RUBICON
RUBICON Robotic UBiquitous COgnitive Network
Project No.: 269914

SEVENTH FRAMEWORK PROGRAMME

D1.3.2 – Final Version of the Communication Layer

Editor:	**Mauro Dragone**	**UCD**
Contributor(s):	**Claudio Gennaro**	**CNR**
	Claudio Vairo	**CNR**
	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)

Issue Date	19/05/2013
Deliverable Number	D1.3.2
WP	WP 1 - Communication Layer
Status	□ Draft □ Working ☒ Released □ Delivered to EC □ Approved by EC

Document history

V	Date	Author	Description
0.1	13/02/2012	Mathias Broxvall	Creation of LaTeX template
0.2	11/03/2013	Claudio Gennaro	First draft
0.3	17/04/2013	Claudio Gennaro	Second draft, data logger and rssi
0.4	18/04/2013	Mauro Dragone	Web-based Testbed Utilities
0.5	19/04/2013	Mauro Dragone	Conclusions
0.6	30/05/2013	Stefano Chessa	Internal Review
1.0	7/05/2013	Kylie O'Brien	QA
1.0	15/05/2013	Mauro Dragone	Final Version

Disclamer 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 deliverable (D1.3.2) takes place at the end of Task 1.3 *Final version of Communication Layer* at M24 of RUBICON WP1. The goal of the RUBICON Communication Layer (CML) is to provide communication and integration mechanisms built upon middleware for WSN/WSANs and robotic ecologies.

Together with D1.2 and D1.4, this deliverable presents a set of requirements and specifications supported by the final version of the software, as delivered at M24.

In addition to this report with an appendix documenting the CML API, the main part of the deliverable consists of the published software, available on the RUBICON code repository and later to be released on the project webpage.

This report furthermore summarizes, in brief, the state and the achievement of all tasks scheduled for M13-M24 of RUBICON WP1.

Contents

Figures

Abbreviations

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

Chapter 1

Introduction

1.1 Target audience

This report is intended for the project consortium as well as members of the public interested in using the provided software to deploy a RUBICON ecology and/or develop application specific software for such an ecology.

1.2 Overview

The CML provides communication and integration mechanisms built upon middleware for WSN/WSANs and robotic ecologies. Specifically, the CML can be used to: (i) support inter-component and inter-layer interaction, (ii) read data gathered from sensors installed in the environment, such as switch sensors signaling when drawers/doors are open or closed, occupancy sensors signaling when users or robots move in certain areas, as well as the output of off-the-shelf components (e.g. used to recognize sounds or generate other events extracted from raw sensor data), (iii) send instructions to effectors, such as lights, blinds, door locks and appliances, (iv) sense the status of these effectors and know when the user interacts with them (i.e.. when she manually switches on/off lights, TV, etc), (v) publish the radio signal strength index (RSSI) measured between communicating WSN nodes, and (vi) recognize when new sensors are added or existing ones are removed (and notify these events to all the other layers of the RUBICON).

The CML consists of TinyOS[1]-based components suitable for programmers of application specific motes for the RUBICON ecology as well as a unix library implementing the RUBICON communications for PC and Robot based application programmers called the PEIS-kernel. In addition to the interfaces for programmers, the communication layer also contains a set of component softwares that are used for the deployment of a RUBICON ecology in order to ensure that all participating devices can collaborate successfully through the communication layer.

[1] see `www.tinyos.net`

1.3 Overview of this document

The remaining chapters of this deliverable describe the main components in the final version of the CML. Specifically, Chapter 2 discusses the set of new requirements addressed by this new version of the CML. Chapter 3 provides an overview of the new design of the CML, while the remaining chapters describe specific tools that have been fitted in the final version of the CML, namely, (i) data logging (in Chapter 4), (ii) radio signal strength index (RSSI) collection (in Chapter 5), (ii) and Web-based access and programming for WSAN testbed (in Chapter 6). Chapter 7 describes how to access, install and configure the software described in this deliverable. Chapter 8 summarize the achievements of RUBICON WP1 and, in particular, of the activities carried out in its second year. Finally, an appendix serves as a detailed reference to the final CML API.

Chapter 2

Requirements for the Final Version of the Communication Layer

In this chapter, we summarize the main requirements for the final version of the Communication Layer. For each requirement we briefly outline the activities undertaken to define the requirements and the problems that the requirements address.

2.1 Flexibility and Reusability [R1v2.FLEXIBILITY]

One of the main requirements of the final version of the Communication Layer is to increase its flexibility, reusability and portability. While the current prototype successfully integrates WSANs inside the RUBICON system, its implementation is strictly dependent on TinyOS and requires a number of interventions by the Communication Layer for every modification necessary to the higher layers of the RUBICON system. Such a model has the disadvantage of not allowing us to decouple the Communication Layer from the application layers (e.g. the Learning, Control and Cognitive layers in RUBICON). The lack of decoupling forces developers to request changes to Communication Layer whenever they need to add a new command or message to the application layer.

2.2 Messages Concurrency Safety [R1v2.MSGSAFETY]

Another limitation of TinyOS is that if the radio is busy transmitting or receiving when a packet transmission is requested, a failure will be raised back to the application component that had issued the communication request. In the first prototype version of the Communication Layer, this problem was left to the applications to handle. Specifically, components in the application layer (including, for instance, the Java-based implementation of the Learning Layer as well as C-based Peis components used as part of the Control Layer) had to store a copy of the message and manage timers to re-try failed transmissions. In order to simplify application development, and provide a unique solution to this problem, the final version of the Communication Layer should include an explicit mechanism to handle packet-retransmission(s).

2.3 Message Reliability [R1v2.RELIABILITY]

The first prototype Communication Layer only supported unreliable data transmission, that is, it was not possible to ask the Communication Layer to confirm the receipt of messages. However, both connectionless communication between components, e.g. as required by control instructions sent by the Control Layer to activate WSAN's actuators, and synaptic channels used by the Learning Layer, need the ability to reliably transmit data via the Communication Layer.

2.4 Multi Island Support [R1v2.MULTIISLAND]

The first prototype Communication Layer only supported single-hop WSAN based on multiple mote-class devices that had to be in communication range with the same sink node. The prototype version only supports a single sink node, which had to be connected to a gateway component to provide a link between the WSAN and the RUBICON Peis tuplespace, as detailed in Deliverable D1.3.1. However, a single sink acted as a bottleneck in the system, since all the motes share the same communication media. In addition, such a solution could not support large deployments, as requested, for instance, by large houses and hospitals, where WSAN must cover areas that could not be covered by a single-hop network. Rather than implementing a multi-hop communication architecture, which is a topic already largely confronted in WSN/WSAN research, we have decided to complement these type of solutions by supporting the creation of multi-island systems, in order to fit the peer-to-peer characteristics of a robotic ecology. Specifically, we achieve a multi-island topology by using multiple sinks connected in a peer-to-peer network, and by letting each sink supervise its own (single-hop / multi-hop) WSAN (the *island*).

Such an multi-island topology allows the RUBICON ecology to easily scale to large WSAN deployments. However, there are issues that need to be considered. Here is a list of requirements that must be met by the multi-island topology implementation.

2.4.1 Dynamic and Independent Island Address [R1v2.MI_ADDRESS]

In order to simplify the configuration and the maintenance of multi-island systems, the address of each mote in the system needs to be dynamic, i.e. it must be set at run time and not at compilation time. Furthermore, RUBICON needs a solution to address collisions so that adding islands to an existing system can be achieved by ignoring the addresses that are already in use in the existing islands.

2.4.2 No Island Interferences [R1v2.MI_NOINTERFERENCE]

Another problem that must addressed in the implementation of the multi island topology, is the possible overlap between distinct islands. A mote may be in communication range with two sinks of two (or more) islands thus potentially creating interferences to normal communication in each one of them. One possible solution is to take advantage of the multiple radio channels of the underlying MAC protocol so that each island uses a different radio channel. However, the maximum number of channels supported by those networks would pose a limit to the actual scalability of the resulting systems. For this reason it is necessary to implement some mechanism to distinguish radio messages from other islands.

2.4.3 Safe Island Join [R1v2.MI_SAFEJOIN]

Finally, whenever a mote is added to an island it must join only one island even if there are more sink nodes in its transmission range.

2.5 Nonfunctional Requirements

2.5.1 Java Support for PCs [R1v2.JAVAAPI]

In order to support the portability of the higher layers of the RUBICON, e.g. to provide PC-based implementations of the modules of the Learning Layer, the Communication Layer should export a Java API to enable those PC-based implementation to seamlessly interact among each other and with TinyOS-based parts of the RUBICON. This API would open the way for the development of RUBICON applications running on devices that are more computationally capable than the current mote class devices considered by the prototype system.

2.5.2 Sensor Data Logging [R1v2.DATALOG]

The WP5 activities, as discussed in D5.1, require a specific data logging tool to support WSAN data collection in the application testbeds. In particular, in D5.1 we have outlined how it is important to i) have a central monitoring station to monitor each experiment while it is being performed, and ii) rely on flash memory installed on each mote to collect the data rather than rely on the network to transmit the data to a central server, in order to avoid communication problems, such as congestion, packet loss, etc.

Furthermore, the logging tool should provide functionalities for retrieving the data stored on each mote, store the data into a database, and visualize it by means of a Graphical User Interface (GUI).

2.5.3 Received Signal Strength Indicator Support [R1v2.RSSI]

During the forwarding phase of the Learning Layer (see D2.1 and D2.2), Received Signal Strength Indicator (RSSI) data must be fed into a synaptic channel exactly as with other sensed data from standard transducers. A tool for instructing a set of mote to send radio beacons (using a different radio channel different from the one used for communicating) and a tool for collecting and preparing the RSSI data is thus required.

