RUBICON
Robotic UBUquitous COgnitive Network

SEVENTH FRAMEWORK PROGRAMME

Project No.: 269914

# D2.3 – Core Learning Services API and Documentation V.2.0



| | | |
|---|---|---|
| **Editor:** | **Davide Bacciu** | **UNIPI** |
| | **Claudio Gallicchio** | **UNIPI** |
| | **Alessio Micheli (Supervisor)** | **UNIPI** |
| | **Claudio Vairo** | **CNR** |
| **Contributor(s):** | **Stefano Chessa** | **UNIPI** |
| | **Mauro Dragone** | **UCD** |



| | |
|---|---|
| Issue Date | 30/09/2013 (M30, MS4) |
| Deliverable Number | D2.3 |
| WP | WP2 – Learning Layer |
| Status | ☐Draft ☐Working ☒Released ☐Delivered to EC ☐Approved by EC |

| 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) |

| Document history | | | |
| --- | --- | --- | --- |
| V | Date | Author | Description |
| 1.0 | 15/10/2013 | Davide Bacciu, Claudio Gallicchio | First deliverable version for internal revision and quality assurance review. |
| 1.1 | 30/10/2013 | Davide Bacciu | Revised version after quality assurance review. Changed status to “Released”. |

**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 the release 2.0 of the “Core Learning Service API” software, presented as deliverable D2.3. We focus here on a description of the design and implementation of this software, and we provide an experimental evaluation of its performance on real-world data from the RUBICON application scenarios in WP5.

This report includes

- External and internal overviews of the learning layer architecture, with a particular focus on progress since D2.2.
- An outline of how to access the software and the technical platform requirements
- Detailed descriptions of the main components of the learning layer software system
    - The Core learning service API v2.0
    - The Learning Network (LN)
    - The Learning Network Manager
    - The Training Manager
- A description of the testing and experimental assessment and the successful results of all the tests being carried out
- Conclusions that mainly describe the compliance to workplan and the impact on the project

In addition to this report with an appendix documenting the Learning Layer 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.

# Contents

**EXECUTIVE SUMMARY** ........ **3**

**ABBREVIATIONS** ........ **6**

**FIGURES** ........ **7**

**TABLES** ........ **10**

**1. OVERVIEW** ........ **11**

1.1 TARGET AUDIENCE ........ 11
1.2 EXTERNAL VIEW OF THE LEARNING LAYER ........ 11
1.3 INTERNAL ARCHITECTURE OF THE LEARNING LAYER ........ 11
1.4 DELIVERED SOFTWARE ........ 13
*1.4.1 Accessing the software ........ 13*
*1.4.2 Software requirements and hardware assumptions ........ 13*
1.4.2.1 Platform dependencies: node hosting the RUBICON Learning Gateway ........ 15
1.4.2.2 Platform dependencies: nodes hosting Learning Modules ........ 15
1.4.2.3 The Learning Layer as a Distributed Software System ........ 15
1.5 DIFFERENCES FROM D2.2 CORE LEARNING SERVICE V1.0 ........ 16

**2. LEARNING LAYER SOFTWARE SYSTEM** ........ **19**

2.1 CORE LEARNING SERVICE (CLS) API V2.0 ........ 19
*2.1.1 The Java Learning Gateway API (LGA) ........ 19*
*2.1.2 The NesC Learning Network API ........ 20*
*2.1.3 The Java Learning Network API ........ 23*
2.2 LEARNING NETWORK (LN) ........ 23
*2.2.1 Overview ........ 23*
*2.2.2 Learning Modules ........ 23*
2.2.2.1 NesC Learning Modules ........ 24
2.2.2.2 Java Learning Modules ........ 24
*2.2.3 Synaptic Connections ........ 26*
2.2.3.1 NesC Synaptic Connections ........ 27
2.2.3.2 Java Synaptic Connections ........ 28
*2.2.4 Forward Computation ........ 29*
*2.2.5 Learning Network Wrapper ........ 31*
2.3 LEARNING NETWORK MANAGER (LNM) ........ 32
*2.3.1 Overview ........ 32*
*2.3.2 Supervisor Interface ........ 33*
*2.3.3 Synaptic Communication Control and Management ........ 35*
*2.3.4 Device and Module Management ........ 35*
2.4 TRAINING MANAGER ........ 37
*2.4.1 Overview ........ 37*
*2.4.2 Feature Selection ........ 39*
2.4.2.1 Feature Filter ........ 40
2.4.2.2 Feature Wrapper ........ 41
*2.4.3 Training of a Learning Module ........ 41*
2.4.3.1 Training through the Network Mirror ........ 41
2.4.3.2 Online learning ........ 43
*2.4.4 Deployment of a Learning Module ........ 45*
2.5 HOW TO ACTIVATE LEARNING IN THE LEARNING LAYER: A SUPERVISOR PERSPECTIVE ........ 46

*2.5.1 How to activate a new learning task (incremental learning) ............................................ 46*
*2.5.2 How to supply an instantaneous supervised/reinforcement teaching signal (refinement learning)........................................................................................................................ 47*
2.6 GRAPHICAL USER INTERFACE ........................................................................................... 48

**3. TESTING ........................................................................................................................ 51**

3.1 OUTLINE.......................................................................................................................... 51
3.2 TESTING LEARNING NETWORK COMPUTATION ......................................................................... 52
*3.2.1 Test Configuration ......................................................................................................... 52*
*3.2.2 Testing Distributed Computation with NesC Learning Modules........................................ 52*
*3.2.3 Testing Distributed Computation with Java Learning Modules......................................... 58*
*3.2.4 Performance Assessment of the Echo State Network model on Real-World RUBICON Tasks 62*
3.3 TESTING FEATURE SELECTION ............................................................................................ 68
3.4 TESTING ONLINE LEARNING ............................................................................................... 73
3.5 TESTING RECOVERY ......................................................................................................... 78
*3.5.1 Test Configuration ......................................................................................................... 78*
*3.5.2 Testing Mote Watchdog and Recovery............................................................................. 78*

**4. CONCLUSIONS ............................................................................................................. 83**

4.1 COMPLIANCE TO WORKPLAN .............................................................................................. 83
4.2 IMPACT ON PROJECT.......................................................................................................... 85
4.3 NEW DEVELOPMENTS AND UNFORESEEN ISSUES ....................................................................... 85

**5. APPENDIX A – REFERENCE MANUAL................................................................................ 87**

# Abbreviations

| | |
|---|---|
| AAL | Ambient Assisted Living |
| API | Application Programming Interface |
| CLS | Core Learning Service |
| ESN | Echo State Network |
| GUI | Graphical User Interface |
| LGA | (Java) Learning Gateway API |
| LN | Learning Network |
| LNM | Learning Network Manager |
| PEIS | Physically Embedded Intelligent Systems (Ecology) |
| RUBICON | Robotic UBIquitous COgnitive Network |
| TM | Training Manager |
| WSN/WSAN | Wireless Sensor Network / Wireless Sensor and Actuator Network |

# Figures

FIGURE 1 EXTERNAL VIEW OF THE LEARNING LAYER WITH AN HIGHLIGHTING OF THE APIS INTERFACING WITH OTHER RUBICON LAYERS. ........ 12

*FIGURE 2 SOFTWARE ARCHITECTURE OF THE LEARNING LAYER: DERIVES FROM THE PRELIMINARY SKETCH IN D2.1 AND REFINEMENTS IN D2.2. LOGICAL SUBSYSTEMS ARE REPRESENTED AS SIMPLE RECTANGULAR BOXES, SOFTWARE COMPONENTS ARE SMALL RECTANGLES WITH THE UML COMPONENT ICON ON THE TOP-RIGHT CORNER, WHILE INTERLAYER AND INTRALAYER INTERFACES ARE DENOTED AS THICK AND THIN ARROWS.* ........ 14

FIGURE 3 ARCHITECTURAL DETAIL OF THE LN SUBSYSTEM: THE MANAGER AND LEARNING MODULE COMPONENTS RUN ON ECOLOGY DEVICES (REPRESENTED AS L-SHAPED BOXES) DISTRIBUTED IN THE ENVIRONMENT. SUPPORTED DEVICES INCLUDE MOTE-CLASS AND JAVA-ENABLED NODE. THE LN WRAPPER COMPONENT IS DEPLOYED ON A PC (REFERRED TO AS LEARNING GATEWAY), REPRESENTED AS A DASHED RECTANGLE. ........ 17

FIGURE 4 ARCHITECTURAL DETAIL OF THE LN MANAGER SUBSYSTEM: THE LN CONTROL AGENT COMPONENT IS DEPLOYED ON THE LEARNING GATEWAY. THE LN MANAGER SUBSYSTEM INTERACTS WITH THE PEIS WRAPPER COMPONENT, RUNNING ON THE SAME PC, THAT INTERFACES THE LEARNING LAYER WITH EXTERNAL PEIS-ENABLED COMPONENTS. ........ 17

FIGURE 5 ARCHITECTURAL DETAIL OF THE TRAINING MANAGER SUBSYSTEM: THE REPOSITORY, THE TRAINING AGENT AND NETWORK MIRROR COMPONENTS ALL RUN ON THE LEARNING GATEWAY, REPRESENTED AS A DASHED RECTANGLE. ........ 18

FIGURE 6 THE PACKAGE DIAGRAM OF THE JAVA LEARNING GATEWAY API. ........ 21

FIGURE 7 SCHEMATIC ILLUSTRATION OF THE WIRING AMONG SOFTWARE COMPONENTS INVOLVED IN THE NESC LN API. ........ 22

FIGURE 8 A SCHEMATIC ILLUSTRATION OF THE INFORMATION FLOW IN ONE STEP OF THE FEEDFORWARD COMPUTATION ON-BOARD A MOTE. ........ 30

FIGURE 9 INFORMATION FLOW IN THE LEARNINGNETWORKWRAPPER COMPONENT: THE DASHED ARROW REPRESENT A SYNAPTIC CONNECTION FEEDING THE JAVA LEARNING MODULE B WITH THE NEURON OUTPUT OF A NESC LEARNING MODULE A (OTHER SYNAPTIC CONNECTIONS ARE OMITTED FOR CLARITY). THE JAVA-SIDE OF THE LN FEEDS AN INPUT BUFFER "JAVA LN" THROUGH PEIS TUPLES, WHILE THE RUBICON GATEWAY PROVIDES INFORMATION FROM THE NESC –SIDE. THE INFORMATION IN THIS TWO BUFFERS IS ROUTED TO THE LNOUTPUTS COMPONENT FOR PUBLISHING AND/OR TO THE "LN BYPASS" MECHANISM, THAT FEEDS IT BACK TO THE TARGET MODULE OF THE LN. THE "VIRTUAL" SYNAPTIC CONNECTION A->B IS IMPLEMENTED BY ARROW 1 (GATEWAYINTERFACE) ENTERING THE NESC BUFFER, BEING RE-ROUTED TO THE "LN BYPASS" WITH ARROW 2 (LOCAL MESSAGE), AND BEING FINALLY DELIVERED TO B THROUGH ARROW 3 (PEIS TUPLE). ........ 32

FIGURE 10 SUMMARY OF THE FUNCTIONALITIES IMPLEMENTED BY THE TWO COMPONENTS OF THE TRAINING MANAGER SUBSYSTEM. THE TRAININGAGENT RELIES ON THE NETWORKMIRROR FOR THE TRAINING AND MODEL SELECTION OF THE LEARNING MODULES. ON THE OTHER HAND, THE NETWORKMIRROR EXPLOITS THE FUNCTIONALITIES OF THE TRAININGAGENT TO DEPLOY A TRAINED COMPUTATIONAL LEARNING TASK TO THE PHYSICAL ECOLOGY. ........ 37

FIGURE 11 SCREENSHOT OF THE LEARNING LAYER GUI SHOWING THE LN LAYOUT. ........ 49

FIGURE 12 SCREENSHOT OF THE LEARNING LAYER GUI SHOWING THE DEVICE MANAGER. ........ 50

FIGURE 13 LEARNING LAYER WAITING DEVICE JOIN AT START-UP. ........ 53

FIGURE 14 GUI SNAPSHOT: ONLY THE SINK DEVICE IS AVAILABLE AT LEARNING LAYER STARTUP ........ 54

FIGURE 15 GUI SNAPSHOT: INFORMATION ON THE JOINED DEVICES ALONG WITH THEIR CAPABILITIES IS STORED IN THE LEARNING LAYER AND DISPLAYED IN THE GUI. ........ 54

FIGURE 16 RECEPTION OF THE WIRING AND TASK INFORMATION THROUGH PEIS. ........ 55

FIGURE 17 RECEPTION OF THE TRAINING SAMPLES THROUGH PEIS. ........ 55

FIGURE 18 MODEL-SELECTION, TRAINING AND DEPLOYMENT OF A TINYOS LEARNING MODULE ON THE EXERCISING TASK. ........ 56

FIGURE 19 SNAPSHOT OF THE GUI SHOWING A DEPLOYED SYNAPTIC CONNECTION BETWEEN TWO MOTES IN THE EXERCISING DEMO. 57

FIGURE 20 STARTING OF THE DISTRIBUTED FORWARD COMPUTATION PRODUCING THE PREDICTIONS OF THE TINYOS LEARNING MODULE FOR THE EXERCISING ACTIVITY. ........ 57

FIGURE 21 MODEL-SELECTION, TRAINING AND DEPLOYMENT OF A JAVA LEARNING MODULE ON THE EXERCISING TASK. ........ 59

FIGURE 22 SYNAPTIC CONNECTION DEPLOYMENT FOR THE EXERCISING TASK WITH JAVA LEARNING MODULES: NOTE HOW THE MODULE REGISTERS IT-SELF AS LISTENER TO THE TUPLE IMPLEMENTING THE INPUT SYNAPTIC CHANNEL (IN RED). ............................ 60

FIGURE 23 WIRING TABLE RESULTING FROM THE DEPLOYMENT OF THE NEW TASK.................................................................. 60

FIGURE 24 STARTING OF THE DISTRIBUTED FORWARD COMPUTATION PRODUCING THE PREDICTIONS OF THE JAVA LEARNING MODULE FOR THE EXERCISING ACTIVITY. ............................................................................................................... 61

FIGURE 25 PUBLICATION OF THE PREDICTIONS FOR THE EXERCISING ACTIVITY ON THE EVENT PUBLISHING TUPLE: THE 3 VALUES DISPLAYED REFER TO THE RUBICON TIME, EVENT OCCURRENCE (+1=OCCURRENCE, -1=NON-OCCURRENCE) AND PREDICTION CONFIDENCE (0 MINIMUM CONFIDENCE, 1 MAXIMUM) , RESPECTIVELY. ...................................................................... 61

FIGURE 26 SNAPSHOT OF THE TUPLES IMPLEMENTING SYNAPTIC COMMUNICATION IN (LEFT) AND OUT(RIGHT) OF THE JAVA LEARNING MODULE. ............................................................................................................................................ 62

FIGURE 27 LEARNING LAYER OUTPUT PREDICTIONS FOR A SET OF EVENTS IN THE AAL HOME LAB SCENARIO. ............................... 64

FIGURE 28 MEAN ABSOLUTE TEST ERROR ACHIEVED BY ESNS ON THE PLANNER WEIGHT PREDICTION FOR LOCALIZATION SYSTEMS – MIRROR TASK. ....................................................................................................................................... 65

FIGURE 29 MEAN ABSOLUTE TEST ERROR ACHIEVED BY ESNS ON THE PLANNER WEIGHT PREDICTION FOR LOCALIZATION SYSTEMS – KITCHEN TASK. ...................................................................................................................................... 65

FIGURE 30 MAP OF THE CORRIDORS INVOLVED IN THE MEASUREMENT CAMPAIGN IN THE WING OF THE STELLA MARIS BUILDING. THE INDICATIVE TRAJECTORY FOLLOWED BY THE ROBOT IS ILLUSTRATED IN RED.................................................................... 66

FIGURE 31 MEAN EUCLIDEAN TEST ERROR ACHIEVED BY ESNS ON THE PRELIMINARY EXPERIMENTS ON THE HOSPITAL ROBOT LOCALIZATION TASK. PERFORMANCE RESULTS ARE REPORTED FOR VARYING RESERVOIR DIMENSIONALITY AND READOUT REGULARIZATION. ........................................................................................................................................ 67

FIGURE 32 MAP OF THE AREA USED FOR THE MEASUREMENT CAMPAIGN FOR THE UNIPI INDOOR ROBOT LOCALIZATION TASK. ...... 67

FIGURE 33 ANGEN EXPERIMENTAL SCENARIO: A MOBILE ROBOT USING LASER RANGE-FINDER LOCALIZATION IS PERFORMING TWO TYPES OF TRAJECTORIES ENDING IN THE KITCHEN, REPRESENTED AS CONTINUOUS AND DASHED LINES. THE PERFORMANCE OF THE LOCALIZATION SYSTEM IS NEGATIVELY AFFECTED BY THE INTRODUCTION OF A MIRROR; THIS EFFECT CAN ONLY BE NOTICED IN THE DASHED TRAJECTORY TYPE. ALL THE SENSORS ARE EQUIPPED WITH PIR SENSORS, BUT ONLY M3 AND M6 ARE IN A LINE-OF-SIGHT THAT IS INFLUENCED BY THE ROBOT MOVING. THE REST OF THE PIRS IS SHIELDED AND IS NOT ACTIVATED BY ROBOT MOTION. ............................................................................................................................................ 69

FIGURE 34 RESULTS OF THE FEATURE SELECTION ALGORITHM ON CONFIGURATION 5 FOR THE ANGEN DATA (SEE TABLE XVI), INCLUDING ALL TRANSDUCERS FROM ALL THE INVOLVED MOTES. THE MAJORITY OF THE RUNNING TIME IS SPENT ON THE COMPUTATION OF THE REDUNDANCY MASK, WHILE THE EFFORT REQUIRED FOR FEATURE FILTERING IS NEGLIGIBLE. ONLY 6 FEATURES ARE RETAINED FROM AN INITIAL SET OF 27................................................................................................ 71

FIGURE 35 PLOT OF THE ELAPSED TIME(MILLISECONDS) FOR FEATURE FILTERING AS A FUNCTION OF THE NUMBER OF INPUT FEATURES. ............................................................................................................................................................ 71

FIGURE 36 RESULT OF THE FEATURE SELECTION MECHANISM WHEN RUNNING INCREMENTAL LEARNING ON THE EXERCISING EVENT.. 73

FIGURE 37 MEAN TEST ACCURACY ON THE AAL HOME LAB-PREPARING COFFEE TASK, ACHIEVED BY ESNS WITH DIFFERENT RESERVOIR DIMENSIONS, TRAINED USING OFFLINE LEARNING (DASHED LINES) AND ONLINE LEARNING (CONTINUOUS LINES), FOR INCREASING NUMBER OF TRAINING SAMPLES CONSIDERED. ........................................................................................... 74

FIGURE 38 MEAN TEST ACCURACY ON THE INDOOR USER MOVEMENT FORECASTING TASK, ACHIEVED BY ESNS WITH DIFFERENT RESERVOIR DIMENSIONS, TRAINED USING OFFLINE LEARNING (DASHED LINES) AND ONLINE LEARNING (CONTINUOUS LINES), FOR INCREASING NUMBER OF TRAINING SAMPLES CONSIDERED ......................................................................................... 74

FIGURE 39 MEAN ABSOLUTE TEST ERROR ON THE ANGEN SCENARIO-MIRROR TASK, ACHIEVED BY ESNS WITH DIFFERENT RESERVOIR DIMENSIONS, TRAINED USING OFFLINE LEARNING (DASHED LINES) AND ONLINE LEARNING (CONTINUOUS LINES), FOR INCREASING NUMBER OF TRAINING SAMPLES CONSIDERED. Y-AXIS IS IN LOGARITHMIC SCALE. ........................................................... 75

FIGURE 40 MEAN ABSOLUTE TEST ERROR ON THE ANGEN SCENARIO-MIRROR TASK, ACHIEVED BY ESNS WITH DIFFERENT RESERVOIR DIMENSIONS, TRAINED USING OFFLINE LEARNING (DASHED LINES) AND ONLINE LEARNING (CONTINUOUS LINES), FOR INCREASING NUMBER OF TRAINING SAMPLES CONSIDERED. STANDARD RLMS WITH KNOWN TARGET IS USED. Y-AXIS IS IN LOGARITHMIC SCALE. ............................................................................................................................................................ 75

Figure 41 Mean test accuracy on the Indoor User Movement forecasting task, achieved by ESNs with different reservoir dimensions, trained using offline learning, before (dashed lines) and after (continuous lines) online learning, for increasing number of training samples considered ........ 76

Figure 42 Outcome of the online learning script after the activation of the online learning process. ........ 77

Figure 43 Outcome of the online learning script during the phase of output prediction. ........ 77

Figure 44 The script awaits joining of at least one device, that is assigned nodeID=720897 and will initially host the Waving learning task (console and GUI output on the left and right, respectively). ........ 79

Figure 45 Training and deployment of the Waving task on the device with nodeID = 720897 ........ 80

Figure 46 The Learning Layer providing predictions on the Waving event using the output generated by the ESN on-board the mote with nodeID = 720897. ........ 80

Figure 47 Another device, with nodeID = 720898, joins the ecology while the other mote keeps providing predictions. ........ 81

Figure 48 Recovery procedure triggered by the failure of device 720897 (left): the cold spare with nodeID 720898 is found to have compatible capabilities and is therefore used as a replacement. The learning module implementing the Waving task is deployed on 720898 and starts providing again its predictions (right). ........ 82

# Tables

TABLE I DESCRIPTION OF THE TUPLE USED TO DELIVER CONTROL INSTRUCTIONS FROM THE LEARNINGNETWORKWRAPPER TO THE JAVA LEARNING MODULES........ 24

TABLE II ENCODING OF THE COMMANDS THAT CAN BE INVOKED THROUGH THE LL_INTERNAL TUPLE. THE ARGS COLUMN DESCRIBE THE ARGUMENTS ASSOCIATED TO THE COMMAND. ........ 25

TABLE III THE MAIN MEMBERS OF THE CLASS GENERICS.ECHOSTATENETWORK........ 26

TABLE IV DEFINITION OF THE INPUTSYNCONNECTION_T DATA STRUCTURE. ........ 27

TABLE V DEFINITION OF THE OUTPUTSYNCONNECTION_T DATA STRUCTURE. ........ 27

TABLE VI DESCRIPTION OF THE TUPLE USED TO REQUEST THE CREATION OF SYNAPTIC CONNECTIONS WHOSE ENDS INCLUDE AT LEAST ONE JAVA LEARNING MODULE ........ 28

TABLE VII EXAMPLE OF A TUPLE TRANSMITTING N COMMA-SEPARATED VALUES FROM A JAVA LEARNING MODULE WITH NETID=-1 TO ANOTHER MODULE WITH NETID=-2 ........ 28

TABLE VIII DESCRIPTION OF THE TUPLE USED TO PUBLISH THE OUTPUTS OF THE LN ........ 30

TABLE IX DESCRIPTION OF THE TUPLE USED TO PUBLISH THE SYMBOLIC NAMES OF THE LN OUTPUTS........ 30

TABLE X DESCRIPTION OF THE TUPLE USED TO DELIVER CONTROL INSTRUCTIONS FROM THE SUPERVISOR TO THE LNM........ 33

TABLE XI ENCODING OF THE SUPERVISOR INTERFACE COMMANDS THAT CAN BE INVOKED THROUGH THE CONTROL_CMD TUPLE. THE ARG1 AND ARG2 COLUMN DESCRIBE THE ARGUMENTS ASSOCIATED TO THE COMMAND. THE DASH "-" INDICATES THAT THE COMMAND DOES NOT REQUIRE AN ARGUMENT (SET IT TO 0 FOR CONVENIENCE). ........ 34

TABLE XII DESCRIPTION OF THE TUPLE USED TO DELIVER WIRING INSTRUCTIONS FROM THE SUPERVISOR TO THE LNM ........ 35

TABLE XIII EXAMPLE OF A WIRING TABLE, USED BY THE TRAINING MANAGER TO CONVERT WIRING INSTRUCTIONS INTO SYNAPTIC CONNECTIONS........ 38

TABLE XIV DESCRIPTION OF THE TUPLE USED TO PUBLISH THE WIRING TABLE ........ 39

TABLE XV DESCRIPTION OF THE TUPLE USED TO REQUEST A WIRING TABLE UPDATE: IF ACTION = "+" THE FOLLOWING WIRING ENTRY IS ADDED TO THE TABLE; IF ACTION = "-", IT IS REMOVED. ........ 39

TABLE XVI DESCRIPTION OF THE TUPLE USED TO DELIVER TRAINING SAMPLES FROM THE SUPERVISOR TO THE LEARNING LAYER. ........ 42

TABLE XVII DESCRIPTION OF THE FIELDS IN THE STRING PAYLOAD FOR THE PEIS TUPLE WITH KEY-STRING LLTRAININGSAMPLE. ........ 42

TABLE XVIII DESCRIPTION OF THE TUPLE USED TO DELIVER ONLINE LEARNING INSTRUCTIONS FROM THE SUPERVISOR TO THE LNM. . 43

TABLE XIX ENCODING OF THE SUPERVISOR INTERFACE COMMANDS THAT CAN BE INVOKED THROUGH THE ONLINE_REFINE TUPLE. ARG1 COLUMN DESCRIBES THE ARGUMENTS ASSOCIATED TO THE COMMAND. ........ 43

TABLE XX MAIN VARIABLES USED FOR ONLINE LEARNING FUNCTIONALITIES IN THE THE LEARNING LAYER NESC API. ........ 45

TABLE XXI FEATURE SELECTION ON THE ANGEN DATA: RESULTS ARE PROVIDED FOR DIFFERENT CONFIGURATIONS OF THE INPUT FEATURES, E.G. M3 MEANS THAT ALL TRANSDUCERS FROM MOTE 3 ARE PROVIDED AS INPUTS. THE GROUND-TRUTH TASK-RELEVANT FEATURES (DETERMINED BY EXPERT KNOWLEDGE) ARE MARKED IN RED. ........ 70

# 1. Overview

## 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. It assumes that the reader is already confident with the concepts described in Deliverable D2.1 about the learning mechanisms in RUBICON. On the other hand, this report is completely self-contained with respect to the contents in D2.2 (preliminary release of the CLS API V1.0). For convenience, the key differences with respect to the contents of D2.2 have been summarized in Section 1.5.

