaptors available on the local machine.

Currently, PEIS components need to be run with root privileges in order to be able to use the bluetooth adaptors, this is necessary to enable RAW interface mode to overcome some of the limitations in the current linux bluetooth interface drivers and libraries.

Multiple PEIS components can be run using the same bluetooth adaptors. However, only one component can use the adaptor for "broadcasting", which announces that this bluetooth adaptor is connected to a PEIS component. Currently, the first PEIS that opens the device gets this privilege, and hence, when that PEIS is closed no other PEIS will make incoming connections to any PEIS on this computer. In the proposed service level BDI architecture of WP3 this is not a limitation since the first PEIS to run on each component, *peisinit,* will also be the last to terminate.

In order to find other PEIS that have bluetooth adaptors, each PEIS with an adaptor performs SCAN operations using the bluetooth adaptor in RAW mode. This gives a list of all other bluetooth devices (not just PEIS) in the vicinity, as well as their signal strengths. The signal strengths are reported as tuples, to be used by higher-level applications to compute locations of devices.

By regularly attempting to connect using L2CAP port 7315 (HELLO connections) to detected bluetooth devices, it can be determined if they are PEIS and basic host information data can be transmitted between the two. This also transmits the port numbers of any ports for incoming connections.

By using the signal strengths, and the results of the periodic HELLO connections, it is possible to determine when a bluetooth connection between two PEIS might be possible. This is reflected by the *bluetoothIsConnectable* function, and is used by the *connection manager* to determine which devices should connect to whom.

3.1.6 The P2P layer

The middle layer of the PEIS-kernel consists of a basic P2P overlay network, which uses any forms of network links provided by the lowermost overlay *linklayer* to create an ad-hoc P2P layer discovering and connecting all PEIS.

It provides these basic services to other parts of the PEIS kernel *and to the MaD-WiSe Stream System layer gateway.*

- routing of messages towards destinations multiple hops away
- port/pin based communication. Any PEIS can send a message on a given port of any other named PEIS or broadcast on a given port to reach all available PEIS in the environment. The ports typically denote different modules in the targets PEIS-kernel. Communication can be

initiated with unknown PEIS by a broadcasted request and continued by sending messages back to the replier.

- long messages: splits up messages longer than *linklayer* limit (1Kb) into multiple packages sent individually towards destination
- reliable messages: see Section 3.1.6.8 for details how this is implemented. This allows for the automatic resending of messages until they have been acknowledged by the receiver, or until a timeout have been reached. In either case, a user defined hook can be triggered to deal with the success/failure, which allows higher modules to take appropriate actions.
- port based hooks: calls user defined functions when a package belonging to a given port passes through or targets this host. Allows the functions to intercept or ignore the packages.

3.1.6.1 Connections

The basic network interface between the P2P layer and the Link layers are the notion of connections that are established between any PEIS devices. Each connection have five queues of packages that are to be transmitted along the connection one at a time through the underlying *linklayer*s. The queues are sorted by priority and all packages on a higher priority queue must be transmitted in full before lower priority queues are transmitted. The higher priority queues must be used very selectively, keeping mind that they may cause lower priority messages to fail and possibly generate further traffic to deal with the failures.

When a higher module requests to send a point-to-point or broadcast message to a specific other *peis-id* and port combination it provides flags that specify which priority should be used and if the message requires an acknowledgement.

3.1.6.2 Routing tables

Internally The P2P layer keeps track of a routing table and propagates these along the neighbours links whenever the routing information changes. This information is stored in a local hash table.

For every other known component, a hash table entry is created with routing information for this host. The routing information consists of the *peis-id* of the host, the magic number of the host, the sequence number of the last known routing information to this host, the estimated hops (metric distance) to this host as well as a timer used by the *knownhosts* updating service.

Additionally, a simplified version of this hash table is stored for each connection, listing the estimated routing information to each host as if it had to go through this connection.

A periodic function is used to update the global routing table by inserting the local node into its own routing table and monotonically incrementing the corresponding sequence number (called *seqno*) by one on each update. Furthermore, on each update a packed representation of the global routing table is computed containing the id, last-seen-sequence-number, magic-number, hops and estimated-connections for each host. The estimated-connections are computed from the *connection manager* structure for this host if available. These packed representations are sent along each connection (split into pages that fit within each package size requirement) to the direct *linklayer* neighbours without any further propagation.

An update to the global routing table is computed on each node whenever they receive routing information from the direct neighbours. The received routing information is decoded and compared

to the current routing hash table for that connection and missing hosts are marked as outdated. Hosts that are not outdated are updated in the connections routing table and a check is made to see if the global routing table is updated. This is done by comparing the values of *seqno – hops* for the routing table, trying to maximize this value in the global table. This can lead to the three cases: (1) no-change required, (2) use this connection instead of the current connection or (3) select another connection.

The magic numbers for hosts are used for detecting the situation when a given *peis-id* is used simultaneously by two different nodes, or to detect when a node have stopped and restarted. It is also used to prevent propagation of incorrect routing information to hosts that have been deleted and restarted.

3.1.6.3 Known hosts

This is a module that uses routed packages to query for and respond to information requests of the *hostinformation* structure which stored in the PEIS-kernel. Whenever a *peis-id* for an unknown host is detected by the routing algorithms, a query is send to retrieve this information. Additionally, when inconsistencies with multiple components using the same id, or a component is suspected as having restarted (detected through the magic numbers) this is verified through the known hosts query mechanism and any old routing information is invalidated.

3.1.6.4 Connection Management

A connection management module is responsible for establishing connections to hosts that have been detected by the periodic announcements in the *linklayer*s or through propagated routing information.

This module attempts to keep the diameter of the P2P network small in order to decrease amount of wasted global bandwidth due to routing and to increases chance of packages being delivered with fewer retries (reducing latencies) – while at the same time keeping a trade off between the number of connections that exist in the network and the possibility of the software to request communications with any host at any time (e.g. without introducing the latency of first establishing a connection).

Secondary goals of it is to the keep number of new connection establishments as small as possible (avoid routing table changes, saves OS time) and to have direct connections to any hosts with which we exchange much data (overall efficiency of network) as well as to attempt to keep the number of connections in within a suitable range.

To minimise used bandwidth this module establish direct connections to hosts to whom we (or hosts routing through us) transmit much information. This module keeps track of how much traffic (bytes/second) are routed to different hosts. This includes all directed messages (not link level packages) and routed packages. For hosts that we route more than X bytes per second over a sliding window (for a constant X), force a connection if possible.

To minimise girth this module finds the value V(c) of each connection c. The value of a connection is the maximum increase in metric distance to any host, which are currently routed through that connection if that connection would be closed. This can be computed from the connections routing tables. The value of connections to whom we have a forced link (data per second > X) have an infinite value.

Next, for each known host which it hasn't connected to recently it finds the value V(h) of establishing a connection to this host h. This value is defined as a random number (0-99) plus a constant (100) times the current metric to this host minus the heuristic metric cost along a direct connection to that host (this metric is dependent on the *linklayer* and is computed for each such low-level address by low-level address pair).

Hosts that have recently been connected to have a value of zero. This allows for a certain randomness in choosing which hosts to connect to, aliasing interference patterns from multiple components attempting to connect to the same hosts.

If we have fewer than the minimum wanted connections (eg. 3) then establish a new connection to the host h with the highest value V(h).

If we have a suitable number of connections (eg. < 7) then connect to the host h with the highest value V(h) if this value is higher than the lowest V(c) for any connection c. If we have more than a suitable number connections (eg. 7) then close the connection with the lowest value (if it not infinite).

3.1.6.5 Clusters

The module for cluster management is a sub-module of the general connection management and tries to ensure that the global ecology will always have at least one path of communication between any two hosts – ie. to avoid the ecology from splitting into separate P2P networks.

To do this the service computes the cluster number, defined as the lowest id of the components to which it has a route. This number is included in the host information for this host (and thus propagated through any link-layer broadcasts). For each host detected by the *linklayer*s it checks to see if the detected host has a lower cluster number. If so, it attempts to force a connection to it if any of its listed lowlevel addresses are connectible.

3.1.6.6 Auto-connect hosts

In conjunction with the *connection manager* there is also the notion of auto-connect hosts. These are hosts to whom we attempt to always maintain a low-level connection. A built in service will monitor this connection and re-establish it whenever it is closed.

The main purpose of establishing auto-connect hosts are to either (a) connect networks which do not talk with multicasts between each other or goes through firewalls or (b) connect to hard-coded leaf nodes. By using one manually established such connection between two PEIS separated by firewalls and/or different netmasks these networks become routable and may even establish direct connections for the case of having any low-level routable interfaces between the devices on these networks.

Currently there is no UDP firewall piercing service in the PEIS-kernel, but this may be implemented later to allow for better establishment of connections through firewalls.

3.1.6.7 Leaf nodes

To minimise the amount of overhead associated with the large number of components it is possible to configure the recology as a network-of-stars, where only one component per host talks to the P2P network and the remainder of the components only communicate through this component. This is done by running these secondary components as leaf nodes and establishing auto-connections to the

master node – which usually is the special *peisinit* component residing on each host and managing which other components can run on it.

3.1.6.8 Acknowledgement handling

Packages sent through the P2P layer of the PEIS-kernel can be, and usually are, sent in "reliable mode". Meaning that they after sending get put on a stack of packages to be acknowledged within a given timeframe or they are retransmitted. In order to not flood this stack, packages will only be retransmitted a maximum of PEISK_PENDING_MAX_RETRIES times. The retransmission are with an increasing period of PEISK_PENDING_RETRY_TIME seconds for the first retry, twice that for the second retry, three times that for the third etc.

The PEIS-kernel saves a copy of the message in a separate queue and will retransmit the message with incrementally increasing period times until either a timeout has passed or until an acknowledgement receipt for this package arrives.

When a package is removed from the acknowledgement queue, a function hook associated to this package is triggered along with a flag signalling if the package succeeded or failed. When messages are sent along a point-to-point connection (never for broadcasts) the user can specify such function.

When the PEIS-kernel receives messages targeted to it, it checks for the presence of the acknowledgement flag. If present, it saves the unique package id and sender of the package on a list of packages to be acknowledged. When this list of packages to be acknowledged for a specific user is large enough or when the oldest message in the list exceeds a threshold a package is send back to the sender with all the acknowledged packages. (This allows for minimising the overhead of acknowledgements while avoiding triggering unnecessary retransmission of packages).

Acknowledgement/failure of large messages (requiring multiple packages) are treated listing all the sub-packages that was send for the large message and invoking the hook (if any) when all of the parts have succeeded or when any of them have failed for the last time.

In the implementation of the island-to-island communication in WP1 we will rely on the acknowledgments mechanisms only for some type of messages and will send messages for which retransmission is undesirable using the non-reliable transmission mechanism.

3.1.7 The Tuple Layer

The tuplespace layer is the third layer of the PEIS-kernel, implemented on top of the mechanisms provided by the P2P layer. It is responsible for all storage, publishing and retrieval of tuples in the distributed tuplespace – effectively creating a distributed database as a blackboard communication/collaboration model. By performing associative wildcard searches, we allow efficient collaboration between any pair of components in the ecology. By introducing concepts of meta-tuples allowing for indirect accesses to data it enables simple and efficient dynamic reconfiguration of inputs/outputs.

Since the tuplelayer is used by the control layer as the basic mechanism for collaboration it will also act as the main interface point between MaD-WiSe Stream System devices and PEIS-network devices. Although not a direct part of the communication layer developed in WP1 we will here describe this tuplelayer in order to explain how the implementation of the gateway devices that link the MaD-WiSe Stream System devices and PEIS-network devices will be developed.

3.1.7.1 Overview of the tuplespace

From the perspective of the tuplespace keys consists of three parts: (name, owner, data) where *name* is a string key for the tuple, *owner* is the address of a PEIS responsible for this tuple (see below) and data is the value of the tuple. The tuples are indexed by *name* and *owner* meaning that tuples with the same name but different owners are allowed to coexist while there can (ideally) only be one instance at a time of tuples with the same *name* and *owner* but different data.

There are (currently) two basic methods by which tuple values are propagated.

- The *owner* PEIS writes to the tuple and the updated value is propagated to all *subscribed* PEIS.
- A PEIS writes to the tuple by sending a message to the *owner* PEIS which stores the last written value and propagates it to all *subscribed* PEIS. The message passing is handled transparently by the PEIS kernel and is not visible to the application other than by a delay before the value is updated in the local tupelspace.

The *owner* of a tuple can always access the latest value of that tuple by a direct memory access. In order for other PEIS to have access the latest value of a tuple they need to be subscribed to the tuple. These subscriptions, which will make sure that subscription messages, are regularly sent to only the relevant parts of the network. Subscriptions come in two varieties:

- Qualified subscription to (*name*, *owner*) tuples. These subscriptions are in essence a direct one way communication from the owner of the tuple to the receiving PEIS which will get the latest value of all written tuple values.
- Wildcard subscriptions (*name*, *) which registers50m a subscription with all PEIS on the network. All matching produced tuples will be propagated to this PEIS.

Corresponding to these two subscription mechanism tuples can also be accessed by a call to *getTuple*. We have three cases:

- A PEIS accessing a tuple with itself as *owner*. This returns the last value written to the tuple by itself or by some other PEIS writing to the tuple (and propagating the message to this owner).
- A PEIS accessing a tuple with another PEIS as *owner*. Returns the last value of the tuple that has been propagated from the owner to this PEIS.
- Accessing a tuple with a wildcard (-1) as *owner*. Return a value for each owner that has propagated a value for this *keyname* to this PEIS.

Note that the last two accesses require active subscriptions in order to receive the *correct* value.

Everyone interested in data should subscribe to key(s) periodically. Everyone producing data should send a copy of it to everyone currently subscribed to it.

3.1.7.2 Storage of and accessing tuples

Each PEIS component has a simple data store where tuples belonging to the tuplespace of the component is stored. Additionally they have a local cached version of the tuples that have been sent to it from remote components. All tuples, both in the local storage and in the cache are subject to the expiry date in the tuples and are deleted when the given timepoint (if specified) have passed.

For the actual storage, a simple hashtable based mechanism is used with primary indices on the tuple key, *with no secondary indices currently implemented.* The indices and the data of the tuples are all stored in RAM memory.

When a user attempts to access a tuple with a fully qualified key it is checked against both the local storage as well as the cached storage. If a fully qualified owner id of the tuple was given this gives up to one unique answer which is returned to the user. If a wildcard was specified for the owner id was given this instead gives a number of answers that are returned in a *result-set* to the user.

If on the other hand access is made with wildcards inside the tuple key the local storage and cache is iterated over to enumerate all matching tuples. These are again returned in a result-set.

> The *getTuple(s)* operation will only consume computational power in the device that executes the instruction.