2.5.4 Memory Usage Optimization [R1v2.MEMORYOPT]

According to the OSI communication layered model, the data that must passed down from a layer to a lower layer must follow an encapsulation process. Headers and Trailers of data at each layer are the two basic forms used to carry information. Headers are prepared to data that has been passed down from upper layer. Trailers are appended to data that has been passed down from upper layers. Each layer may add a Header and a Trailer to its Data, which consists of the upper layer's Header.

Unfortunately, such a model implies a memory copy to exchange data between one level and another. This produces a severe waste of memory, which is not acceptable on mote-class computationally constrained devices. The Communication Layer addresses this problem in its internal design.

2.5.5 Re-programming and Testbed facilities [R1v2.TESTBED]

In order to facilitate experimentation with the Communication Layer within the research laboratories in the RUBICON consortium, and to support the testing and the tuning of its features in the final applicative scenarios during the 3rd year of the project, we will need mechanisms to easily re-program a remote WSN, possibly over the Internet. Specific utilities should be implemented to enable the sharing of sensing and acting infrastructures among different research institutes, and speed up normally costly maintenance operations, such as the update or the debugging of the WSN embedded software.

Chapter 3

Development of the Second Version of the Communication Layer

3.1 Introduction

This chapter is devoted to explaining how the functional and nonfunctional requirements have been addressed in the final version of the Communication Layer.

3.2 Flexibility and Reusability [R1v2.FLEXIBILITY]

With the first prototype version of the Communication Layer, each time we wanted to create a new type of message to be sent by means of the connectionless interface (for example a new command to an actuator), according to the TinyOS AM model, we had to create a new AM type. This required modification and recompilation of the Communication Layer. In this section, we describe how this issue has been addressed by separating the packet formatting from the communication logic layer.

The component-oriented design of TinyOS gives developers the ability to easily compose different applications. However, this also entails a disadvantage associated with its use of the Active Messages (AM) system - the TinyOS 2.x message buffer abstraction that allows datagrams to be passed between different link layers[1]. An AM contains the name of an application-level handler to be invoked on a target node, and the payload. The receiving node dispatches the message using events to one or more handlers that are registered for this message. The AM model works like a distributed event model, by allowing nodes to send events to other nodes and to specific components wherein. Such a model has the disadvantage of not allowing us to decouple the Communication Layer from the application layers (e.g. the Learning, Control and Cognitive layers in RUBICON). Each time we want to create a new type of message or modify an existing one (for example a control command to an actuator or a learning instruction), according to the AM model we have to create or modify an AM type, and therefore to some extent change the Communication Layer. Separating packet formatting layers from communication logic layers would

[1] see http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html

increase the modularity of the Communication Layer. In particular, applications would not have to know the details of the TinyOS APIs, thus more easily supporting further extension of the software and also facilitating its porting to other languages or operating systems.

In Figures 3.1 and 3.2, we depict a schematic comparison of the *send* and *receive* operations of the first and final version of the Communication Layer using the Connectionless component. Suppose the Control Layer wants to issue a command to a remote mote equipped with actuator capable of opening a door. As mentioned above, with the first release of the Communication Layer, we needed to create a new TinyOS AM type, with the disadvantage of bringing out in the application level some details of the TinyOS operating system. This aspect makes difficult the realization of a possible version of Communication Layer that uses different systems of communication and languages (eg, Java over TCP/IP).

With the final version of the Communication Layer, instead, the application (in this example the Control Layer) defines its own new message, i.e. any sequence of bits that the lower levels will ignore. This allows us to exploit the well-established model of communication called *data encapsulation*, in which a level does not know the details of the information that the upper level wants to send, but the information content becomes a simple payload to be encapsulated and sent to the lower levels along with the necessary information that the level abstraction implements. The opposite path, when the message is received, is characterized by the reverse procedure called *data decapsulation*.

Figure 3.2 shows this process. The message in the example, which we have called `OPEN_DOOR` is sent to the Connectionless component that is responsible for sending messages to individual remote devices. This message is encapsulated in a message of AM type *application* that is accompanied by a tag *APPID = CL*, which is to say that the message was generated by an instance of a Control Layer. The message that is sent through the internal interface of the transport layer, called *Message Dispatcher* that encapsulates it in a new transport message with the tag *TRANSID = CLESS*, which is to say that the message was generated by the Connectionless component. The Message Dispatcher sends the message to the *Network layer*, which is in charge of creating an active message of type *AMType = Network* by inserting the transport message in the payload of the network message. Note that it is only at this point that TinyOS active message is created, so that if in the future we want to replace the network level with a new network to another type of media, we would need to change only this bottom part of the system, without modifications to the core Communication Layer.

When the message has been delivered to its intended recipient, it goes upwards to the application level. At each step, the message is subjected to a decapsulation process to get back the original message. All tags APPID, TRANSID, and AMType are exploited to make the internal routing in the receiving node to bring the message to the right application process that must receive it (similar to what happens with TCP/IP ports).

3.2.1 Memory Usage Optimization [R1v2.MEMORYOPT]

As explained above in the general communication layered model, the data that must passed down from a layer to a lower layer must follow the encapsulation process. This implies a memory copy to exchange data between one level and another and produces a severe waste of memory, which is not acceptable on mote-class computationally constrained devices. This problem is solved by using the message structure presented in Section 7.3.6.

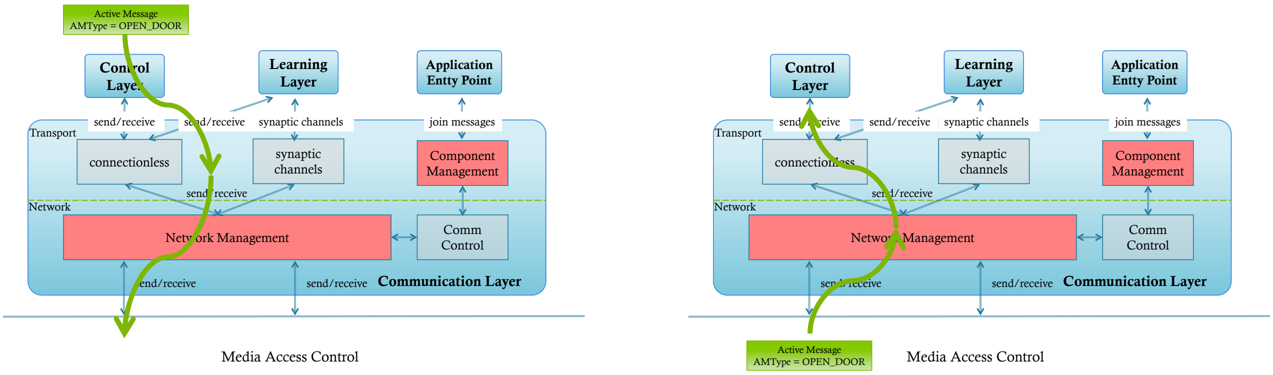


Figure 3.1: The *send* and *receive* operations of the Connectionless component of the first version of the Communication Layer.

3.3 Messages Concurrency Safety [R1v2.MSGSAFETY]

To solve the problem of buffering multiple, concurrent messages in the Communication Layer, we implemented separate message buffers structured as static circular queues for the Network messages. In the first prototype of the Communication Layer, if the radio was busy when the application had to transmit a message, the application had manage a timer and try later to send messages once the radio becomes available again. With the circular buffer, this management is no longer required. A similar buffer is used in reception to temporarily queue all messages received from the network and send them to the application layers if they are busy.

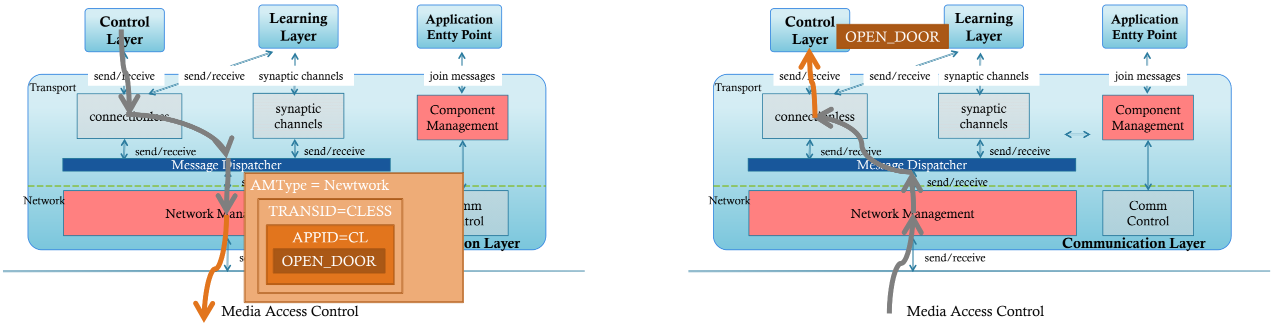


Figure 3.2: The send and receive operations of the Connectionless component of the final version of the Communication Layer.

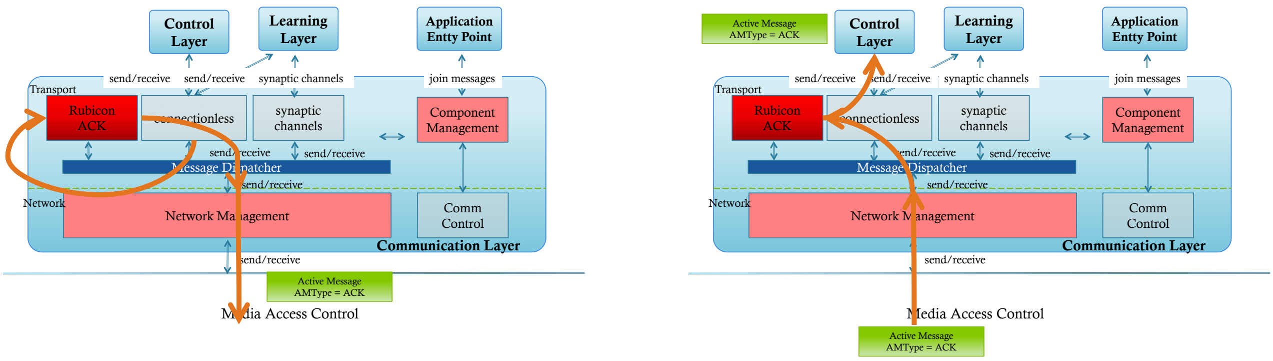


Figure 3.3: The transmission of acknowledgement messages by the Ack component introduced in the final version of the Communication Layer.