## 1.2 External view of the Learning Layer

*"Aka. Blackbox view of the Learning Layer."*

The RUBICON Learning Layer is a distributed software system, executed on a range of devices with heterogeneous computational, sensing and actuator capabilities, that provides a distributed, adaptive, and self-organizing memory for the RUBICON ecology.

The RUBICON Learning Layer processes **streams** of sensor data gathered by the ecology transducers and delivered to the distributed software components of the Learning Layer through the communication infrastructure provided by the Communication Layer, whose final version has been made available through D1.3.2 and D1.4. A Synaptic Connection mechanism is built on the top of the Synaptic Channel API (described in Sect. 2.2.1, D.3.1.1, and whose final version is in D3.1.2) provided by the Communication Layer (see Figure 1). It allows the delivery of transducer readings local to the learning module onboard a device, as well as the construction of a network of cooperating learning modules that implement a distributed neural computation over the ecology nodes. The Learning Layer further uses the "Connectionless Message Passing" mechanism of the Communication Layer (Sect. 9.2.1, D.3.1.2) to configure and control its components.

The streams of sensor data are processed by the Learning Layer to produce a set of **predictions** that concern the state of the environment/users monitored by the RUBICON ecology, e.g. events and actions plans in Figure 1. Such predictions are delivered to the Control and Cognitive layers by publishing tuples, using the Tuplespace API described in D1.3.1, Sect. 2.1 (see Figure 1.). The same Tuplespace mechanism is used by the Learning Layer to implement the Supervisor Interface, that receives control, configuration and training information from the higher levels of the RUBICON ecology.

## 1.3 Internal architecture of the Learning Layer

*"Aka. Whitebox view, opening up the Learning Layer."*

The Learning Layer is a complex software system comprising several software components, implemented with different programming languages depending on the underlying hardware and OS support. Deliverable D2.1 has provided a detailed architectural sketch of the Learning Layer software system, that has been further refined in D2.2, Sect. 1.3. For the sake of self-containment of this document, the final software architecture of the Learning Layer is reported in Figure 2.

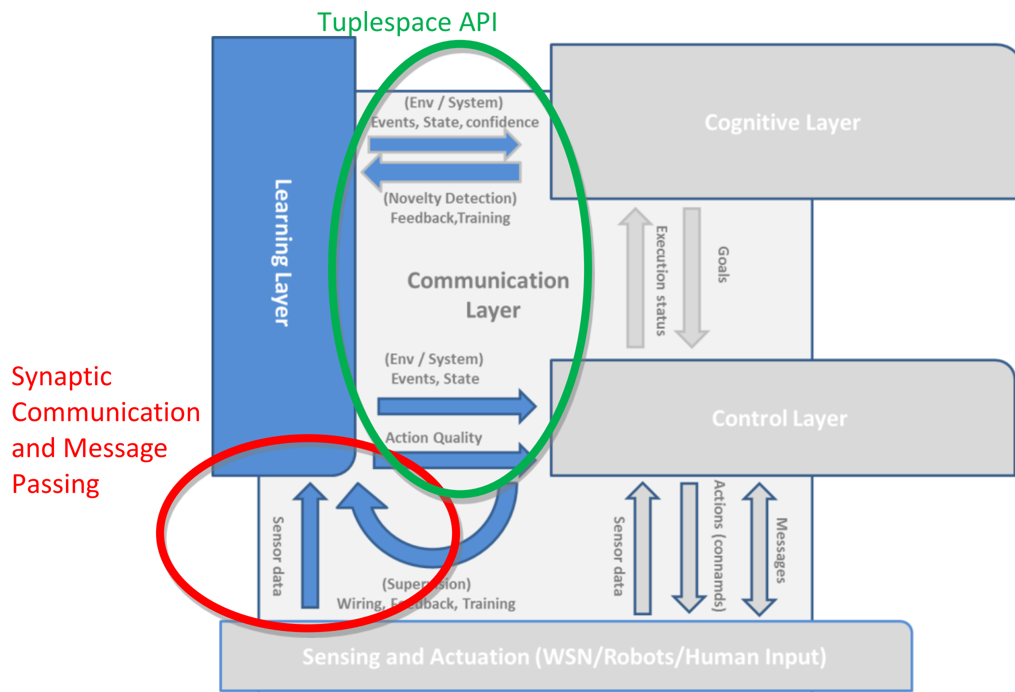



Figure *1 External view of the Learning Layer with an highlighting of the APIs interfacing with other RUBICON layers*.

The Learning Layer is organized into 3 logical subsystems, represented as light-blue boxes in the architectural sketch in Figure 2. Each subsystem is made up of a variable number of software components, depicted as simple rectangular boxes in Figure 2, that are distributed over a heterogeneous networked architecture comprising both resource constrained devices (e.g. sensor nodes) as well as more powerful Java-enabled nodes.

The Learning Layer subsystems, whose architectural detail is shown in Figure 3 to Figure 5, are as follows:

1. The **Learning Network** (LN) realizes the ecology memory by means of a distributed Echo State Network (ESN) residing on devices with heterogeneous computational capabilities, denoted as L-shaped boxes in the architectural detail in Figure 3. Each device hosts a **Learning Module**, implementing the Echo State Network (ESN), that is controlled and configured by a **Manager** component. In D2.3, these are available both as NesC software, targeted to low-power TinyOS 2.1 devices, as well as Java components, targeted to the less constrained nodes of the ecology. The **LN Wrapper** is a component that abstracts the distributed nature of the LN. It is a Java-based component that receives configuration, learning and control related information from the LNM and the TM and forwards them to the appropriate Manager components in the **Learning Network**. Further, it realizes the LN distributed neural computation by bridging and synchronizing the TinyOS and Java learning modules deployed in the ecology nodes.

2. The **Learning Network Manager** (LNM) is responsible for the configuration and control of the Learning Layer. The LNM is implemented by a single software component, the **LN Control Agent**, implemented as a Java-based agent and hosted on a gateway device, as shown in Figure 4. The **PEIS Wrapper** component, also shown in Figure 4, implements the interface between the Learning Layer and the Control and Cognitive Layer (Supervisors), realized

through the PEIS tuplespace system (see D1.3.1). The PEIS Wrapper is a Java-based software component that realizes this interface by providing mechanisms for writing into the appropriate interface tuples and for notifying the reception of messages to the appropriate Java components of the Learning Layer. The PEIS Wrapper is used by the LNM, the TM and the LN subsystems.

3. The **Training Manager** (TM) controls the learning phases of the Learning Layer: it receives training information from the Supervisor and uses it to update the Learning Modules in the LN subsystem. The TM functionalities are implemented by two Java-based software components, i.e. the **Training Agent** and the **Network Mirror**, and by a **Repository** that is used to store training data and information concerning the computational learning tasks deployed in the ecology. All the TM components are deployed on a PC, as shown in Figure 5.

## 1.4 Delivered software

### 1.4.1 Accessing the software

The software of this deliverable have been stored in the subversion repository of RUBICON and will be published on the RUBICON public website following the final project review.

### 1.4.2 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 nodes that are either WSN devices (also called *motes*) or Java-enabled devices; nodes in the same island are within direct communication range with each other.

- Each sink-mote must be connected through a USB serial to a PC running
  - The Communication Layer API from D1.3.2
  - The Proxy-Specific software from D1.4

- A PC hosting the RUBICON Gateway software (D1.3.2) and the Proxy-Specific API (D1.4)

- Zero or more nodes in each such island. A mote-class device must deploy the TinyOS based Communication Layer API (see D1.3.2 and D1.4) and the Learning Layer API. A Java-enabled device must deploy a PEIS-init and PESI-java component (see D1.3.1, D1.3.2 and D1.4) as well as the Java Learning Module component (see Section 2.2.2.2).

- One sink-node WSN mote per island deploying the TinyOS based Communication Layer API (see D1.3.2) and the Learning Layer API.

- A PC acting as RUBICON Learning Gateway that runs
  - The Gateway Wrapper software enclosed within the Learning Layer API;
  - The Training Manager and LN Manager subsystems of the Learning Layer API, together with the LN Wrapper component.
  - The Communication Layer API from D1.3.2
  - The Proxy-Specific software from D1.4
  - A PEIS-init component (D1.3.2);
  - A working PEIS-java distribution.

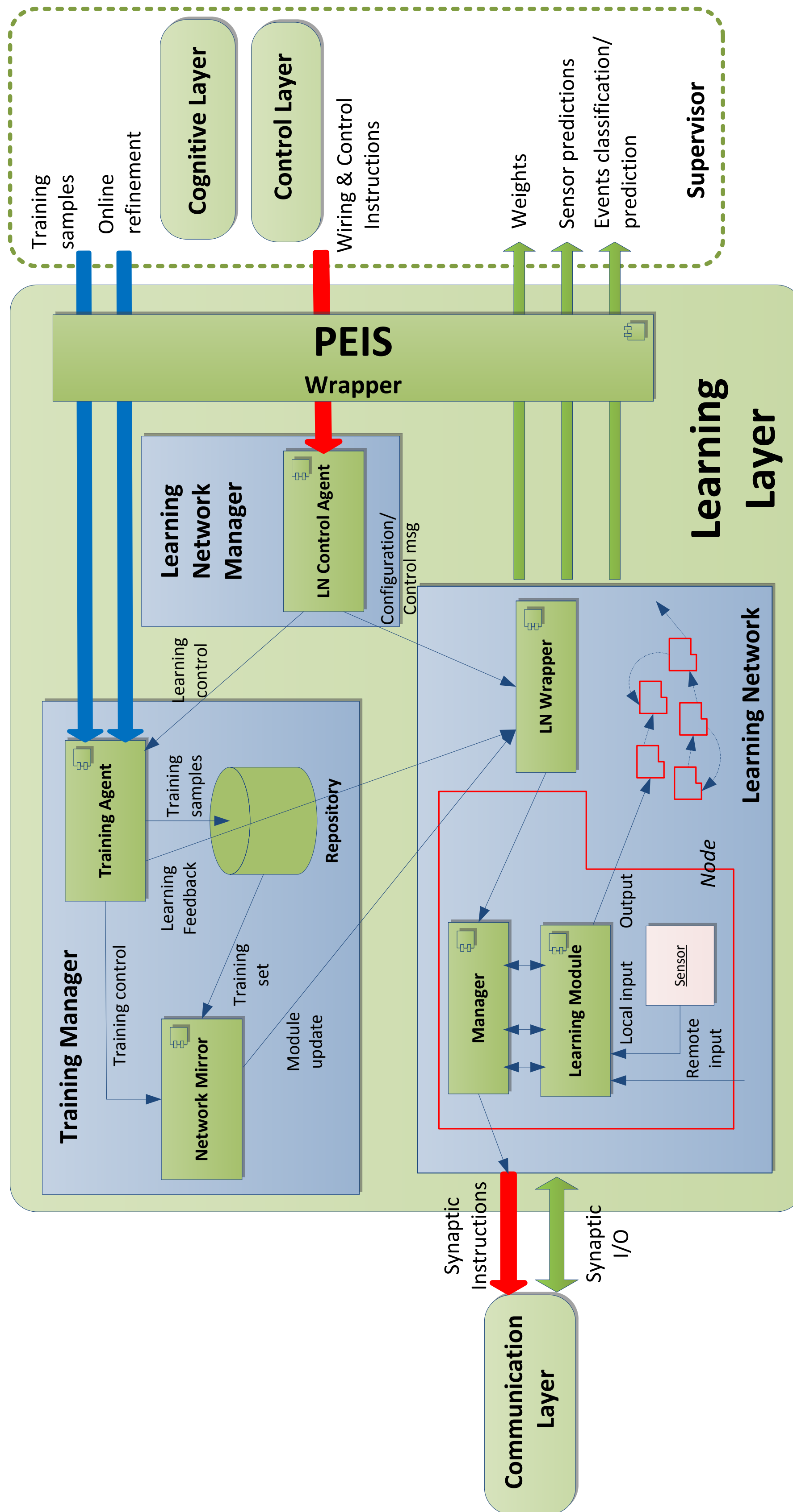



*Figure 2 Software architecture of the Learning Layer: derives from the preliminary sketch in D2.1 and refinements in D2.2. Logical subsystems are represented as simple rectangular boxes, software components are small rectangles with the UML component icon on the top-right corner, while interlayer and intralayer interfaces are denoted as thick and thin arrows.*

#### *1.4.2.1 Platform dependencies: node hosting the RUBICON Learning Gateway*

The primary target for the RUBICON Learning Gateway 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 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.

#### *1.4.2.2 Platform dependencies: nodes hosting Learning Modules*

Two classes of nodes are supported to host the RUBICON Learning modules

- WSN/WSAN motes
- Java-enabled devices.

The supported architecture for WSN/WSAN motes is 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 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.

The target architecture for Java-enabled nodes is any device capable of running the Oracle Java implementation. The software has been tested on an Ubuntu 11.10 based systems running on Intel x86 compatible hardware in 32/64-bit modes.

#### *1.4.2.3 The Learning Layer as a Distributed Software System*

The information on the system requirements provided in the Sections above conveys the picture of the Learning Layer as a distributed software system whose components (introduced in Section 1.3) are hosted by different nodes of the ecology. Here we provide a summary of the distributed allocation of the Learning Layer subsystems:

1. The **Learning Network** is, by all means, a distributed subsystem whose composing elements, i.e. the learning modules and the associated management components, resides on ecology nodes with heterogeneous computational capabilities, including both TinyOS operated devices and more powerful Java-enabled nodes. Such devices co-operate through a Synaptic Communication mechanism to realize a fully distributed neural computation that provides the predictions of the Learning Network.

2. The **Learning Network Manager** is a software component implemented as a Java-based agent and hosted on the Learning gateway device which, nevertheless, interacts with the distributed Learning Network and all high-level components of the RUBICON ecology that may be hosted by remote devices.

3. The **Training Manager** is also implemented by a Java component allocated on the Learning gateway device. Similarly to the LNM, it interacts with the distributed Learning Network for the purpose of deploying newly trained learning modules and delivering refinement teaching signals to the ESN on-board the ecology nodes. It also interacts with the remote high-level components of the RUBICON ecology for receiving learning-related information.

The CLS API V2.0 provides an implementation of a distributed learning systems for devices with heterogeneous computational capabilities that is innovative with respect to the solutions available in literature, in particular as regards the following aspects:

- Approaches in literature tend to propose the use of a learning model to solve a very specific task, whereas the CLS implements a system that can instantiate learning-based solutions to several ecology or WSN-specific tasks (what we call “general-purpose”).
- Approaches in literature typically use static learning models that do not account for the sequential nature of data or exploit linear approaches, whereas the CLS implementation founds on a model in the class of recurrent neural networks, i.e. the Echo State Network, that has been specifically designed to deal with time-series data within the class of non-linear approaches, and which is more rarely exploited in these contexts.
- Our proposal is characterized by an efficient design of the recurrent neural network models, based on ESN learning models, that are compatible with the low-computational capacity of the cheap sensor motes used in the RUBICON ecology, while they allow the deployment of more complex computational learning task on more powerful Java-enabled nodes.
- The CLS API V2.0 does not only implement a learning system, whereas it implements a whole set of functionalities and mechanisms supporting the automated system configuration and reconfiguration, including support for learning module recovery, as well as tools for facilitating access to the system, e.g. a Graphical User Interface (GUI).
- The CLS API V2.0 makes available an actual learning system that can be downloaded and installed by anyone interested. We are not aware of any software with comparable features that has been made openly available to the scientific community.

## 1.5 Differences from D2.2 Core Learning Service V1.0

This report describes the release 2.0 of the “Core Learning Service API” software, whose version 1.0 has been distributed as D2.2 in month M18 of the project. For the sake of completeness and self-containment, this report includes all the relevant information *inherited* from D2.2 and appropriately extended to document the new functionalities introduced by V2.0. This section has been designed as a quick guide to the major changes introduced in D2.3. with respect to D2.2, and it is intended for knowledgeable readers already acquainted with the contents of D2.2.

The **Learning Network** subsystem has been extended to allow a distributed neural computation comprising both NesC learning modules for TinyOS operated devices (already in D2.2) and newly-developed learning modules for Java-enabled devices. To this end, it has been developed a stand-alone component that can be run in any Java-enabled device of the Rubicon ecology and that can allocate and control several learning modules on-board the device (where the maximum number of deployable learning modules is determined by characteristics of the host). Details of the new Java component can be found in Section 2.2.2.2.

The **forward computation** phase has been extended from D2.2 to allow a distributed computation where learning modules (of any type) residing on different nodes are allowed to interact with each other: this aspect is discussed in Section 2.2.4. The introduction of Java learning modules has required the extension of the **Synaptic Communication** mechanism to enable communication between Java learning modules and/or with TinyOS modules. This has been realized exploiting the PEIS-tuple mechanisms, as detailed in Section 2.2.3.

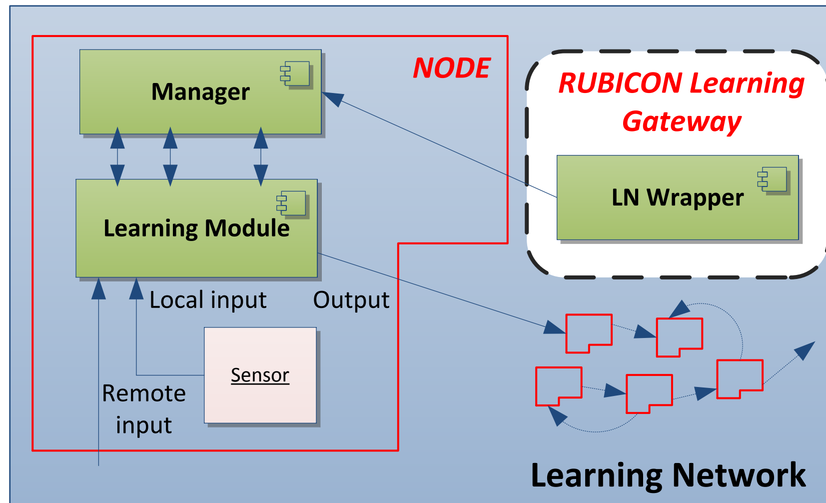



*Figure 3 Architectural detail of the LN subsystem: the Manager and Learning Module components run on ecology devices (represented as L-shaped boxes) distributed in the environment. Supported devices include mote-class and java-enabled node. The LN Wrapper component is deployed on a PC (referred to as Learning Gateway), represented as a dashed rectangle.*

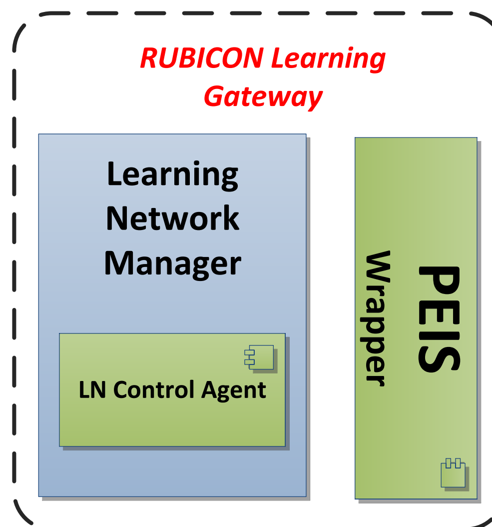



*Figure 4 Architectural detail of the LN Manager subsystem: the LN Control Agent component is deployed on the Learning Gateway. The LN Manager subsystem interacts with the PEIS Wrapper component, running on the same PC, that interfaces the Learning Layer with external PEIS-enabled components.*

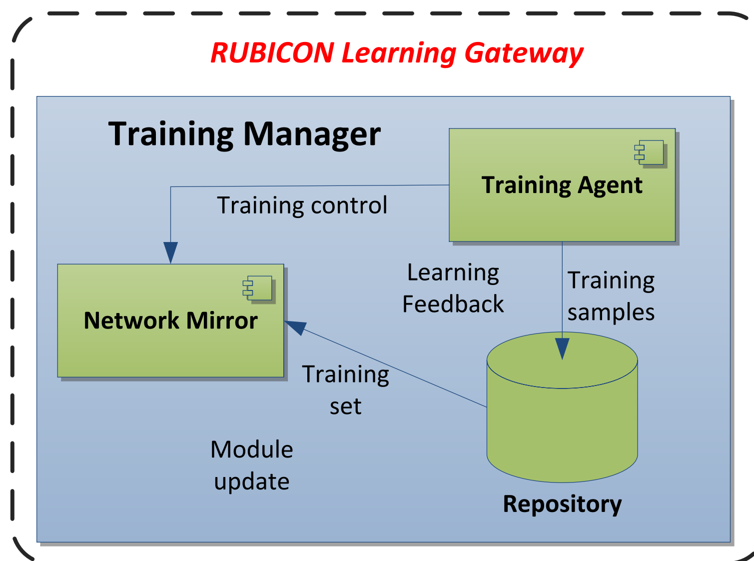



*Figure 5 Architectural detail of the Training Manager subsystem: the Repository, the Training Agent and Network Mirror components all run on the Learning Gateway, represented as a dashed rectangle.*

The **Learning Network Manager** subsystem has been extended to accommodate the changes to the Synaptic Communication mechanism, as detailed in Section 2.3.3. Further, the LNM has been provided with **recovery** routines for learning modules (TinyOS or Java) failing due to device disconnection: details can be found in Section 2.3.4.