In order to populate the local cache of tuples, the user must create subscriptions that are propagated to all relevant components in the ecology. These components will then publish all their matching tuples into the local cache of the subscriber and will keep publishing all newly generated tuples into the subscriber until the subscription is cancelled.

> Due to the time delay inherent in the network subscriptions must be made in advance before being used.

Due to the simple implementation of the databases and search operations, unnecessary use wildcard searches are discouraged.

> A *subscription* operation with wildcards may consume significant computational power due to the wildcard search and size of tuple database of each other component when executed or when the component joins the ecology. However, it will only consume a moderate additional computational power when new tuples are inserted in the other components.

3.1.7.3 Callback functions

Input streams are often accessed by explicitly querying and reading the latest value of a tuple in the tuple space within the main loop of a robotic program. Although this method is convenient and easy for beginning programmers a more efficient method with fewer drawbacks such as a risk of missing some values or reading the same value twice is to use callback functions.

By registering a callback function with an *abstract tuple* as a prototype for the kind of tuples of interest a given function will be guaranteed to be invoked with the matching tuples as they arrive into the local cache of the PEIS-kernel. In addition to avoid the risks of missing tuples this also avoid the problems with mutability of the tuples since the callback function will be executed from within the main thread of the kernel. However, tuples should still be cloned and a return from the callback function must be made if the processing time exceeds 50ms to avoid blocking the execution of the PEIS-kernel stepping.

3.1.7.4 Meta Tuples

A meta tuple is a tuple which gives the owner and name of other tuples. Thus meta tuples provide a mechanism for *indirect reference* which allows components to be dynamically reconfigured by rewriting the references during execution. For details of the use and implementation of meta tuples we refer the reader to Deliverable D3.1.

In the development of the gateway devices that interface between MaD-WiSe Stream System to PEIS-ecology devices we will use the meta-tuple *concept on the gateway devices and/or the proxy devices to allow the services provided by the WSN to be dynamically configured from the perspective of the control layer.*

3.1.7.5 The tuplespace as communication channel

The tuple space has been and is still used actively as the main communication channel and collaboration mechanism for the PEIS Ecology project and the robotic devices developed at AASS. As a rough estimate of the uses of it, the shared data range from single reads of small static configurations, to repeated ascii or binary data sensor data and even sharing video frames of image data at high frame rates. The later typically uses frames of 100KByte to 10MByte at frame rates ranging from 5-25Hz from one or more cameras.

For the communication between MaD-WiSe Stream System to PEIS-network devices the bottleneck on the bandwidth used will be on the IEEE 802.15.4 side. As such no special considerations need to be made to accommodate for the size of these messages on the PEIS side.

3.2 Proxied objects

Although developed as part of the previously mentioned PEIS ecology initiative, the notion of *proxied objects* is a general concept that can be used to empower simple devices to operate in conjunction with most forms of advanced robotics middlewares [1, 9].

3.2.1 Introduction to proxied objects

The basic idea of proxied objects is to give a representation of everyday objects which is consistent with how other objects such as other robots are represented. By giving a uniform mechanism for interfacing to any kind of object, we can easily query it for its capabilities and properties or ask it to perform tasks regardless if it is a fancy robot or a simple coffee cup. As lofty as this goal may seem, and in spite of the limited capabilities of simple devices (regardless of the representation, a coffee cup just cannot brew itself), we will see that this slight shift in viewpoint gives a number of advantages.

In this section we give a brief reminder to the general design pattern for implementing this viewpoint and outline the various components and the needed information flow for this to happen.

The notion of proxied objects is based around the use of a *proxy*: a process hosted by a component of the distributed robot system, which acts as a representative of the simple object inside the middleware. The role of these proxies is to integrate everyday objects, such as RFID tagged objects or very simple wireless sensors and actuators lacking the capability to run any middleware on board, into a larger networked robot system. In these cases, the proxy is used to create an image of the external object, which is made accessible to the middleware. This image is maintained using a dedicated communication channel (e.g., an rfid-reader or a ZigBee radio module) to synchronize this information with the actual object.

From a hardware point of view we require, obviously, the objects to be proxied. These objects can be any kind of object to which we have any form of communication channel. Examples include, for instance, everyday objects equipped with RFID tags or simple sensors and actuators connected with ZigBee modules or Radio modems to transmit/receive analog and digital signals.

In the RUBICON project these devices correspond to the WSN motes for purely sensor based objects, WSAN motes for objects with limited actuation capability (such as triggering a light switch) as well as the classical RFID, and ZigBee type of proxied robotic objects as presented in the literature [9].

In addition to these objects, we also require components that can perform the communication with the proxied objects, eg. an RFID reader or a ZigBee radio link, and that it can relay these communications to the general middleware and connected robotic devices. We call these devices *interface* components. These devices need not perform any major operations or interpretations on the communications, but rather only transform in into a form usable by other components. This is necessary since the same proxied object may be communicating with different interface components at different timepoints or even simultaneously.

In the RUBICON project these interface points *correspond to the gateway devices that are developed for passing messages between PEIS components and WSN devices.*

3.2.2 Design pattern

The design pattern consists of the following ingredients (we refer the readers to Rashid et. al [9]) for the detailed explanation and experimental implementation):

Proxied Objects: These are objects with very limited computational capabilities but augmented with a communication channel, eg. RFID tagged household items or simple ZigBee sensors, that we want to include as peers in the network of devices. *In the case of RUBICON we will consider here not only computationally passive devices but also the WSN devices required for higher level control for this purpose.*

Interface: We refer to any component that provides an end-point for communication with the the proxied object as an interface. These interfaces are used to access the information about the object and include e.g. rfid readers for the case of rfid tagged objects or ZigBee radio base stations for the case of ZigBee equipped sensors. These components need not perform any major operations or interpretations on the communicated data, but only to relay the raw data to make them accessible to devices in the networked robot system. We do not require the same interface component to communicate with a proxied object at all time, but only that some interfaces can communicate with the object some of the time. *For the case of RUBICON the interface to the proxied WSN motes are the gateway. Each gateway provides a list of the WSN motes with which it can communicate and facilities the exchange of messages with the WSN.*

Proxy: This is the actual software component that represents the proxied object and interfaces with the middleware.

The proxy receives any data flow from the proxied hardware via the interface, parses them and publishes to the middleware with proper semantics. This information is called *direct information*.

The proxy also provides *prior information*, given implicit in the nature of the proxy or being given as argument to the proxy when instantiated. Typical prior information includes the capacities and physical properties of the proxied objects. For instance, a proxy for RFID tagged objects can make a static database lookup to give these properties when instantiated for an RFID tag with a specific ID.

Properties from interfaces which indirectly allow us to deduce properties of proxied object based on the context or the nature of the interfaces is indirect information. For example, deducing a proxied object's coarse location based on the available interfaces signal strength and the location of the interface device itself is a form of indirect information.

Proxy manager: Coping in a dynamic environment requires dynamic creation or starting of proxies when proxied objects appear in the environment.To deal with this dynamic creation of proxies, we are using a *proxy manager*, which is responsible for the instantiation of proxy components as soon as interfaces detect new objects. In order to decide if the corresponding object, or percept, corresponds to an already existing proxy or constitutes a new object the proxy manager relies on any already running proxies to consume the percept. Each proxy is assumed to contain mechanisms to assess whether the data from the interface (percept) pertains to the object beeing proxied or not.

If the percept is not consumed by an existing proxy the proxy manager must instantiate a new proxy that corresponds to this object. To do this, the proxy manager relies on the *signature* of the percept, which must be provided by all interface components for each object with which it communicates. This signature is searched in a database of available latent proxy components. If a match is found, proxy manager starts the corresponding proxy program.

Signature: A signature is a unique, semi-structured identifier used for identifying the class of an object. It is composed of the used interface channel's type and a hardware unique identification number for that channel, eg. MAC addresses or RFID id's and performs a partial matching on these to determine a suitable proxy class to be used for the corresponding object. This proxy class is used to instantiate the right proxy for the object to be used. Note that class here denotes only a category of object and does not necessarily imply an object oriented programming paradigm. In fact, for the PEIS Ecology reference implementation the proxy exists as separate processes (implemented in any language) where the signature is used to determine which program to launch for the corresponding object.

Figure 4 shows the anatomy of a proxy for simpler everyday objects such as a milkbox with a built-in RFID tag. Here objects are proxied using only one communication channel at a time, eg. through one of the RFID readers, even though multiple interfaces exists in the environment. The proxy component is responsible for selecting the most suitable interface, when multiple interfaces simultaneously perceive the proxied object.

The proxy communicates to the actual hardware via this selected interface. The hardware data, which are received from and sent to the proxied object, are translated in the proxy with proper semantics and are forwarded using this selected interface to the proxied object.

These communication data are direct information. On the other hand, position information, which comes from the location of the most suitable interface is an example of indirect information.

The prior information is determined by a table lookup when the item is first sensed and contains the proxied object's properties and capabilities, such as colours, shapes, grasping points, abilities to perform actions, which are used by the deliberative components.

An example of a traditional proxied object that can perform actuated actions would be a simple ZigBee mote with a PWM output connected to servos in the environment (for eg. actuating window blinds or lamps).

Another example of a proxy would be a simple RFID tagged piece of grocery where the proxy process would receive a serial number when instantiated. During run time, it would look for information from all RFID readers in the environment and when it finds a reader which can perceive the tag with that serial number, it uses the reader to read further tag data to compute properties of this specific item, eg. translating the hexcode of the first few bytes into a representation of what type of food, expiry

date, shape of packaging etc. that it has. Additionally, it also computes its own position from the position of the RFID reader.

When the interface components communicating with this proxied object is changed, for instance by moving the object, the proxy should automatically be configured to use any new interface(s).

During task T1.4 we will use the above design pattern and augment it with the new requirements as posed within RUBICON. See Section 4.2.5 for further details on these requirements. The outcome of this task will be described in deliverable D1.4.

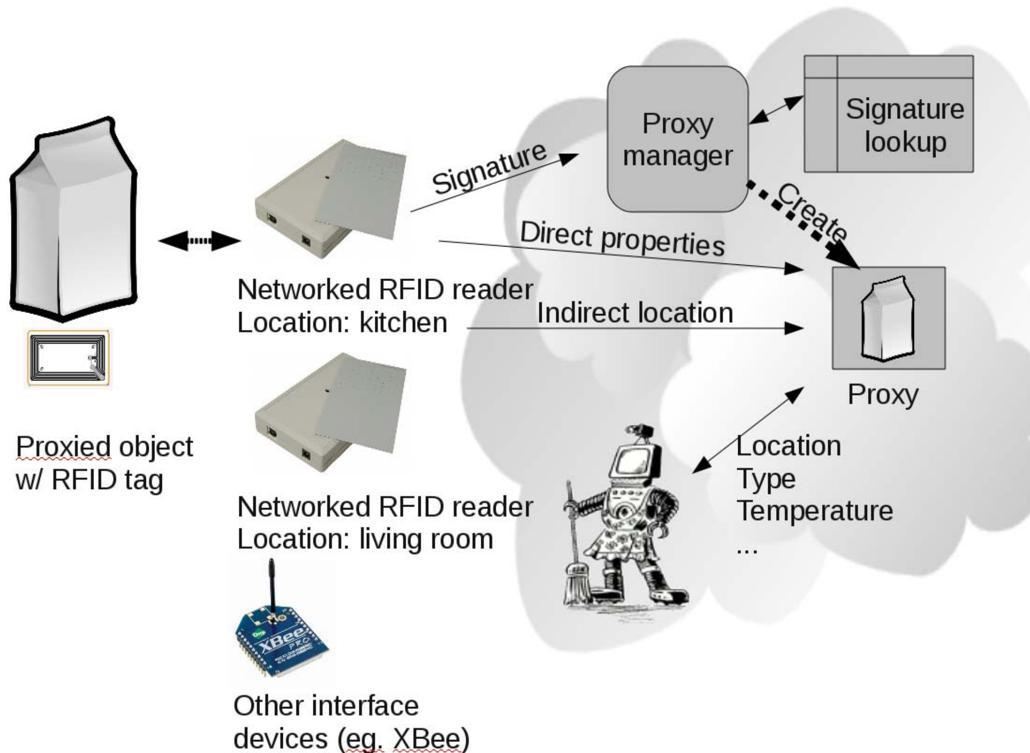

Figure 4: Example of a proxy using RFID as interface channel. Proxy manager receives signatures from all interfaces, looks up and creates the proxy for the detected object. The robot communicates with the proxy like for any other device in the ecology.

3.3 MaD-WiSe Stream System layer

3.3.1 Introduction

The Stream System is the transport layer of the MaD-WiSe framework [1]. It is a software module that abstracts the intra-sensor and inter-sensor communication mechanisms. Applications running on the sensors rely on the Stream System to disregard the actual implementation details of collecting transducer readings and passing such data to other local or remote computational entities.

The Stream System functionalities reflect the high dynamic variability of data and distributed access/processing observed in sensor networks, and can be used as a building block for higher level applications, such as sensed data management or database software. Applications can also address

local as well as remote transducers through a uniform interface that hides the details related to communication and buffering.

The Stream System provides a uniform paradigm for the execution of data acquisition, communication and local processing activities. Having these three activities defined under a uniform framework allows the applications to concentrate on its logic, to ignore the specific data sources used (transducers, radio, local queues) and to easily change data management strategies without affecting the entire application.

The Stream System offers a unidirectional data collection and data communication abstraction to higher layers. The basic concept of stream represents a generic unidirectional data channel that is able to carry data records. The Stream System provides functionalities for creating, destroying, writing and reading records to/from streams.

The context where we envision the use of streams is that of operator-driven computations. Operators are seen as independent agents that have inputs, perform some operations on those inputs and possibly produce an output. In our model, operator inputs and outputs are records read from/written to streams.

3.3.2 The Stream System

The Stream System offers three types of streams: sensor streams, local streams and remote streams. Sensor streams are the basic abstraction for collecting readings from transducers. They can only be read by operators since the writing is carried out by the associated transducers (these can be thought of as virtual operators writing to sensor streams). Local streams represent a local data channel where read and write operations must occur on the hosting sensor. Remote streams require cooperation between two nodes since they intend to provide a data channel between two different nodes. Write operations can be carried out on one of them (the stream write-end) and read operations can take place on the other (the read-end). Figure 5 illustrates these concepts.

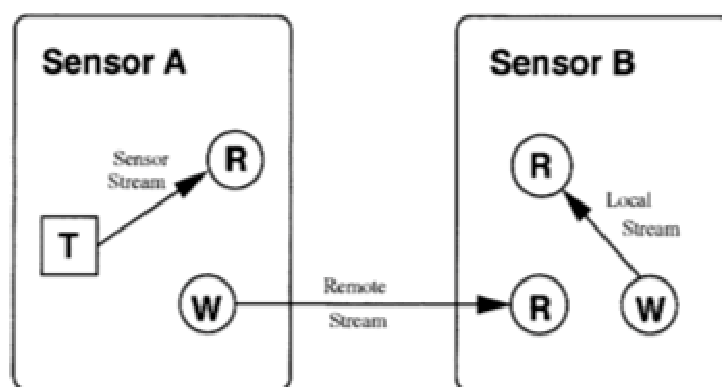

Figure 5: Stream types. T = transducer, W = writing operator, R = reading operator.