3.4 Message Reliability [R1v2.RELIABILITY]

The message reliability had been implemented by building on top of the connectionless communication mechanisms already available in the first version of the Communication Layer. Specifically, the Communication Layer API has been extended, by allowing client components to specify if they need to receive a confirmation of receipt from the intended recipient of the messages they send. In addition, confirmation messages are different in nature from the normal communication messages as they must be processed by the transport layer that generates them automatically and notify them to the application. If they were generated as normal messages, they should be created and managed by the application as the lower levels do not have permission to read the contents of the messages. For this reason we have created a specific component, called *Ack*, which deals with the generation and the management of the exchange of acknowledgement messages in a dedicated protocol.

Figure 3.3 shows how the Ack component works. When a sender component (in this case the connectionless) requires an acknowledgement, the receiver component (shown in Figure 3.3, left panel) automatically generates a message with *AMtype* = *ACK* and sends it through the network. When the sender receives the *ack* (shown in Figure 3.3, right panel) it notifies the application that requested it.

3.5 Multi Island Support [R1v2.MULTIISLAND]

Multi Island Support fits the requirements for the Communication Layer for supporting a growing number of devices without degradation of performance in communications. The addressing scheme used in the multi island RUBICON system ([R1v2.MI_ADDRESS], [R1v2.MI_NOINTERFENCE]) is described in detail in Deliverable D1.2, while details concerning the use of the Peis middleware to create the peer-to-peer skeleton of the multi-island topology are discussed in Deliverable D1.4.

3.5.1 Java Support for PCs [R1v2.JAVAAPI]

The Java implementation of the Communication Layer is presented in Chapter 7.

Chapter 4

The Data Logger [R1v2.DATALOG]

4.1 INTRODUCTION

In this chapter, we present the tool for the real-time acquisition of data from a set of mote sensors and for the storage of the acquired data in the flash memory of the sensors that was required for Rubicon [R1v2.DATALOG]. The tool provides functionalities to retrieve the stored data from the sink node, to store the data into a PC-based database and to visualize it by means of a graphical user interface that runs on a linux-based PC. The tool that we propose consists of both the software that runs on the PC and the software to be installed on each mote in the RUBICON WSAN.

The tool described in this chapter allows experimenters and developers to remotely control a set of notes for both real-time and off-line data acquisition. The tool is compatible with the IRIS, MicaZ and TelosB platforms. With this tool, the user can request a series of measurements from the sensors installed on each mote in the WSAN under study (e.g. light, temperature, accelerometer, humidity, passive infrared (PIR), magnetic, and microphones). The data collected by the motes can be subsequently forwarded to the sink, which can store them in a local database, enabling the user for off-line analysis and processing.

4.2 Features

The system we have developed has the following features:

- **Topology** - Figure 4.1 shows the network architecture of our WSAN. Each mote in the WSAN acquires the requested data and send data updates to the sink mote either in a real-time fashion, or on-demand when requested. The sink interfaces the WSAN with the user: it routes the data received from the other motes to the base-station (the PC) where the data is stored in a local database and visualized to the user by means of a Graphical User Interface (GUI) component, which is also included in the logger tool.
- **Mote Management and Setting** - The number of motes involved in the query can change over time, and each of them can be managed independently from the others. The tool allows operators to activate, deactivate and configure the logging operations to be performed by specific motes, without affecting the oper-

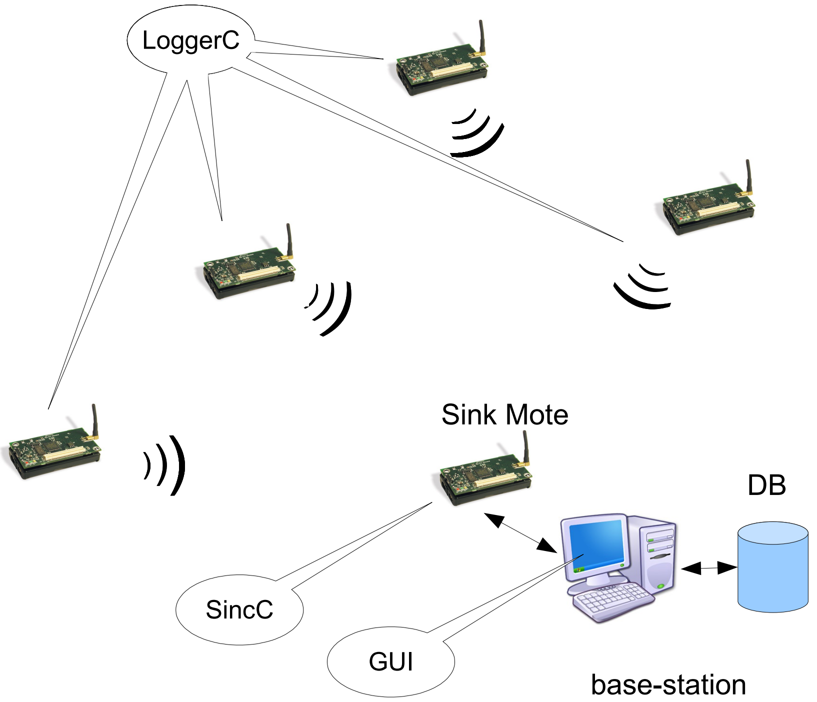


Figure 4.1: The topology of the data logger.

ation of the other motes in the network. At the same time, the tool allows the operator to activate, deactivate or reset all motes on the network simultaneously. The main parameters that can be set for each mote are the following:

- Sampling period. It indicates the period of time (expressed in milliseconds) between two consecutive readings of the transducers.
- Type of transducer. It allows the user to choose which transducers of the sensor board the mote has to sample.

- **On-line and Off-Line Data Collection** - The monitor application on the motes can be executed in two modalities: *on-line* and *off-line*. The on-line mode provides the data collected in real-time for being immediately stored in the central database when they are received by the base-station. In the off-line mode, on the other hand, the data is temporarily stored within the mote's flash memory and it can be downloaded at a later time, upon a specific request from the operator.

- **Data Storage** - The data sampled by transducers mounted on motes is stored in a central database located on the base-station, allowing the operator to access the data and analyze it in off-line mode. Notably, in most WSAN application scenarios, including those tackled by RUBICON, the amount of acquired data can be very large, so this feature is particularly useful.

- **Graphical User Interface** - The GUI allows the user to remotely configure all the needed parameters on the motes and to activate/deactivate all the functionalities provided by the logger. Moreover, the GUI visualizes the data received while it is being stored in the database, so a first quick analysis of the data can be executed without having to access the database. This is useful in particular for the real-time operation of the tool.

4.3 Architecture

The Data Logger consists of four separate software components (see Figure 4.1):

- **SincC** is the gateway between the WSN and the user that runs on the sink mote.
- **LoggerC** is the component designated to collect and store the data sampled by each mote.
- **GUI** is the user interface that runs on the base-station.
- **DB** is the database for the storage of the data collected that runs on the base-station.

4.3.1 SincC

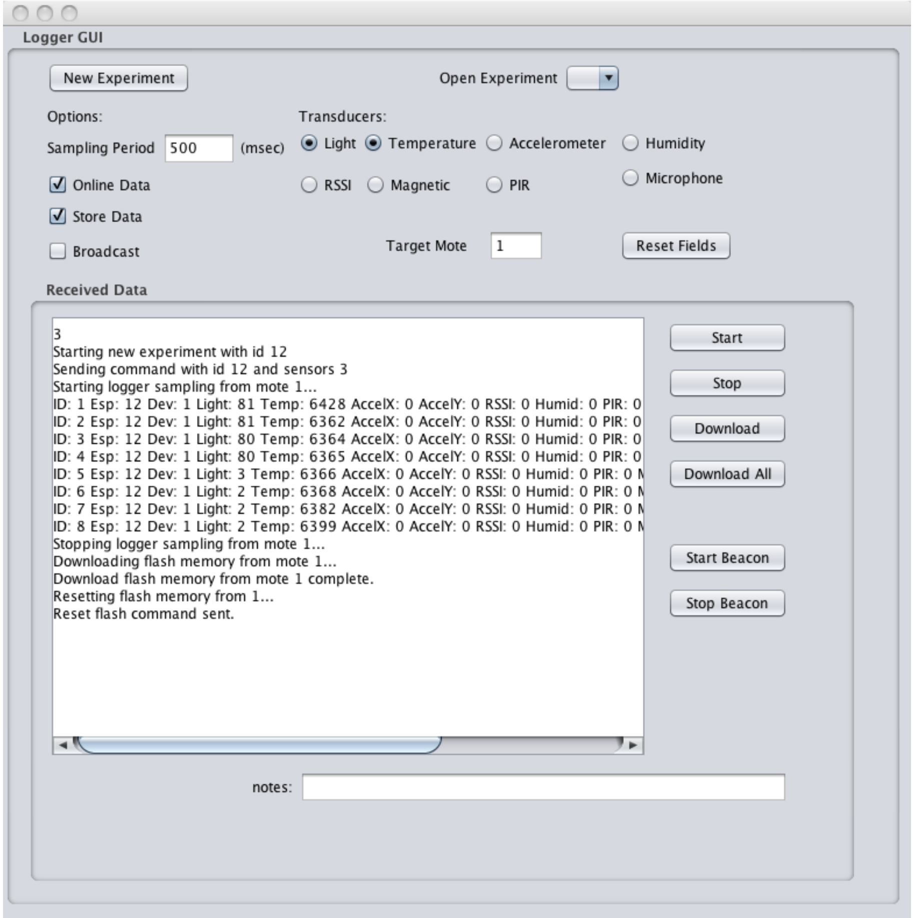
The sink mote forwards the commands received from the base-station to the other motes of the network through the wireless communication channel. It also routes all the data acquired by the motes in the network towards the base-station to which it is connected through a serial channel. The nesC component, that implements such functionality similar to a hub, is called *SincC* and it has to be installed on the sink mote.