The **Training Manager** subsystem has undergone considerable changes, mostly to accommodate novel learning mechanisms and to support Learning Layer self-configuration capabilities by automatizing the training and management of the learning modules. A **feature selection** mechanisms has been implemented to learn to automatically filter-out irrelevant inputs for a computational learning task, with the objective of reducing the associated computational and communication effort, while increasing the quality of the information provided as input of a computational learning task. A two-level feature selection system has been made available in CLS API V2.0, comprising a preliminary step of redundancy reduction based on filter methods (see Section 2.4.2.1), followed by the application of a wrapper approach tailored to optimizing the predictive and generalization abilities of the Echo State Networks (ESNs) in the LN (see Section 2.d.ii). The learning mechanisms have been extended to provide: (i) an on-mirror validation procedure to batch training and selection of the best performing ESNs configuration for a Rubicon learning task; (ii) an **online refinement** mechanism providing adaptation based on instantaneous teaching/reward information and that is performed directly on-board the learning module. These mechanisms are overviewed in Section 2.4.3.1 and 2.4.3.2, respectively.

Finally, the whole of Section 3 has been revised to document **testing and performance** of the novel functionalities introduced in D2.3 and discussed above.

# 2. Learning Layer Software System

## 2.1 Core Learning Service (CLS) API V2.0

The Core Learning Service (CLS) API V2.0 provides the **implementation of the RUBICON Learning Layer**, whose specification is discussed in detail in D2.1. The current release provides the management, control and training functionalities specified in D2.1, Sections 5.2-5.5.

The CLS API is articulated into three software libraries:

1. A Java library, i.e. the **Learning Gateway API**, comprising software that runs on a PC, referred to as the **RUBICON Learning Gateway**, and provides the implementation for the LN Manager and Training Manager subsystems, as well as the wrapper objects to access the PEIS and the LN interface.

2. A NesC library, i.e. the **NesC Learning Network API**, that provides the implementation of the distributed LN subsystem, including the Synaptic Connection mechanism, targeted to TinyOS devices.

3. A Java library, i.e. the **Java Learning Network API**, that provides the implementation of the distributed LN subsystem, including the Synaptic Connection mechanism, targeted to Java-enabled devices.

In the remainder of this section, we discuss the main functionalities implemented as part of the CLS API. Rather than discussing the three software libraries in separation, we provide a description of the functionalities following the structure of the specification document D2.1, to allow a straightforward mapping from the specification to the implementation. The key aspects and the design choices of the libraries are discussed briefly in the following two subsections, whereas a more technical reference manual of the implemented functionalities is available for download together with the code.

### 2.1.1 The Java Learning Gateway API (LGA)

The LGA implements the gateway-side functionalities of the Learning Layer: these are provided by several threads that communicate by means of message queues and dispatcher-listener mechanisms. A unique interface `LLRunnableInterface` has been defined to allow unified control over thread activation, management and termination. The dispatcher-listener mechanism (e.g. see the `LNInformationDispatcher` and `LNInformationListener` interfaces) allows any component controlling the main `LearningLayer` object (e.g. a GUI) to subscribe to its publishing service and to receive information on the status of the layer. The message queue system, on the other hand, is used internally to the layer to allow thread communication.

The LGA requires the RUBICON Communication Layer and PEIS-java libraries to be installed into the system where it is deployed. The LGA comprises 6 key packages

- `generics` – Includes the data structures shared across the Learning Layer subsystems, the standard interface of the Learning Layer threads, the inter-thread communication mechanisms as well as some general macros.
- `learningnetwork` – Includes the Java wrapper for accessing the distributed LN (both Java and NesC components) and an interface to the RUBICON gateway.
- `main` – Includes the definition of the main Learning Layer object, as well as the LN output interface and the wrapper to access the PEIS functionalities.
- `manager` – Includes the implementation of the LN Manager subsystem.

- `training` – Includes the implementation of the Training Manager subsystem by two main components, i.e. the `TrainingAgent` and the `NetworkMirror`.
- `test` – Includes the example scripts for testing and experimenting with the library.

Figure 6 shows the package diagram of the API, detailing the interfaces, the classes and the package import dependencies among the packages.

In addition to that, the software includes a Java package gui implementing a Graphical User Interface (GUI) for the configuration and control of the Learning Layer. The GUI is not officially part of D2.3 as per-DoW description, but it is released as an extra-functionality of D2.3 providing a tool to facilitate deployment and experimentation with the RUBICON Learning Layer.

The LGA depends on a set of RUBICON packages that implement the communication primitives exploited by the Learning Layer. In particular, it requires the peisjava.jar library, that provides a Java interface to the PEIS functionalities. In its current version, the LGA also needs the `tinyos`, `cnr.rubicon.cml`, `ucd.rubicon.network` and `ucd.rubicon.network` packages, redistributed along with the LGA, that serve to provide access to the Communication Layer functionalities in the RUBICON Gateway. We expect that, in future releases, these will be available as a unique jar distribution: the LGA package dependencies will be updated accordingly.

### 2.1.2 The NesC Learning Network API

The NesC LN API provides the **mote-side** functionalities of the Learning Network. These are implemented as ESN learning modules for TinyOS devices, including all the data structures and functions used for the feed-forward computation, online learning mechanisms, implementation of the synaptic connections, upload/download and activation/stop of the ESN modules. The NesC LN API also includes the implementation of the Synaptic Connection abstraction, which is used to properly route the input data towards the units of the neural network, by exploiting the Synaptic Channel mechanism delivered by the Communication Layer.

The NesC LN API requires TinyOS 2.1.1 to be installed into the devices where it is deployed, supporting both mobile and sink devices.

The NesC LN API consists in the files

- `learning.h` – Contains the data structures and macro definitions for the implementation of the learning network onboard the TinyOS devices.
- `LearningP.nc` – Contains the NesC implementation of the learning modules.
- `LearningC.nc` – Contains the NesC configuration for the learning modules.

Figure 7 describes the structure of the NesC LN API, showing the wiring among the different NesC components involved. More details on the implementation of the NesC Learning Modules are provided in Section 2.2.2.1.

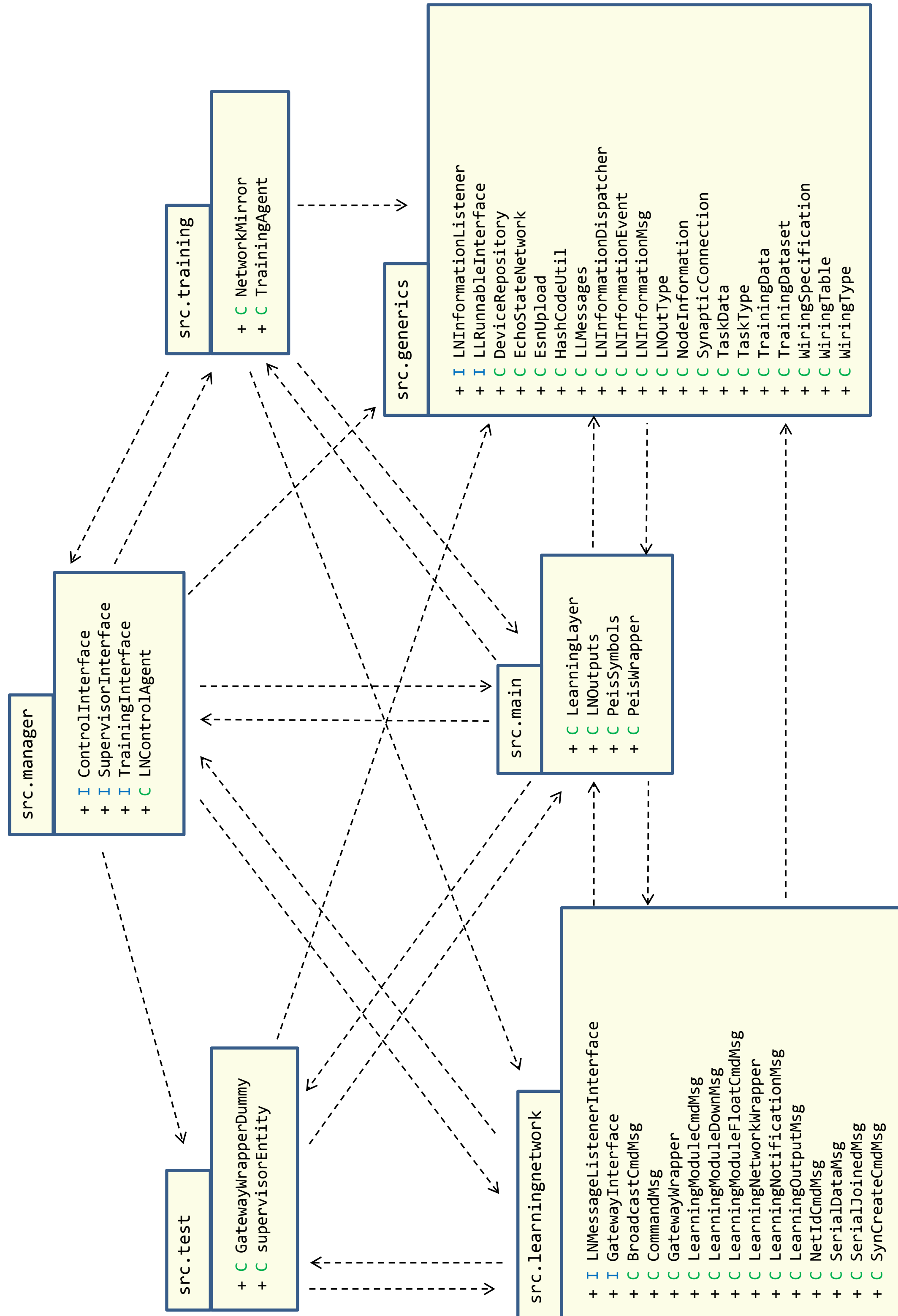



*Figure 6 The package diagram of the Java Learning Gateway API.*

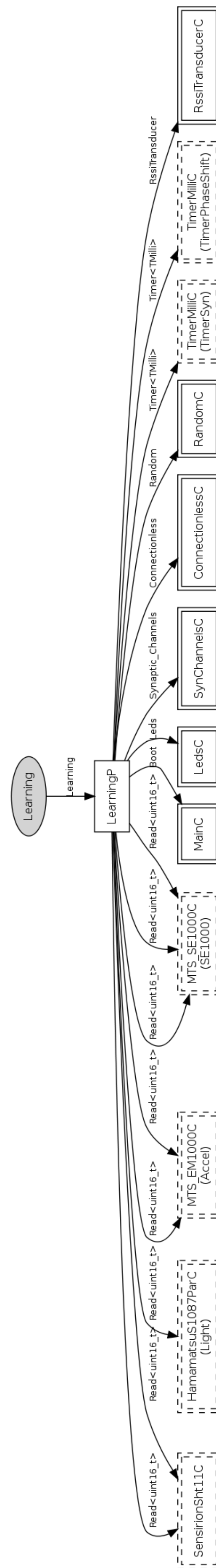



*Figure 7 Schematic illustration of the wiring among software components involved in the NesC LN API.*

### 2.1.3 The Java Learning Network API

The Java LN API provides the **PC-side** functionalities of the Learning Network, through a Java implementation of the ESN learning modules and Synaptic Connection abstraction discussed above for the NesC version.

The Java LN API is a standalone package with respect to the LNA library, but requires a working installation of PEIS-init and PEIS-java in the devices where it is deployed to.

The Java LN API is made of a single package, whose key classes are

- `PCLMmanager.java` – Defines the agent (one for each device) that manages all the learning modules running on the device.
- `LearningModulePC.java` – Defines the implementation of a learning module object, that encapsulates the ESN as well as the ingoing/outgoing Synaptic Connections.
- `EchoStateNetwork.java` – Defines the ESN object and the associated methods for training, prediction and online learning.

More details on the implementation of the Java Learning Modules are provided in Section 2.2.2.2.

## 2.2 Learning Network (LN)

### 2.2.1 Overview

The LN subsystem implements the environmental memory of the RUBICON ecology by means of a network of learning modules distributed on the ecology devices (e.g. WSN motes, gateways, PC, etc). It computes the Learning Layer predictions through a distributed neural computation (referred to as *forward computation*) realized by the single learning modules interconnected and cooperating through synaptic connections.

The functionalities of the LN subsystem are implemented partly by the Java LGA and partly by the Java and NesC LN API. The `learningnetwork` package of the LGA implements most of the gateway-side mechanisms and data-structures of the LN. The main Java component is implemented by the `LearningNetworkWrapper` class (discussed in Section 2.2.5): it provides methods to interact with the LN and allows abstracting from the technical details of its implementation and deployment. Further, the package defines an interface detailing the RUBICON gateway functionalities used by the Learning Layer (see `learningnetwork.GatewayInterface`) as well as a stand-alone component implementing the interface (see `learningnetwork.GatewayWrapper`).

The actual learning modules executed on the ecology nodes are implemented by the Java and the NesC LN APIs, where the latter is used for mote-class TinyOS devices while the former supports Java enabled devices. In addition, the two APIs provide the mechanisms supporting Synaptic Communication (see Section 2.2.3).

### 2.2.2 Learning Modules

The Java and NesC LN API provide the stand-alone implementation of the learning modules that are embedded in the ecology nodes. They comprise an implementation of the Leaky-Integrator ESN model, including functions for initialization, forward computation, online learning mechanisms, upload/download and activation/stop of the learning network. Note that the same ESN model used in the stand-alone APIs is implemented in the Network Mirror component of the Java LGA, where it serves to perform batch training of the ESNs parameters for incremental learning purposes and to maintain copies of the modules deployed in the ecology.

Both the Java and the NesC learning modules perform a distributed computation synchronized by a clock signal, referred to as the RUBICON clock. This clock is unique for all the devices in the ecology and is its clock signal is made available through the Communication Layer facilities.

In the following sections, an overview of the learning modules is provided.

#### *2.2.2.1 NesC Learning Modules*

The NesC LN API implements the learning modules for TinyOS-enabled devices, mainly through the component `LearningP.nc`. This file defines the data structures that are used to represent the ESN parameters and computational components of the network. These include input, reservoir and readout dimensions, internal weight values of the neural network (input-to-reservoir, recurrent reservoir and reservoir-to-readout weight matrices), reservoir state transition function parameters, the readout regularization parameter, online learning related data structures and parameters, and the input, reservoir state and output of the network. The weight values for the reservoir part of the ESN are encoded by resorting to a finite alphabet of possible weights.

#### *2.2.2.2 Java Learning Modules*

A stand-alone implementation of the learning modules for Java-enabled devices is provided by the Java LN API. The class `PCLMmanager` implements the agent that manages the CLS functionalities of a generic Java-enabled node of the ecology. It defines a `main()` method that subscribes the host device to the Learning Layer by posting a formatted join command on the predefined `PeisSymbols.CONTROL_CMD` tuple, passing the device address as argument of the command. More details on the use of the `CONTROL_CMD` tuple are provided in Section 2.3.2.

After the join protocol, the `PCLMmanager` agent subscribes itself to the tuple `PeisSymbols.LL_INTERNAL`, where it can receive control and configuration commands from the `LearningNetworkWrapper` component in the RUBICON Learning Gateway, including instructions for

- creation/destruction of a learning module ;
- activation/stopping forward computation ;
- loading of ESN parameters from the remote `NetworkMirror`.

The general format of the `LL_INTERNAL` tuple is provided in Table I: a varying number of comma-separated arguments are provided depending on the command. The `CMD_CODE` values for the different commands are defined in `main.PeisSymbols.java` and reported in Table II for the sake of completeness. Note that the `PCLMmanager` class provides `static` methods for each command in Table II that automatically generate the corresponding well-formatted string and post it to the `LL_INTERNAL` tuple.

*Table I Description of the tuple used to deliver control instructions from the `LearningNetworkWrapper` to the Java Learning Modules*

| **Tuple String** | **Peis Name** | **Payload (Byte[] encoding a string of 3 comma-separated int)** |
|---|---|---|
| LLInternalCmd | LL_INTERNAL | CMD_CODE,ARG1,ARG2,…,ARGN |

*Table II Encoding of the commands that can be invoked through the* `LL_INTERNAL` *tuple. The Args column describes the arguments associated to the command.*

| **Name** | **Value** | **Args** | **Description** |
|---|---|---|---|
| PC_LEARN_MODULE_CREATE | 25 | *nodeID, netID, [argEsn]* | Creates a new ESN with arguments argESN=[inputDimension, reservoirDimension, readoutDimension, leakyParameter, reservoirConnection] |
| PC_LEARN_MODULE_START | 26 | *nodeID* | Activates forward computation on the device |
| PC_LEARN_MODULE_STOP | 27 | *nodeID* | Stops forward computation on the device |
| PC_LEARN_MODULE_ACTIVATE | 28 | *nodeID, netID* | Registers the callbacks of a target learning module (for Synaptic communication) |
| PC_LEARN_MODULE_DEACTIVATE | 29 | *nodeID, netID* | Unregisters the callbacks of a target learning module |
| PC_LEARN_MODULE_LOAD | 30 | *nodeID, netID, matID, mat* | Loads a parameter matrix *mat* of type *matID* (e.g. input, reservoir, readout or scaling parameters) into a target learning module. |
| PC_LEARN_MODULE_DESTROY | 31 | *nodeID, netID* | Destroys a target learning module |
| PC_LEARN_MODULE_REFINE | 32 | *type, nodeID, netID, neurID* | Provides an online refinement signal (of the specified type) directed to a target neuron of a learning module |

The `PCLMmanager` agent allocates a new `LearningModulePC` object upon reception of a valid `PC_LEARN_MODULE_CREATE` command: therefore, every device runs a single instance of the agent but can allocate a variable number of learning modules depending on the available memory. A Java Learning Module is associated to a unique negative integer identifier `netID`. A `LearningModulePC` object encapsulates the actual learning machinery, i.e. the ESN, as well as the associated Synaptic Connections (more details on Synaptic Communication are provided in Section 2.2.3).

The Java implementation of the Leaky-Integrator ESN module is provided by the class generics.`EchoStateNetwork`, which is used both in the LGA and in the Java LN API. This class stores the parameters of the ESN, including the input, reservoir and readout dimensions, the weight values for the input-to-reservoir, recurrent reservoir and reservoir-to-readout connections. The weight matrices are represented both in a standard form, as matrices of `float`, and in a mote-version which resembles the NesC implementation, using the same weight alphabet. Other members of the `EchoStateNetwork` model the input, the state, the output of the neural network and two vectors of `float` values used for normalization of the input signals for the ESN. The class also includes the ID of

the computational task associated. Moreover this class includes the definition of all the variables and data structures required for online learning computation, in particular the forgetting lambda_forgetting, the weight update matrix Phi, and an onlineLearningMode flag. The signature of the most relevant members of the EchoStateNetwork class is provided in Table III.

*Table III The main members of the class generics.EchoStateNetwork.*

```
Class EchoStateNetwork

      //Forward Computation
      private float[] state;
      private float[] output;
      private float[] input;
      private float leakyParameter;
      private int reservoirDimension;
      private int inputDimension;
      private int outputDimension;

      //Weight values
      private float[][] Win,W,Wout;
      private short[][] WinIndices;
      private short[][] WIndices;
      private short[][] WPositions;

      //Mote-embedded representations of weight values
      private float[] weightAlphabet;
      private int weightAlphabetLength;
      private short reservoirConnections;
      private short[][] WinE;
      private short[][] Wa;
      private short[][] Wb;
      private short maxDistanceReservoir;

      //Training
      private float readoutRegularization;
      private int taskID;

      //Input scaling parameters for zero-mean unit-variance
      //normalization
      private float scalingMean[];
      private float scalingStd[];

      //Online learning related variables and data structures
      private float lambda_online; //forgetting factor
      private  float[][] Phi; //online weight update
      private int onlineLearningMode; //flag for online mode
```

### 2.2.3 Synaptic Connections

Synaptic Connections are a multiplexing/demultiplexing mechanism that is used to route input/ output information towards/from the neurons of the distributed learning network. They serve as an abstraction of the underlying communication mechanism that is responsible of the actual transmission of neural information across the ecology. Synaptic Connections are realized by the NesC and Java LN API: a brief overview of their implementation in the two APIs is provided in the following.

Nevertheless, the `LNControlAgent` component maintains a repository of `SynapticConnection` objects, that holds information on the deployed connections for management purposes.

#### *2.2.3.1 NesC Synaptic Connections*

NesC Synaptic Connections are built on top of the Synaptic Channel mechanism implemented by the Communication Layer. A Synaptic Connection is used to route the input and the output information towards the right units of the local learning network or towards the right positions in the Synaptic Channels buffers. Synaptic Connections can be remote or local.

The `inputSynConnection_t` data structure describes the input of Synaptic Connections by specifying the ID of the associated Synaptic Channel, and the index in the corresponding ChIn buffer reserved for the Synaptic Connection, the ID of the source device in the ecology and the ID of source and destination neurons for the connection. The structure definition of the `inputSynConnection_t` data structure is reported in Table IV.

Analogously, the `outputSynConnection_t` data structure describes the Synaptic Connection at the source-side, i.e. the ID of the corresponding Synaptic Channel, the index in the ChOut buffer and the ID of the source Neuron. The structure definition of the `outputSynConnection_t` data structure is reported in Table V. Both `inputSynConnection_t` and `outputSynConnection_t` are defined in the header file `learning.h`.

*Table IV Definition of the inputSynConnection_t data structure.*

```
typedef struct inputSynConnection {
      syn_ch_id_t synChannelIndex; // id of the synaptic channel
      uint8_t chIndex; // index in the synaptic channel buffer
      WSNnodeID_t sourceNode; // id of the source node
      uint16_t sourceNeuron; // id of the source neuron
      uint16_t destinationNeuron; // id of the destination neuron
      synConnectionStatus status; // status
} inputSynConnection_t;
```

*Table V Definition of the outputSynConnection_t data structure.*

```
typedef struct outputSynConnection {
      syn_ch_id_t synChannelIndex; // id of the synaptic channel
      uint8_t chIndex; // index in the synaptic channel buffer
      uint16_t sourceNeuron; // id of the source neuron
      synConnectionStatus status; // status
} outputSynConnection_t;
```

Remote input Synaptic Connections are created by the function `configure_syn_connect_in(nodeId, neuronsId_out, neuronsId_in, numNeurons, params)`, requesting the configuration of a remote synaptic connection among the `numNeurons` `neuronsID_out` in `nodeId` and the `neuronsId_in` in the local node. Analogously, the creation of an output Synaptic Connection is requested by invoking the function `configure_syn_connect_out(nodeId, neuronsId, numNeurons, params)`, which configures an output Synaptic Connection from the `numNeurons` units `neuronsId` in the local node to the node with identifier `nodeId`. Functions `configure_syn_connect_in` and configure_syn_connect_out are both defined in LearningP.nc.

In the case of a local Synaptic Connection the source of the data is represented by local sensor transducers. This particular case is handled as a special case of the remote Synaptic Connection case, and corresponds to the creation of an input and an output Synaptic Connections, in which the ID of the transducer is obtained as a special encoding of the field describing the ID of the source neuron.

#### *2.2.3.2 Java Synaptic Connections*

The Java Synaptic Connections build a similar mechanism with respect to their NesC counterpart, but they rely on a communication mechanism exploiting PEIS tuplespace. In summary, an input Synaptic Connection of a Java learning module subscribes itself to a PEIS tuple that plays the role of a Synaptic Channel: at each RUBICON clock, the tuple value is updated and the Synaptic Connection mechanism routes the updated values to the appropriate neurons similarly to what happens in NesC. Likewise, an output Synaptic Connection writes a specific PEIS tuple that allows propagating the values to other learning modules.