In the following we use a command/event-based notation to define streams and their operation. A key concept in command/event-based systems is that of split-phase operations. The call requesting

to start the operation returns immediately, without waiting for the operation to complete. When the operation completes an event is fired to notify it. Split-phase operations are typically involved when interacting with hardware devices where blocking is not allowed. Operations involving simple data structure manipulation can immediately complete without requiring a later event notification (i.e., they are not split-phase).

3.3.2.1 Basic Stream Operations

An application identifies a stream through its stream descriptor, which is returned at stream construction. Among the operations that can be applied to a stream, command

```
write(desc,buffer,length);
```

allows the user to write a record to a stream. A stream maintains data in a finite size queue and `write()` appends a value to the end of the queue. If it finds the queue full, it simply discards the first (oldest) queue element before appending its own. Argument `desc` supplies the stream descriptor while arguments `buffer` and `length` give the starting address of the buffer containing the data and its length in bytes, respectively.

Data stored in a stream can be accessed with command

```
read(desc);
```

where argument `desc` identifies the stream, as usual. `read()` is split-phase: the call immediately returns. As soon as a record is available in the stream queue (possibly immediately), the Stream System removes and returns the oldest one to the user by signalling event

```
readDone(desc,buffer,length);
```

The arguments have the same meaning as for `write()`.

Finally, when a stream is no longer needed, it must be destroyed with the command

```
close (desc);
```

3.3.2.2 Sensor Streams

Sensor streams provide an abstraction for collecting readings from local transducers. It is not possible for an application to write to a sensor stream since its write-end is automatically associated with a transducer. The Stream System writes data to the stream on the basis of readings collected from the transducer. Sensor streams come in two flavors: periodic and on-demand.

With periodic sensor streams the Stream System periodically commands a reading from the associated transducer with a fixed, timer-driven rate and writes a record of 3 elements to the stream: (nid, ts, val). nid is the (network unique) node identifier, ts is a timestamp of the reading and val is the actual reading. The user can later retrieve these records through the command/event sequence `read()/readDone()`.

With on-demand sensor streams the Stream System performs a transducer sampling only when explicitly requested to do so with a `read()` command. When the transducer supplies its reading, the Stream System writes a record (with the same content as for periodic sensor streams) to the stream and immediately signals the event `readDone()`.

The sensor stream constructor is command

```
openS(dev,qsize,type,rate);
```

Argument `dev` encodes one of the transducers available on the node (e.g., light, temperature, humidity, acceleration, magnetism, etc.). `qsize` gives the size of the queue data structure used to store data records. The stream type (periodic or on-demand) is specified through argument `type`. For periodic sensor streams the user must also supply the sampling rate in `rate`. The call immediately returns the assigned stream descriptor.

3.3.2.3 Local Stream

The read-end and write-end of a local stream are on the same node. An application can both write to and read from a local stream. Of course, different tasks of the applications are expected to read and write on such streams. Creation of a local stream is achieved through command

```
openL(qsize);
```

that immediately returns the assigned stream descriptor. As for sensor streams, argument `qsize` gives the queue size.

3.3.2.4 Remote Stream

Remote streams interconnect two distinct sensors in a unidirectional way: writing can only happen on one of them (the write-end) and reading can only happen on the other one (the read-end). To request the opening of a remote stream from sensor A to sensor B, the application modules on both sensors must create their own stream end with command

```
openR(dest,qsize,s_id);
```

The Stream System takes care of interacting with a lower level Network module and setting up the remote stream.

On sensor A, argument `dest` gives the identity of remote sensor B. Various approaches could be used to identify nodes, including node ids, coordinates, roles. On sensor B, `dest` simply identifies the local node. From an implementation point of view, the Stream System module on sensor A (the stream write-end) turns to the Network module asking it to setup a communication channel to sensor B. On the other hand the Stream System module on sensor B (the read-end) plays a passive role and just keeps track of the open request received. Argument `dest` allows the Stream System module to decide whether it is the write-end or the read-end. Arguments `s_id` is a symbolic id, it is

the same on both stream endpoints and it is used to identify the same stream in the two nodes. Finally, argument `qsize` gives the queue size.

The opening of a remote stream is split-phase to cope with possible network latencies. The command immediately returns but when the stream has been set up the Stream System modules on the two ends independently notify this by signalling the event

```
openRDone(desc,s id);
```

Argument `desc` contains the assigned stream descriptor and `s_id` is needed by the application module to associate the event with the previous `openR()` command.

3.3.3 Implementation Details

3.3.3.1 The Network Module

In order to implement remote streams, the Stream System module needs assistance from a Network module offering a connection oriented service. When opening a remote stream, the Stream System asks the Network module to establish a unidirectional communication channel (a connection) to a remote sensor with the following command

```
connect(dest);
```

Argument `dest` serves to identify the remote end. The command returns a locally allocated, network unique, channel id.

Since operation outcome depends on the availability of resources on each node on the path to the destination, it cannot be determined at the time the call returns. The connection establishment procedure is implemented with a connect message going through the network towards the destination and reserving the necessary resources along the way. A connect-ack message is then sent back to the originating sensor confirming that the connection has been accepted by the destination Network module. Arrival of the latter message at the source Network module, causes the signalling of event

```
connectDone(c_id);
```

to inform the user. Argument `c_id` is the same returned by the previous `connect()` and is used by the Stream System to identify the pending connection request.

The user can now send packets on the established connection invoking the command

```
send(clid,buffer,length);
```

where the arguments serve to identify the connection and the data to send. The service offered by the Network module is not necessarily reliable: reliability could be provided by other levels or be completely missing. For this reason sending a packet is not contemplated as a split-phase operation in the network interface and return of the command call does not mean that the message has been received by the destination.

Upon receiving a message from a remote sensor over an established connection, the Network module must pass the payload to the upper level. It achieves this by signalling the event

```
receive(c_id,buffer,length);
```

where the parameters identify the channel as well as the received data and its size. Finally the command

```
disconnect(cid);
```

is used on the source sensor (write-end) to ask the Network module to shut down (i.e., terminate) an existing connection. Servicing this request implies deallocating channel resources on all nodes of the data path and it is performed by sending a special disconnect message along the path. Optionally, also an expiration timer for each connection can be set on all the sensors of the path to free resources of not closed channels. There is no event signalling upon operation completion.

The Network module also uses a novel algorithm [3] that attempts to turn off the radios on the basis of application-provided information. When an application needs to periodically send fixed size/rate packets to another node trough a given path (multi-hop communication), the algorithm exploits communication timing information, transmission times, and average medium access delays, to optimally schedule radio activities in the path.

To achieve energy efficient communication the application module provides additional arguments to command `open()`, indicating when it will start sending packets on the remote stream, the interval between consecutive packets and the size of each packet. The Stream System simply passes these values to the Network module as additional arguments to `connect()`.

The implementation of our Network module relies on greedy routing [4] for the connection establishment procedure and uses the algorithm described in [5] to assign three-dimensional virtual coordinates to the sensors.

The Network module uses greedy routing to send the connect message when it needs to establish a new connection. All sensors along the path allocate an entry in their connection forwarding table associating the channel id with a neighbour (the next hop). The sensors also configure their radio activity intervals as required by the energy efficiency algorithm. The destination replies to the connect message with a connect-ack message that returns (along the reverse path) to the connect message source and confirms connection establishment.

After connection setup each sensor on the path only turns on its radio when the next message is expected and for up to some maximum amount of time. It forwards the message to the next hop on the basis of the contained channel id and its connection forwarding table.

3.3.3.2 Streams

Internally, a stream is implemented as a finite size queue. The number of elements in the queue is specified when the stream is opened, by means of argument `qsize` to commands `openS()`, `openL()` and `openR()`. In case of local and sensor streams the queue data structure is allocated on the node where the stream resides. For remote streams the queue data structure resides on the read-end node. Thus, writing a record to a remote stream means passing it to the Network module that will transport it to the destination node, which will finally store it in the queue.

The procedure for setting up a remote stream is rather complex since it involves several interactions between the Stream System module and the underlying Network module. Figure 6 depicts the temporal sequence of commands (full arrows) and events (dashed arrows) that occur when a remote stream is opened from sensor A to sensor B. To avoid excessive cluttering only significant arguments to command/event calls are reported. Dotted lines indicate frame generation in the MAC modules, the actual sending, network traversal, reception in the destination MAC module and passing up to the Network module.

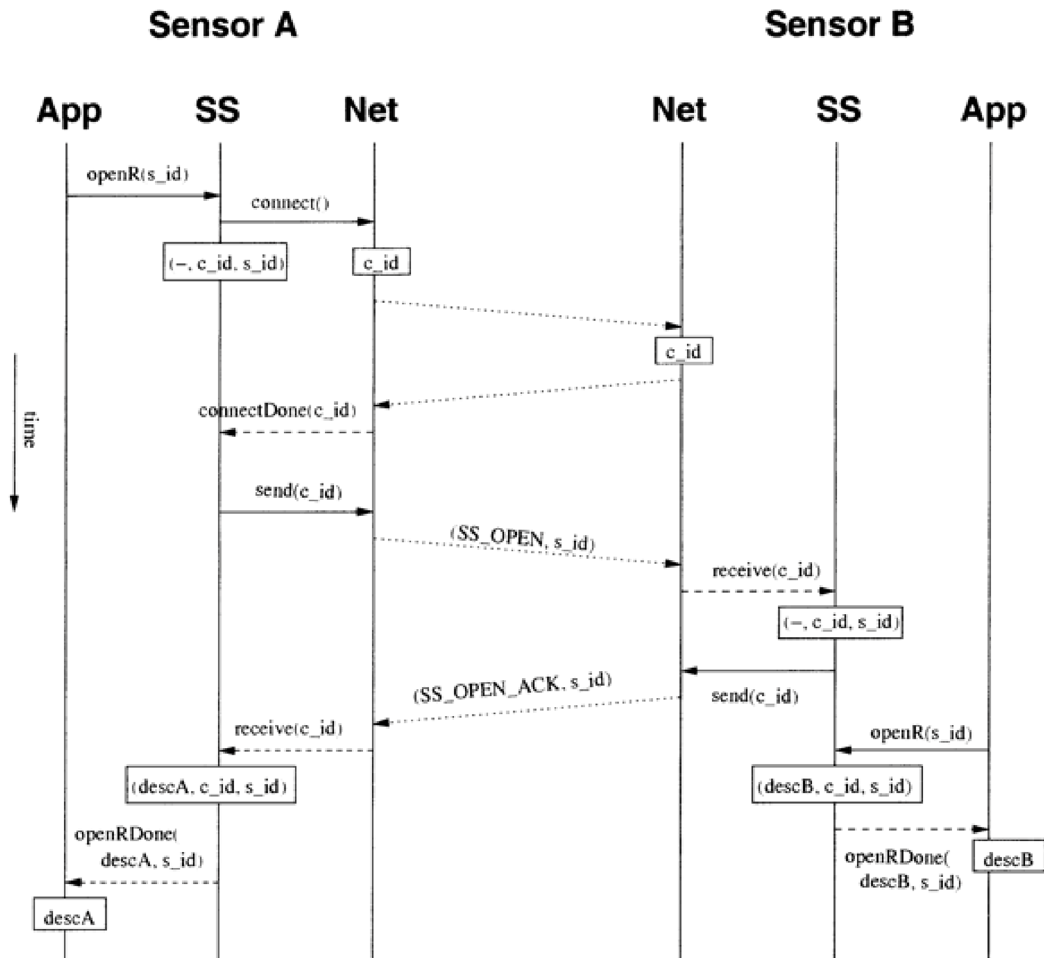

Figure 6: Command/event sequence when opening a remote stream from sensor A to sensor B (App = Application, SS = Stream System, Net = Network).

Rectangles report on local data structures as they appear after the preceding command or event completes.

In response to an `openR()` command, A’s Stream System module invokes command `connect()`, asking the Network module to establish a data channel to B's Network module. `connect()` returns a channel id (`c_id`) and instructs the Network module to send a connect message. At this point

`openR()` stores the channel id together with the stream symbolic id (`s_id`) and returns control to the application module.

When A's Network module receives the connect-ack message from B, it fires the `connectDone()` event. `connectDone()` retrieves the data structures for the remote stream by means of the channel id, prepares an `SS_OPEN` message that includes the stream symbolic id and hands it to the Network module for sending on the established channel. Upon receiving the message, B's Network module signals the event `receive()` to B's Stream System module. Assuming B's application module did not request to open the stream yet, the Stream System module associates the stream symbolic id with the channel id and replies with a `SS_OPEN_ACK` message. A's Network module passes up such message to the Stream System module which retrieves the stream data structures and signals event `openRDone()` to the application, passing the stream symbolic id and a newly allocated stream descriptor.

When B's application module invokes command `openR()`, the Stream System module discovers that it already received the `SS_OPEN` message from the other side and signals event `openRDone()` with the stream symbolic id and a locally allocated stream descriptor as arguments.

Note that if B's application module invokes `openR()` before the Stream System module has received message `SS_OPEN` from its peer in A, it cannot associate any channel id with it, yet. It only keeps track of the symbolic id specified in the `openR()` call. Upon arrival of the `SS_OPEN` message from A it will signal event `openRDone()` after filling its data structures with the channel id.

Writing a record to a remote stream means sending the record over the network to a remote sensor. The Stream System modules on both stream endpoints interact with the local Network modules in order to implement this operation. Figure 7 illustrates the sequence of calls and events that take place. A's application module writes a record to the remote stream invoking the command `write()` and passing the stream descriptor as well as the actual data record. The Stream System looks up its data structures and retrieves the channel id associated with the user supplied stream descriptor. It prepends a header (`SS_WRITE`) and asks the Network module to send the message over the channel (`send()`).