4.3.2 LoggerC

The rest of the motes in the network have the same nesC application installed on board called *LoggerC*. Their typical function is to read the values from the transducers integrated on their sensorboard, store the acquired data in their internal flash memory and/or send them to the sink.

4.3.3 GUI

Through the GUI, the operator is able to remotely instruct the LoggerC components to start/stop the sampling, to download the data stored in the flash memory of the corresponding mote and to erase the memory after the download has been completed. The GUI also allows the operator to specify the parameters needed for the monitoring task: sampling period at which the data have to be acquired, transducers to sample, acquisition modality (on-line and/or off-line), the ID of the target mote or, in case of storing in the flash memory and erasing the flash memory content, to send the command in broadcast to all the motes. The sink connected to the base-station redirects the command specified in the GUI to the target mote. In case of on-line mode operation, the acquired data are immediately sent back to the base-station and they are displayed on the GUI and stored in the DB. In case of off-line mode operation, instead, the data is only stored in the flash memory of the mote. In order to access these data, a specific download command is sent and, once the data is received, it is stored in the DB. In Figure 4.2, a screenshot



Sampling Period 500 (msec)

☑ Online Data

☑ Store Data

□ Broadcast

● Light ● Temperature ○ Accelerometer ○ Humidity

○ RSSI ○ Magnetic ○ PIR ○ Microphone

Target Mote 1

```
3
Starting new experiment with id 12
Sending command with id 12 and sensors 3
Starting logger sampling from mote 1...
ID: 1 Esp: 12 Dev: 1 Light: 81 Temp: 6428 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0
ID: 2 Esp: 12 Dev: 1 Light: 81 Temp: 6362 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0
ID: 3 Esp: 12 Dev: 1 Light: 80 Temp: 6364 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0
ID: 4 Esp: 12 Dev: 1 Light: 80 Temp: 6365 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0
ID: 5 Esp: 12 Dev: 1 Light: 3 Temp: 6366 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0 M
ID: 6 Esp: 12 Dev: 1 Light: 2 Temp: 6368 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0 M
ID: 7 Esp: 12 Dev: 1 Light: 2 Temp: 6382 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0 M
ID: 8 Esp: 12 Dev: 1 Light: 2 Temp: 6399 AccelX: 0 AccelY: 0 RSSI: 0 Humid: 0 PIR: 0 M
Stopping logger sampling from mote 1...
Downloading flash memory from mote 1...
Download flash memory from mote 1 complete.
Resetting flash memory from 1...
Reset flash command sent.
```

Figure 4.2: Screenshot of the data logger GUI and data sampling

of the GUI of the data logger is provided. It shows the sampling of data from a mote and the downloading of the information to the database.

4.3.4 DB

The DB component is an SQL relational database. In order to facilitate offline analysis of the data collected with the aid of the Logger, its database stores both meta-information describing each experiments (e.g. reporting how the WSAN and the logging operations were configured at the time the experiment was performed, in the *Experiments* entity), and the data recorded by the sensors (in the *DataSample* entity). Figure 4.3 shows the entity-relationship schema of the Data Logger database. Each experiment is associated with an ID, called *ExpID*. Therefore we have a tuple for each experiment in the Experiments table and the association that links the two tables is provided by ExpID, which is a foreign key in the DataSample table.

Here is the list of attributes of tables with related information. Experiments table:

- *ExpID*. Unique identifier of the experiment (primary key).

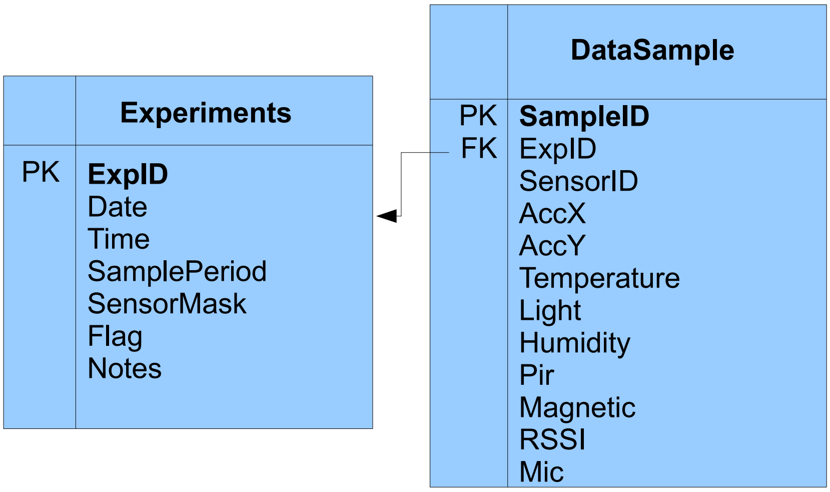


Figure 4.3: The Entity-Relationship diagram of the database.

- *Date*. Starting date of the experiment.
- *Time*. Starting time of the experiment.
- *SamplePeriod*. Sample Period of the experiment.
- *SensorMask*. A bit-mask used representing which transducers were active during the experiment.
- *Flag*. Indicates the choice of making an on-line/off-line communication.
- *Notes*. Contains any notes entered by the user when creating the experiment.

SampleDate table:

- *SampleID*. Unique identifier of the sample data (primary key).
- *ExpID*. Unique identifier of the experiment for which the sample belongs to (foreign key).
- *SensorID*. Unique identifier of the Mote in the network.
- *AccX*. Sampled value from the x-axis accelerometer.
- *AccY*. Sampled value from the y-axis accelerometer.
- *Temperature*. Sampled value from the thermometer transducer.
- *Light*. Sampled value from the light transducer.
- *Humidity*. Sampled value from the humidity transducer.
- *Pir*. Sampled value from the Pir transducer.

- *Magnetic*. Sampled value from the Magnetic transducer.
- *RSSI*. RSSI value of the communication between the sink and the target Mote.
- *Mic*. Sampled value from the microphone.

4.4 Implementation and Usage

4.4.1 Setting Up an Experiment

Here we describe the use of the GUI of the data logger shown in Figure 4.2.

- **New Experiment** button. This button is used to create a new entry in the *experiments* table in the DB.
- **Sampling Period** text box. It allows the user to setup up the sampling period of the transducers readings (expressed in msec).
- **Transducers** radio buttons. Eight radio buttons that allow the user to select which transducers must be active on the motes during the experiment.
- **Online Data** check button. If selected, the data are sent to the sink in real-time when they are acquired by the target mote.
- **Store Info**. If selected, the acquired data are stored in the flash memory of the target mote (these two last options can be jointly checked).
- **Target mote** text box. It specifies the ID of the mote from which to acquire data. Note that, in order to avoid collisions, only one mote is allowed if the *Online Data* modality is set, that is the real-time data can be received from one mote at a time.
- **Broadcast** checkbox. This option is available only for the *Store Info* mode and to erase the flash memory of the motes. By checking it, the storing of data in the flash memory, or its deletion, can be requested on all the motes in the network at the same time.
- **Note** textbox. It allows the operator to associate the experiment entry with a textual comment.
- Command buttons **Start**, **Stop**, **Download**, and **Reset Flash**. Once a new experiment entry has been created, the sampling task is activated by clicking on the *Start* button. As consequence, the mote specified in the *Target Mote* text-field will start sending its acquired data and they will be visualized in the GUI. If the *Store Info* modality is set, the blue LED toggling on the motes notifies the writing of the acquired data into the flash memory. The *Stop* button terminates the sampling task. This command can also be sent in broadcast mode by checking the *Broadcast* checkbox. In order to download the data stored in each mote into the database, the button *Download* is used. In order to avoid collisions, this operation is allowed for only one mote at a time. The green LED toggling on the target mote notifies the reading from the flash. The received data is stored in the *DataSample* table of the database in the same order as they have been acquired by each mote and the data is stored with the reference to the experiment in the table *Experiments* that contains

the meta-data describing the experiment, including the exact time at which the sampling started and the sampling frequency. The *Reset* button erases the flash memory of the mote. This operation may take a long time and its progress is signaled by the red LED on the mote performing it. This command can also be sent in broadcast.

4.5 Accessing the Software

Besides the RUBICON software repository, the software described in this section is freely downloadable at `www.nmis.isti.cnr.it/gennaro/logger.zip` and runs on Linux where Postgres DBMS (`www.postgresql.org`) is installed.

Chapter 5

RSSI Support [R1v2.RSSI]

In many scenarios of RUBICON, the Learning, Cognitive and Control layers make use of user and robot localization information that, in some cases, is produced by means of Received Signal Strength Indicators (RSSI). The RSSI is produced by exchanging beacon packets among a set of fixed sensors (called anchors) and a mobile sensor deployed on the user or on the robot. The mobile sensor measures the signal strength associated with the beacon packets received by the anchors and gives this data in input to the Learning Layer (normally to an instance of the Learning Layer installed on the mobile sensor itself) that is trained to output a location information in terms of a pair of coordinates $< x, y >$.