More in detail, the instantiation of a set of Java Synaptic Connections is initiated (as in the NesC case) by a create instruction from the `LNControlAgent` component. This request is delivered to the interested learning module by posting an appropriately formatted string on the predefined tuple `PeisSymbols.LL_SYN`, to which all the `LearningModulePC` objects subscribe at construction time (see the format in Table VI). This request is delivered in broadcast to all the Java Learning Modules in the ecology, but only the learning modules that are either source or destination will process it. In particular, the `LearningModulePC` object that is the source of the Synaptic Channel, will allocate a new tuple following a predefined naming convention. A tuple implementing a Synaptic Channel between two Learning Modules with identifier `sourceNetID` and `destNetID` will be named as

```
LL_SN + sourceNetID + : + destNetID
```

where "+" is the string concatenation operator. Then, the source `LearningModulePC` object will allocate appropriate output Synaptic Connections that will write to the tuple at each tick of the RUBICON clock, following the string encoding in
Table *VII*. In parallel, the destination `LearningModulePC` object will allocate the input Synaptic Connections that will subscribe to the same tuple and deliver its content to the appropriate ESN units.

Note that 0 is a reserved learning module identifier that is used to denote the RUBICON Learning Gateway. For instance, the prediction of an ESN with identifier `netId=-1` is delivered to the Learning Gateway (e.g. for propagation to the other RUBICON layers) through a tuple with name "`LL_SN-1:0`".

*Table VI Description of the tuple used to request the creation of Synaptic Connections whose ends include at least one Java Learning Module*

| **Tuple String** | **Peis Name** | **Payload (a string of comma-separated arguments)** |
|---|---|---|
| LLSynConCmd | LL_SYN | SourceNodeID,DestNodeID,SourceNetId,DestNetId,[SourceNeuron1#...#SourceNeuronN],[DestNeuron1#...#DestNeuronN] |

*Table VII Example of a tuple transmitting N comma-separated values from a Java Learning Module with netID=-1 to another module with netID=-2*

| **Tuple String** | **Payload (String)** |
|---|---|
| LL_SN-1:-2 | VALUE1,VALUE2,…,VALUEN |

### 2.2.4 Forward Computation

The distributed forward computation is realized by the instances of the NesC and Java LN API running on-board the RUBICON nodes and interacting through the Synaptic Communication mechanisms discussed in Section 2.2.3. Within the NesC API, this is realized by the `LearningP.nc` component: the control-flow is schematically shown in Figure 8. In particular, each time the RUBICON clock fires:

1. the input for the ESN module is read from the input Synaptic Connections using the function `read_syn_connection(inConnection);`
2. the routine `do_feedforward_step()` computes the current ESN output ;
3. the output of the ESN module is propagated through a write operation `write_syn_connection(outConnection)` into the Output Synaptic connections, from where it is further transmitted to other distributed components.

Similarly, the `LearningModulePC` component realizes the forward computation within the Java LN API, by exploiting the functionalities implemented in the class `EchoStateNetwork`. Every time a LN_SNx tuple posts new information, the `LearningModulePC` object that is the destination of the synaptic communication collects the new data and routes it to the appropriate positions of the input buffer of the associated `EchoStateNetwork` object. When the RUBICON clock fires, the `EchoStateNetwork` computes the new ESN outputs and the `LearningModulePC` takes care of propagating the outputs to other components by writing them in the appropriate LN_SNx tuples.

The outputs predicted by the various ESNs distributed in the RUBICON ecology can serve as inputs to other learning modules and/or can be published by the Learning Layer as LN predictions for the other RUBICON layers. In order for these predictions to be published, the corresponding ESN outputs distributed in the LN are routed to the `LearningNetworkWrapper` component in the RUBICON Learning Gateway, again by means of NesC or Java Synaptic Communication (see Sect. 2.2.5 for the details). The `LearningNetworkWrapper` object, in turn, forwards the ESN outputs to the `LNOutputs` component, implemented through a java class in the `main` package of the LGA, that maintains information concerning the LN predictions, including their symbolic names and their current values. These can be accessed through getter methods by any component possessing a reference to the `LNOutputs` object.

Nevertheless, the information in the `LNOutputs` component is made available to any ecology participant (including Control and Cognitive Layer) through a set of PEIS tuples. To receive information concerning all the LN predictions and their symbolic names, a PEIS-enabled participant needs only to subscribe to the tuples with key-string “LLoutputIDValue” and “LLoutputID”, respectively. Table VIII and Table IX show the format of the two tuples as well as the “PEIS Name” macro defined in `main.PeisSymbols.java.` In particular, the values predicted for K LN outputs (i.e. a float array of size K) are serialized to string of comma-separated values (one for each output) and published to as a tuple with a binary encoding as a byte array (see Table VIII). Symbolic names are concatenated into a unique string, using the placeholder “#” to separate between names (see Table IX). A time-stamp with the current RUBICON time is appended to the head of these two tuples.

The `LNOutputs` component also provides a specialized access to the LN outputs that correspond to the prediction of events. These are published by two additional tuples, with key-string "LLeventID" and "LLeventIDValue", where the former provides information on the event symbolic names, while the latter provides the predicted values. In practice, for each event, the `LNOutputs` component publishes a couple of values (and names), where the former element relates to the classification of the event, while the latter concerns the confidence about the classification. The RUBICON time-

stamp is also attached to the head of these two tuples, whose format is the same shown in Table VIII and Table IX for the “LLoutputIDValue” and “LLoutputID tuples.

Publishing of all the output-related tuples is achieved by the `main.PeisWrapper` component, which takes care of creating all the tuples and updates their value at each tick of the RUBICON clock.

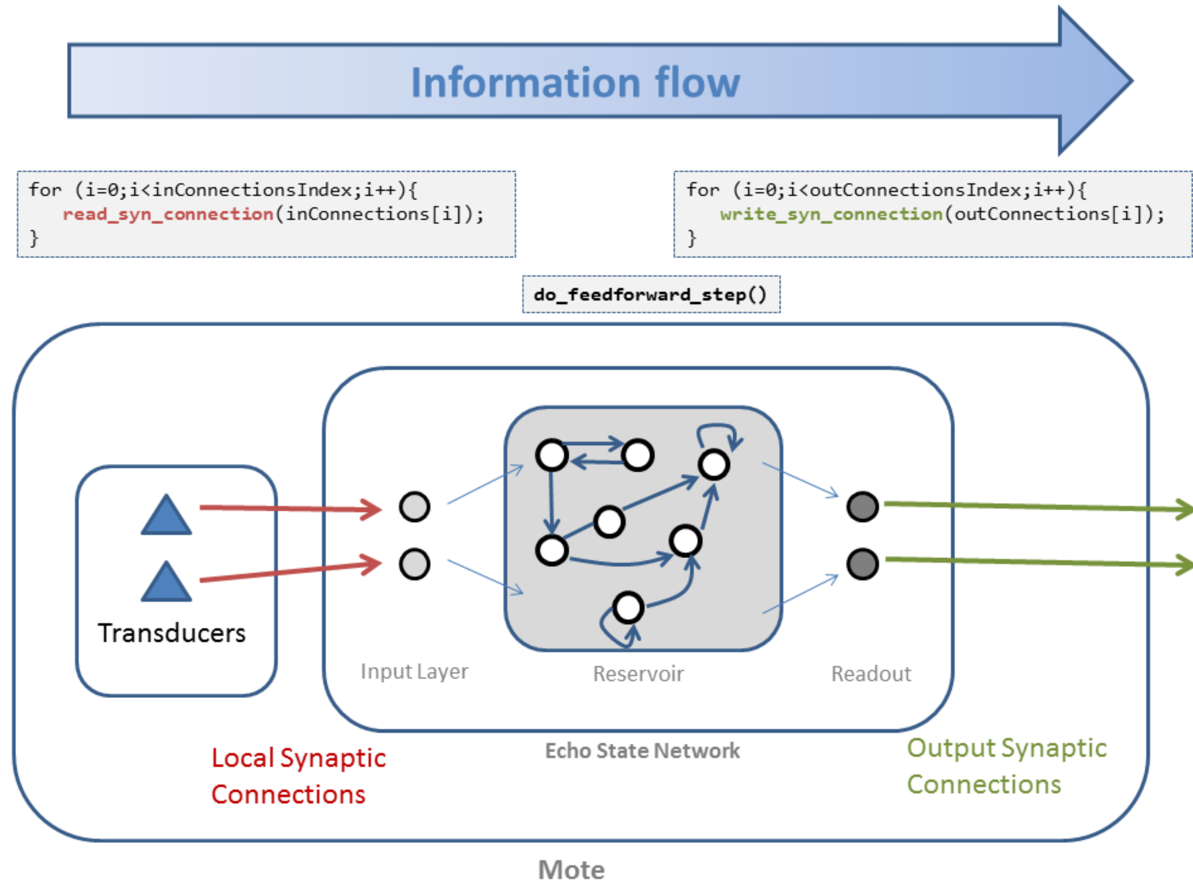



*Figure 8 A schematic illustration of the information flow in one step of the feedforward computation on-board a mote.*

*Table VIII Description of the tuple used to publish the outputs of the LN*

| **Tuple String** | **Peis Name** | **Payload (Byte[] encoding a string of comma-separated floats)** |
|---|---|---|
| LLoutputIDValue | OUTPUT_ID_VALUE | TIME,OUT_1,OUT_2, …,OUT_K |

*Table IX Description of the tuple used to publish the symbolic names of the LN outputs*

| **Tuple String** | **Peis Name** | **Payload (String)** |
|---|---|---|
| LLoutputID | OUTPUT_ID | TIME#OUT_NAME1# OUT_NAME2#...#OUT_NAMEK |

### 2.2.5 Learning Network Wrapper

The `LearningNetworkWrapper` class is defined in the `learningnetwork` package of the LGA: it provides methods to interact with the LN and allows abstracting from the technical details of its implementation. The `LearningNetworkWrapper` implements an agent that receives control and configuration commands from the `LNControlAgent` and the `NetworkMirror` components, it formats them into appropriate messages and forwards them to the LN through the most adequate communication mechanism. In particular, messages directed to NesC learning modules exploit the Connectionless Communication primitives provided by the RUBICON Gateway. The `LearningNetworkWrapper` interaction with the RUBICON gateway is defined by the methods listed in the `learningnetwork.GatewayInterface` interface. This defines a set of `send()` operations to direct messages to the NesC learning modules, as well as a list of command types. The messages directed to Java learning modules are, instead, posted by the `LearningNetworkWrapper` agent on the LL_INTERNAL PEIS tuple as discussed in Section 2.2.2.2.

The `LearningNetworkWrapper` also collects information from the LN and forwards it to the appropriate Learning Layer components: e.g. it receives the LN predictions and communicates them to the `LNOutputs` component. In fact, the `LearningNetworkWrapper` component plays a key role in implementing the distributed forward computation, as it acts as a bridge between the NesC and the Java learning modules. In particular, it enables Java-based ESN to receive inputs from any neuron belonging to a NesC learning module, and vice-versa. To this end, to receive information from the NesC side of the LN (through the RUBICON gateway), the `LearningNetworkWrapper` implements the `LNMessageListenerInterface` method `messageReceived()`. On the other hand, to receive information from the Java learning modules, the `LearningNetworkWrapper` subscribes to all the "`LL_SNsourceNetID:destNetID`" PEIS tuples whose destination identifier equals the reserved value 0. At each tick of the RUBICON clock, the `LearningNetworkWrapper` collects updated neural readings from both its NesC and Java input interfaces. Then, it demultiplexes this information by determining if these readings need to be forwarded to a NesC/Java module, and/or if they need to be sent to the `LNOuputs` component to be published as LN predictions. Figure 9 summarizes the synaptic information flow entering and leaving the `LearningNetworkWrapper` component in a scenario where a NesC learning module A provides an input to the Java learning module B.

The `LearningNetworkWrapper` agent also maintains a list of pending control and configuration commands sent to the LN (i.e. the `requestState` structure), for the purpose of supporting reliable delivery of the requests. For each non-broadcast message, the `LearningNetworkWrapper` awaits the reception of an acknowledgment of the state of the pending command, that is used to update the `requestState` structure and to inform the `LNControlAgent` and `NetworkMirror` of the outcome of their requests.

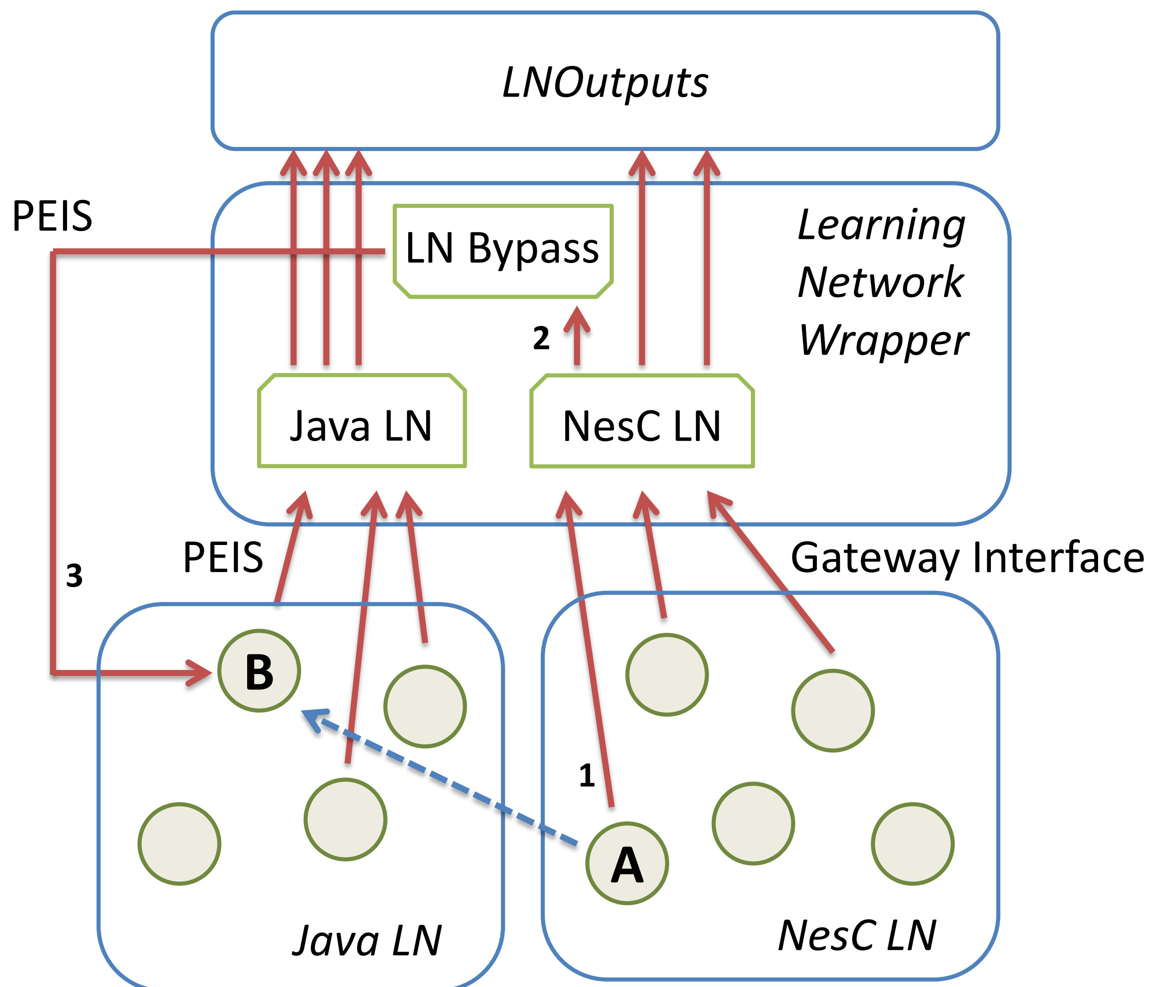



*Figure 9 Information flow in the `LearningNetworkWrapper` component: the dashed arrow represent a Synaptic Connection feeding the Java learning module B with the neuron output of a NesC learning module A (other synaptic connections are omitted for clarity). The Java-side of the LN feeds an input buffer "Java LN" through PEIS tuples, while the RUBICON Gateway provides information from the NesC –side. The information in this two buffers is routed to the `LNOutputs` component for publishing and/or to the "LN Bypass" mechanism, that feeds it back to the target module of the LN. The "virtual" Synaptic Connection A->B is implemented by arrow 1 (GatewayInterface) entering the NesC buffer, being re-routed to the "LN Bypass" with arrow 2 (local message), and being finally delivered to B through arrow 3 (PEIS tuple).*

## 2.3 Learning Network Manager (LNM)

### 2.3.1 Overview

The main goal of the LNM subsystem is to configure and manage the Learning Layer. It acts as an interface towards the RUBICON layers performing as Supervisors, by receiving their instructions (e.g. wiring information) and transforming them into control and configuration actions (e.g. synaptic connection setup) that are delivered to the appropriate Learning Layer components (e.g. the LN Wrapper).

The functionalities of the LNM subsystem are mainly implemented in the `LearningLayerApi.manager` package of the Java LGA. The `LNControlAgent` class implements the agent controlling the subsystem, which advertises 3 types of input interfaces

1. SupervisorInterface - Provides methods that are invoked by the Supervisor component.
2. ControlInterface – Provides additional methods that can be (optionally) invoked by a Supervisor component (e.g. the GUI), to have a finer grained control on the Learning Layer.
3. TrainingInterface - Provides methods that are invoked by the Training Agent in the TM.

The LNControlAgent advertises information concerning the internal status of the Learning Layer through 3 information dispatchers (LNInformationDispatcher)corresponding to the 3 input interfaces described, i.e. the superDisp, trainDisp and controlDisp fields. In addition to that, the LNControlAgent provides an information dispatcher, i.e. the gui field, that publishes data for visualization through the Learning Layer graphical user interface (see Section 2.5).

For the purpose of LN management, the LNControlAgent maintains

- synConList: a repository of synaptic connections deployed on NesC learning modules;
- synPC: a repository of synaptic connections deployed on Java learning modules;
- deviceRepository: a repository of available ecology nodes;
- pclm: a repository of deployed Java learning modules.

### 2.3.2 Supervisor Interface

This interface allows the Learning Layer to interact with the entities acting as Supervisor, e.g. other RUBICON layers or the GUI. The prototype methods that pertain to this interface are listed in SupervisorInterface.java. Additional (optional) management operations can be found in ControlInterface.java.

The set of methods pertaining to the Supervisor interface can be invoked either

- Directly, by calling the appropriate method of the SupervisorInterface (or ControlInterface) implemented in the LNControlAgent; this approach can only be used by those software entities possessing the handler to the LNControlAgent object (e.g. the GUI);
- Remotely, by posting a well-formatted request on the appropriate tuple in PEIS; this approach can be used by any software entity that is integrated in PEIS (e.g. the Control Layer or the Cognitive Layer).

A PEIS-enabled entity can invoke methods of the Supervisor Interface by means of the tuples with key-string “LLControlCmd” and “LLWireCmd”: Table X and Table XII show the expected tuple format as well as the “PEIS Name” macro defined in main.PeisSymbols.java.

The CONTROL_CMD tuple (Table X) expects a String of 3 comma-separated int, such that the first integer encodes the command, while the second and the third integers denote the (optional) command arguments. The CMD_CODE values for the different methods are defined in main.PeisSymbols.java and reported in Table XI for the sake of completeness. E.g. to place a stop_learn_module(netID) request, the Supervisor should fill the CONTROL_CMD tuple with the int value 4 (that encodes the stop command), followed by the netID value, followed by 0 (since the stop command has a single argument), e.g. the String “4,1,0” if netID = 1.

*Table X Description of the tuple used to deliver control instructions from the Supervisor to the LNM.*

| Tuple String | Peis Name | Payload (Byte[] encoding a string of 3 comma-separated int) |
|---|---|---|
| LLControlCmd | CONTROL_CMD | CMD_CODE,ARG1,ARG2 |

The WIRING_CMD tuple (Table XII) is used to post the command (see specification 5.3.1 in D2.1)

```
learn_new_task(WiringType wiringInfo, TaskType taskInfo)
```

that requests the allocation of a new learning task and provides the necessary wiring and task information through the objects of type `WiringType` and `TaskType`, respectively. These two classes pertain to the TM subsystem and their details are discussed in Sect. 2.4: they provide serialization methods to transform the object into a string representation that can be used as a payload for the tuple (Table XII). Hence a component that is willing to post a `learn_new_task()`request needs only to serialize the wiring and task information and publish it in the WIRING_CMD string.

The `LNControlAgent` object receives the commands posted in the PEIS Supervisor Interface through the `main.PeisWrapper` component, which takes care of subscribing to the CONTROL_CMD and WIRING_CMD tuples and notifies the `LNControlAgent` by invoking its local methods.

*Table XI Encoding of the Supervisor Interface commands that can be invoked through the* `CONTROL_CMD` *tuple. The Arg1 and Arg2 column describe the arguments associated to the command. The dash "-" indicates that the command does not require an argument (set it to 0 for convenience).*

| **Name** | **Value** | **Arg1** | **Arg2** | **Description** |
|---|---|---|---|---|
| ACTIVATE_FORWARD | 1 | - | - | Broadcast message to activate forward computation |
| STOP_FORWARD | 2 | - | - | Broadcast message to stop forward computation |
| ACTIVATE_MODULE | 3 | *nodeID* | *netID* | Activates a target learning module |
| STOP_MODULE | 4 | *nodeID* | *netID* | Stops a target learning module |
| RESET_MODULE | 5 | *nodeID* | *netID* | Resets a target learning module |
| CONNECT_NODE | 6 | *nodeID* | - | Connects a target device to the LN |
| DISCONNECT_NODE | 7 | *nodeID* | - | Disconnects a target device to the LN |
| SET_CLOCK | 8 | *clock* | - | Sets the RUBICON clock |
| STOP_TRAINING_DATA | 9 | *taskID* | - | Signals that no more training data will be sent for the new learning task |
| TASK_FEAT_ON | 10 | *taskID* | - | Activates feature selection for the task in argument |
| TASK_FEAT_OFF | 11 | *taskID* | - | Stops feature selection for the task in argument |
| JOIN_DEV | 12 | *nodeID* | - | Encodes the joining message of a Java-Enabled device |
| PC_LEARN_MODULE_READY | 13 | *netID* | - | Signals that the learning module is ready to provide predictions |

*Table XII Description of the tuple used to deliver wiring instructions from the Supervisor to the LNM*

| Tuple String | Peis Name | Payload (String) |
|---|---|---|
| LLWireCmd | WIRING_CMD | Wiring+Task<br><br>FORMAT = <WIRING>[OUTID]{TASK} = <s1,d1>...<sn,dn>[outid1]…[outidk]{type}{granularity}…{datatype}{outtype} |

### 2.3.3 Synaptic Communication Control and Management

The LNControlAgent provides mechanisms for the deployment of the synaptic connections and maintains updated information concerning their status. The deployment of a set of synaptic connections listSC associated to a learning task with identifier taskID is requested, by the TrainingAgent, by means of the method

```
deploy_synaptic_connections(int taskID, List<SynapticConnection> listSC).
```

The LNControlAgent determines the type of synaptic connection (local or remote, see D2.1) and forwards the appropriate requests to the LearningNetworkWrapper object. Also, it stores the synaptic connections into the synConList and synPC repositories and sets their status to synapticState.INITING. The LNControlAgent will be notified by the LearningNetworkWrapper upon the successful deployment of a synaptic connection, whose status will be then changed to synapticState.READY. Single synaptic connections can also be requested by using the create_synaptic_connections() method in the ControlInterface.

The class generics.SynapticConnection defines the synaptic connection objects used by the LNControlAgent. It maintains information concerning the type (i.e. remote, local or involving Java Learning Modules), source and destination endpoints of the connection as well as the QoS parameters, as specified in Sect. 5.2.2 of D2.1. Further, it stores management related information such as the ID of the associated learning task (optional), a deployment ID to manage its setup phase and the state information discussed above. Although the SynapticConnection object encapsulates information concerning the connection type being Java or NesC, this is not exploited by the LNControlAgent, given that the LearningNetworkWrapper object provides methods to handle them transparently.

### 2.3.4 Device and Module Management

The LNControlAgent provides mechanisms for configuration and control of the devices participating in the Learning Layer, as well as of the Learning Modules hosted on such devices.