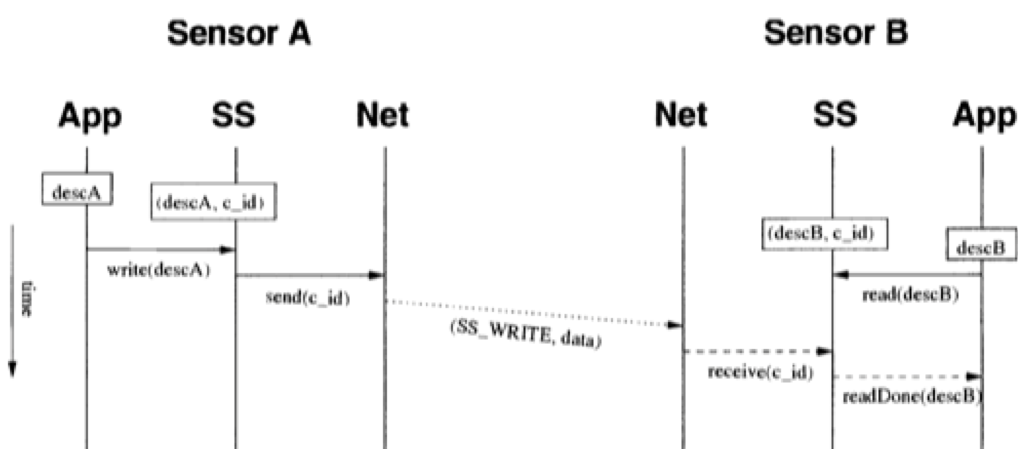

Figure 7: Command/event sequence when data is written to a remote stream (App = Application, SS = Stream System, Net = Network)

Upon receiving this message, B's Network module signals event `receive()` passing the channel id as an argument. B's Stream System module retrieves the stream descriptor that is associated with the channel id and writes the record into the stream data structure. If B's application module previously commanded a read from the stream, the Stream System module can now signal the event `readDone()`, passing the stream descriptor and the data record as arguments (Figure 7 shows this case). Otherwise, the Stream System module simply stores the record in the stream data structure and signals `readDone()` when a record is requested with a later `read()` command.

3.3.4 Conclusions

The Stream System is a nesC module that offers (implements) some functionalities according to a well defined interface and signals events to notify of conditions. It relies on the existence of a Network module implementing a network (lower level) interface and providing interconnections between any two sensors in a multi-hop network. It also interacts with a Transducer Abstraction Module that abstracts the transducer devices on the sensor, providing commands to read any specific transducer and signalling events when readings are available.

Figure 8 illustrates how the Stream System module fits into the system and how it interacts with the other modules.

The Stream System is a data collection and data communication abstraction model suitable for sensor networks. An application built on top of this system can be structured as a set of operators distributed among the nodes of the sensor network. Operators read inputs, do some processing and produce some output. The output of an operator becomes input of another by means of a stream interconnection. A special type of stream serves as a transducer data source. The application totally disregards issues concerning transducer operation as well as moving data from one node to another.

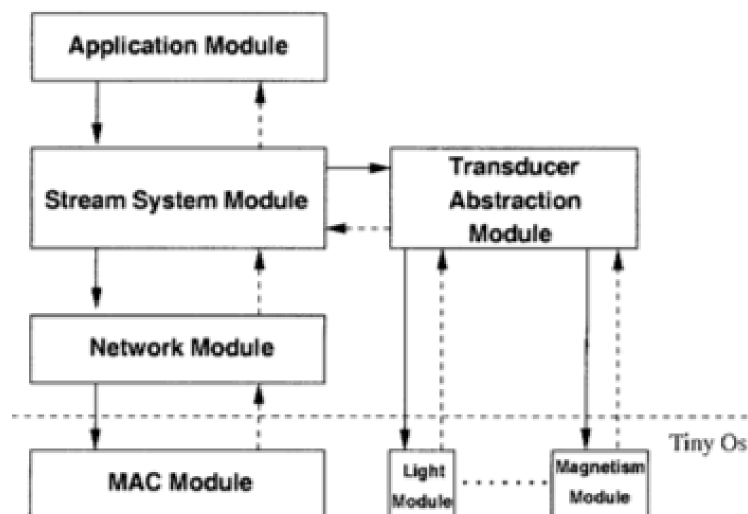

Figure 8: Stream System Module interaction diagram. Full arrow lines indicate commands while dashed arrow lines indicate events.

3.4 SMEPP Light

3.4.1 Introduction

SMEPP Light is a middleware for Wireless Sensor Networks (WSNs) based on mote-class sensors. It is derived from the specification developed under the framework of the Secure Middleware for Embedded Peer-To-Peer (SMEPP) project [6], to deal with the hardware and software constraints of WSNs.

SMEPP’s main objective is to hide the complexity of the underlying infrastructure while providing open interfaces to third parties for secure application development. SMEPP middleware is secure, generic and highly customizable, and it is adaptable to heterogeneous devices (from PDAs to embedded sensor/actuator systems) and domains (from critical systems to consumer entertainment).

SMEPP Light is the version of SMEPP tailored for WSNs. This because a sensor can hardly face the technical problems arising in the implementation of the whole SMEPP specification. It addresses a limited but yet significant and coherent subset of SMEPP primitives. In particular, differently from the full SMEPP specification, SMEPP Light does not support services. On the other hand it features group management, group-level security policies, mechanisms for query injection and data collection based on a subscribe/event mechanism, and adaptable energy efficiency mechanisms.

SMEPP Light targets a Mote-class hardware platform for sensors. A typical example of sensors in this class is the MEMSIC IRIS sensor [7]. It has an 8 bit, 8 MHz processor, 128 KBytes of program memory, 8 KBytes of RAM and 512 KBytes of storage memory.

SMEPP Light is developed on top of the TinyOS operating system [8]. A TinyOS application is a set of components linked together to form an executable. Each component consists of an interface and its implementation. The interface specifies a set of commands implemented by the component and a set of events that the component can signal. Hereafter, to avoid confusion with the term event that is also used for other purposes in SMEPP Light, we will call signals the TinyOS events. TinyOS relies on the concept of split function for energy saving. Split functions are functionalities split into pairs command/signal: when a component A invokes a split function of another component B implemented by a command, the command enqueues the request and returns immediately. Only when the result is available B generates the corresponding signal to give the result to A.

3.4.2 Requirements

The main feature of SMEPP (and thus of SMEPP Light) is that the peers in the same network organize themselves into groups. The existence of different groups is useful since it enables the definition of different security and communication domains, that involve only peers owning the appropriate credentials.

The peers of a group in SMEPP Light interact via a publish/subscribe mechanism. A peer can subscribe for events of other peers that belong to the same group, so that it automatically receives the relevant events whenever they become available. From the security point of view, a group can be open or closed, private or public. In closed groups, a key is necessary to access the group, while the appropriate key is necessary to discover private groups.

SMEPP Light also provides a two-level security based on symmetric cryptography: network-level and group-level. The network-level security exploits two keys: one for packets' confidentiality (used to encrypt the packet) and one for packets' integrity (used to compute a MAC to be attached to the packet). The two keys are set by the application and can be changed at run time. The group-level security exploits three keys, namely the masterKey, the sessionKey, and the sessionMAC. The masterKey is used to restrict the access to the closed group, so that a peer can join a closed group only if it owns the right masterKey. Once a peer joins a group it receives the sessionKey and the sessionMAC.

```
command peerId smepp newPeer(netwKey, netwMAC)
```

```
command smepp createGroup(groupDescription)

command smepp_getGroups(groupDescription)

signal getGroups_result(groupId[])

command smepp_getGroupDescription(groupDescription)

command smepp_joinGroup(groupId, masterKey)

signal peerJoined(peerDescription)

signal joinGroup_result(groupId, subscriptions[], result)

command peerDescr[] smepp_getPeers(groupId)

command smepp_leaveGroup(groupId)

signal peerLeft(peerId, groupId)
```

```
command smepp subscribe(eventName, groupId, expirationTime?, rate?)

signal subscribed(eventName, groupId, expirationTime, rate, offset)

command smepp_unsubscribe(eventName?, groupId)

signal unsubscribed(eventName, groupId)

command smepp_event(groupId, eventName, value)

command smepp_receive(groupId, eventName, frequency)

signal receive_result(sender, groupId, eventName, value)
```

Table 1: Interface of SMEPP Light

These keys are used to enforce data confidentiality and data integrity in all the communications within the group, and they can be updated at run time, however the masterKey must be known in

advance and is set at compile time. Since a peer can join several groups, in all its communications it must specify in clear text the identifier of the group to which the message is directed, so that each peer receiving that message can use the right key to check the message integrity and to decrypt it.

Another important requirement of SMEPP is that each peer must be described by an XML document. In SMEPP Light, the peer description contains the list of the transducers the sensor is equipped with and it is compressed into a bitmask that the sensors can easily store and exchange.

For energy management purposes, each group defines its own duty cycle that drives the radio activity of the peers. Thus each peer in the group operates the energy management according to this duty cycle (e.g., it responds to the messages only when it is active). However, any received subscription can request the peer to use also another duty cycle for environmental sampling. For this reason SMEPP Light manages all the duty cycles (which should coexist) by turning on or off the peer (and thus the radio) whenever necessary.

3.4.3 Specification

3.4.3.1 Interface

SMEPP Light provides primitives for peer initialization, group management, and event transmission. The set of the (main) primitives is shown in Table 1. The peer initialization is executed by the primitive `smepp_newPeer`. It takes in input the network and the MAC keys (hence these keys are established by the application) and returns to the application the peer identifier. This identifier is unique within the network and corresponds to the sensor identifier used by TinyOS and assigned at compile time to the sensor.

The group management primitives support the creation of groups, the search and the join to existing groups. The `smepp_createGroup` primitive creates a group according to the group description taken in input. The description contains the security keys of the group and a set of flags expressing the group security policies in terms of closeness and privacy.

The primitives for group discovery are used to retrieve groups that match search criteria. The command `smepp_getGroups` accepts a group description partially filled up, and returns, via the signal `getGroups_result`, the id of matching groups. Then `smepp_getGroupDescription` is used to read the group descriptions. Group discovery can be based on group name or on the security properties.

The command `smepp_joinGroup` takes in input the master key of the group to be accessed. The result of the join protocol (either success or failure) is returned to the application by means of the `joinGroup_result` signal. If the joining is successful, the peer also receives the session keys used for the communications in the group, the list of the peers belonging to the group, and the list of the subscriptions currently active within the group. The list of subscriptions is also notified to the application layer that can thus begin raising any relevant event. Furthermore, as a result of the join protocol, all the peers in the group are notified with the signal `peer_joined` reporting the identifier of the newly joined peer. The descriptions of the peers into the group can be accessed via the `smepp_getPeers` primitive.

The command `smepp_leaveGroup` is a split function that enables a peer to leave a group. This disassociation is automatically notified to all the peers in the group with the `peerLeft` signal.

The main event management primitives offer functionalities for event subscription and event notification. Any peer in a group can invoke the event subscription that is then issued to all the peers in the group. When one of the peers detects an event matching the subscription it sends the event back to the peer (or the peers) that subscribed for it. The command `smepp_subscribe` takes in input the name of the event to be subscribed and a group id to which the subscription is directed. The event name may encode an arbitrary monitoring task to be run on the peers in the group. The primitive also takes two optional parameters: the expiration time and the rate of the subscription. The rate defines the sampling rate of the monitoring task associated to the event name, which also implies the maximum rate at which the events can be sent back to the subscriber. Each subscription results in the creation of a routing tree spanning all the peers in the group and rooted in the subscriber. This tree is used to route the events to the subscriber. After the expiration time the subscription (and consequently the associated routing tree) expires and the subscriber should issue again another subscribe if it is still interested. The presence of a subscribe request is notified to the application layer of all the peers in the group by means of the `subscribed` signal that provides to the application layer the parameters of the subscription (event name, group id, rate, expiration time) and an offset time, that is the time in which the subscribe has been generated and that is used to synchronize all the peers in the group about that subscription. After the invocation of the command `smepp_subscribe`, the subscriber can invoke the command `smepp_receive` to start waiting for the corresponding events.

If a peer detects an event matching a subscription, it sends the event to the subscriber by using the command `smepp_event`, that routes the event using the routing tree constructed by the subscribe. When the event reaches the subscriber, SMEPP Light provides the event to the application layer by raising the `receive_result` signal that provides to the application layer the value associated with the event along with the event name, the identifier of the peer that detected the event, and the identifier of the group where the event was detected.

3.4.3.2 Architecture

SMEPP Light is composed by three main components, namely the Peer Identification, Group Management and Event Management, that implement the SMEPP Light primitives, and three components that provide support to security, networking, and energy efficiency. The interaction among these components is shown in Figure 9.

The Peer Identification component maps to the peer initialization primitives and it interacts with the Security component to set the network keys.

The Group Management component manages the topology of the groups and maps to the group management primitives. It is in charge of the group descriptions and duty cycle management and it interacts with all the support components: it sets the group master key, it sends and receive data from the Network component when most of the primitives are executed, and it interacts with the Energy Efficiency component to set the information for the management of the peers' duty cycle.

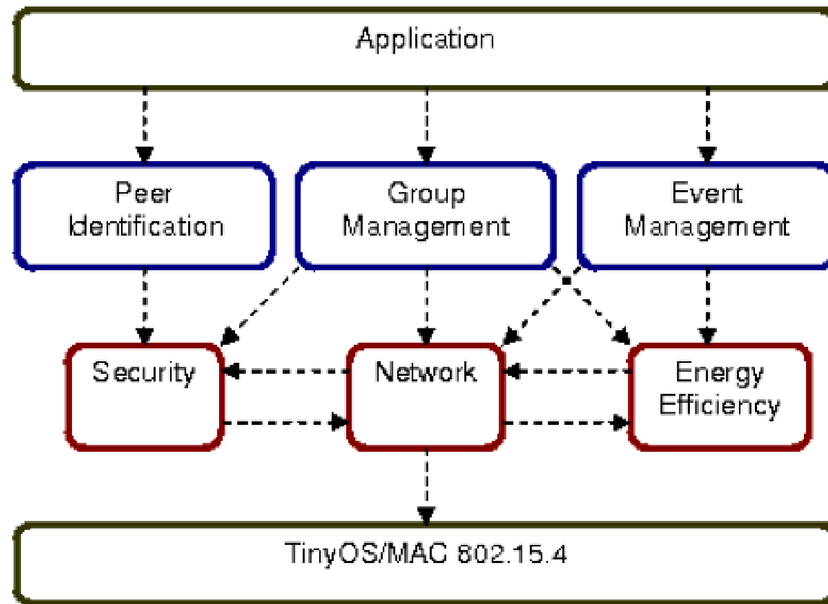

Figure 9: Components in SMEPP Light architecture

The Event Management component maps to the event management primitives and it is in charge of subscriptions and events. This component interacts with the Network component to access the wireless medium and to set up the routing trees associated with subscriptions, and it interacts with the Energy Efficiency component to configure the duty cycle of the peer according to the subscribes generated or received.

The Security component manages the keys for all the security issues related to the network and to the group layers. It keeps the network keys set by the Peer Identification component, the group master key set by the Group Management component, and the group session keys received from the Network component during the join protocol. This component also manages the protocol for the dynamic refresh of the session keys.