In practice, since localization is mostly used to track a mobile target, the beacons should be emitted by the anchors at a given, constant rate, so that for each complete set of received beacons the Learning Layer can produce the localization information. From the point of view of the RUBICON Learning Layer, each value of RSSI is equivalent to a data read by a sensor from any of its transducers (i.e., the sensor has a transducer for measuring RSSIs), hence the Learning Layer can be fed with this information by making use of synaptic connections and synaptic channels. On the other hand, from the point of view of the Communication Layer, each RSSI measurement is produced by combining the action of two sensors: the anchors that emit the beacon and the mobile sensor that receives the beacon and produces the RSSI. This fact is however hidden to the upper layers that are aware only of the RSSI measurements as if they were transducers readings. A second aspect is that, in order to produce a meaningful value of localization, several RSSI measurements referring to different anchors should be collected all together in a single tuple and encapsulated in a synaptic channel to be given in input to the Learning Layer.

Based on these requirements, we extended the Communication Layer in order to enable such a behavior. In this implementation, a set of sensors act together as a single sensor measuring RSSI tuples. We decided to implement the beacons by means of sensors that run a very simple protocol, in which the anchor nodes send their beacon following a turn; this is implemented by using a virtual token that tours among the anchors. Furthermore, the anchors make use of timeouts to avoid that a node failure stops the entire protocol. Since the emission of beacons may easily saturate the wireless channel thus preventing the other communications of RUBICON, the beacons operate on a separate channel (channel A) than the one used for the normal operations of RUBICON (channel R).

The mobile node switches between the channels used to received the beacons and the channel used for RUBICON communications. It remains for a period in receive mode on the channel A and in this period it accumulates the measurements of RSSI in a tuple. At the end of this period it switches to channel R and sends the RSSI tuple through a synaptic connection to the Learning Layer, then it switches back to channel A to start collecting again the RSSI values for the next period.

From the point of view of the interface with the above layers, there are no differences with respect to the use of normal, built-in transducers of the sensor.

Chapter 6

Web-based Testbed Utilities [R1v2.TESTBED]

6.1 Introduction

The Communication Layer jointly presented in this and in the other WP1 deliverables released at M24 (D1.2 and D1.4) includes a number of features aimed at supporting important application requirements, such as scalability, computational/bandwidth constraints, and extensibility. We have also implemented a set of Web-based testbed utilities. These utilities allow any developer to easily re-program a remote WSN, over the Internet. Together with the data logging feature described in the previous section, these utilities enable the sharing of sensing and acting infrastructures among different research institutes, and speed up normally costly maintenance operations, such as the update or the debugging of the WSN embedded software.

Unlike traditional computers, reprogramming the processors in a WSN is more challenging because of the limited power, processing, storage and communication capabilities. Once the WSN motes are installed in a testbed, possibly wired to external sensors and actuators (e.g. reed switches and relays), it becomes infeasible to remove them every time there is a need to update their software, for instance, before testing or tuning new features, to upload a new sensor processing component, or to install new versions of the RUBICON Communication and/or Learning layers. Deluge [3], the standard reprogramming facility included in the TinyOS embedded operating system used in RUBICON, allows developers to install new programs on each mote, by transmitting their full image over the network. However, this requires the base-station to transmit the new program with Deluge, for every node in the network. This consumes significant communication, processing and storage resources, since a big chunk of the memory of each processor must be used to host a copy of the Deluge nesC program. Alternative solutions, based on modularization of the code, such as SOS [2] and LiteOS [1], provide solutions for operational WSNs but do not substantially change these overheads.

For these reasons, and in order to focus our efforts on the development and the testing of our embedded solutions, our test-bed utilities leverages ”wired” WSNs. ”Wired” WSNs are WSNs where each mote is powered and interfaced through a USB connection. While such an organization cannot be used in a final application, since it obviously negates the basic advantages of having wireless motes, it suits experimental test-beds, where it is more important to have a fast and easy way to re-program the WSN while limiting the down-time involved in those maintenance operations. The biggest advantages of a wired test-bed are that (i) each mote can be easily re-programmed by simply and rapidly flashing a new program image through its USB connection, and (ii) each mote is powered

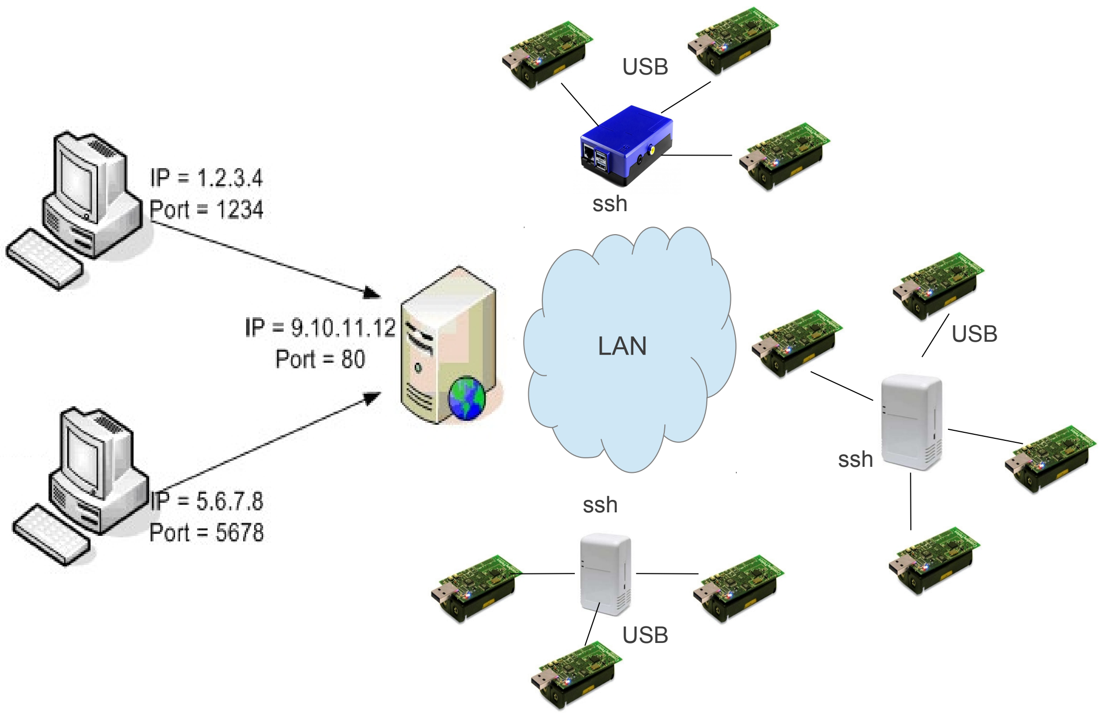


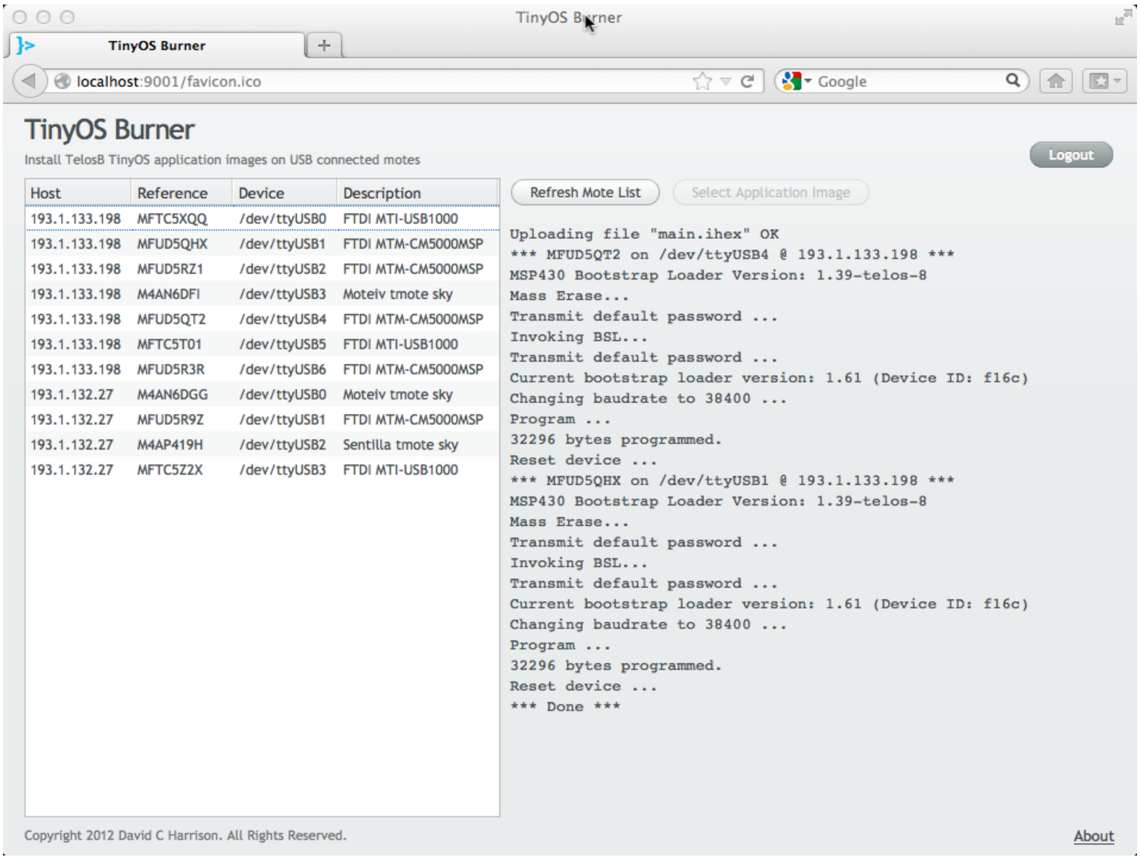
Figure 6.1: Architecture of the Web-based testbed.

through the same USB connection, so that experimenters do not need to worry about having to change the batteries in what are normally battery-operated devices. This reduces the downtime and the cost of the maintenance of the test-bed while developers can still use it to assess the energy efficiency of their solutions, for instance, by using simulation-based models of power consumptions, or by availing of dedicated hardware for power monitoring.