Information concerning the devices is maintained in the deviceRepository structure (defined in generics.DeviceRepository): an element of such repository provides information on the ID of the device, on the ID of the (optional) on-board learning module and on its current state (INITING, CONNECTED, DISCONNECTING). Additionally, the repository stores information on the device capabilities in terms of on-board sensors, transducers and type of API supported (Java or NesC). Such information is communicated to the LNControlAgent through the SupervisorInterface method

```
connect_node(int nodeID, NodeInformation nodeSpec).
```

Such a notification is automatically provided by the device itself when it joins the ecology. A joining message containing the nodeSpec information by the Communication Layer to the LearningNetworkWrapper which, in turns, activates the connect_node() method above.

The nodeSpec information provided by the connect_node() is transformed by the LNControlAgent into entries of a table, named WiringTable (see Section 2.4.1), that maintains an association between transducers, actuators and neurons in the ecology and their unique symbolic name. Such symbolic names facilitate the referencing of specific information sources in the Learning Layer and are used, for instance, by the Supervisor to instruct the Learning Layer on which input sources should be used for training on a new computational learning task (see the example in Section 2.5.1). Note that, upon disconnection of a device from the ecology (method LNControlAgent .disconnect_node(int nodeID)), the LNControlAgent maintains the WiringTable consistency by removing the associated entries created at the time of node join. The table is implemented by the WiringTable object within the TrainingAgent. An up-to-date copy of the table is maintained in a PEIS tuple for the purpose of sharing it with the other PEIS-enabled components of RUBICON (see more details in Section 2.4.1). In addition to the physical devices, the LNControlAgent maintains also a repository pclm of Java Learning Modules deployed in the ecology which, from an architectural point of view, can be seen as virtual nodes of the ecology, abstracting from the fact that they can be hosted on the same physical device.

The LNControlAgent provides methods (in its SupervisorInterface) to trigger the activation, stop and reset of the single learning modules distributed on the ecology device (according to specifications 5.2.4.4-5.2.4.6 in D2.1). The activation of a learning module netID on the device nodeID is achieved through the method activate_learn_module(nodeID, netID): note that this is a NECESSARY step in order to enable the reception of other Learning Layer commands on the device, which will be otherwise ignored. Activation and stopping of the forward computation in the LN is triggered by calling the LNControlAgent methods activate_forward_computation() and stop_forward_computation(). This result in a broadcast message sent by the LN Wrapper and processed by all the learning modules that have received the activation message.

The LNControlAgent implements a best-effort recovery mechanism catering for the disconnection of devices embedding deployed learning module. The sudden disconnection of a device and its associated learning module may result in the LN failing to provide some of its predictions (e.g. if these were computed by the failed module) or can affect the predictive performance of other learning modules that were using information produced by the failed device. The recovery routine is triggered when the LNControlAgent receives from the LN a NODE_FAIL message for a device that is currently in use by the Learning Layer. First, the LNControlAgent tries to determine if any of the joined devices is a compatible replacement for the missing node. To this end, it uses the getCompatibleDevice(capability) method of the deviceRepository component, that determines which devices in the repository possess the sensing capabilities passed as argument. If a compatible device is available, the LNControlAgent activates the RecoveryESN() routine, that retrieves the NetworkMirror copy of the learning module in the failed device and transfer it on-the-fly to the replacement node. Then, the RecoverySC() routine takes care of recovering the Synaptic Connections and Channels needed to put the replacement learning module into operation. As soon as the Synaptic Communication is in place, the LNControlAgent activates the replacement learning module that starts participating to the distributed forward computation. The recovery routine is performed while the rest of the LN keeps operating, with no interference on its distributed computation. The LNControlAgent also allows to place a watchdog on selected Synaptic Connections through its setRec() method. The watchdog triggers the recovery routine if the values read by the watched Synaptic Connections are not updated for a predefined timespan.

## 2.4 Training Manager

### 2.4.1 Overview

The Training Manager is implemented by two main software components, which are the Training Agent and the Network Mirror. These two components implement all the functionalities required to train and deploy a computational learning task to the RUBICON ecology. Figure 10 shows how the management, control and training functionalities of the Training Manager subsystem are partitioned among these two components. At a high level, the `TrainingAgent` relies on the `NetworkMirror` for training and model selection of the learning modules implementing a computational learning task. On the other hand, the `NetworkMirror` exploits the functionalities of the `TrainingAgent` to deploy a trained computational learning task to the physical ecology. A detailed description of the implementation of the functionalities in Figure 10 is provided in the remainder of this section.

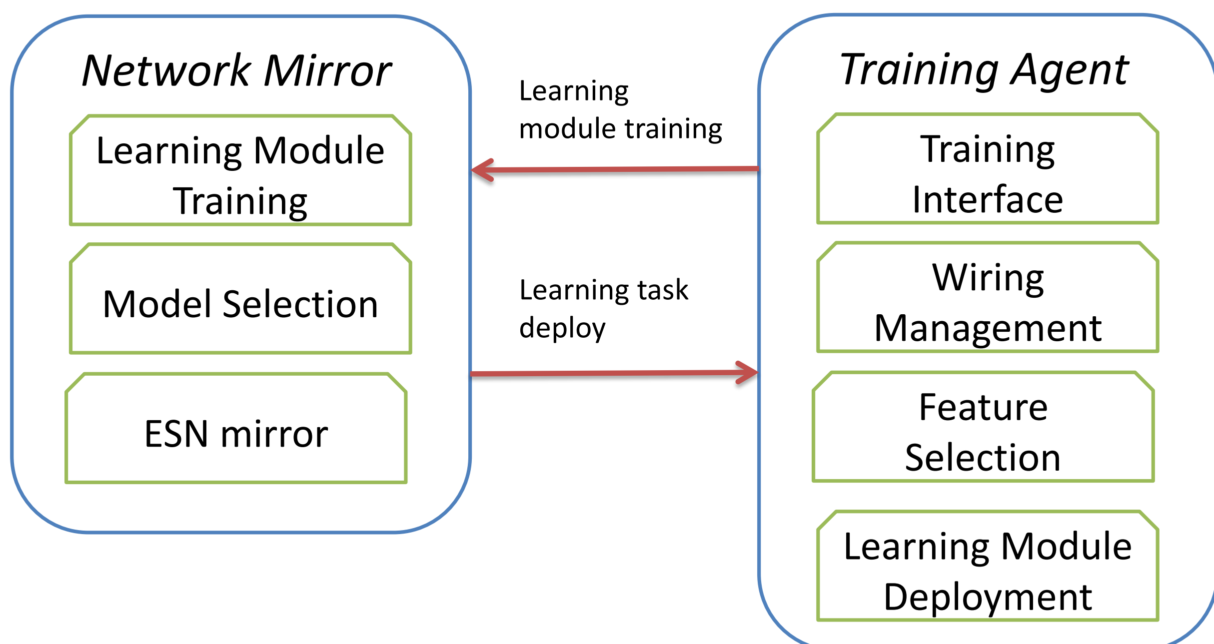



*Figure 10 Summary of the functionalities implemented by the two components of the Training Manager subsystem. The* `TrainingAgent` *relies on the* `NetworkMirror` *for the training and model selection of the learning modules. On the other hand, the* `NetworkMirror` *exploits the functionalities of the* `TrainingAgent` *to deploy a trained computational learning task to the physical ecology.*

The Training Manager receives wiring and task information for the creation and deployment of new computational tasks. In order to manage the wiring instructions, the Training Manager manages a Wiring Table (see an example in Table XIII) to convert the wiring instructions into a set of Synaptic Connections.

Wiring instructions are implemented by the class `WiringType`, containing `ArrayList` of symbolic names for source, destination (in couples) and output entities for the wiring instruction. The entries of the Wiring Table are implemented by the class `WiringSpecification` which, for each element, specifies the symbolic name, the ID of the node, the ID of the ESN, the ID of the unit within the ESN and some possible associated information.

*Table XIII Example of a wiring table, used by the Training Manager to convert wiring instructions into synaptic connections.*

| Symbolic Name | Node ID | Net ID | Sensor/Neuron ID | Associated Information (optional) |
|---|---|---|---|---|
| T_LIGHT_1 | 1 | 0 | 1 | |
| EVENT_FIRE2 | 4 | 8 | 45 | |
| LOCATION_PREDICT_X | 7 | 10 | 30 | |
| LOCATION_PREDICT_Y | 7 | 10 | 31 | |
| … | … | … | … | … |

Task information are specified through the class `TaskType`, which entails information on the computational task, namely the task type (a String which may be equal to "classification" or "regression"), the granularity (a String which may be equal to "sequence-to-sequence" or "sequence-to-element"), the data type (a String which may be equal to "event" or "sensory") and the output type (a String which may be equal to "weight", "event", "sensor").

Training data are used to train the ESN modules. Training examples are implemented by the class `TrainingData`. In the constructor of the class `TrainingData`, the input and the desired (target) output are specified through two vectors of float. An ArrayList of IDs of the tasks to which the training sample is associated should be provided as well. Training datasets are implemented by the class `TrainingDataset`, which entails an ArrayList of `TrainingData` objects.

The Training Agent is implemented by the class `TrainingAgent`. It manages a `WiringTable` object in order to realize the functionalities related to the wiring instructions conversion. Wiring instructions in the form of `WiringSpecification` objects can be added to the wiring table by using the method `add_wiring_specification(w)`. Analogously, the entries in the wiring table can be removed by invoking the method `remove_wiring_specification(w)`. As anticipated in Section 2.3.4, the `WiringTable` content is dynamically updated to reflect the current LN situation. In particular, the `LNControlAgent` ensures that appropriate wiring entries are added/deleted from the table when a device joins/leaves the ecology. Similarly, the `TrainingAgent` ensures that, whenever a new learning task is deployed to the LN, the `WiringTable` is updated using the information provided in the associated wiring instructions. In particular, the `allocate_wiring_entries(-,-,WiringType wiring)` method ensures that the Wiring Table is updated with the symbolic names of the ESN inputs and outputs, that are received from the Supervisor through the `wiring` argument.

The `TrainingAgent` also maintains a shared, up-to-date copy of the `WiringTable` object in the `PeisSymbols.WIRING_TABLE` tuple, to allow any PEIS-enabled RUBICON component to access it (e.g. the Supervisors). To this end, it suffices to subscribe to the `WIRING_TABLE` tuple, that is updated every time the local copy of the `WiringTable` object is modified by the Learning Layer. Serialization of the `WiringTable` object to the `PeisSymbols.WIRING_TABLE` tuple is obtained by concatenation of the string generated by the `WiringSpecification.toPeisString()` method for each table entry (see Table XIV).

The Wiring Table can be also modified by RUBICON layers other than the Learning Layer, by posting a request in the `PeisSymbols.WIRING_TABLE_UPD` tuple. The format of the tuple string is shown in Table XV: the value of the ACTION field determines if the request is for addition (i.e. ACTION = "+") or removal (i.e. ACTION = "-") of the entry passed as argument. The update of Wiring Table object is performed in a thread-safe fashion by the `TrainingAgent` and its later propagated to the PEIS copy in the `WIRING_TABLE` tuple. The `TrainingAgent` also provides a multi-threaded implementation of

the Wiring Table update actions that exploits worker threads to avoid overloading of the caller thread in case of particularly extensive modifications of the table.

*Table XIV Description of the tuple used to publish the Wiring Table*

| **Tuple String** | **Peis Name** | **Payload (String)** |
|---|---|---|
| LLWiringTable | WIRING_TABLE | {SymbolicName1,nodeId1,netId1,neuronId1,[AssocInfo1]} {SymbolicName2,nodeId2,netId2,neuronId2,[AssocInfo2]} … |

*Table XV Description of the tuple used to request a Wiring Table Update: if ACTION = "+" the following wiring entry is added to the table; if ACTION = "-", it is removed.*

| **Tuple String** | **Peis Name** | **Payload (String)** |
|---|---|---|
| LLWiringTable Update | WIRING_TABLE_ UPD | ACTION,{SymbolicName,nodeId,netId,neuronId,[AssocInfo]} |

The Training Agent manages a collection of tasks, through an ArrayList of `TaskData` objects. A new task is allocated by calling the method `allocate_task(wiring, taskInfo, nodeID)`, where the wiring instructions, the information on the task and the ID of the RUBICON node on which to deploy the task, are specified respectively by the `WiringType` object `wiring`, the `TaskType` object `taskInfo` and the integer `nodeID`.

All the information pertaining to a computational task and necessary for the Training Agent, is collected within the class `TaskData`, which comprises a `WiringType` object, a `TaskInfo` object, a `TrainingDataset` object (collecting the training data that should be used for training on the task), a List of `SynapticConnection` objects (in which the wiring instructions can be converted using the wiring table) and a status. The `TrainingAgent` class manages a repository of tasks and related information by containing an `ArrayList` of `TaskData`. Training data examples are passed to the Training Agent by the function `new_training_data(sample)`, which stores the `TrainingData` object sample in the `TrainingDataset` object associated to all the tasks with identifiers indicated in the sample itself.

The Network Mirror is implemented by the class `NetworkMirror`. This class contains an ArrayList of `EchoStateNetwork` objects (which correspond to the deployed ESN modules), and manages a queue of incoming messages from an associated `LearningNetworkWrapper` object. The creation of a new ESN is requested to the Network Mirror by invoking the method `addESN(taskID, inputDimension, outputDimension, nodeID)`, which creates and stores a new `EchoStateNetwork` object using the specified information. When the creation of a new ESN is requested, the `WiringTable` object in the `TrainingAgent` is also updated with the neural network ID assigned by the `NetworkMirror`.

### 2.4.2 Feature Selection

The `NetworkMirror` component provides feature selection functionalities to learn to filter irrelevant inputs for the learning modules, contributing to enforce the self-configuration capabilities of the Learning Layer by avoiding the deployment of unnecessary inter-node synaptic communication and, thus, reducing the communication and computational burden of the distributed forward

computation. To this end, the `NetworkMirror` component implements a two-level feature selection system, comprising

1. A redundancy reduction procedure, realized through a filter algorithm using linear cross-correlation time-series analysis to identify and remove those input streams providing overlapping information.
2. A relevance-guided procedure tailored to optimizing the predictive and generalization abilities of the LN, realized through a wrapper algorithm using model selection guided by a search procedure that uses the outcome of the redundancy reduction step.

The details of the two feature selection procedures are provided in Section 2.4.2.1 and 2.4.2.2, respectively.

#### *2.4.2.1 Feature Filter*

The redundancy reduction procedure is implemented by the `training.FeatureFilter` class of the Java LGA. This implements a forward selection-elimination algorithm (in the `runFilter(TrainingDataset dataset)` method) that filters out redundant inputs measured by a linear time-series cross-correlation metric. The algorithm is based on the iterative application of a set of 4 selection/elimination rules, backed-up by the following intuitions

- A variable that is not correlated with any of the other features, should be selected.
- A variable that is correlated with all the variables that have already been selected is a good candidate for elimination.
- If we are left with a set of mutually correlated variables, act conservatively and maintain all those features that are less correlated with the selected ones.

The skeleton of the algorithm is as follows

1. Compute a Boolean matrix of feature redundancy, i.e. `computeRedundancyMasks(TrainingDataset T)`:
    a. For each sample in the dataset, compute the maximum absolute normalized cross-correlation between the pairs of input sequences . If a pair has a sufficiently high value, assume the features are correlated on the sample.
    b. Compute the percentage of training samples in which each pair of feature is correlated.
    c. Set an entry (i,j) of the Boolean matrix to true, if the corresponding pair of features i and j is correlated on more than TH_SEQP% (e.g. 20%) of the training samples.
2. Iteratively apply, until termination, the following selection-elimination rules (in order of application priority)
    a. RULE 0 - If any of the rows in the matrix is uncorrelated with all the others:
        i. save the corresponding features in the selected subset;
        ii. update the matrix by removing the row and column corresponding to the saved features;
        iii. If an uncorrelated feature is generated as result of the previous step, remove it from the matrix and save it to the deleted subset.
    b. RULE 1 - If any of the rows in the matrix is correlated with all the others and the matrix does not contain only true values:
        i. put the corresponding features in the deleted subset;
        ii. update the matrix by removing the row and column corresponding to the deleted features;

c. RULE 2 - If all the features left are mutually correlated with each other, i.e. the matrix contains only true values,
   i. Select the feature that is less correlated with those saved until now in the selected subset;
   ii. Move the rest of the features to the deleted subset and terminate.
d. RULE 3 - If neither of RULE 1 or RULE 2 apply,
   i. extract the row i that is correlated with the minimum number of other features still in the matrix;
   ii. define S(i) as the subset of features correlated with i and select feature j in S(i) as the maximally correlated with those saved until now in the selected subset;
   iii. save feature i to the selected subset and move feature j to the deleted subset;
   iv. update the correlation matrix by removing rows and columns corresponding to i and j.

Once that the filter algorithm is converged, the `NetworkMirror` can query the `FeatureFilter` object to obtain the set of select/deleted features (e.g. `get_selected_features()`) and to retrieve a clean `TrainingDataset` object without redundant features (i.e. `getFilteredDataset(TrainingDataset dataset)`).

#### *2.4.2.2 Feature Wrapper*

The `training.FeatureWrapper` class implements a *supervised* feature selection mechanism, which exploits model selection (see Section2.4.3.1) to reduce the input dimensionality of a given training dataset while aiming at optimising the predictive performance of the learning module.

Given a specific training `dataset` (an object of the class `TrainingDataset`), the Feature Wrapper generates a number of datasets derived by `dataset`, by progressively reducing the input dimensionality. A model selection pass is then applied to choose the set of input features which leads to a smaller error on a validation set. The clean dataset is retrieved by calling the method `getFilteredDataset(TrainingDataset dataset, TaskType taskType)`. The indices of the selected features then be retrieved by calling the method `get_selected_features()`.

### 2.4.3 Training of a Learning Module

The ESN modules composing the RUBICON Learning Network can undergo two different processes of training, i.e. offline training and online training. Offline training is managed through the functionalities implemented in the Network Mirror component, allowing to perform a standard training of the ESN modules once a dataset has been made available. Online training functionalities, implemented both in the NesC and Java API, allow any entity, acting as Learning Layer Supervisor, to request an adjustment of the output of the learning modules while they are producing their predictions (i.e. in their forward computation mode). Note that both the mechanisms can be used for the refinement of already learnt tasks or for the incremental learning of new computational tasks.

#### *2.4.3.1 Training through the Network Mirror*

Local and distributed offline training is managed by the functionalities implemented by the Network Mirror. Offline training requires a dataset (suitable for supervised learning) to be collected and provided to the Learning Layer.

Training samples are provided by the Supervisor by publishing a PEIS tuple with key-string `LLTrainingSample`, following the format specified in Table XVI. The format of the `String` in the

Payload of this tuple is detailed in Table XVII. Alternatively, training information can be passed through a direct call to the method `new_training_data` in class `TrainingAgent`, specifying an object of the class `TrainingData` as argument.

*Table XVI Description of the tuple used to deliver training samples from the Supervisor to the Learning Layer.*

| Tuple String | Peis Name | Payload (String) |
|---|---|---|
| LLTrainingSample | TRAINING_SAMPLE | FORMAT = <input>[output]{taskID_1,…, taskID_k} |

*Table XVII Description of the fields in the String Payload for the PEIS tuple with key-string LLTrainingSample.*

| Field | Description |
|---|---|
| input | A `float[]` input value converted to `String` |
| output | A `float[]` target output value converted to a `String` |
| taskID_i | The `int` ID of the i-th task associated to the training example, converted to a `String` |

The Supervisor can signal to the Learning Layer that the process of data gathering for a specific task has terminated, by posting a `STOP_TRAINING_DATA` command (see Table XI), specifying the appropriate `taskID` as an argument. This triggers the invocation of the method `do_training(int taskID)` in the `TrainingAgent` class. Training is performed only if training data for the task has already been provided. In this case the `TrainingAgent` retrieves the `TrainingDataset` corresponding to the task and calls the method `do_training(taskID, T)` in the `NetworkMirror`, specifying the ID of the task and the `TrainingDataset`.

Before the actual training process takes place, appropriate scaling constants are computed for each input dimension in the training dataset, using the function computeScaling Constants(`TrainingDataset T`) in class EchoStateNetwork. Training is then performed in the `EchoStateNetwork.train()` method, using standard offline linear methods, i.e. Moore-Penrose pseudo-inversion and ridge regression [1]. When training on a specific task has been completed, the corresponding learning modules are deployed in the RUBICON Learning Network (see Section 2.4.4).

The Java API also includes a mechanism for model selection. Using such mechanism, among a set of different ESNs with different hyper-parameters values, it is possible to choose a specific ESN module which minimizes an error function on a validation set. Several model selection configuration settings are possible, ranging from an extensive cross validation, to a setting without model selection. The cross validation type is specified through the enumeration `CvType` in the class `TrainingAgent`, it can assume the values `MOTELM` (a simple scheme designed for learning modules embedded on the motes, i.e. considering small dimensional reservoirs), `JAVALM` (a simple scheme designed for PC learning modules, i.e. considering also large dimensional reservoirs), `GENERAL` (general extensive cross validation, i.e. considering the largest grid for the values of the hyper-parameters) and `NONE` (no cross validation). The cross validation scheme can be set by using the method `setCvConfiguration` in the class `TrainingAgent` (default value is `GENERAL`). The model selection procedure is implemented by the method `CrossFold_Validation_training` in the class `NetworkMirror`. Such method exploits the functionalities developed in the helper class

Trainer, which provides methods for organizing the data-folds and the search-grid for the ESN hyper-parameters, as well as for the computation of the validation and test performance of the ESN modules.

#### *2.4.3.2 Online learning*

Learning modules are equipped with online learning mechanisms both in the NesC API and in the Java API. Such mechanisms are designed to allow the Supervisor to provide online teaching signals to the Learning Layer for adjusting the predictions pertaining to specific tasks and are based on the Recursive Least Mean Square (RLMS) [2] algorithm[1].

The Supervisor can activate the online learning mechanisms by posting a specific command for online learning through the tuple with key-string “LLOnlineRefine” (whose macro is in PeisSymbols.ONLINE_REFINE). Table XVIII shows the expected tuple format and the PEIS name defined in the class main.PeisSymbol.

An ONLINE_REFINE tuple expects a String of 2 comma-separated values, where the first one is an int that encodes the online learning command, while the second is a String that specifies the symbolic name of the output prediction to which the online learning command is directed (see Table XVIII). The CMD_CODE values for the different methods are defined in main.PeisSymbols and reported in Table XIX.

*Table XVIII Description of the tuple used to deliver online learning instructions from the Supervisor to the LNM.*

| **Tuple String** | **Peis Name** | **Payload (Byte[] encoding a string of one int and one String separated by a comma)** |
|---|---|---|
| LLOnlineRefine | ONLINE_REFINE | CMD_CODE,ARG1 |

*Table XIX Encoding of the Supervisor Interface commands that can be invoked through the* ONLINE_REFINE *tuple. Arg1 column describes the arguments associated to the command.*

| **Name** | **Value** | **Arg1** | **Description** |
|---|---|---|---|
| ONLINE_EVENT | 14 | *predictionName* | Activates the online learning functionalities for the LL output prediction corresponding to the specific symbolic name predictionName, in the case of occurrence of an event (i.e. classification task with positive target) |
| ONLINE_NOEVENT | 15 | *predictionName* | Activates the online learning functionalities for the LL output prediction corresponding to the specific symbolic name predictionName, in the case of non-occurrence of an event (i.e. |

[1] The RLMS algorithm implemented here for the case of regression tasks is a variant of the standard RLMS algorithm [1], in which the target output is modulated by a delta_output value, which can be used to increase or decrease the output of the ESN module.

| | | | |
|---|---|---|---|
| | | | classification task with negative target) |
| ONLINE_INCR_FEEDBACK | 16 | *predictionName* | Activates the online learning functionalities for the LL output prediction corresponding to the specific symbolic name predictionName, when the output of the module is requested to be incremented (used e.g. for regression tasks) |
| ONLINE_DECR_FEEDBACK | 17 | *predictionName* | Activates the online learning functionalities for the LL output prediction corresponding to the specific symbolic name predictionName,, when the output of the module is requested to be decremented (used e.g. for regression tasks) |
| STOP_ONLINE | 18 | *predictionName* | Stops the online learning functionalities for the LL output prediction corresponding to the specific symbolic name predictionName. |

Possible commands are:

- ONLINE_EVENT, used to request online learning for an event classification task which is occurring;
- ONLINE_NOEVENT, used to request online learning for an event classification task which is not occurring;
- ONLINE_INCR_FEEDBACK, used to request online learning for a regression task (e.g. weight prediction), when the output of the learning module is considered too small;
- ONLINE_DECR_FEEDBACK, used to request online learning for a regression task (e.g. weight prediction), when the output of the learning module is considered too large;
- STOP_ONLINE, used to stop online learning mechanisms for a specific task.