The Network component implements the communication between peers. Its main mechanisms are the network broadcast used to implement the subscriptions and the management of the routing trees associated to them. It also provides a 1-hop broadcast protocol used to implement the group discovery and the join mechanisms.

The Energy Efficiency component manages the duty cycles of the peer. In particular it manages the on/off periods of the radio interface according to the duty cycles associated to the subscribe messages received or generated by the peer. It should be observed that the management of the radio is transparent to the other components, since this component makes sure that the radio is activated before it is used by other components and that it is turned off soon after its use.

3.4.3.3 Protocols

For the sake of brevity we describe only the group creation/join protocols (Figure 10) and the subscribe/event protocols. We illustrate the protocol of group creation and join referring to the diagram of Figure 10, where it is shown the case where Node B creates a group and Node A joins the group of B.

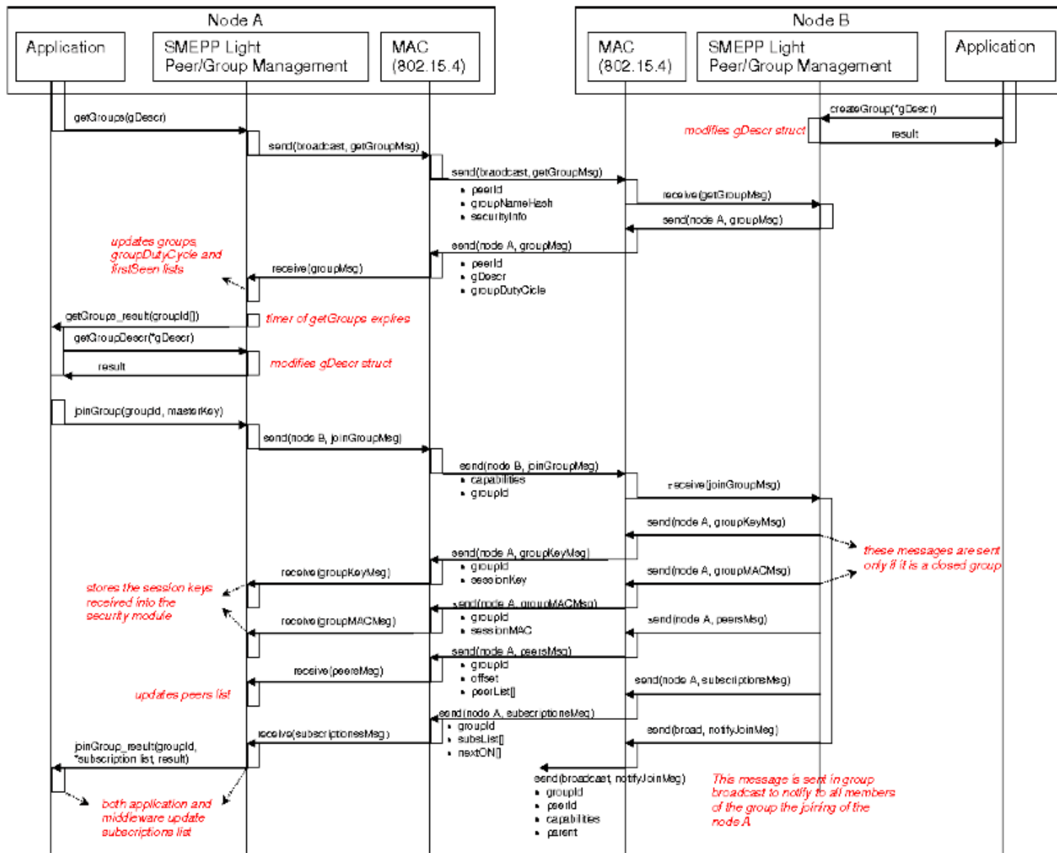

Figure 10: Creating a group

Node B creates the group using the `createGroup` command. The group creation does not involve communications since it consists in setting a few data structures in SMEPP Light and in setting the master and the session keys in the B's Security component, hence it can immediately provide the result of the operation without resorting to the split function mechanism. The application layer of Node A performs the search for existing groups by invoking the `getGroups` command. This command sends to all the A's neighbors (in local broadcast) a message requesting the group descriptions of the existing groups. At Node B, SMEPP Light replies to this request (without involving the application layer) by sending to A the descriptions of all the groups known to Node B (in this case only one). At Node A SMEPP Light keeps all the received descriptions and after a timeout it notifies to the application layer the list of identifiers of all the groups detected. The application at Node A can then choose a group ID and it can access the corresponding group description by invoking the `getGroupDescription` primitive.

Now the application layer of Node A can invoke the `joinGroup` primitive to join the group. As a consequence, SMEPP Light sends in unicast to Node B the request to join the group. This message is notified to SMEPP Light in Node B, that, in turn, decides whether to accept the request of A. In our case the request is accepted, hence SMEPP Light in Node B sends a set of messages to A. These messages contain the session keys, the list of peers belonging to the group, and the list of

subscriptions that are currently active into the group. All of these messages are used by SMEPP Light in Node A to update its internal data structures, and once this phase is completed SMEPP Light raises a `joinGroup_result` signal to the application layer of the same node to notify that the join protocol is completed. In the meantime, SMEPP Light in Node B sends in broadcast to all the other peers in the group a message notifying that Node A joined the group.

The subscribe protocol is initiated by any peer in a group that wants to receive a given type of event generated by other peers in the group. Consider for example the case where Node A subscribes for an event that is generated by a Node B. To this purpose Node A invokes a subscribe command, that broadcasts the subscription to all the peers in the group. Then Node A invokes the `receive` command to prepare SMEPP Light in Node A to receive events related to this subscribe. When Node B receive the subscribe message, SMEPP Light in Node B notifies this request to the application layer by means of the `subscribed` signal. This signal gives to the application layer the event name for which the subscription holds, hence it is responsibility of the application to start any relevant monitoring task to detect the events matching the subscribe. This monitoring task should be activated at the sampling rate contained in the subscribe message. Whenever the application in Node B detects an event it sends the event to the subscriber using the `event` command. This primitive sends a message containing the event to SMEPP Light in Node A that, in turn, notifies the application in Node A by means of the `receive_result` signal. This protocol continues until the subscription expires or Node A cancels it using the `unsubscribe` command.

If a primitive fails, the middleware notifies the error to the application to let it implement fallback policies. There are however some exceptions to this general behavior when the middleware cannot identify the fault. One example is the removal of a peer from a group without the call of the `leave` primitive. This case may result in broken routing trees, stopping data flows related to some subscription. SMEPP Light copes with this problem via regular refreshing of subscriptions. There are also some situations that cannot be coped with, for example when the sudden removal of a peer results in a group partition. In this particular case, the group splits up, but the peers that remain connected continue working without the sensors in the partitioned branch.

3.4.4 Energy Efficiency

The Energy Efficiency component of SMEPP Light saves sensors' energy by keeping the radio off whenever possible, i.e. when the sensors do not expect to receive or send data. This component manages the radio by means of user duty cycles that are implicitly determined by the subscribe messages (recall that these messages contain the subscription rate and an expiration time), and a management group duty cycle that enables the sensors in the group to exchange control messages (in particular join and subscribe messages). Each of these duty cycles defines periodic intervals when all the sensors should turn on the radio.

As a side-effect of this approach, when a sensor executes the `getGroups` primitive it needs to keep sending request messages until one of these messages is sent during a period of activity of a group. However during the `getGroups` protocol the sensor receives enough information to synchronize with the other sensors of the group, so the subsequent communications are performed according to the management duty cycle of the group. To make this approach effective, the sensors need to be (weakly) synchronized, for this reason synchronization information is periodically exchanged among the peers in the group.

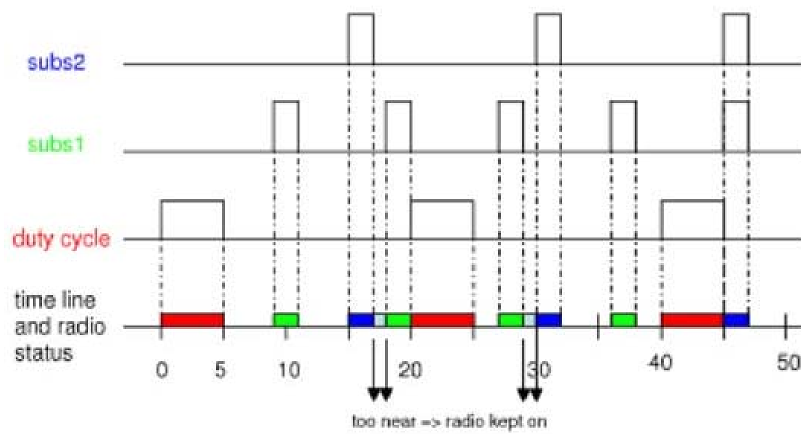

Figure 11: Duty cycles and subscriptions

Figure 11 shows an example of the status of the radio of a node that belongs to a group in which are active two subscriptions (subs1 and subs2). The three lines on the top show the activity windows of the radio for the user duty cycles corresponding to the two subscriptions and for the group management duty cycle. The last line shows the overall radio status. The Energy Efficiency component computes the union of all the duty cycles, and decides when the radio should be turned off and on, according to two parameters (tolerance and radio-delay) that provide some flexibility to the system. The tolerance parameter specifies the minimum distance (in milliseconds) between the end of an active radio window and the start of the next one: if two windows are too close the radio is kept on until the second window ends. The radio-delay parameter expresses the time used to anticipate and delay the radio commutations from off to on and from on to off, respectively.

The performance of the energy efficiency mechanism has been evaluated by measuring the periods of active radio of a sensor. In the experiments we used 4 MicaZ motes (s1, s2, s3, s4) connected in a line, i.e. s1 is connected to s2, s2 to s3 and s3 to s4. We measured the radio status of node s2, s1 is the node that produces the subscriptions and s4 generates the events. We repeated four sets of experiments with a number of subscriptions ranging from 0 to 4. The rate of each subscription has been set randomly in each experiment. Each experiment has been repeated 10 times for 180 seconds, and in each experiment we measured the average period of time in which the radio of sensor s2 was inactive, ready, receiving and sending. With these data we computed the average energy consumption of sensor s2 (expressed in mA-hr) in every set of experiments, as shown in Figure 12. For a comparison the figure reports the energy consumption estimated with the TOSSIM simulator and the energy consumption in the case where the energy efficiency module is disabled. The figure shows that the energy efficiency strategy enables significant energy savings, and that the energy consumed grows sub linearly with the number of subscriptions.

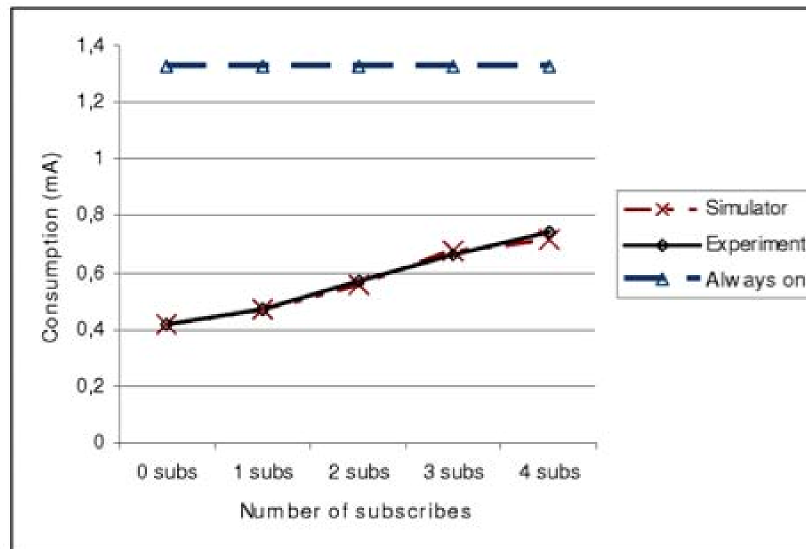

Figure 12: Energy consumption (in mA-hr)

4. High-Level Design

4.1 Design overview

As described in Section 1, the Communication Layer will make available different paradigms of communication on the basis of the type of hardware involved in the communication.

From the responsibilities of WP1 we envisage that the communication layer must enable synaptic communications between all four combinations of mote/PC to mote/PC. For exchange of raw sensor readings and actuation commands it must enable PC to PC communication as well as some combinations of PC to mote communications and mote-to-mote communications. However, the form of direct non-synaptic communications in the mote-to-mote scenario is different from the PC to mote and/or PC-to-PC scenarios.

For communication involving Motes-to-PC and Motes-to-Motes, we will use three types of communications:

1. **Synaptic Communication**: This type of communication is used exclusively by the Learning Layer and enables two ESNs (Echo State Network), running on different nodes, exchanging data. In particular it enables the transmission of the output of a set of neurons from a source ESN to a destination ESN.
2. **Data Stream Communication**: This type of communication is used exclusively by the Control Layer and enables the point-to-point communication between two specific devices, typically for reading data from remote mote transducers.
3. **Connectionless Message Passing**: This type of communication is used exclusively by the Control Layer and enables the point-to-point communication between two specific devices, typically for sending commands to remote mote-actuators.

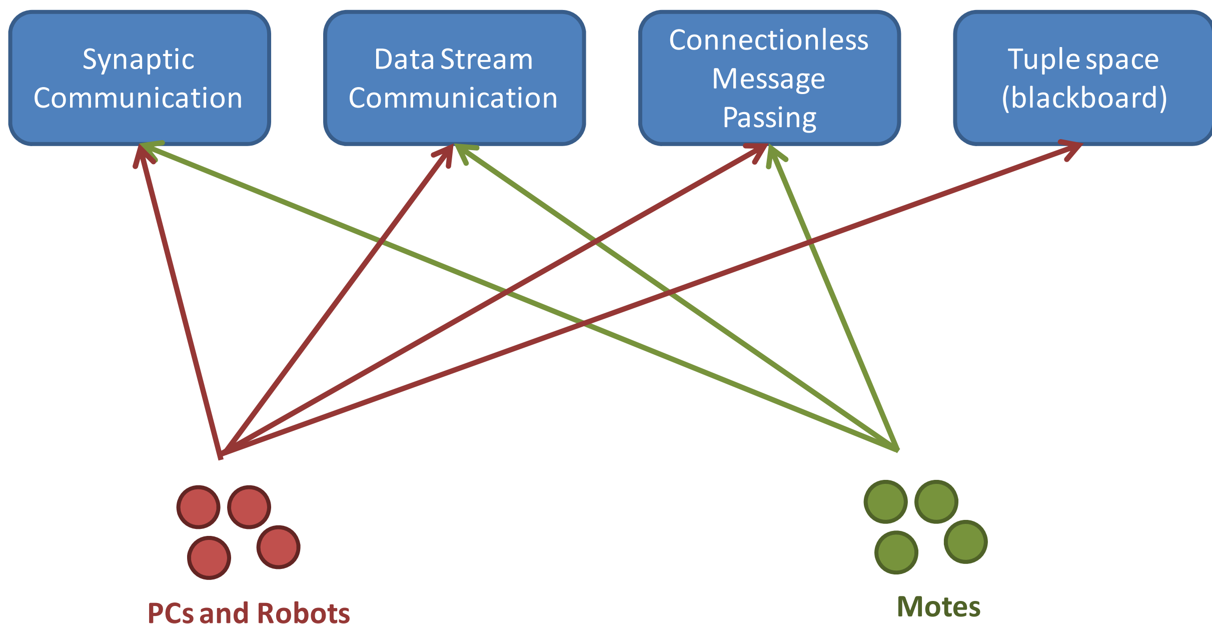

Figure 13 - Type of communications and devices involved.