6.2 Design

Figure 6.1 illustrate the organization of a test-bed availing of the facilities described in this chapter. We adopt a cluster-based architecture whereas groups of motes that are physically close to each other (e.g. an island in the multi-island organization of the RUBICON system) are wired (via USB and possibly availing of USB extensions and powered USB hubs) to mini-pcs hosting both the gateway-side re-programming software of our test-bed and application-specific software (e.g. the RUBICON Proxy component described in Deliverable D1.4). Each mini-pc is connected to the same LAN, either via wired or wireless network. A server hosts a Web-server interface to the test-bed. Specifically, the Web-server interface can be used over the Internet by remote operators to:

- list all the motes available in the test-bed connected with the Web-based interface. This is achieved by querying each of the mini-pc in the test-bed (via ssh, using the *motelist* TinyOS command to retrieve the list



TinyOS Burner

Install TelosB TinyOS application images on USB connected motes

Host	Reference	Device	Description
193.1.133.198	MFTC5XQQ	/dev/ttyUSB0	FTDI MTI-USB1000
193.1.133.198	MFUD5QHX	/dev/ttyUSB1	FTDI MTM-CM5000MSP
193.1.133.198	MFUD5RZ1	/dev/ttyUSB2	FTDI MTM-CM5000MSP
193.1.133.198	M4AN6DFI	/dev/ttyUSB3	Moteiv tmote sky
193.1.133.198	MFUD5QT2	/dev/ttyUSB4	FTDI MTM-CM5000MSP
193.1.133.198	MFTC5T01	/dev/ttyUSB5	FTDI MTI-USB1000
193.1.133.198	MFUD5R3R	/dev/ttyUSB6	FTDI MTM-CM5000MSP
193.1.132.27	M4AN6DGG	/dev/ttyUSB0	Moteiv tmote sky
193.1.132.27	MFUD5R9Z	/dev/ttyUSB1	FTDI MTM-CM5000MSP
193.1.132.27	M4AP419H	/dev/ttyUSB2	Sentilla tmote sky
193.1.132.27	MFTC5Z2X	/dev/ttyUSB3	FTDI MTI-USB1000

```
Uploading file "main.ihex" OK
*** MFUD5QT2 on /dev/ttyUSB4 @ 193.1.133.198 ***
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
32296 bytes programmed.
Reset device ...
*** MFUD5QHX on /dev/ttyUSB1 @ 193.1.133.198 ***
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
32296 bytes programmed.
Reset device ...
*** Done ***
```

Copyright 2012 David C Harrison. All Rights Reserved.

Figure 6.2: Snapshot from the Web-based interface used to access the testbed programing utilities described in Chapter 6.

of mote-class devices connected to the USB serial of each mini-pc). The developer can associate a symbolic name to each mote, which is internally identified via its MAC address. This means that the motes can be easily moved across different islands, without affecting the way they can be accessed within the test-bed.

- Install new version of the software on a list of motes. Remote operators can do this by simply selecting (i) the list of target motes from the Web-based interface, and (ii) the embedded program image to install to those motes from their local computer. After that, the Web-based interface will upload the selected image and supervise its installation (always via ssh) to the mini-pcs used to access each mote in the list.

Figure 6.2 is a snapshot of the Web-based interface showing the list of motes connected to a remote test-bed (on the left) and the result of a re-programming request (on the right).

6.3 Accessing the Software

Both the re-programming shell scripts used in the mini-pcs and the Web-based interface (built with the Vaadin Java-based Web Application framework, see https://vaadin.com/home) are included in the RUBICON code repository and in all the pre-compiled images available from the project WiKi, together with detailed usage instructions.

The respective folders have the following structure in the file system:

```
.../1-Comm/Testbed-Utilities/server
.../1-Comm/Testbed-Utilities/mini-pc
```

Chapter 7

Installing and Using the Communication Layer (NesC and Java version)

This chapter explains how to compile and install the code for the Communication Layer in nesC and Java ([R1v2.JAVAAPI]).

7.1 Accessing the software

The software of this deliverable has been stored in the subversion repository of RUBICON. In addition, precompiled and ready to be used images of TinyOS and each element of the RUBICON Communication Layer have been uploaded on the project. WiKi. These include:

- A Ubuntu 11.10 Virtual Machine.
- SD-Card images of Ubuntu 9.04 for Sheeva-Plug mini-pcs (see http://www.plugcomputer.org/) and Rasbian, a version of Debian optimized for the Raspberry Pi hardware (see http://www.raspberrypi.org/ and http://www.raspbian.org/)

7.1.1 Software requirements and hardware assumptions

Successful operation of this software requires the deployment of the following hardware and software configuration.

- Zero or more islands (groups) of WSN nodes (also called motes) where each mote is within direct communication range of each other mote.
- Zero or more motes in each such island. Each motes must deploy the TinyOS based Communication Layer API. One sink-node WSN mote per island deploying the TinyOS based Communication Layer API.
- The sink-mote must be connected through a USB serial to a PC running

- The RUBICON Proxy component (D1.4);
- A working peis and peisjava distribution;

7.1.2 Platform dependencies, PC side

The primary target for the PC side of the RUBICON Communication Layer is an Ubuntu 11.10 based systems running on Intel x86 compatible hardware in 32/64-bit modes and with the Oracle Java implementation. The software has been tested and guaranteed to work on these systems, but have also to a lesser extent been verified to work with a range of other Posix conformant operating systems such as other linux based systems as well as Macintosh based systems. The Communication Layer has also been tested with ARM-compatible mini-pcs in both the Sheeva-Plug and the Raspberry Pi family. More details on these embedded options are given in Chapter 6.

7.1.3 Platform dependencies, WSN/WSAN side

The primary target for the deployment of Communication Layer over WSN/WSAN are mote-class devices based on a micro-controller and Bluetooth radio stack capable of running TinyOS 2.x and the Communication Layer API. Furthermore, we assume that the motes have a programming flash memory in addition to a minimum of 10KB of on-board RAM memory. The software has been tested and guaranteed to work on the TelosB clone CM3000 by Advanticsys: the mote is equipped with and MSP430 processor, 48KB program flash, 10KB data RAM and 1MB external flash.

7.2 Installation of the Communication Layer

7.2.1 Get the source code

An installation of TinyOS is required to build and install the Communication Layer (this can be checked by issuing the command `tos-check-env` and compile an example such as *Blink*). Developers also need to have SVN correctly installed with a RUBICON account. The project WiKi contains further details. After checkout/Update from the SVN, the software will have the following structure on the file system:

```
.../1-Comm/Mockup-Sink
.../1-Comm/Mockup-WSN
.../1-Comm/CommunicationLayer
.../1-Comm/ucd.rubicon.gateway
.../1-Comm/ucd.rubicon.network
.../1-Comm/ucd.rubicon.proxy
```

7.2.2 Compiling the NesC Source Code

In order to compile and flash the Sink code, go to `RUBICON/1-Comm/Mockup-Sink` folder and type `make telosb install.0 bsl,YOUR_SERIAL_PORT`. The parameter `YOUR_SERIAL_PORT` is typically `devttyUSB0`.

Where X is the *ID* to be given to the mote, which must be greater than 0.

7.2.3 Compiling the Java Source Code

In order to compile the Java Network go to `RUBICON/1-Comm/CommunicationLayer/java` and type `./build.sh`. In order to compile the Proxy and the Gateway go to `RUBICON/1-Comm/ucd.rubicon.proxy`, edit the *gateway.properties* file by adding your serial port in the *network.source* line: `network.source=serial@/dev/ttySERIAL_PORT:telosb`, then type `ant build`.

7.2.4 Launch the Proxy

In order to launch the Proxy component shut down all the motes first. Therefore, switch on or reset the Sink and start the Proxy by typing `./testProxy.sh`. Switch on the motes and you should see that they automatically join the island. You can see this either in the shell or in Peis tupleview console. In this latter case, run tupleview before starting the Proxy by typing in another console `tupleview`. The motes will flash their green LED to signal when they have successfully completed the join protocol and joined a RUBICON island.

7.3 Overview of the New API of Connectionless Component

7.3.1 The Send Command

The Send command can be used to send single messages to a remote devices. In the following, we specify its interface:

```
uint16_t send(r_addr_t dest, int8_t* payload, uint8_t nbytes,
              bool reliable, uint8_t comp_id),
```

where:

- `dest` is the RUBICON address of the destination node (in the form `<peis-id, mote_addr>`);
- `payload` is message content to be sent as a vector of bytes. Note that `payload` is `App_msg` structure that is described later;
- `nbytes` is the size of the application payload;
- `reliable` is a flag that specifies if an acknowledgment is need or not;
- `comp_id` is an ID that specifies the sender application (e.g., Control layer, Learning Layer, etc.).
- The function returns the sequence number generated by the network for that message.

7.3.2 The Receive Event

When the receiving node receive a message the following event is called

```
event void receive(r_addr_t sender, int8_t* payload, uint8_t nbytes),
```

where:

- `sender` is the RUBICON address of the sender;
- `payload` is message content (payload) to be sent as a vector of bytes;
- `nbytes` is the size of the application payload.

7.3.3 Using the Connectionless Interface

The application that want to use the connectionless interface must use the `Connectionless` component. In order to use the Connectionless in NesC in module file use interface Connectionless. In the configuration file use the component `ConnectionlessC` and call `Application.Connectionless-> ConnectionlessC.Connectionless[APP_ID]`. In Java create an instance of `Connectionless` class and register to listener for receiving the `receive()` event.

7.3.4 Rubicon_Ack Interface

When a send with `reliable=TRUE` is invoked, the sender subsequently receives a notification of receipt of the message by the remote node. The event is specified in the following:

```
event void receive_ack(r_addr_t sender, uint16_t seq_num),
```

where

- `sender` is the RUBICON address of the node that generated the acknowledgment message;
- `seq_num` is sequence number of the acknowledged message. This number must be equal to the one that the `send` command that required the acknowledgment returned.