When an ONLINE_REFINE command is received, the `TrainingAgent` takes care of transforming the symbolic name of the output prediction into a `nodeID`, `netID`, `neuronID` address, by accessing its Wiring Table. The online-learning method in the `LearningNetworkWrapper` component that matches the command code in the tuple is then called, e.g. `ln_online_event(nodeID, netID, neuronID)`. This method checks if the online learning command is directed to a TinyOS or to a Java learning module. In the former case a proper TinyOS message is sent to the target mote, through the `GatewayInterface` implementation. In the latter case, an appropriately formatted command is posted on the `PeisSymbiols.LL_INTERNAL` tuple (see Section 2.2.2.2), to signal the online refinement to the target `PCLMmanager` object.

In the NesC API, the online learning mechanism is implemented in the file LearningP.nc. The main variables involved are reported in Table XX.

*Table XX Main variables used for online learning functionalities in the the Learning Layer NesC API.*

```
//Variables involved in online learning

uint8_t onlineLearningMode; //flag for online learning

float delta_output; //delta used to modulate the target for

                    //regression tasks

float lambda_online; //forgetting factor
```

Each time the RUBICON clock fires, the value of the `onlineLearningMode` flag is used to selectively activate the online learning functionalities (whenever `onlineLearningMode` is set to 1) or the standard feedforward computation (see Section 2.2.4). The forgetting factor `lambda_online` can assume strictly positive values not larger than 1. In our implementation we use a value `lambda_online = 0.999`, as in [2].

When online learning is activated, at each RUBICON clock, the function `do_onlineLearning_step()` is called in place of the standard `do_feedforward_step()` method. The former includes a step of forward computation followed by a step of weights' update for the connections from the reservoir to the readout, according to the RLMS algorithm. Such algorithm determines an update for the readout weights values, based on the state of the reservoir, which encodes the recent history of the ESN inputs (defined by the wiring instructions associated to the tasks), and on the online output target.

In the Java API, the same online learning functionalities are implemented in the class `EchoStateNetwork`. Analogous data structures and variables as in the NesC implementation (see Table XX) are used, and one step of the RLMS algorithm is implemented in the method `onlineLearningStep`().

Both in the NesC and Java implementation of the RLMS, the computation of the weight update depends on the value of a target output which is set according to the last online learning instruction that has been received. For classification tasks (e.g. event predictions) the target value is set to +1 or -1 in correspondence to the commands `ONLINE_EVENT` or `ONLINE_NOEVENT`, respectively. In case of regression tasks (e.g. weight predictions), the target is set to a value which is equal to the value of the output of the model incremented or decremented by a `delta_output` quantity (set to 0.1 in initialization phase), for the commands `ONLINE_INCR_FEEDBACK` or `ONLINE_DECR_FEEDBACK`, respectively. Note that this strategy allows also to treat the case of reinforcement learning as a special case in which the Supervisor simply requests to increase or decrease the output of the ESN depending on the success or failure of the action resulting from the prediction.

### 2.4.4 Deployment of a Learning Module

Once that the model selection and training processes have been completed, the `NetworkMirror` needs to deploy the resulting learning module to a physical device. This entails the selection of an appropriate device for hosting the learning module, the transfer of the matrices holding the trained ESN parameters and the instantiation of appropriate synaptic connections to feed the input neurons and to propagate predictions from the output neurons. A short summary of the technical actions needed to implement this process, is provided in the following.

The successful completion of the training procedure on a task with handler taskID is signalled by the NetworkMirror to the TrainingAgent, by calling the method training_complete(taskID). Subsequently, the TrainingAgent method deploy_task(taskID) is invoked to start the deployment of the learning modules. Within this method, the TrainingAgent uses the information in the WiringTable to find the best device to store the new computational learning task. In particular, the allocate_nodeID_from_wiring(int netID, WiringType wiring) method seeks a matching between the wiring instructions for the task (i.e. the wiring argument) and the information in the WiringTable, with the objective of deploying the learning module as close as possible to the source of its inputs. For instance, a learning module using light and humidity transducers from a give mote will have an high chance of being deployed on the same device.

Once that the computational learning task is associated to a device with a given nodeID, the TrainingAgent calls the method deploy_modules(taskID) in the NetworkMirror, which calls the upload_module(nodeID, netID, ESN) function in the associated LearningNetworkWrapper object for each EchoStateNetwork object which is associated to the task. When the upload of a module has been completed, the LearningNetworkWrapper object sends a LEARN_MODULE_READY message to the NetworkMirror. When a LEARN_MODULE_READY message has been received for all the ESN modules involved in a task, the NetworkMirror signals the completion of the upload procedure to the TrainingAgent, by calling the method upload_complete(taskID).

At this point, the TrainingAgent component updates the WiringTable by adding entries corresponding to the new ESN inputs and outputs. Then, it uses the information in the updated WiringTable to translate the wiring instructions, originally provided by the Supervisor and possibly refined by the feature selection phase), into a list sc of SynapticConnection objects by means of the prepare_synaptic_connections(WiringType wiring, TaskType task) method. Finally, the TrainingAgent requests the deployment of such connections by a call to the function deploy_synaptic_connections(taskID, sc) in the LNControlAgent object.

## 2.5 How to Activate Learning in the Learning Layer: a Supervisor Perspective

In the following, it is provided a succinct guide for triggering the learning mechanisms in the Learning Layer from the perspective of a Supervisor component. A summary of the steps needed to activate incremental learning of a new computational learning task is provided in Section 2.5.1, while Section 2.5.2 discusses how to supply instantaneous teaching signal for the online refinement of an existing learning task.

### 2.5.1 How to activate a new learning task (incremental learning)

This section summarizes the steps to be taken by the Supervisor to trigger learning of a new computational task:

1. **Construct the wiring instructions**
   Construct a String wiring, specifying the wiring instructions according to the format “<source1,destination1>…<sourceN,destinationN>[output1]...[outputM]”. Alternatively, create a WiringType object using the constructor WiringType(source, destination, symbolicNames), where source, destination and symbolicNames are ArrayList of String and use the toPeisTuple() method to obtain an encoding of the WiringType object as an appropriately formatted string.
2. **Construct the task information**
   Construct a String task, specifying the task information according to the format

"{type}{granularity}{data_type}{output_type}" . Alternatively, create a TaskType object using the constructor TaskType(type, granularity, dataType, outputType), where type, granularity, dataType and outputType are String objects, and use the toPeisTuple() method to obtain its string encoding.

3. **Allocate the task**
   Allocate the new task by posting the String concatenation of wiring+task on the PEIS tuple PeisSymbols.WIRING_CMD.
4. **Wait for the ID of the allocated task**
   Wait and listen on the PeisSymbols.SUPERVISOR_INFO tuple for the identifier of the new computational learning task (i.e. the argument of the PeisSymbols.MSG_TASK_LEARN_READY command).
5. **(OPTIONAL) Set the feature selection preferences**
   Provide preferences on whether the feature selection mechanism should be switched on/off for the task. This is obtained by posting an appropriately formatted command on the PeisSymbols.CONTROL_CMD tuple (see Table X).
6. **Provide the training data**
   Provide training examples for the computational task by allocating a TrainingData object and obtaining its string encoding through its toPeisTuple() method. Post the resulting string in the PeisSymbols.TRAINING_SAMPLE tuple.
7. **Train the Learning Modules**
   Acknowledge termination of training data for a task by posting the PeisSymbols.STOP_TRAINING_DATA command in the PeisSymbols.CONTROL_CMD tuple, adding the task identifier as first argument (see Table X).

As a result of this process, a new learning task is allocated and, when training is completed, the associated learning modules and supporting synaptic connections are automatically deployed (see section 2.4.4), and the Forward Computation is activated (see section 2.2.4).

### 2.5.2 How to supply an instantaneous supervised/reinforcement teaching signal (refinement learning)

This section summarizes the steps to be taken by the Supervisor to provide online learning/refinement signals for a computational task:

1. **Activate the online learning mode**
   Activate the online learning mode by posting an appropriate command in the PeisSymbols.ONLINE_REFINE tuple, adding the symbolic name of the output prediction in question (see Table XVIII and Table XIX). The posted command should be chosen among those descripted in Section 2.4.3.2. For the case of an event prediction task, the command is PeisSymbols.ONLINE_EVENT if the event is occurring or PeisSymbols.ONLINE_NOEVENT if the event is not occurring. For a regression task (e.g. weight prediction) the command to post is PeisSymbols.ONLINE_INCR_FEEDBACK if the predicted output for the task is considered too small, or PeisSymbols.ONLINE_DECR_FEEDBACK if the predicted output for the task is considered too large.
2. **Continue providing online learning signals**
   When an online learning command is received from the Supervisor, it is continuously applied at each tick of the RUBICON clock, until another online learning command is received to change it, or until the online learning mode is stopped with the appropriate instruction. For example, in the case of an event classification task, a PeisSymbols.ONLINE_EVENT command should be posted when the event in question is first observed to be occurring, while a command PeisSymbols.ONLINE_NOEVENT should be posted as soon as the occurrence of the event is not observed anymore.

3. **Stop the online learning**
   Stop the online learning by posting the command `PeisSymbols.STOP_ONLINE` in the `PeisSymbols.ONLINE_REFINE` tuple, adding the symbolic name of the target output as argument (see Table XVIII and Table XIX).

As a result of this process, the weight values for the reservoir-to-readout connections in the ESN learning modules will be adapted to changes in the environmental conditions, as described by the online learning/refinement signals supplied by the Supervisor.

## 2.6 Graphical User Interface

A GUI for configuration and control of the Learning Layer is delivered together with the CLS Java API 2.0: the Java package `gui`, available in the WP2 folder of the RUBICON repository, contains all the classes needed to instantiate the GUI. The GUI is not officially part of D2.3 as per-DoW description, but it is released as an extra-functionality of this deliverable D2.3 to help the configuration and testing of the Learning Layer. The GUI allows the user to organize and access the distributed learning system implemented by the CLS API, providing a straightforward means for adjusting the Learning Layer configuration, controlling its execution and monitoring the LN outputs. The key class of the GUI is `gui.main.StartWindow`, which also includes the implementation of the `main()` method for running the interface. The GUI accesses the LN through the CLS API: it reads the LN predictions from the `LNOutputs` component and invokes the methods defined in `manager.SupervisorInterface` and `manager.ControlInterface`.

A screen-shot of the typical appearance of an experiment in the GUI is shown in Figure 11. Here, the structure of the LN is represented as a graph in which nodes are devices and arcs are Synaptic Connections between them. Each arc is labelled with the symbolic names of the neurons connected in the two devices. As connections between modules are usually quite rich, the interface allows to select a single device to focus only on its connections (the red node in the picture). Figure 11, for instance, shows a red node receiving three incoming Synaptic Connections for light, *position_x* and *position_y*, and several outgoing Synaptic Connections to other devices.

The GUI provides a device manager where the user can access all the parameters of the learning module on-board the device. The device manager is activated by double-clicking the node (Figure 12). Each device manager can be docked on the right part of the interface or detached to a new window to ease user interaction. Through the device manager, the user can adjust the learning module parameters defined in the `generics.EchoStateNetwork` class (e.g. the size of the reservoir) as well as its connection topology with other learning modules (i.e. the Synaptic Connections).

Synaptic Connections can be established both via the device manager in Figure 12 as well as graphically by drawing a connection in the canvas displaying the LN network, as in Figure 11. In both cases, the user needs to specify the identifiers of the input and output neurons to be connected. A new device is automatically added to the GUI when it joins the ecology.

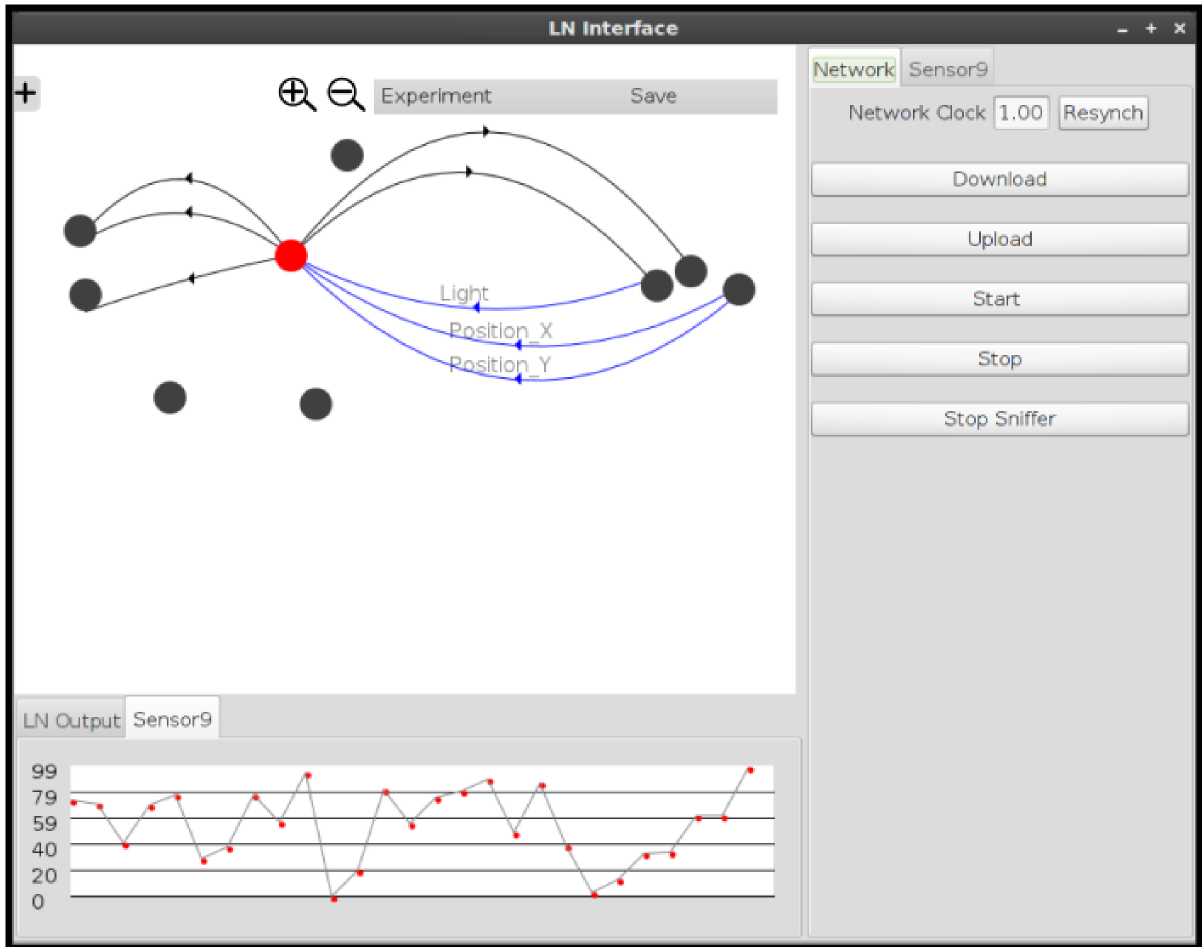



*Figure 11 Screenshot of the Learning Layer GUI showing the LN layout.*

Commands affecting the whole network can be issued from the "Network" tab menu on the right side of the GUI (see Figure 11): for instance, it is possible to adjust the global RUBICON clock. The download and upload buttons are designed to allow the user to deploy the whole LN configuration to the actual devices, or to retrieve the configuration from the actual LN. The start/stop buttons activate and disable the forward computation, while the whole LN network can be restarted, possibly using a different configuration, using the resynch button. The LN output menu allows to select which neuron output is shown at a given time.

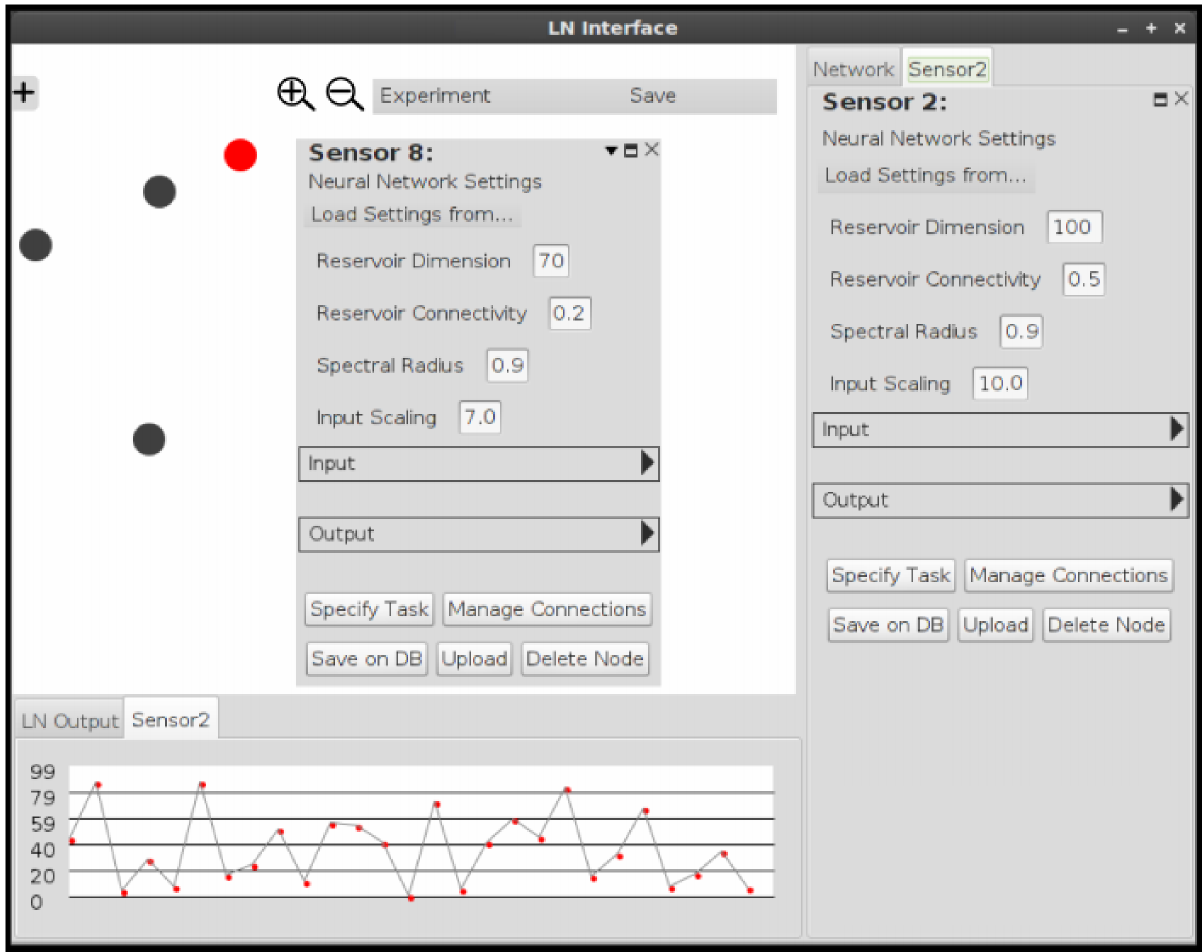



*Figure 12 Screenshot of the Learning Layer GUI showing the Device Manager.*

# 3. Testing

## 3.1 Outline

The distributed neural computation realized by the Learning Layer is an articulated process realized by a pipeline of independent mechanisms running of different devices of the ecology. A detailed testing of the independent steps of this pipeline has been presented in deliverable D2.2. This report focuses on testing and assessing the result of the full pipeline, showing how the Learning Layer can manage high-level commands on its Supervisor interface by automatically generating the most appropriate configuration and control instructions for its software components. Further, all the tests have been performed using real-world data collected on experimental scenarios compatible with the RUBICON applications in WP5.

The debugging and assessment process described in this section focuses on testing the CLS API V2.0 functionalities with respect to the 4 characterizing aspects of D2.3, as per RUBICON DoW description, that are

1. Distributed Learning Network Computation: this pertains testing the dynamic deployment of computational learning tasks entailing a distributed neural computation between multiple nodes of the ecology. A new computational learning task is deployed from scratch by using the incremental learning mechanism provided by the Learning Layer. This includes reception of wiring instructions and training data, learning module training and automated model selection, automated generation of synaptic instructions from wiring and selection of target devices and the final, over-the-air deployment of the learning module on the target devices. The process is tested for both NesC and Java learning modules.
2. Feature Selection: this pertains testing the feature selection mechanism that filters out redundant/irrelevant inputs for the learning task from those specified by the wiring instructions.
3. Online Learning: this pertains testing the learning mechanism that allows the Supervisor to provide instantaneous teaching signals targeted at refining the predictions of an existing computational learning task. The performance of this online learning mechanism is benchmarked against the batch training mechanisms using for incremental learning.
4. Recovery: this pertains testing the forward recovery and watchdog mechanisms, that allow to identify and replace failed motes by recruiting a compatible substitute device to host the learning module and synaptic connections of the failed node.

These 4 aspects have been assessed with targeted tests scripts whose experimental setup and results are presented in detail in the following subsections. Nevertheless, all the scripts needed to reproduce the tests described here can be found in the `LearningLayerApi.test` package of the CLS Java API. The key classes of this package are

- `supervisorEntity` – a stand-alone implementation of the Supervisor for testing and experimentation purposes; it provides a set of example script including control commands that exercise the various functionalities and configuration options of the Learning Layer.
- `GatewayWrapperDummy` - a stub components that wraps the Rubicon Gateway functionalities needed by the Learning Layer and allows testing the Java objects without needing to be connected with an actual LN; it collects all command and configuration commands from the `LearningNetworkWrapper` component and responds with appropriate acknowledgment messages. This component is not used in the present deliverable but it has been maintained for compatibility with the CLS API V1.0.

- `TaskDataParser` - an helper class to parse data files storing real-world training samples, from an event prediction task in an Ambient Assisted Living (AAL) scenario, that are used in the scripts.
- `AngenDataParser` - an helper class to parse data files storing real-world training samples, from a planner weight prediction task, that are used in the scripts.

## 3.2 TESTING LEARNING NETWORK COMPUTATION

### 3.2.1 Test Configuration

The assessment of the distributed incremental learning and computation in the CLS API V2.0. entails testing

1. Instantiation of a new computational learning task activated through the incremental learning mechanisms.
2. Automated training and model selection of the most appropriate learning module for the task.
3. Automated generation of the synaptic instructions, over-the-air deployment of the learning module on an appropriate ecology node and instantiation of the synaptic communication channels.
4. Collection and publishing of the LN predictions on the output interface component.

The hardware configuration used in the tests comprises:

- 1 PC (coherent with the platform requirements in 1.4.2.1) running the JLN API.
- 1 PC (coherent with the platform requirements in 1.4.2.1) running the Java LM API.
- 1 sink mote (coherent with the platform requirements in 1.4.2.2) attached to the PC through USB and running the sink-specific code of the NesC LM API.
- 2 motes (coherent with the platform requirements in 1.4.2.2) equipped with light, temperature, accelerometer and humidity transducers and running the NesC LM API.