Figure 13 shows the relationships between the devices used in RUBICON and the type of communication paradigm used by them.

4.1.1 Gateways and Proxies

For communication involving PC-to-PC, we will use the **Tuple space** (blackboard) mechanism, which provides a repository of tuples in a distributed shared memory that can be accessed concurrently.

For the PC to mote scenario we will rely on a *basestation,* which sits on the border of the PEIS ecology and is directly connected (by means of a USB or serial cable) to a WSN *sink.* This *gateway* device will translate a *tuplespace* representation of actuation commands and raw sensor readings and the *connectionless message passing* mechanism on the WSN side.

In practical terms this is done by continuously publishing tuples with the latest received (raw) sensor readings from each WSN that transmits such messages. The name of these tuples will contain the unique ID of the corresponding WSN mote. Furthermore, it will listen for tuples containing raw messages on tuples containing these ID's and will translate the content of these tuples as *connectionless message passing* commands to the *basestation*.

This provides a generic light-weight interface between the PC and mote. The component that performs this task of WP1 is called the ***gateway***.

To fulfil the R1.SCALABILITY and R1.DISTRIBUTION requirements, the Communication Layer will be developed to support a star-topology (see Figure 14), in which a cluster of motes, called islands, will be interconnected by mean of the PEIS-network. A special mote acting as sink node, connected via USB with a *basestation* (a PC) will host the gateway component. Please refer to Section 4.2 for additional information on how the gateway component will be implemented.

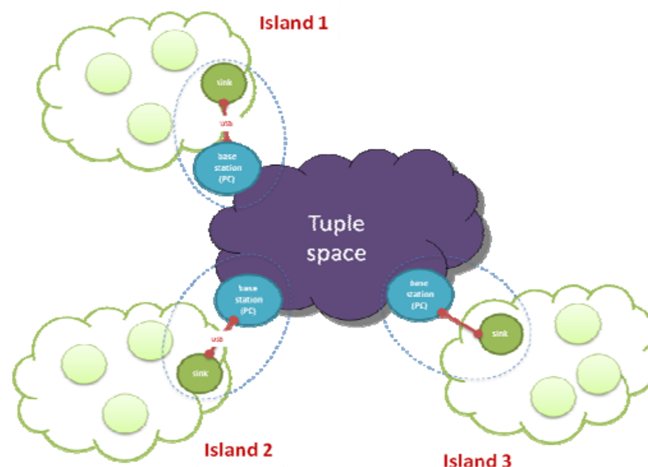

Figure 14 - Topology of the RUBICON Ecology

From the implementation point of view, the Communication Layer will exploit both middlewares, the PEIS middleware and the TinyOS middleware. The details of these two middlewares are provided in Section 4.2, what it is important to know at this stage is how they are used in scenarios when the motes involved in the communication belong to the same island or belong to remote islands.

Figure 15 depicts the first case, which is the simplest one, when the communicating motes lie in the same islands, in which TinyOS is used.

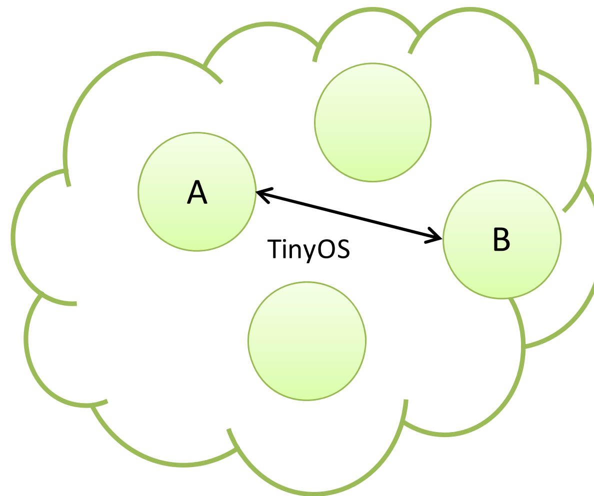

Figure 15 - Communication between two Motes (A-to-B) in the same island.

The second case is represented in Figure 15, in which we show how two processes running on two motes lying on remote islands exploit the PEIS middleware to communicate. To be able to communicate across these islands we require a special mote called a *sink* connected with a PC by means of a serial link (such as USB). This PC is the *basestation* that hosts the gateway component and routes the messages between the PC/Robot and the motes.

In the example scenario below, two *basestations*, which manage one island each, can communicate with each other through the PEIS middleware. By exploiting the P2P network of PEIS these devices can route messages originating in the islands to each other and forward such messages to the destination motes.

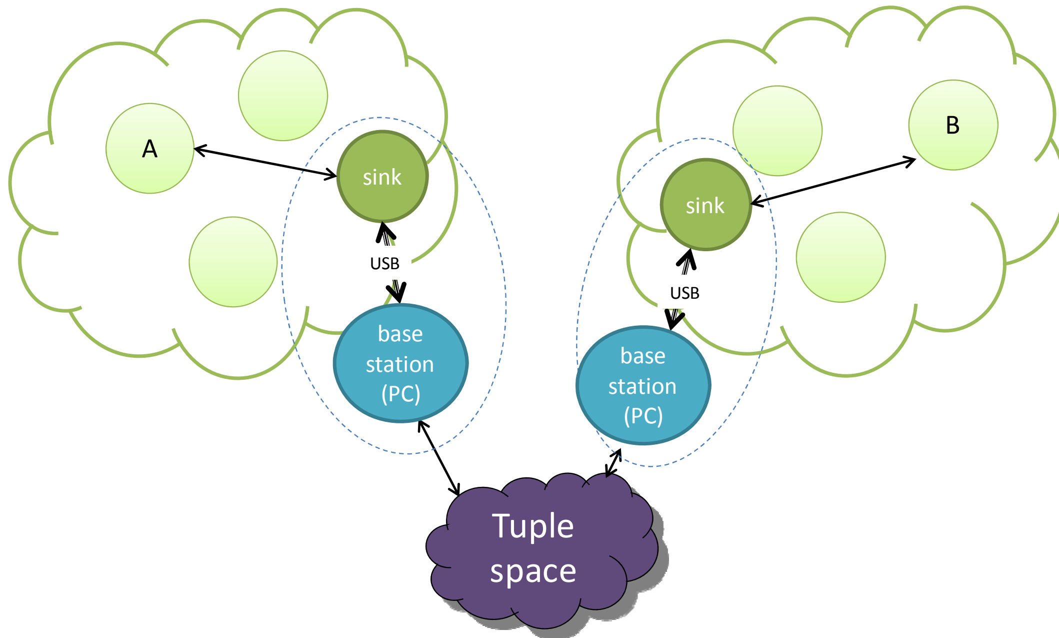

Figure 16 - Communication between two Motes (A-to-B) in remote islands

Finally, the last case is when a mote must communicate with a PC or a robot. In this case, we again use the PEIS-middleware as a means for connecting the gateway with the remote PC/robot, see Figure 17, but give a tuplespace-based interface for reading/sending raw messages.

In order to accommodate for more generic and high-level integration between the networked robotic software and the WSN we will use the notion of proxied devices to give higher level representations of the available WSN motes to the robotic devices.

In practical terms this corresponds to PEIS components, *aka.* ***proxies***, that subscribe to the tuples provided by the gateway devices in order to receive raw sensor readings. These sensor readings are parsed in a mote-specific way and translated into the general (human readable) format used on the PEIS network. Furthermore these proxies will subscribe to *meta tuples* that correspond to inputs that will be transmitted to the actuation devices – also by means of the message passing functionality of the routing devices.

It is important to have a clear idea of the difference between gateway and proxy. The gateway has two responsibilities:

- It communicates with the sink mote and can send data to the other gateways connected to other motes. (i.e. it can bridge them over the PEIS P2P network).
- It can accept tuples and translate them into raw messages that are sent to motes.

Notably, the gateway does not have any logic for "understanding" what a mote can do. It cannot create semantic representations of the motes (i.e. it does not export XML or JSON tags describing what functionalities a given motes export, what type of messages they can accept or the data-format that is produced by them). This task is instead delegated to a proxy.

Table 2 summarizes the relationship between the middleware and the paradigm of communication used for different combination of communicating peers.

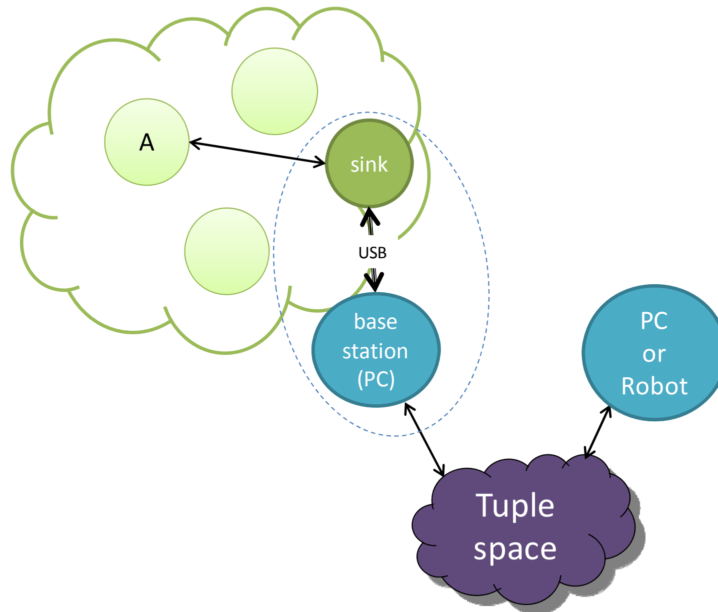

Figure 17 - Communication between a mote and a remote PC/ Robot.

	PC-to-PC	Mote-to-Mote	PC-to-Mote
Synaptic Communication *(point-to-point)*	Streams over Peis Kernel	TinyOS / TinyOS+Gateway+PEIS	TinyOS / TinyOS+Gateway+PEIS
Data Stream Communication *(point-to-point)*	Peis Middleware	TinyOS / TinyOS+Gateway+PEIS	TinyOS / TinyOS+Gateway+PEIS
Connectionless Message Passing	Peis Middleware	TinyOS / TinyOS+Gateway+PEIS	TinyOS / TinyOS+Gateway+PEIS
Tuplespace *(blackboard)*	Peis Middleware	*Not supported*	*Not supported*

Table 2 - Summary of the communication relationships

4.2 Implementation issues

4.2.1 High level architecture overview

In order to give a practical guide for the design of Communication Layer components, in this section we analyze more in detail for the most complex last two cases where the communication involves tinyOS and PEIS. This guidance is given for Control Layer, however, can be adopted in other Layer.

In Figure 18, we show how the control layer program running on mote in an island send a message to the control layer running in a remote PC. This is an additional level of detail of the Figure 17. First the communication layer installed in the source mote forward the message to communication layer of the sink responsible for its islands. The gateway part running on the sink mote uses the USB connection to forward the message to the gateway of the *basestation*, which in turns publish the message as a tuple for the control layer of the remote PC.

Figure 19 shows the same sketch also the communication of two motes belonging to different islands. This is an additional level of detail of the Figure 16.

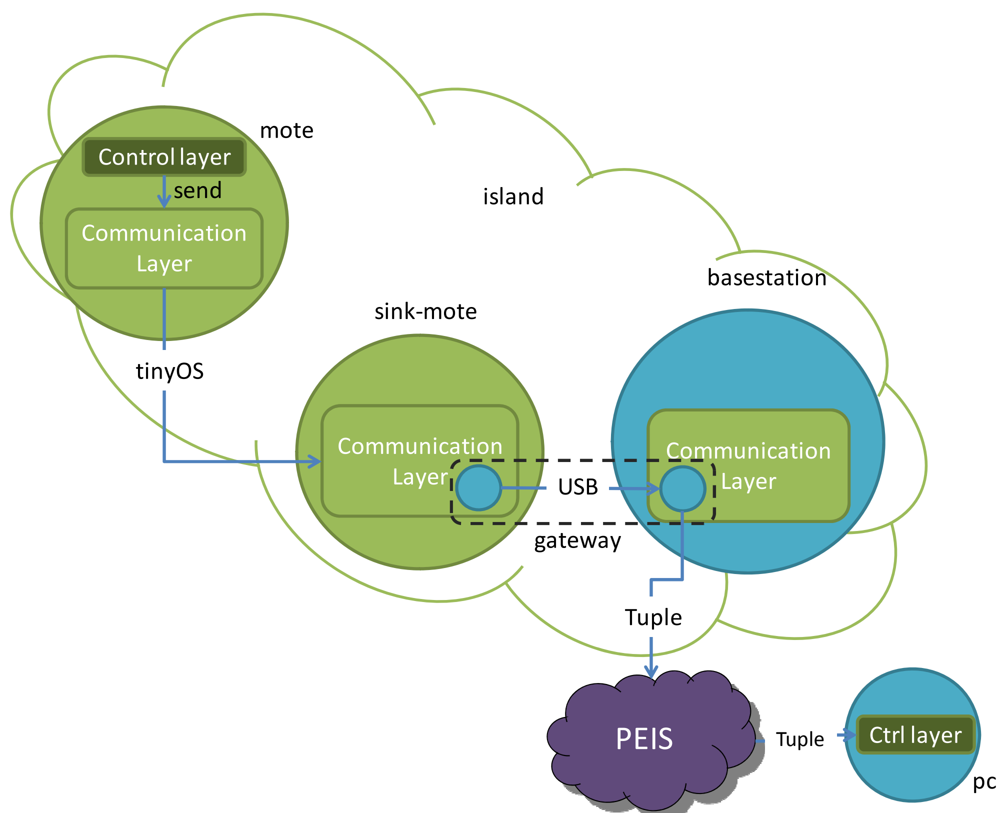

Figure 18 - The communication messages between a mote and a PC in details

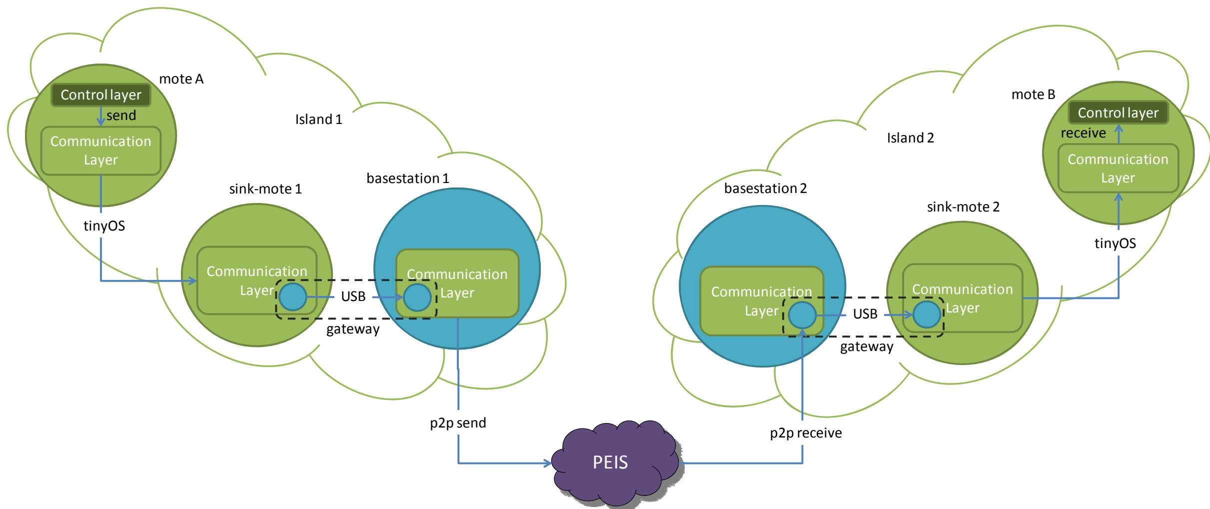

Figure 19 - The communication messages between two foreign motes in details.