7.3.5 Using Rubicon_Ack Interface

The applications using Connectionless interface and willing to send reliable messages, must use the `Rubicon_Ack` component and must implement the `receive()` event. In order to use the `Rubicon_Ack` in NesC in module file use interface `Rubicon_Ack`. In the configuration file use the component `Rubicon_AckC` and call `Application.Rubicon_Ack->Rubicon_AckC.Rubicon_Ack[APP_ID]`. In Java create an instance of `Rubicon_Ack` class and register to listener for receiving the `receive_ack()` event.

7.3.6 The App_msg structure

The `App_msg` is the structure that application has to use to interact with the Communication Layer. This structure must be allocated by the sender before calling the `send` command. This is unusual for a communicating interface at the application level, but this is needed to fulfill the requirement [R1v2.MEMORYOPT] and avoids copies of messages for each call of the lower level calls (transport, network, et.).

- `TRANS_ID`: 1 byte field
- 1 byte field: `APP_ID`
- `payload` of maximum size `APP_PAYLOAD` = 100.

The first two fields do not have to be set, and are only needed to allocate the space for these fields as explained above. In Java this optimization is not needed, since the code will be used in a more powerful hardware environment. Therefore, we can just prepare the message that we send along with Java interface. However, in order to make the two worlds TinyOS and Java interoperable, we must agree on the format of the messages. The user application will define its own structure in nesC as a message to exchanged, and this will be converted to Java through the MIG tool provided by the environment TinyOS. In order to prepare the application-level payload in Java, we use the Java class nesC generated by the MIG tool that extends `net.tinyos.message.Message`. For instance, suppose the application wants to send the of the following structure:

```
typedef nx_struct demo_msg {
nx_uint8_t msg_type;
nx_uint16_t sensorsBitmask;
nx_uint16_t period;
} demo_msg_t;
```

The MIG will generate the class with the following interface:

```
public class DemoMsg extends net.tinyos.message.Message
    public void set_msg_type(short value)
public void set_sensorsBitmask(int value)
public void set_period(int value)
}
```

The application then have to do the following operations:

```
DemoMsg dmsg = new DemoMsg();

//set fields of the application message
dmsg.set_msg_type(demo_msg_type);
dmsg.set_sensorBitmask(sensorsBitmask);
dmsg.set_period(period);
```

Namely, instantiate the class `DemoMsg` and fill various fields corresponding to the structure in nesC. Conversely, if the massage is received from a mote by means of the Java interface, we need to do exactly the inverse steps for extracting the information from the payload , i.e.:

```
event void receive(RubiconAddress sender, byte[] payload, short nbytes) {
switch (payload[0]) {
case DEMO_MSG:
//instantiate a DemoMsg using the payload received
DemoMsg dmsg = new  DemoMsg(payload);
//accessing the fields of the application level message
int sensorBitmask = dmsg.get_sensorBitmask();
int period = dmsg.get_period();
break;
case ...
}
}
```

Chapter 8

Conclusions

We conclude this report with the an overview of the planned tasks achieved in the second year of RUBICON within WP1, Task 1.3: *Implementation and test of the communication infrastructure*, as documented in the Description of Work, and the performed work as documented in this deliverable and in second interim progress report.

Task 1.3 has been performed as planned and finished in March 2013. Based on the feedback we have received from the partners following the first release of the software (at M12, as documented in D1.3.1), the work in Task 1.3 on the second year of the project have focused on i) collecting new requirements, ii) addressing them by refining of the design of the Communication Layer and its components, and iii) supervising and coordinating the necessary modifications to the Communication Layer and the creation of new features in the activities carried out in Task 1.2 and Task 1.4, which are documented respectively, in deliverables D1.2 and D1.4.

This deliverable is a prototype of the fully implemented and integrated communication layer, offering solutions for integrating WSAN and robotic components and both data and synaptic channels.

Overall, this work has produced the final version of the Communication Layer that will be used in the final integration of the RUBICON framework and in the application scenarios. To this end, the software has been fitted with support for heterogeneous networks. This has been done by providing it with integration mechanisms toward existing middleware (e.g. the Tecnalia SALAD middleware and the Robotic Operating System), which together allows the RUBICON system to interact with all the devices used in the two application testbeds. Learning, control and scalability requirements have been addressed by providing reliable communication mechanisms to be used to communicate learning and control instructions, and by integrating the Peis middleware with heterogeneous networks in order to support the creation of re-configurable, cluster-based and peer-to-peer system systems. Furthermore, the software has been redesigned and extended to support further extensions and to be as much as possible independent from specific networks and applications, thus also increasing its potential for outliving the duration of the RUBICON project. Finally, the experimentation and further testing of the Communication Layer, and its usability by both RUBICON partners and third parties has been facilitated with a number of dedicated software utilities, including a WSN logger for data collection, and a WEB-based WSN reprogramming tool.

8.1 Impact

The software described in this deliverable is accessible in the software repository containing all the code and examples needed for the execution of the Communication Layer.

The software is currently in active use by all the RUBICON partners, in both research and application testbed, where it is being used to collect datasets to drive the final advancements planned within the project, and to test the fully-integrated RUBICON architecture. Based on the feedback of the partners, further work is expected in order to validate the performance and tune all the communication and integration mechanisms presented in this deliverable during the final demonstration stage. Data-collection, re-configuration, re-programming and monitoring of the communication infrastructure during this final stage will be supported by availing of the tools described in this deliverable.

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

Bibliography

[1] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society.

[2] W. Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *ICC'10*, pages 1–6, 2010.

[3] David C. Chu Prabal K. Dutta, Jonathan W. Hui and David E. Culler. Securing the deluge network programming system. In *5th Internaltional Conference on Information Processing in Sensor Networks (IPSN'06), April 25-27, 2006*, 2006.

Chapter 9

Appendix A

9.1 Private APIs and main components of the Communication Layer

9.1.1 The Network Component

The Network Component is responsible for sending messages over the communication interface to a given destination device. It can be a mote, a PC or a Robot, and it can be located either in the local island or in a remote island. The destination device is addressed by a RUBICON address, defined as follows:

```
//RUBICON address
typedef r_addr {
uint16_t pid;
uint16_t devid;
} r_addr_t;
```

where pid is the PEIS identifier and it addresses software components in a PC or a Robot, and devid is the device identifier. The former identifies also the island where the PC is connected, while the latter is used only to address a mote displaced in the island identified by the pid.

The value 0xFFFF is used to address the broadcast island or mote, respectively; the value 0xFFFE is used by the Synaptic component to address the local island and local mote, while the value 0xFFFD as pid is used to address the Proxy component in the join protocol and as devid to specify the NULL_MOTE in case the message is not for a mote device.

The Network component exposes an external interface (defined in the following section) that is used by the upper levels to invoke the network functionalities.

It also have some internal functions for the proper routing and handling of the messaged to be sent.

External Interface

command uint16_t **send**(r_addr_t dest, int8_t* payload_TDM, uint8_t nbytes_TDM, bool reliable);

Sends the given message to the specified destination. It generates and returns a sequence number.

Parameters:

dest - (input) The RUBICON address of the destination of the message. If the devid field is NULL_MOTE, it addresses a PC or a Robot.

payload_TDM - (input) Pointer to a region where the data to be sent are stored.

nbytes_TDM - (input) Number of bytes to be sent.

reliable - (input) TRUE if an ack is requested for the current message.

Returns:

The sequence number generated for that message.

Implementation: it checks if the dest.pid == LOCAL_ISLAND and dest.devid != NULL_MOTE (that means that the destination is a mote in the same island) and in this case sets the device_dest parameter of the TOSSend to the devid field of the input dest parameter, otherwise it is set to the SINK address.

Then it prepares the new payload (payload_net = network_header + payload_TDM) by adding the network_header at the beginning of the received payload. The network_header is defined as follows:

1. r_addr_t dest
2. r_addr_t sender
3. bool reliable
4. uint16_t seq_num
5. uint8_t nbytes

It also updates the size of the message to be sent. Then it invokes the TOSSend as follows:

call TOSSend(device_dest, payload_net, size_net);

event void **receive**(r_addr_t sender, int8_t* payload_TDM, uint8_t nbytes_TDM);

It notifies the reception of the given message from the specified source. The implementation of this event MUST be implemented in the upper levels, the Network component just notifies this event.

Parameters:

sender - (output) The RUBICON address of the source of the message.

payload_TDM - (output) Pointer to the received message.

nbytes_TDM - (output) Number of bytes received.

Implementation: it is signaled to the upper level at the end of the implementation of the TOSReceive() event.

command uint16_t **join_island**(mote_desc_t* desc);

It initiates the protocol to join the island. The application calls this command by passing the pointer to the mote descriptor that contains the following information:

typedef nx_struct mote_desc {

nx_uint8_t mote_type;

nx_uint16_t transducers; //bitmask

nx_uint8_t actuators; //bitmask

} mote_desc_t;

where transducers and actuators are bitmask representing, respectively, the transducers and actuators embedded in the mote.

Parameters:

desc - (input) Pointer to the descriptor of the mote joining the island. It is only stored in the Network layer at this stage, it will be sent to the SINK when sending the Join_Ack.

Returns:

The sequence number generated for that message. The value 0xFFFF means that the mote is already joined.

Implementation: the mote sends the request to the sink that replies with a message containing the r_addr_t given to the mote. As consequence, the network layer automatically sends an acknowledgement message to the sink and notifies the application of the joined event.

event void **joined**(r_addr_t addr, error_t success);

It notifies the completion of the join protocol. The implementation of this event MUST be implemented in the upper levels, the Network component just notifies this event.

Parameters:

addr - (output) The given RUBICON address received from the sink.

success - (output) Whether the joining was successfull.

Implementation: it is signaled to the upper level when the ack to the join_ack message is received from the sink.