### 3.2.2 Testing Distributed Computation with NesC Learning Modules

The first round of tests is intended to assess incremental learning and distributed neural computation capabilities of the NesC learning modules. To this end, we have employed real-world data collected by a WSN in an AAL scenario. In particular, the test script focus on the incremental acquisition of a new computational learning task associated with the prediction (classification) of the Exercising activity performed by the user. The input data used in the test include 2D accelerometer information from 2 motes deployed on the user wrist and ankle: a total of 30 positive examples and 30 negative examples have been provided as training dataset. The reservoir size of the learning modules has been limited to 50 neurons to allow deployment on a mote-class device.

#### *Test 1 Description*

The first test script, implemented by the `supervisorEntity.set_configuration_demo_motes()` method, is organized as follows

1. Creates the main `LearningLayer` object and the associated components and waits for the bring-up message on the `PeisSymbols.SUPERVISOR_INFO` tuple.

2. Triggers learning of a new task by sending wiring and task instructions on the `PeisSymbols.WIRING_CMD` tuple: source transducers include the 2D accelerometers of the wrist and ankle motes.
3. Loads training data from file and forwards it to the Training Agent component through the `PeisSymbols.TRAINING_SAMPLE` tuple.
4. When all training data has been sent, a new task is allocated, a TinyOS learning module is trained through a cross-validation procedure and deployed on one of the two motes and the synaptic communication is appropriately configured.
5. Starts the synaptic streaming using by posting the `PeisSymbols.ACTIVATE_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple.
6. Collects and publishes LN predictions generated by the learning module on-board one mote, using inputs both from its onboard transducers and from the other mote. Keeps streaming predictions until the user enters the keyboard combination "q+RETURN".
7. Stops the synaptic streaming by posting the `PeisSymbols.STOP_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple, and activates termination signals in the Learning Layer components.

At start-up, the Learning Layer activates its PEIS and Gateway interface to receive messages from the other RUBICON layers: it waits for nodes joining the ecology (see Figure 13), as only the sink device, connected to the Rubicon Gateway, is available at start-up (see the yellow-node in Figure 14).

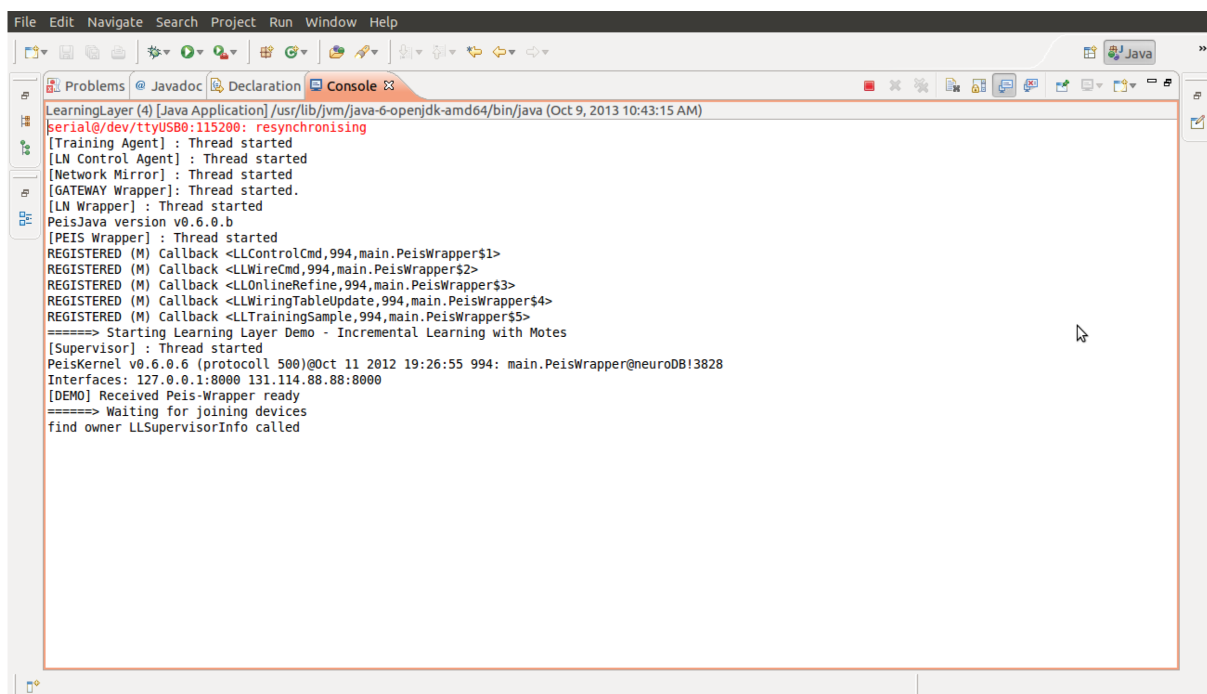



*Figure 13 Learning Layer waiting device join at start-up.*

Upon switching on of mote devices, the Learning Layer receives the joining message of the nodes, the `LNControlAgent` stores the information in its `deviceRepository`, while the `TrainingAgent` updates the local and shared `WiringTable` object with the symbolic links associated with the mote transducers. The device having joined the ecology, along with their capabilities are displayed in the GUI as in Figure 15.

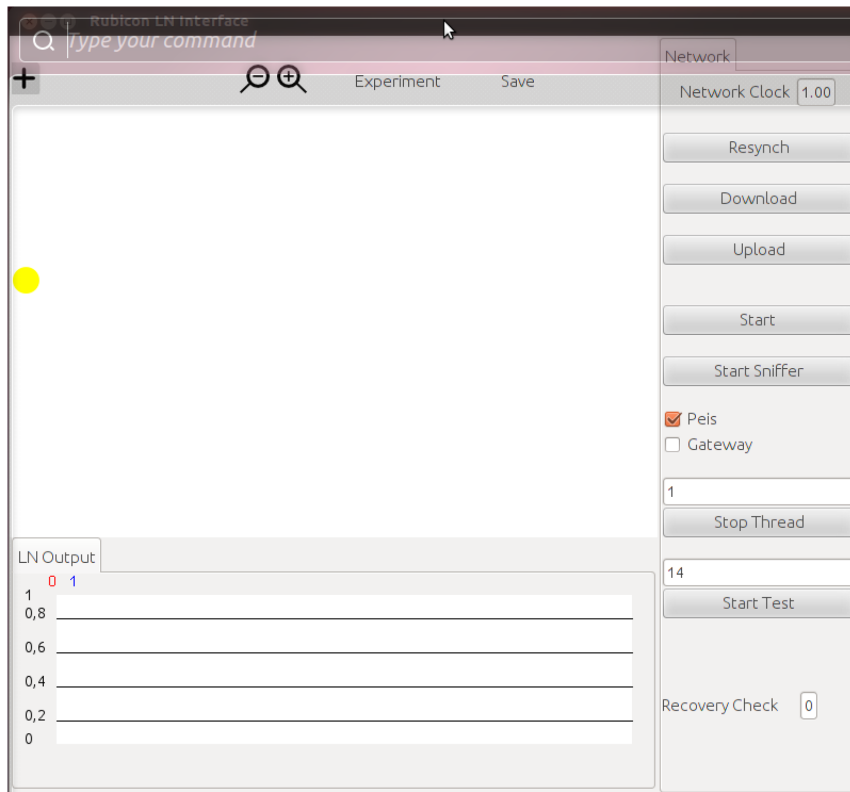



*Figure 14 GUI Snapshot: only the sink device is available at Learning Layer startup*

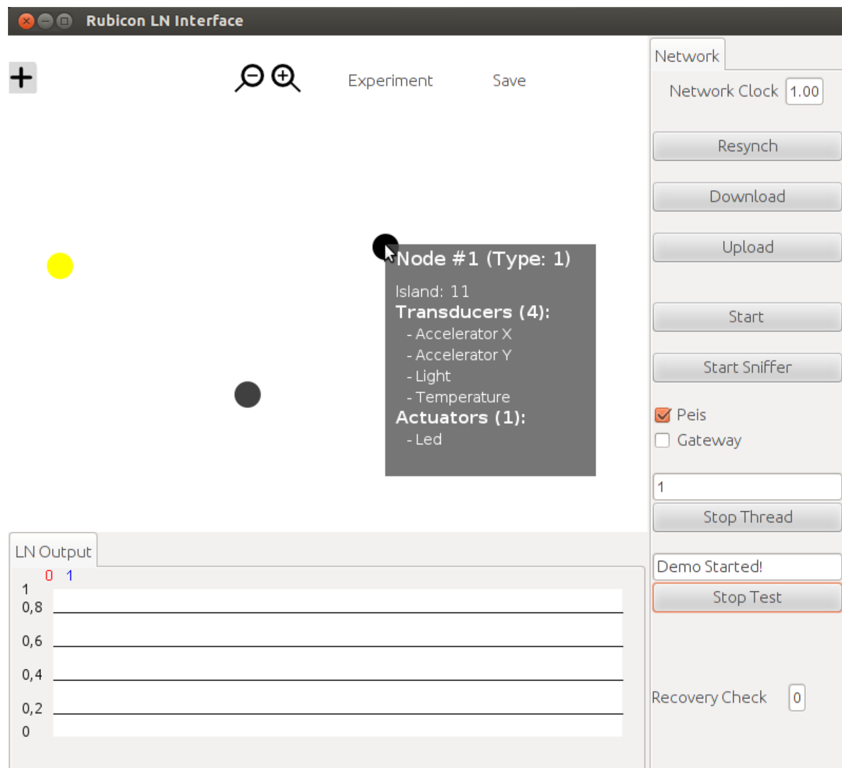



*Figure 15 GUI Snapshot: information on the joined devices along with their capabilities is stored in the Learning Layer and displayed in the GUI.*

Upon reception of the wiring and task information on the `PeisSymbols.WIRING_CMD` tuple (see Figure 16), the `TrainingAgent` allocates the new learning task and returns its identifier on the `PeisSymbols.WIRING_CMD` tuple. Similarly, training samples are received by the `TrainingAgent` as shown in Figure 17.

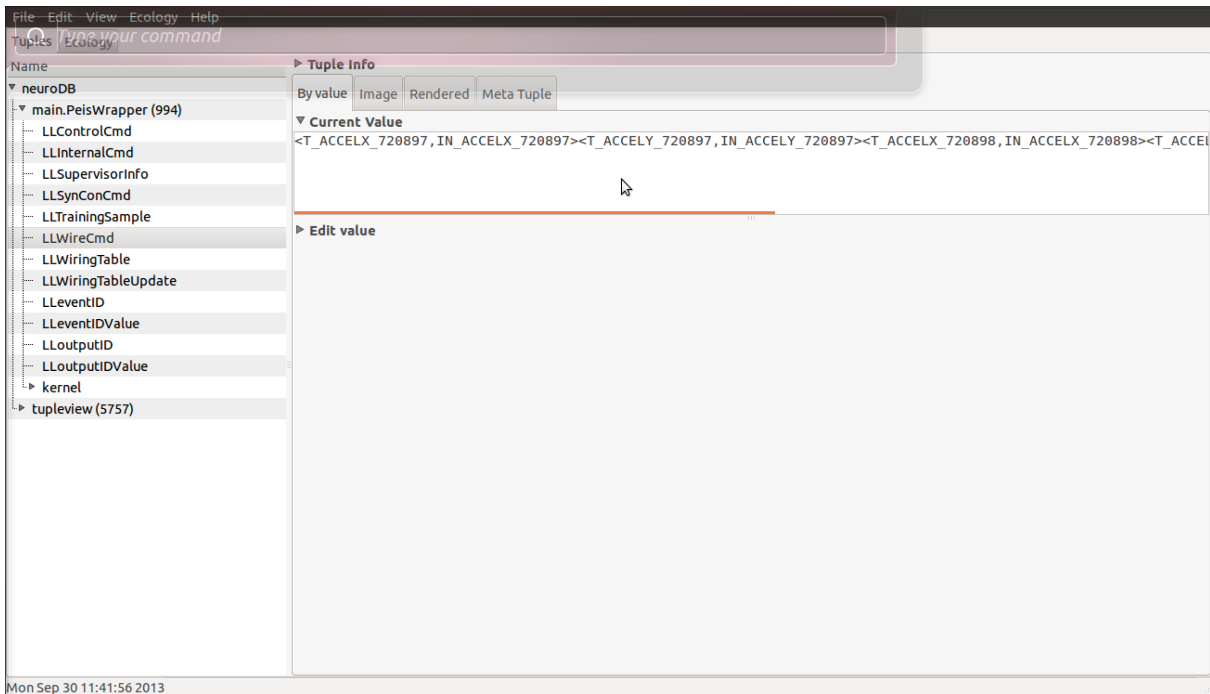



*Figure 16 Reception of the wiring and task information through PEIS.*

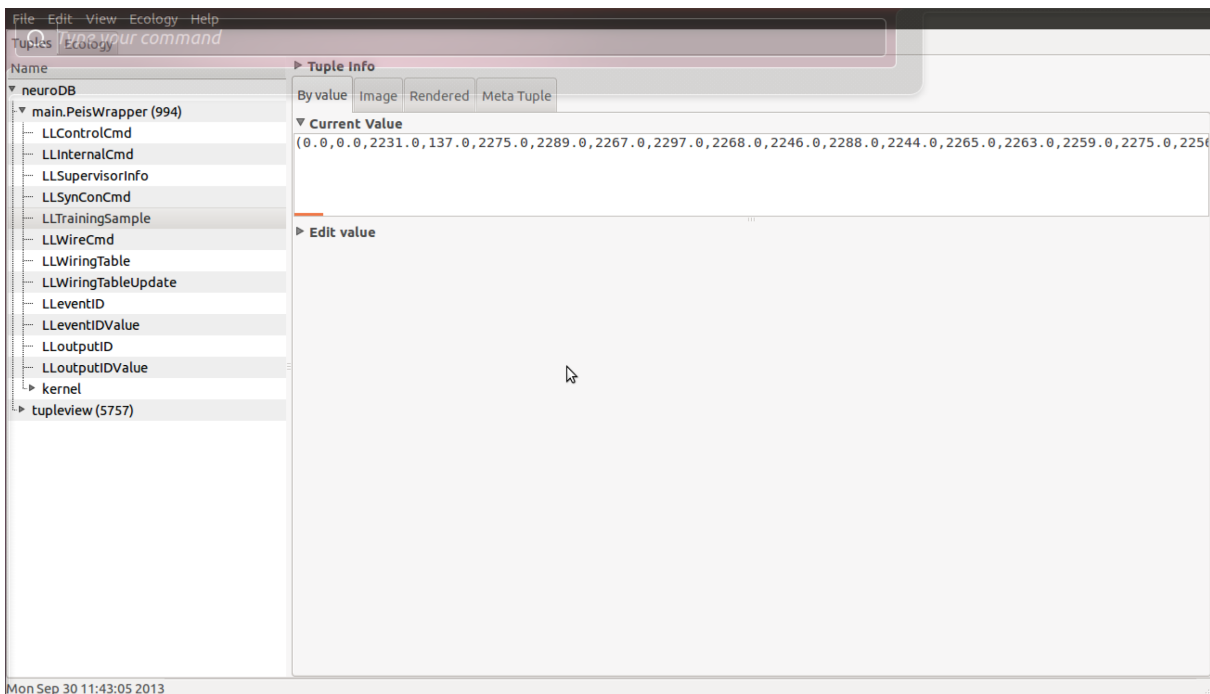



*Figure 17 Reception of the training samples through PEIS.*

Upon reception of all the training data, the `NetworkMirror` performs the model selection procedure, whose results are shown in Figure 18: here the number of configurations explored by the cross-

validation scheme has been limited, for the sake of simplicity, to a single reservoir size of 20 neurons and to four possible values for the regularization parameter (0.1, 0.01, 0.001 and 0.0001). Figure 18 shows that the ESN configuration resulting from the model selection procedure achieves a mean classification accuracy of 80.81% on a randomly selected test set.

```
File Edit Navigate Search Project Run Window Help
Problems @ Javadoc Declaration Console
LearningLayer (4) [Java Application] /usr/lib/jvm/java-6-openjdk-amd64/bin/java (Oct 9, 2013 10:43:15 AM)
[PEIS Wrapper] : Got sample (0.0,0.0,130.0,2285.0,2281.0,2282.0,2286.0,2289.0,2270.0,2312.0,2267.0,2
[PEIS Wrapper] : Got sample (0.0,0.0,2231.0,137.0,2275.0,2289.0,2267.0,2297.0,2268.0,2246.0,2288.0,2
======> Train and deploy task when enough data is available
[NETWORK MIRROR] Starting Cross-Validation on 4 models
[NETWORK MIRROR] Training ESN Configuration 0 (ReservoirSize:20, Regularization:0.1, Leaky:1.0)
[NETWORK MIRROR] Trained ESN n.0 with CV performance 0.6953572
[NETWORK MIRROR] Trained ESN n.1 with CV performance 0.84630954
[NETWORK MIRROR] Trained ESN n.2 with CV performance 0.69922626
[NETWORK MIRROR] Training ESN Configuration 3 (ReservoirSize:20, Regularization:0.01, Leaky:1.0)
[NETWORK MIRROR] Trained ESN n.3 with CV performance 0.6547025
[NETWORK MIRROR] Trained ESN n.4 with CV performance 0.78636914
[NETWORK MIRROR] Trained ESN n.5 with CV performance 0.7660119
[NETWORK MIRROR] Training ESN Configuration 6 (ReservoirSize:20, Regularization:0.001, Leaky:1.0)
[NETWORK MIRROR] Trained ESN n.6 with CV performance 0.8135714
[NETWORK MIRROR] Trained ESN n.7 with CV performance 0.808631
[NETWORK MIRROR] Trained ESN n.8 with CV performance 0.804881
[NETWORK MIRROR] Training ESN Configuration 9 (ReservoirSize:20, Regularization:1.0E-4, Leaky:1.0)
[NETWORK MIRROR] Trained ESN n.9 with CV performance 0.79130954
[NETWORK MIRROR] Trained ESN n.10 with CV performance 0.8901786
[NETWORK MIRROR] Trained ESN n.11 with CV performance 0.7988095
[NETWORK MIRROR] ESN number 10 has the maximum Accuracy
[NETWORK MIRROR] Average accuracy on the test samples = 0.8081019
======> New task deployed and ready to predict
[NETWORK MIRROR] End of Cross-Validation
[TRAINING AGENT] Training completed for Task 1
[LN Wrapper] : Inserted Request 5
[LN Wrapper] : Inserted Request 6
[LN Wrapper] : Inserted Request 7
[LN Wrapper] : Inserted Request 8
[LN Wrapper] : Inserted Request 9
[GATEWAY Wrapper]: Sending a message
[LN Wrapper] : Received request to remove Seq 5
[LN Wrapper] : Removed Request 5
[GATEWAY Wrapper]: Sending a message
```

Model Selection and Training

*Figure 18 Model-selection, training and deployment of a TinyOS learning module on the Exercising task.*

An ESN with the selected configuration is then trained on the full dataset provided by the supervisor and deployed to a TinyOS learning module. The `TrainingAgent` takes care of translating the wiring instructions into appropriate synaptic communication setup commands: one of the resulting (remote) synaptic connection is shown on the GUI snapshot in Figure 19.

Upon reception of the broadcast command activating forward computation, the newly deployed TinyOS learning module starts streaming its predictions on the Exercising activity (see Figure 20).

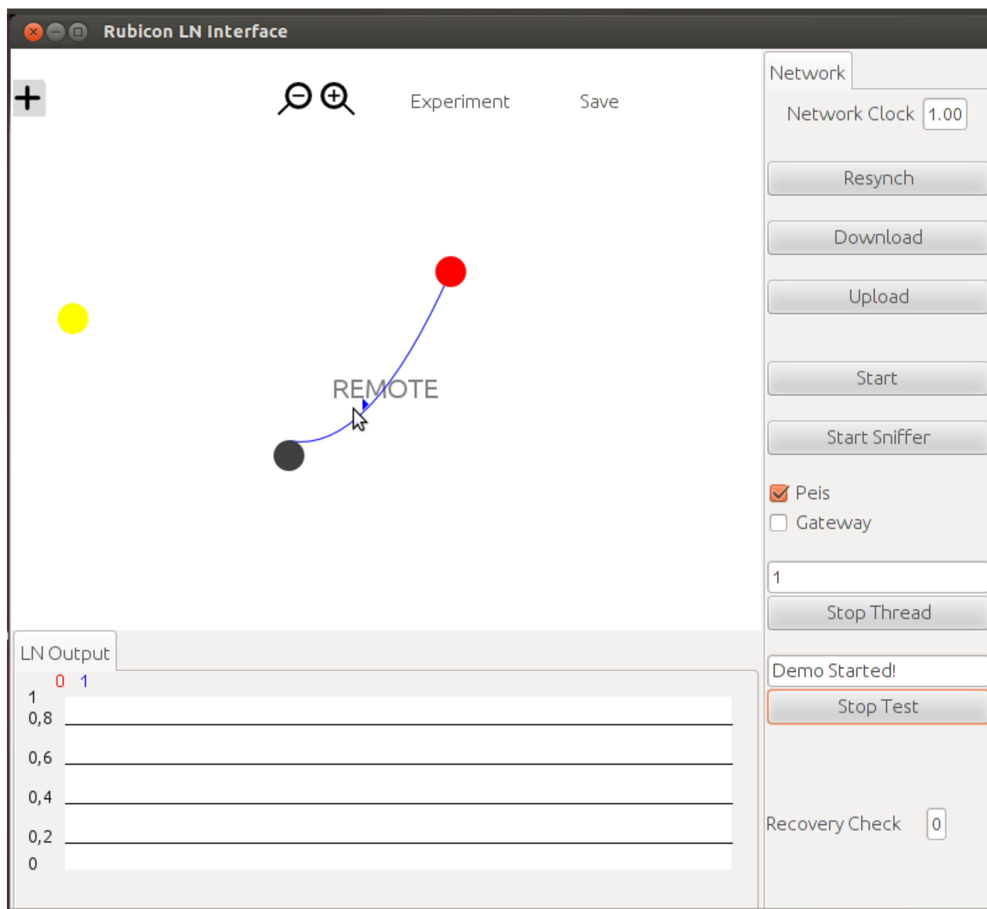



*Figure 19 Snapshot of the GUI showing a deployed synaptic connection between two motes in the Exercising demo.*

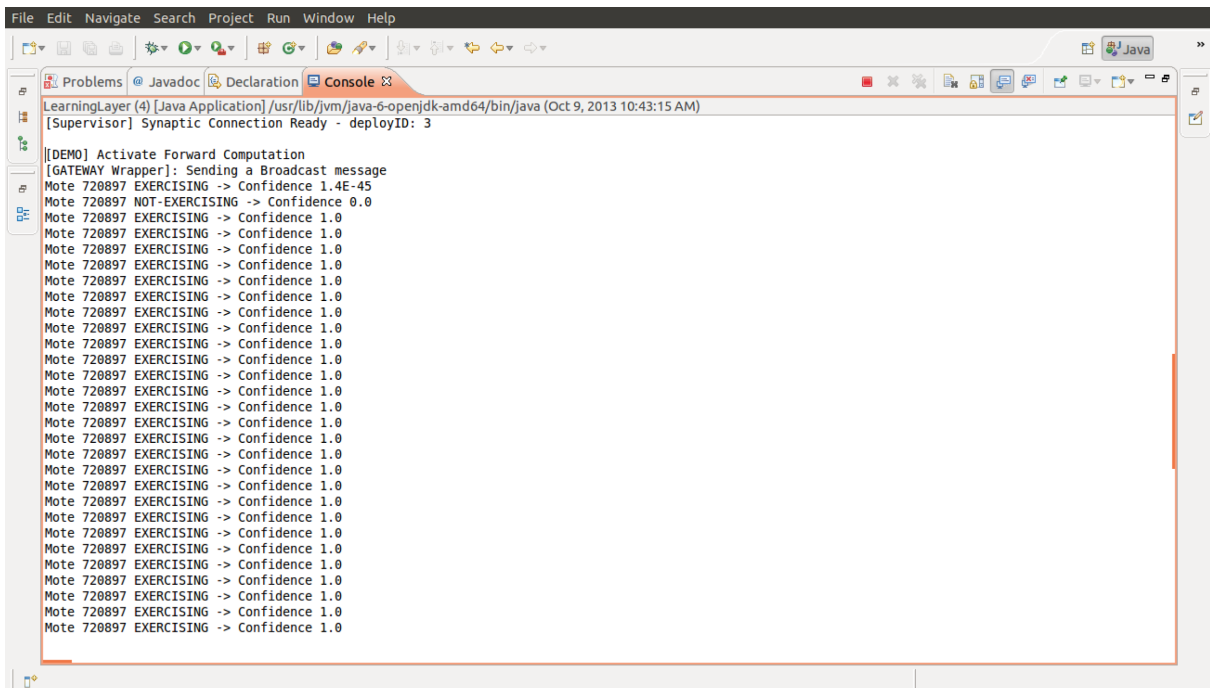



*Figure 20 Starting of the distributed forward computation producing the predictions of the TinyOS Learning module for the Exercising activity.*

### 3.2.3 Testing Distributed Computation with Java Learning Modules

The second round of tests is intended to assess incremental learning and distributed neural computation capabilities of the Java learning modules in the AAL scenario described in Section 3.2.2.