Suppose that a process of the Control Layer running on the mote A belonging to the **Island 1** needs to send a message to a process of the Control Layer running on the mote B located in the **Island 2**. Figure 20 shows the temporal sequence diagram of the communication process steps by highlighting the role of the two Layers involved. In this figure, we also show the separation between the API call and the message invocation.

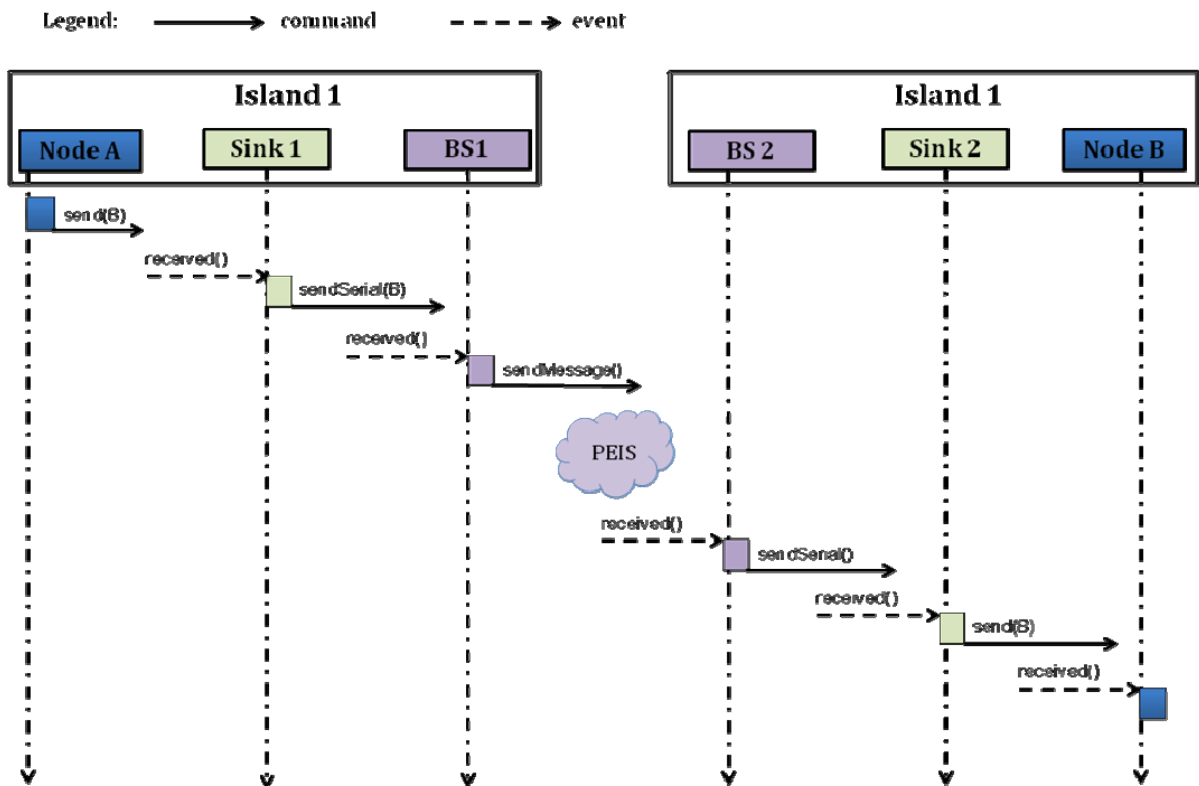

Figure 20 - Sequence diagram of the communication between two motes of the Control Layer

The Control Layer in mote A invokes a call (`sendSerial()`) to the Communication Layer in the same mote to pass the message to be sent. This message contains the identifier of the mote B as destination of the message. The Communication Layer in mote A is notified of the incoming message as consequence of the `sendSerial()` (this method is implemented in the Communication Layer implementing, the Control Layer simply invokes it).

According to the addressing mechanism adopted, Communication Layer implementing in mote A determines that the destination mote does not belong to the same island, so it encapsulate the message in a new message with *sink 1* as destination and transmits it by means of a nesC command (`send()`).

The Communication Layer implementing in *sink 1* receives this message and forwards it on the serial interface to the gateway component in the *basestation 1*. The gateway just receives a notification of the incoming message and must implement the handling of this message.

In addition to the target mote, this message consists of a unique identifier for the island to which the target mote is connected as well as the data of the message. The gateway utilizes the `peisk_sendMessage()` functionality to transmit the message to the corresponding gateway in *basestation 2,* which is connected to the target mote B.

Basestation 2 in the **Island 2** is notified of this message. It forwards this message to the Communication Layer in *sink 2* by invoking a call to `sendSerial()`. *Sink 2* finally knows a route to mote B and the Communication Layer at this mote forwards the message to destination.

4.2.2 Synaptic channels

Requirement **R2.INPUTSTREAM** is needed in order to provide the requested output prediction at each RUBICON clock tick, the Learning Network needs to have all the inputs readily available at its input interface. For this purposes, all the Motes involved in the Learning Network, i.e., on which the Learning Layer is installed, will be have an internal clock synchronized. The synchronized will be guaranteed by means of special broadcast message, which will be periodically sent in order to keep the motes' clocks aligned. For the sake of flexibility, the transducer will be used as local clock for a mote. For the mote that do not have a transducer, as fake transducer will implemented in order to simulate the local clock.

4.2.3 Multitasking

In TinyOS TCP/IP Sockets are not available to support multitasking communications. The reason is that it would employ too much memory for buffering. To cope with this problem and to satisfy **R1.MULTITASKING**, in our implementation we use the mechanism active messages and interfaces.

Every message in TinyOS contains the name of an event handler. The Sender must declare a buffer storage in a frame and in the receiver the event handler is fired automatically in a target node.

A TinyOS application is a set of components bound by bidirectional interfaces. A component can either USE or PROVIDE an interface. Each interface provides COMMANDS that a component using that interface can CALL (for example Send()), and EVENTS that a component using that interface

MUST implement (for example Receive()). The component providing an interface is responsible to implement the COMMANDS (in case of the Send() COMMAND implementing the actual transmission of a packet over the air) and to NOTIFY the EVENTS (for example notify the component using the interface when a new message is received). The component using an interface simply can CALL a COMMAND to execute a task (for example call Send()), but it must implement an event handler for each EVENT defined in the interface (for example event Receive()).

4.2.4 Discovery

In order to satisfy the requirement **R1.DISCOVERY**, we ,must implement a mechanism that allows the ecology to know when a new mote joins a WSN. To this end, we envisage the implementation of a special agent of the control layer, which is responsible to signal the *basestation* that there is a new node on the island and all the abilities of transduction and actuation that are present in this new mote. In particular, this agent on the mote is also in charge of performing the actuations and to route the streams coming from the transducers. When the *sink* of the island receives a join message from a new mote, it sends an event to the gateway of the *basestation* connected to it via the serial port, which in turn publish a tuple on the PEIS Tuplespace with all the metadata describing the functionality of the joining mote. This tuple can be accessed by other entities of the ecology and allow them to discover of the motes present in all the islands.

4.2.5 Implementation of proxied objects and processes

The design pattern of proxied objects, as described in Section 3.2 will be used within RUBICON to simplify the *high level integration* of simple devices with the robotic devices in the ecology. This concept will be used also to simplify the integration of *application dependent* objects into already deployed RUBICON ecologies. As such it will allow for customization of the available hardware devices without any need for re-implementation of any of the robotic software already running in the ecology.

This concept will be used within RUBICON for two main purposes; (1) to provide higher level computational capabilities and representations to motes that lack the resources and (2) to provide all computational and representation capabilities to devices that have none. While originally only the later use of proxies was considered, we have in this designed opted also for the former in order to tackle the challenges required by the limited amount of RAM and computational capability of the motes. Additionally, since the planner expected to be deployed on *all* agents (see D3.1) will not be be able to run onboard such motes we will rely on this proxy mechanism to outsource such ego-centric planning tasks.

As such, the design pattern will be augmented to satisfy a number of new requirements within RUBICON.

First of all, due to the distributed nature of the learning layer and the agent architecture within WP3 we have a requirement of the proxy mechanism to allow for the computational representations of a WSN mote to be split between the physical mote and a proxy process running on computationally more powerful devices. We thus have a requirement on the extended design pattern for proxies to allow for a seamless integration between these two parts of the WSN based agent.

Secondly, the proxied motes must be able to communicate both with other WSN agent as well as with the main robotic ecology. Since these two types of destinations used different mechanisms for communications (Streams vs. tuples) this pose an additional challenge for the implementation of proxies.

When implementing the proxied objects concept for WSN motes in the RUBICON ecology we will use the gateway devices to support message passing between the proxy processes running anywhere in the ecology and the individual motes. Thus the interface device to the proxied WSN motes, as defined in the design pattern for proxies, are the gateway. Each gateway provides a list of the WSN motes with which it can communicate and facilities the exchange of messages with the WSN.

As described in Section 3.2 the proxy manager for RUBICON must be able to detect the presence of new motes. For the case of RUBICON and WP1 this process is trivially implemented for the WSN motes since discovery of the motes and messages from the motes inherently contain a unique sender ID which is associated with each unique proxy id. The proxy manager for RUBICON will rely on these descriptions in order to detect when a new mote have appeared or disappeared and notify the currently running proxy processes accordingly. Thus satisfying the requirement R3.DISCOVERY in WP1.

We require a *signature* for each mote in order to allow the proxy manager to know the type of proxy process that should be started. In RUBICON a signature of a WSN mote is simply a type identifier that identifier the type of mote and configuration with which it communicates. These signatures are represented by an integer given by the WSN mote where proxy programs (processes that can be launched when encountering a matching mote) can claim responsibility for a subset of the space of all the possible types. These signatures are provided by a unique type identifier that is compiled into each WSN mote when assembled for a specific application.

For the point of view of WP1 the implementation of proxied WSN motes will occur during task T1.4. During this task we will address the new requirements as described above and any other constraints posed by the RUBICON ecology. We will deal with the distribution of the planning and control as required for the individual agents for both WSN based proxies as well as traditional proxies of everyday objects. The details of this will be covered in deliverable D1.4.

5. Interface Specification

5.1 System overview

The Communication Layer is a complex software system organized into two logical sub-layers (using the terminology of the OSI reference model), separated by a dashed line in the box representing the communication layer (see Figure 21):

1. The Transport sublayer provides transparent transfer of data between end users, providing reliable data transfer services to the upper layers.
2. the Network sublayer provides the functional means of transferring variable length data sequences from a source entity on one network to a destination entity on a different network, while maintaining the quality of service requested by the Transport Layer.

A sublayer is realized by a variable number of software components, depicted as simple rectangular boxes in Figure 21, that are distributed over an heterogeneous networked architecture comprising both resource constrained devices (e.g. sensor nodes) as well as powerful gateways.

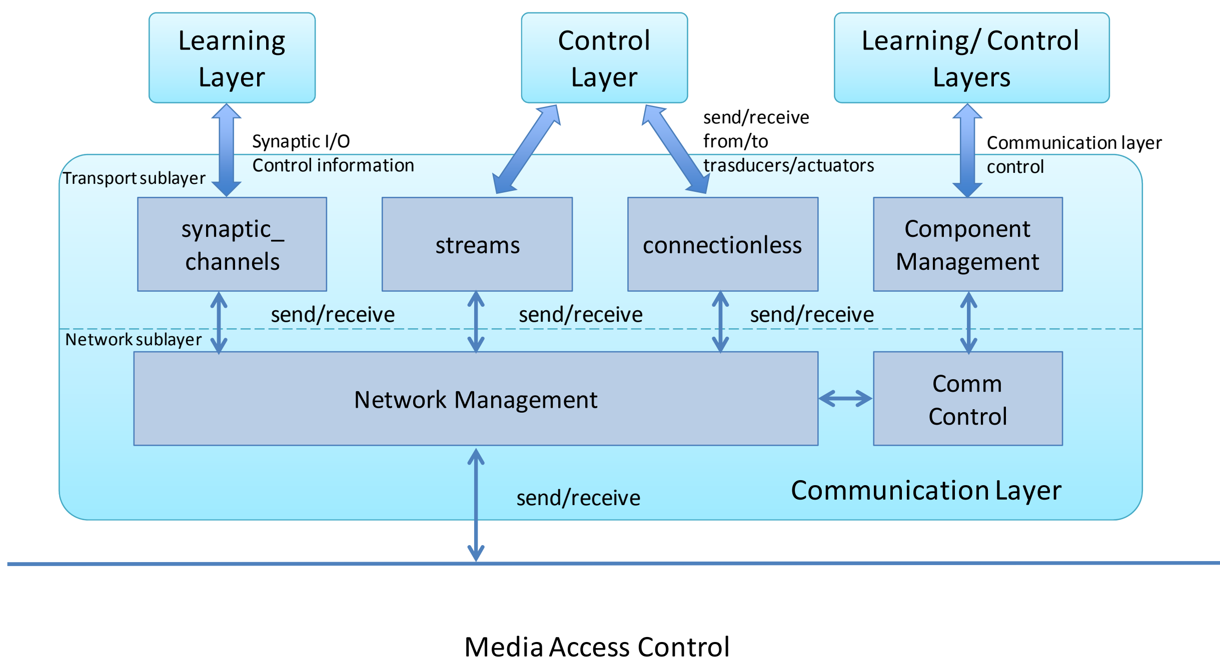

Figure 21 - Software architecture for the Communication Layer: logical sublayers are separated by a dashed line, software components are small rectangles.