Internal Functions

//TinyOS send

command error_t **TOSSend**(uint16_t device_dest, int8_t* payload_net, uint8_t size_net)

It is called from the network at the end of the implementation of the Network.send().

//TinyOS receive

event void **TOSReceive**(int8_t* payload_net, uint8_t size_net)

It extracts the network_header from the payload_net and checks if the destination is the current device. If so, it notifies the reception of the message to the upper level by signaling a Network.receive(sender, payload, nbytes) event. In this case, it also checks the reliable flag, and if it is true, it sends back the acknowledgment for that message (by using the sequence number received). Otherwise (this means that the mote receiving this message is the sink) it forwards the message over the serial interface to the Gateway by calling M_sendUART(payload_net, uint8_t size_net);

//TinyOS serial send

command error_t **M_sendUART**(int8_t* payload_net, uint8_t size_net)

It is called from the network at the end of the implementation of the TOSReceive().

// TinyOS serial receive

event void **M_receiveUART**(int8_t* payload_net, uint8_t size_net)

It extracts the network_header from the payload_net and checks if the dest.devid == TOS_NODE_ID (that means that the destination is the SINK). In this case it notifies the reception of the message to the upper level by signalling a Network.receive(sender, payload_TDM, nbytes_TDM) event. In this case, it also checks the reliable flag, and if it is true, it sends back the acknowledgment for that message (by using the sequence number received). Otherwise it forwards the message to the destination mote by calling the TOSSend (dest.devid, payload_net, size_net).

PC Functions

//TinyOS moteIF send

PC_sendUART(byte[] payload_net)

It is called from the network at the end of the implementation of the GWReceive().

//TinyOS message listener for messages coming from serial interface

PC_receiveUART(byte[] payload_net)

It extracts the network_header from the payload_net and checks if the dest.pid == LOCAL_ISLAND (that means that the message has reached its destination). In this case it notifies the reception of the message to the upper level by signalling a Network.receive(sender, payload_TDM, nbytes_TDM) event. In this case, it also checks the reliable flag, and if it is true, it sends back the acknowledgment for that message (by using the sequence number received). Otherwise it sends the message to the destination island by means of PEIS (either Tuplespace or PEIS_kernel) by calling the GWSend(dest, payload_net, size_net).

GWReceive(byte[] payload_net, uint8_t size_net)

It extracts the network_header from the payload_net and checks if the dest.devid == NULL (that means that the destination is the current device and not a mote). In this case it notifies the reception of the message to the upper level by signalling a Network.receive(sender, payload_TDM, nbytes_TDM) event. In this case, it also checks the reliable flag, and if it is true, it sends back the acknowledgment for that message (by using the sequence number received). Otherwise it forwards the message to the SINK over the serial interface by calling the PC_sendUART(payload_net, size_net).

Info to Gateway

API Implement the following method that sends the given payload to the destination PC or robot through PEIS (either by means of Tuplespace or PEIS_kernel):

void **GWSend**(RubiconAddress dest, byte[] payload_net, short size_net) where the maximum size of the payload is 114 bytes.

Implement also the reception of the sent message at destination point and signal the reception of the message to the Network by signalling the following event:

event **GWReceive**(byte[] payload_net, short size_net)

9.1.2 Transport Message Dispatcher (TMD) Component

It is responsible of dispatching incoming messages to the proper upper-level component. To this purpose, upon receiving a request to send a message from above components, it adds a flag into the payload that is used at destination to determine to which component the message should be delivered.

External Interface

command uint16_t **send**(r_addr_t dest, int8_t* payload, uint8_t nbytes, bool reliable, uint8_t comp_id);

Adds the flag for the message dispatching at destination and forwards the given message to the Network layer. It returns the sequence number generated by the Network Layer.

Parameters:

dest - (input) The RUBICON address of the destination of the message. If the devid field is null, it addresses a PC or a Robot.

payload - (input) Pointer to a region where the data to be sent are stored.

nbytes - (input) Number of bytes to be sent.

reliable - (input) TRUE if an ack is requested for the current message.

comp_id - (input) The identifier of the Transport-level component sending the message. It is used at destination side to dispatch the message to the right Transport-level component.

Returns:

The sequence number generated for that message, received from the Network layer.

Implementation: it prepares the new payload (payload_TDM = TRANS_ID + payload) by adding the comp_id for the message dispatching at the beginning of the received payload. The TRANS_ID is a byte. It also updates the nbytes parameter accordingly. Finally it forwards the message to the Network layer by calling the Network.send(dest, payload_TDM, nbytes_TDM, reliable), and forwards to the upper layer the sequence number received from the Network layer.

event void **receive**(r_addr_t sender, int8_t* payload, uint8_t nbytes);

It forwards the received message to the proper upper-level component according to the TRANS_ID contained in the message. The implementation of this event MUST be implemented in the upper levels, the TDM component just notifies this event.

Parameters:

sender - (output) The RUBICON address of the source of the message.

payload - (output) Pointer to the received message.

nbytes - (output) Number of bytes received.

Implementation: it is signaled at the end of the execution of the Network.receive() event.

Internal Functions

It also implements the following event signalled by the Network layer:

event void **Network.receive**(r_addr_t sender, int8_t* payload_TDM, uint8_t nbytes_TDM)

It extracts the TRANS_ID contained in the payload_TDM and dispatches the received message to the proper upper-level component, according to the TRANS_ID, by signaling the **receive**(sender, payload, nbytes) event.

9.2 Public APIs of the Communication Layer

9.2.1 Connectioless Component

This component enables the point-to-point communication between two specific devices, typically for sending commands to remote mote-actuators or pcs.

External Interface

command uint16_t **send**(r_addr_t dest, int8_t* payload, uint8_t nbytes, bool reliable, uint8_t comp_id);

Adds the flag for the message dispatching at destination and forwards the given message to the Transport Message Dispatcher. It returns the sequence number generated by the Network Layer.

Parameters:

dest - (input) The RUBICON address of the destination of the message. If the devid field is null, it addresses a PC or a Robot.

payload - (input) Pointer to a region where the data to be sent are stored.

nbytes - (input) Number of bytes to be sent.

reliable - (input) TRUE if an ack is requested for the current message.

comp_id - (input) The identifier of the Application-level component sending the message. It is used at destination side to dispatch the message to the right Application-level component.

Returns:

The sequence number generated for that message, received from the Network layer.

Implementation: it prepares the new payload (payload_CLS = APP_ID + payload) by adding the comp_id for the message dispatching at the beginning of the received payload. The APP_ID is a byte. It also updates the nbytes parameter accordingly. Finally it forwards the message to the Transport Message Dispatcher layer by calling the TransportMD.send(dest, payload_CLS, nbytes_CLS, reliable), and forwards to the upper layer the sequence number received from the Network layer.

event void **receive**(r_addr_t sender, int8_t* payload, uint8_t nbytes);

It forwards the received message to the proper upper-level component according to the APP_ID contained in the message. The implementation of this event MUST be implemented in the upper levels, the Connectionless component just notifies this event.

Parameters:

sender - (output) The RUBICON address of the source of the message.

payload - (output) Pointer to the received message.

nbytes - (output) Number of bytes received.

Implementation: it is signaled at the end of the execution of the TransportMD.receive() event.

Internal Functions

It also implements the following event signalled by the Network layer:

event void **TransportMD.receive**(r_addr_t sender, int8_t* payload_CLS, uint8_t nbytes_CLS)

It extracts the APP_ID contained in the payload_CLS and dispatches the received message to the proper upper-level component, according to the APP_ID, by signaling the **receive**(sender, payload, nbytes) event.

9.2.2 Component Management

The API is unchanged (see Deliverable D1.3.1), only the join message has been changed. The new version of the join message is the following:

typedef nx_struct serial_joined_msg {

nx_uint16_t island_addr;

nx_uint16_t mote_addr;

nx_uint8_t mote_type;

nx_uint16_t transducers;

nx_uint8_t actuators;

} serial_joined_msg_t;

where mote_type identifies the type of mote (advanticsys, kneex, or other), transducers and actuators are bitmasks representing, respectively, which transducers and actuators are available on the mote. These information have to be provided at compile time.

9.2.3 Rubicon Ack

It is a transport-level component responsible of manage the sending/reception of the acknowledgement Connectionless messages. When the Connectionless component receives a message that has to be acknowledged, it calls the send_ack() command to send back the ack message to the source. At the sender side, when an ack message is received by the network, it is notified to this component (as transport layer notification), and it notifies the application of the reception of the ack by means of the receive_ack() event.

command void **send_ack**(r_addr_t dest, int8_t* payload, uint8_t nbytes, bool reliable, uint8_t comp_id);

Parameters:

dest - (input) The RUBICON address of the destination of the message. If the devid field is null, it addresses a PC or a Robot.

payload - (input) Contains only the header and the seq_num.

nbytes - (input) Number of bytes to be sent.

reliable - (input) Always FALSE, it is left in order to be compliant with the TransportMD interface.

comp_id - (input) Always RUBICON_ACK, it is left in order to be compliant with the TransportMD interface.

Implementation: Sends the ack for a message. The payload has to be an app_msg_t contaning, in its payload, only the seq_num of the message to be acknowledged, therefore the expected nbytes is 4 (2 bytes for the APP_ID and TRANS_ID and 2 bytes for the seq_num). This component intefaces with the TransportMD, as all the other Transport component.

event void **receive_ack**(r_addr_t sender, uint16_t seq_num);

Signal that the acknowledgment of the message identified by the given sequence number from the given source. The implementation of this event MUST be implemented in the upper levels, the Network component just notifies this event.

Parameters:

sender - (output) The RUBICON address of the sender of the acknowledgement message.

seq_num - (output) The sequence number of the acknowledged message

9.2.4 Synaptic_Channels Component

The API is unchanged (see Deliverable D1.3.1).

9.2.5 Streams Component

The API is unchanged (see Deliverable D1.3.1).