The software components realizing the functionalities of the subsystems within the Communication Layer interact through local function calls as well as with remote messages: at this point, we abstract from such distinction and we represent all *intralayer* interactions as thin arrows in Figure 21.

In the following, we summarize the role and the functionalities associated to the components of the Transport sublayer.

5.2 Transport sublayer and addressing

The Transport sublayer provides transparent transfer of data between end users, providing reliable data transfer services to the upper layers. The Transport Layer controls the reliability of a given link through flow control, segmentation/desegmentation, and error control. Some protocols are state- and connection-oriented. This means that the Transport Layer can keep track of the segments and retransmit those that fail. The Transport layer also provides the acknowledgement of the successful data transmission and sends the next data if no errors occurred.

The addresses of mote in the Ecology is formed by the address of the island and the address of the mote in the island. The address of the island is a unique number that corresponds to the *peis-id* of the *basestation*. It is codified with a byte, that is interpreted as un unsigned integer. The value 255 is used to indicate "myself". The address of the mote is a two bytes unsigned integer and is a unique number that corresponds to the TinyOS-ID. The value 65535 is used to indicate a broadcast inside the island, and the value 65534 is used to indicate "myself".

5.3 Specification of the Mockup Layer

In this section we specify the interfaces of the Transport sublayer, which represents the interface of the Mockup Layer. The implementation of this Mockup interface is aimed at verifying its correctness of the specified components of the communication infrastructure in its early stage. The implementation of the communication functionality will proceed through two main refinement steps. The first step will produce a non-optimized, fully functional version that will be used by the other WPs to test their solutions in the first part of the project. The second phase will produce the final, optimized version of the layer that will be used in the final

5.3.1 Specification of the functions of the Synaptic_Channels component

Function: ***create_syn_channel_out****(dest, size, params[DIM_SYNCHANNEL_PARAM])*

Synopsis: Setup of a Synaptic Channel between local node and a specified destination node.

inputs:

- *dest*: Destination mote of the Synaptic Channel.
- *size*: Number of neuron readings whose values have to be transmitted to dest (i.e. the channel size).
- *params*: Channel parameters like type, QoS..

return:

- *FAIL* if opening the stream is impossible because, for example, Stream System layer data structures have no space to store a new stream or the stream is already opened.
- *SUCCESS* means a **createSynChannelOutDone** event will be signaled once the Synaptic Channel has been opened.

Event: ***createSynChannelOutDone****(channel, success)*

Synopsis: Signal the outcome of the output Synaptic Channel creation request.

<u>inputs</u>:

- *channel*: Descriptor of the opened Synaptic Channel.
- *success*: it tells if the Synaptic Channel creation was successful.

<u>Function</u>: ***create_syn_channel_in****(src, size, params[DIM_SYNCHANNEL_PARAM])*

<u>Synopsis</u>: Setup of a Synaptic Channel between a remote src node and the local node.

<u>inputs</u>:

- *src*: Source node of the Synaptic Channel.
- *size*: Number of neuron readings whose values have to be transmitted to dest (i.e. the channel size).
- *params*: Channel parameters like type, QoS..

<u>return</u>:

- *FAIL* if opening the stream is impossible because, for example, Stream System layer data structures have no space to store a new stream or the stream is already opened.
- *SUCCESS* means a **createSynChannelInDone** event will be signalled once the Synaptic Channel has been opened.

<u>Event</u>: ***createSynChannelInDone****(channel, success)*

<u>Synopsis</u>: Signal the outcome of the output Synaptic Channel creation request.

<u>inputs</u>:

- *channel*: Descriptor of the opened Synaptic Channel.
- *success*: it tells if the Synaptic Channel creation was successful.

<u>Function</u>: ***dispose_syn_channel****(channel)*

<u>Synopsis</u>: Closes the specified Synaptic Channel that resides in the node where the function is invoked.

<u>inputs</u>:

- *channel*: The descriptor of the Synaptic Channel to be closed.

<u>return</u>:

- *FAIL* if opening the stream is impossible because, for example, Stream System layer data structures have no space to store a new stream or the stream is already opened.

<u>Function</u>: ***start_syn_channel****(channel)*

<u>Synopsis</u>: Starts transmitting the stream of neuron output vectors on the specified channel. A channel that is activated with this primitive becomes an active synaptic channel.

<u>inputs</u>:

- *channel*: The descriptor of the Synaptic Channel whose stream of neuron output has to be started.

<u>return</u>:

- *FAIL if the operation can not be completed (for example the channel does not exists); SUCCESS otherwise.*

<u>Function</u>: ***stop_syn_channel****(channel)*

Synopsis: Stops transmitting the stream of neuron output vectors over the specified active synaptic channel.
inputs:
- *channel*: The descriptor of the Synaptic Channel whose stream of neuron output has to be stopped.

return:
- *FAIL if the operation can not be completed (for example the channel is not active, or does not exists); SUCCESS otherwise.*

Function: ***write_output** (channel, k, val)*
Synopsis: Fills the k-th element of the ChOut vector associated to the Synaptic Channel channel with the value val.
inputs:
- *channel*: The descriptor of the Synaptic Channel that uniquely identifies a ChOut vector structure in the output interface of the local mote.
- *k*: The index of the ChOut element where to write the value.
- *val*: The value to be written in the ChOut vector.

return:
SUCCESS if the write operation is completed successfully, FAIL otherwise.

Event: ***syn_input_update** (channel, index, status)*
Synopsis: Notifies the status update of an input synaptic connection.
inputs:
- *channel*: he descriptor of the Synaptic Channel that uniquely identifies a ChIn vector structure in the input interface of the local mote.
- *index*: The index of the ChIn element whose status update has to be notified.
- *status*: The status of the synaptic connection.

Function: ***read_input** (channel, index)*
Synopsis: Read the specified value in the ChIn vector of the input interface of the ESN running on the local mote.
inputs:
- *channel*: The descriptor of the Synaptic Channel that uniquely identifies a ChIn vector structure in the input interface of the local mote.
- *index*: The index of the ChIn element whose value has to be returned.

return:
the requested value.

Event: ***clock_tick** (currentClock)*
Synopsis: Signals the tick of the current distributed RUBICON clock T.inputs:
- *currentClock*: The current value of the RUBICON clock.

5.3.2 S Specification of the functions of the Streams component

Function: ***open_remote*** (*island_addr, mote_addr, symbolic_name, stream_rate, qos*)
Synopsis: Request that a remote stream is opened.
inputs:
- *island_addr*: The address of the island destination of the message.

- *mote_addr*: Destination of stream (if *open_remote* is invoked by destination, dest is the node itself).
- *symbolic_name*: Code univocally identifying the stream to be opened.
- *stream_rate*: Rate at which source (destination) of the stream is going to write (read) data into (from) the stream. It is expressed in milliseconds.
- *qos*: Quality of service of the stream.
- *buffer_size*: Maximum size of data (in bytes) that can be sent over the stream.

return:

- *FAIL* if opening the stream is impossible because, for example, Stream System layer data structures have no space to store a new stream or the stream is already opened.
- *SUCCESS* means a ***openRemoteDone*** event will be signaled once the stream has been opened.

Event: ***openRemoteDone****(stream_desc, symbolic_name, success)*

Synopsis: Signal that the remote stream has been opened.

inputs:

- *stream_desc*: Descriptor of the opened stream.
- *symbolic_name*: Code univocally identifying the opened stream.
- *success*: Whether the ***open_remote*** was successful.

Function: ***open_sensor*** (*sensor_tid, stream_rate, sampling_type*)

Synopsis: Open a sensor stream.

inputs:

- *sensor_tid*: Identifier of the transducer whose sensor stream has to be opened.
- *stream_rate*: Rate at which sensor must sample environment. It is expressed in milliseconds.
- *sampling_type*: If sampling is periodic or on demand.

return:

- *stream descriptor if opening has been completed successfully;*
- *-1 otherwise (for example no available space to store a new stream or the stream is already opened)*

Function: ***open_local*** ()

Synopsis: Open a sensor stream.

inputs:

return:

stream descriptor if opening has been completed successfully;
-1 otherwise (for example no available space to store a new stream or the stream is already opened)

Function: ***close*** (*desc*)

Synopsis: Close the selected stream.

inputs:

- *desc*: Descriptor of the stream to be closed.

return:

FAIL if the operation can not be completed;
SUCCESS otherwise.

Function: ***read*** (*desc, nbytes*)

Synopsis: Request that *nbytes* bytes are read from the selected stream.

inputs:

- *desc:* Descriptor of the stream to be read.
- *nbytes*: Number of bytes to be read.

return:

FAIL if the stream is not open or the read can not be performed (for example, attempt to read from a remote stream opened in write mode).
SUCCESS otherwise.

Event: ***readDone****(desc, buf, nbytes, success)*

Synopsis: Signal that the data is ready.

- *desc:* Descriptor of the stream.
- *buf:* Pointer to a region where read data was put; meaningful only if success = SUCCESS.
- *nbytes*: Number of actually read bytes; meaningful only if success = SUCCESS.
- *success:* Whether the read was successful.

Function: ***write*** (*desc, buf, nbytes*)

Synopsis: Write nbytes bytes from buf into the stream identified by *stream_desc*

inputs:

- *desc:* Descriptor of the stream to be written.
- *buf*: Pointer to the buffer to be written.
- *nbytes*: Number of bytes to be written in the stream.

return:

return the number of actually written bytes; -1 if the write fails.

5.3.3 Specification of the functions of the Connectionless component

Function: *send(island_addr, mote_addr, buf, nbytes, reliable, am_type)*

Synopsis: Send a message to the destination mote in a specified island.

inputs:

- *island_addr*: The address of the island destionation of the message.
- *mote_addr*: The address of the mote destination of the message.
- *buf*: Pointer to a region where the data to be sent are stored.
- *nbytes*: Number of bytes to be sent.
- *reliable*: TRUE if an ack is requested for the current message.
- *am_type*: type of the message. It identifies the interface to be used to send the message.

return:

FAIL if it is not possible to send the message, SUCCESS otherwise.

Event: ***ack*** *(success)*

Synopsis: Signal that the acknowledgment of a previously sent message.

inputs:

- *success*: whether the ack was successful.

Event: ***receive** (src_island_addr, src_mote_addr, buf, nbytes, success)*

Synopsis: Signal that the data is ready.

inputs:

- *src_island_addr*: The address of the island destination of the message.
- *src_mote_addr*: The address of the mote destination of the message.
- *buf:* Pointer to a region where the received data is stored; meaningful only if success == SUCCESS.

- *nbytes*: Number of actually received bytes; meaningful only if success == SUCCESS.
- *success:* Whether the receive was successful.

5.3.4 Component Management

Function: ***initCommunications()***

Synopsis: Start both radio and serial interface.

inputs:

Event: ***initCommunicationsDone**(success)*

Synopsis: Signal the completion of both radio and serial component activation.

inputs:

- *success*: Whether the radio and serial activation was successful.

Function: ***joinIsland**(description);*

Synopsis: Request the *basestation* of the current island to be joined. As consequence the mote will receive the *mote_addr* and the *island_addr.*

inputs:

- *description*: Description of the capabilities (transducers or actuators) installed on the mote.

Event: ***moteJoined** (success)*

Synopsis: Signal that a new mote joined the RUBICON system.

inputs:

- *mote_addr:* Identifier of the joined mote.
  description: Description of the capabilities (transducers or actuators) installed on the mote.

5.4 Interface specification for PEIS-WSN gateway

Tuplename / Example	Example data	Description
`devices`	{1,7,11}	Lists the identifier for all detected WSN/WSAN
`devices.$`*ID*`.type`	42	Contains a 4 byte type identifier for each identified device. One tuple for each device detected

devices.7.type		
devices.$*ID*.input.$*CH* *devices.7.input.0*	<binary data>	Writes a binary message on the given channel on the given WSN
devices.$*ID*.output.$*CH* *devices.7.output.0*	<binary data>	Is written by the gateway when a message is sent out from the given WSN at the given channel. Gateway subscribed to this channel iff any PEIS are subscribed to this tuple.

Each island is given a unique number that corresponds to the *peis-id* of the gateway that is connected to the island. For the case of WSN that are connected to multiple gateways this gives multiple paths of addressing such WSN using any of the corresponding gateway ID's.

Unless specified otherwise all binary data is transferred in *network byte order*.

<u>Messages passed from WSN to gateway when intended to be sent to another island</u>

uint32	PEIS ID of the corresponding island to which message is sent
uint16	WSN ID of the target WSN mote
uint16	WSN channel of the target channel on the target WSN
payload (up to 256 bytes)	Payload of message

Corresponding messages passed from gateway to gateway when transmitting a island-to-island message

uint16	WSN ID of the target WSN mote
uint16	WSN channel of the target channel on the target WSN
payload (up to 256 bytes)	Payload of message

6. Acknowledgements

The authors wish to thank Dr. Susanna Pelagatti (UNIPI) for its careful quality assurance review of this deliverable.

7. References

[1] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B.S. Seo, Y.J. Cho. The PEIS-Ecology Project: Visions and Results. In Proceedings of The 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008), pp. 2329-2335, Nice, France, September 2008

[2] Giuseppe Amato, Stefano Chessa, and Claudio Vairo “MaD-WiSe: A Distributed Stream Management System for Wireless Sensor Networks”, Software Practice & Experience, 40 (5): 431-451 (2010).

[3] G.Amato, P.Baronti, and S.Chessa, "Connection-Oriented Communication Protocol in Wireless Sensor Networks," Istituto di Scienza e Tecnologie dell'Informazione - CNR, Tech. Rep. 2005-TR-10, 2005.

[4] B. Karp and H. T. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," in Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiConm 2000), Boston, MA, USA, August 2000, pp. 243-254.

[5] A. Caruso, S. Chessa, S. De, and A. Urpi, "GPS Free Coordinate Assignment and Routing in Wireless Sensor Networks," in Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2005), Miami, FL, USA, March 2005, pp. 150-160.

[6] EU FP6 ”SMEPP” project, http://www.smepp.org/

[7] http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html

[8] http://www.tinyos.net/

[9] J. Rashid, M. Broxvall, A. Saffiotti. Digital Representation of Everyday Objects in a Robot Ecology via Proxies. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2008)*, pp. 1908-1914. Nice, France, September 2008.

[10] http://www.aass.oru.se/~peis