#### *Test 2 Description*

The second test script, implemented by the `supervisorEntity.set_configuration_demo_java()` method, is organized as follows

8. Creates the main `LearningLayer` object and the associated components and waits for the bring-up message on the `PeisSymbols.SUPERVISOR_INFO` tuple.
9. Triggers learning of a new task by sending wiring and task instructions on the `PeisSymbols.WIRING_CMD` tuple: source transducers include the 2D accelerometers of the wrist and ankle motes.
10. Loads training data from file and forwards it to the Training Agent component through the `PeisSymbols.TRAINING_SAMPLE` tuple.
11. When all training data has been sent, a new task is allocated, a Java learning module is trained through a cross-validation procedure and deployed on a target device and the synaptic communication is appropriately configured.
12. Starts the synaptic streaming using by posting the `PeisSymbols.ACTIVATE_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple.
13. Collects and publishes LN predictions generated on the Java learning module using inputs from remote WSN devices. Keeps streaming predictions until the user enters the keyboard combination “q+RETURN”.
14. Stops the synaptic streaming by posting the `PeisSymbols.STOP_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple, and activates termination signals in the Learning Layer components.

#### *Test 2 Outcome*

Execution of this script progresses as discussed in Section 3.2.2 for the NesC case, until learning module training. Figure 21 shows the results of the model selection procedure: here the number of configurations explored by the cross-validation scheme has been limited, for the sake of simplicity, to a single reservoir size of 100 neurons and to two possible values for the regularization parameter (0.01 and 0.001). Nevertheless, the `NetworkMirror` offers more general (and computationally intensive) schemes that explore a larger number of configurations. Figure 21 shows that the ESN configuration resulting from the model selection procedure achieves a mean classification accuracy of 82.06% on a randomly selected test set. An ESN with the selected configuration is then trained on the full dataset provided by the supervisor and deployed to a Java learning module with identifier netID=-1 hosted by the device with nodeID = 72099.

The `TrainingAgent` takes care of translating the wiring instructions into appropriate synaptic communication setup commands (see the messages in Figure 21 and Figure 22). These include both instructions target to TinyOS devices, i.e. for reading the accelerometer transducers, as well as PEIS messages addressed to the Java learning module (see Figure 22).

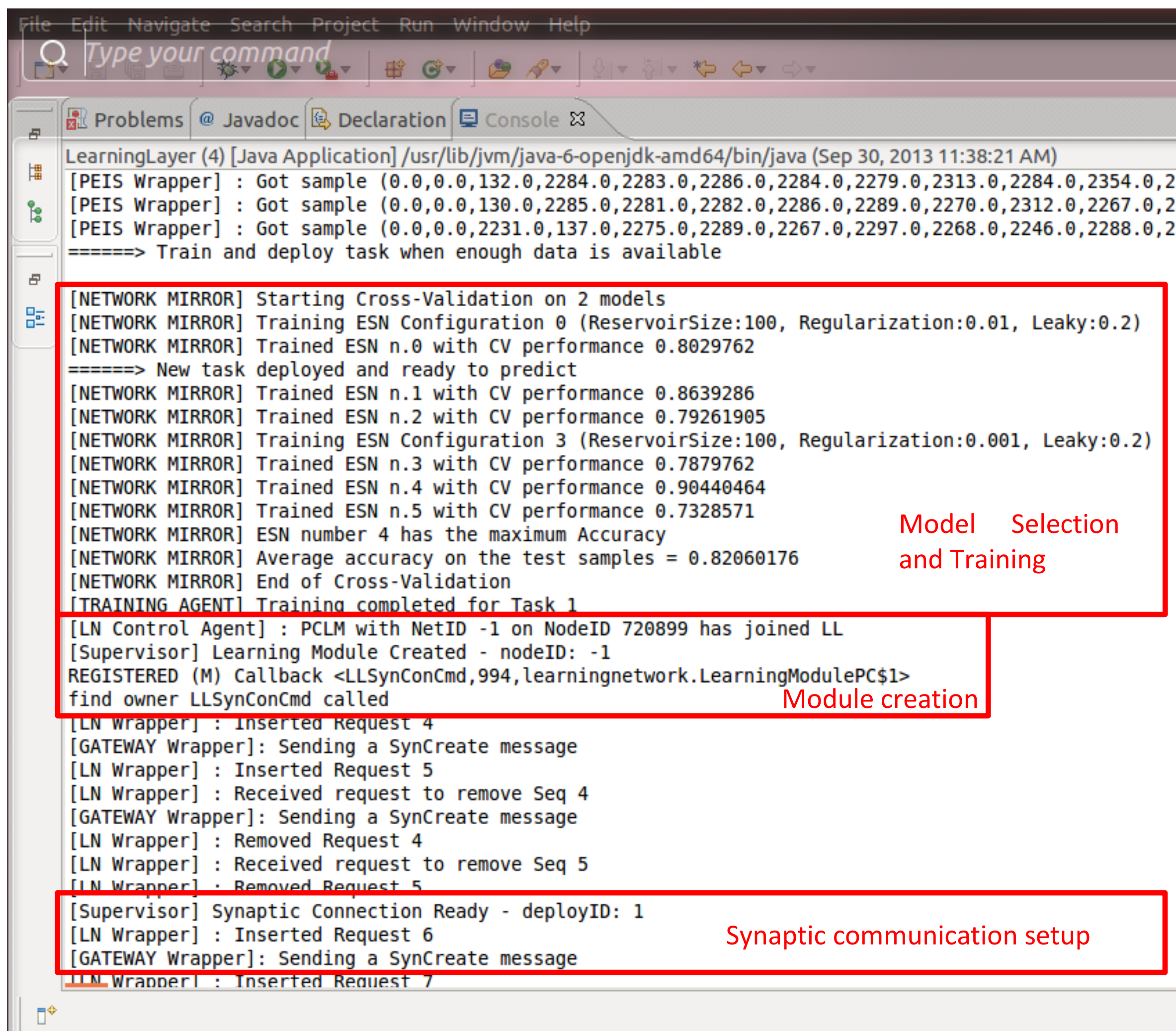
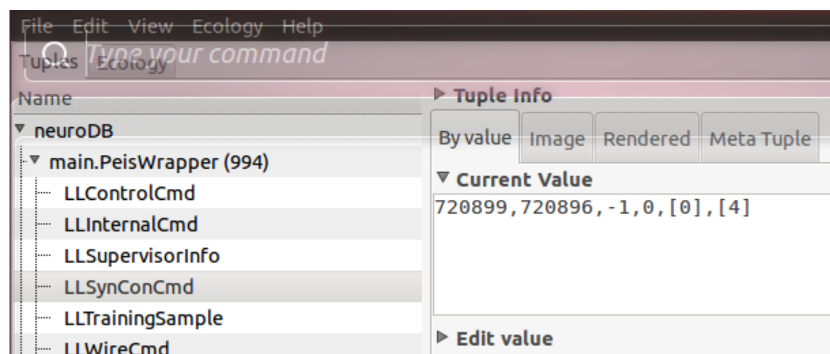



*Figure 21 Model-selection, training and deployment of a Java learning module on the Exercising task.*

In particular, the `LearningNetworkWrapper` instructs the learning module to subscribe to the tuple implementing its input synaptic channel. This is achieved by posting an appropriately formatted string in the `PeisSymbols.CONTROL_CMD` tuple as discussed in Section 2.2.3. A snapshot of the tuple posted to register the synaptic channel of the Java learning module is as follows:



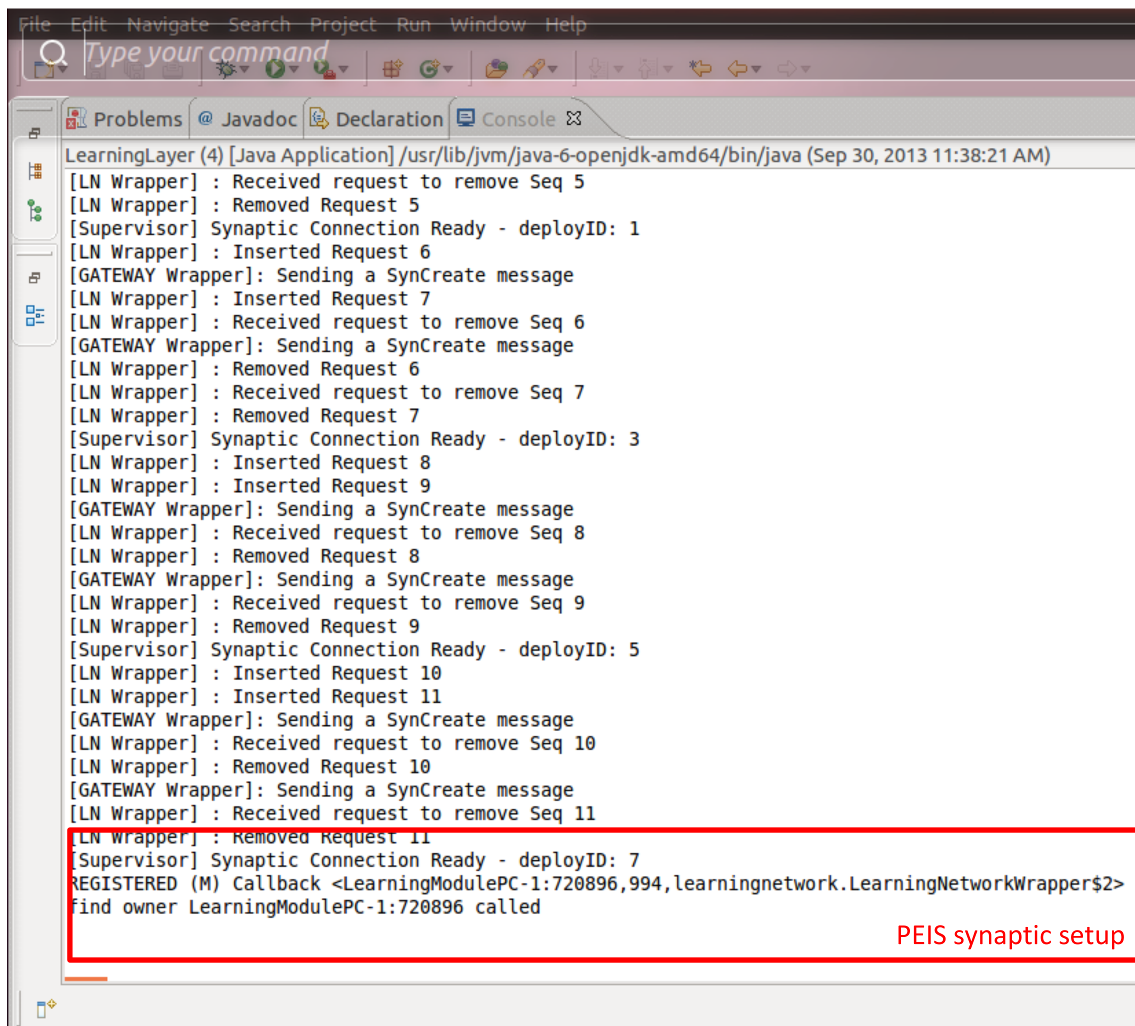



*Figure 22 Synaptic connection deployment for the Exercising Task with Java learning modules: note how the module registers it-self as listener to the tuple implementing the input Synaptic Channel (in red).*

To complete the deployment phase, the `TrainingAgent` also updates the local and global copy of the wiring table, to introduce symbolic names for the newly deployed task. These include the Symbolic name for the task output (i.e. the neuron providing the actual prediction) as well as those for the input neurons of the learning module. A snapshot of the updated wiring table is provided in Figure 23.

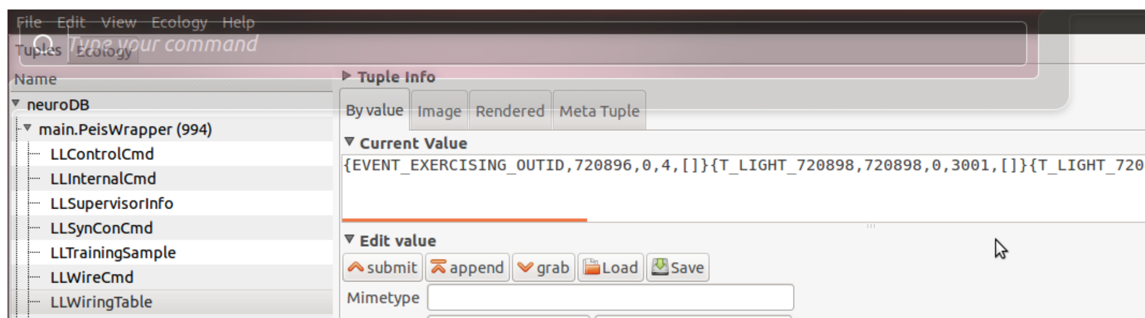



*Figure 23 Wiring table resulting from the deployment of the new task.*

Upon reception of the broadcast command activating forward computation, the newly deployed Java learning module starts streaming its predictions on the Exercising activity (see Figure 24), that are forwarded to the appropriate interfaces tuples of the Learning Layer where they are made available to the other RUBICON layers (see the event publishing tuple in Figure 25).

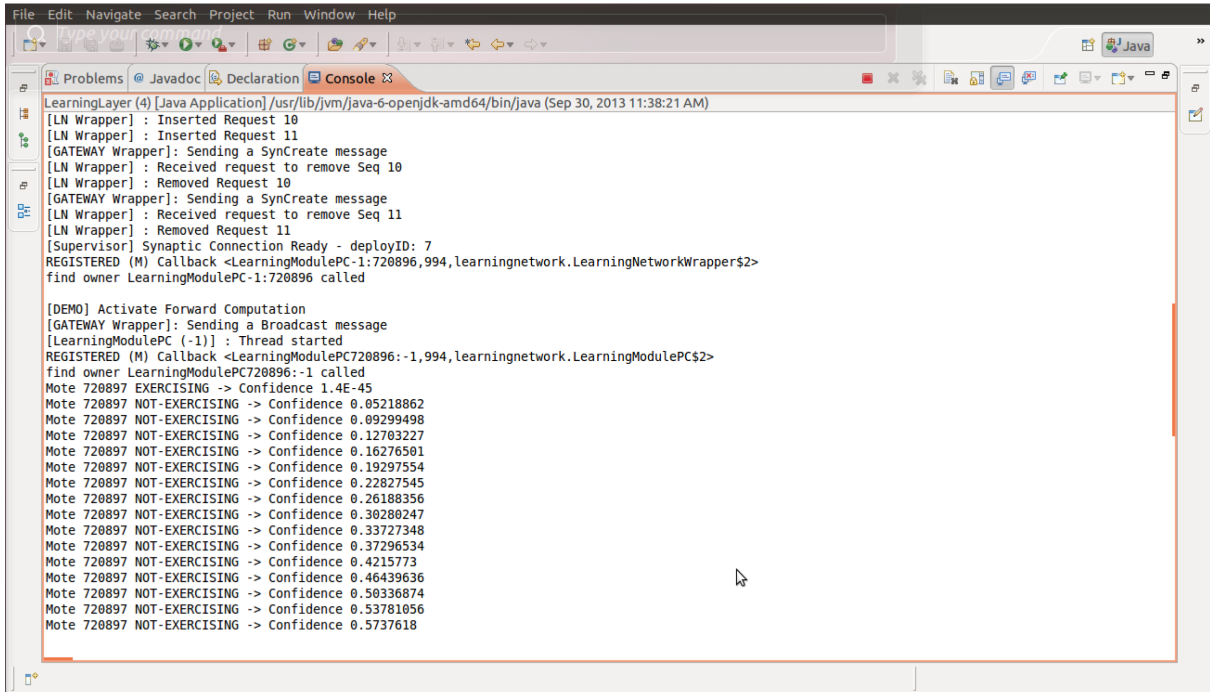



*Figure 24 Starting of the distributed forward computation producing the predictions of the Java Learning module for the Exercising activity.*

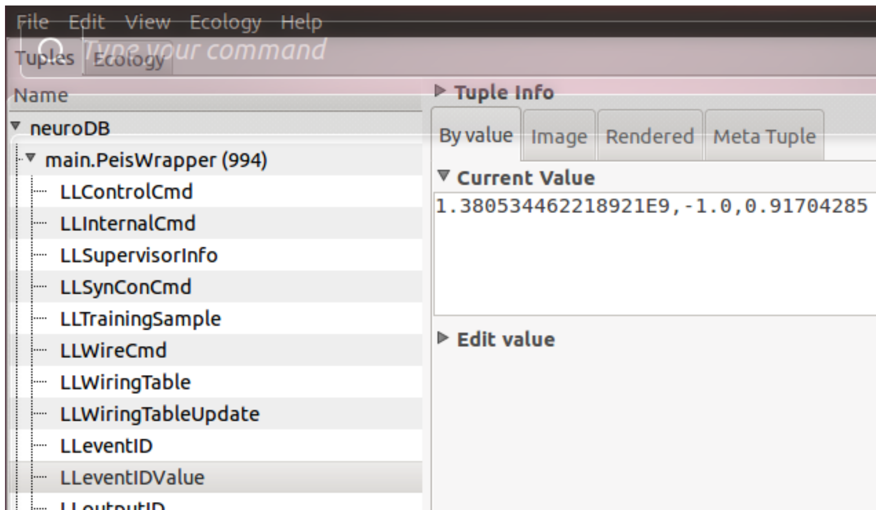



*Figure 25 Publication of the predictions for the Exercising activity on the event publishing tuple: the 3 values displayed refer to the RUBICON time, event occurrence (+1=occurrence, -1=non-occurrence) and prediction confidence (0 minimum confidence, 1 maximum) , respectively.*

For the sake of completeness, note, in Figure 26, a snapshot of the tuples implementing the input synaptic connections from the WSN motes to the Java learning module (left) and the output communication from the module to the Learning Gateway (right).

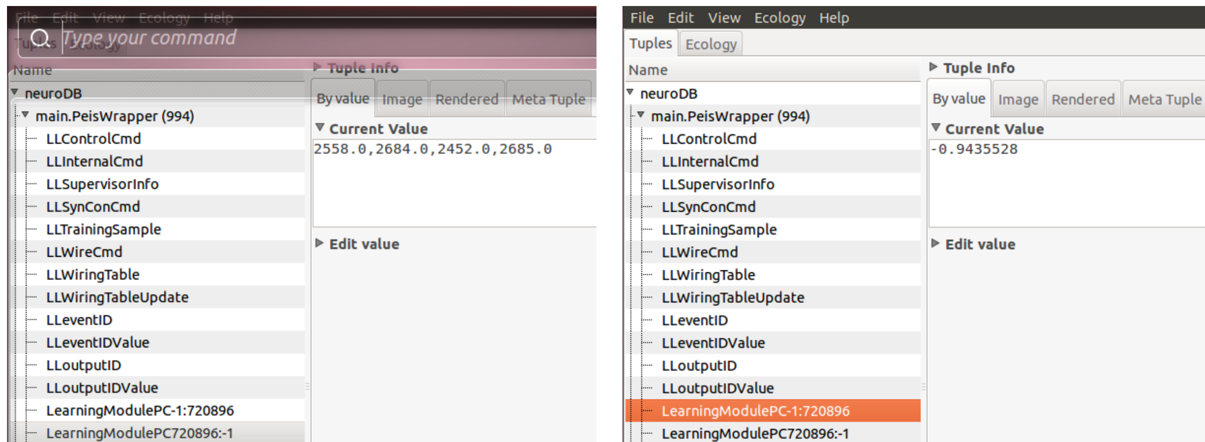



*Figure 26 Snapshot of the tuples implementing synaptic communication in (left) and out(right) of the Java learning module.*

### 3.2.4 Performance Assessment of the Echo State Network model on Real-World RUBICON Tasks

The effectiveness and the appropriateness of the ESN models used for realizing the RUBICON Learning Network, and of the online learning methods implemented in the Learning Layer API, have been tested on a variety of *real-world* tasks in the AAL field. In this section, for each task, we provide a brief description and a summary of the performance assessment results.

**Indoor User Movement Forecasting**

- **Type:** Classification of sequences
- **Input considered:** RSSI
- **Target type:** +1/-1 class label
- **Ground-truth:** Manually gathered
- **Description:** The task consists in predicting user movements (room change vs. room preservation) in an indoor environment composed of a couple of rooms separated by a corridor. The dataset contains data pertaining to 3 different couples of rooms.
- **Dimension of the dataset:** 104 sequences for the first couple of rooms, 106 for the second couple of rooms, 104 for the third couple of rooms.
- **Sampling frequency:** 8 Hz
- **Availability on the data:** The task can be downloaded freely from http://ala.isti.cnr.it/paolo/index.php/dataset/6rooms
- **Performance assessment:** Experimental results in [4, 8] showed that the ESN models can reach an extremely good accuracy on this task (up to 96% of test accuracy), and that ESN performances scale well with the cost of deployment of the network. Experimental results in [5] showed the generalization ability of the ESNs to unseen ambient configurations. In [6] was presented a study the relationship between the predictive performance on this task and the architectural parameters influencing the computational cost of the ESN models, including the reservoir dimension. In the same work, the use of a reduced encoding scheme (adopted in the NesC API Learning Layer) was introduced, showing analogous performance results to the case of standard (more costly) weight storage schemes. A complete empirical assessment of several distributed ESN configurations, based on realistic WSN-induced layouts can be

found in [3]. Further experimental results on this task, concerning the online learning process, are reported in Section 3.4.

**Indoor Robot Localization**

- **Type:** Regression on sequences
- **Input considered:** RSSI
- **Target:** (x,y) localization coordinates
- **Ground-truth:** SLAM/Gmapping
- **Description:** The task consists in predicting the (x,y) position of a robot moving in a laboratory environment in Robotnik.
- **Dimension of the dataset:** 4 sequences.
- **Sampling frequency:** 10 Hz
- **Performance assessment:** The experimental analysis presented in [7] showed promising performance results with an Euclidean test error of approximately 1 m, close to the ground-truth precision.

**AAL Home Lab Scenario**

- **Type:** Step by step classification on sequences
- **Input considered:** RSSI, pir, accelerometer, light, switch, pressure, sound recognition, other KNX sensors
- **Target:** +1/-1 class label for each event
- **Ground-truth:** Manually gathered
- **Description:** The AAL Home Lab Scenario comprises a set of event classification tasks corresponding to different user activities in a home environment, including “prepare food”, “prepare coffee”, “set the table”, “eating”, “washing dishes”, “exercising”, “get drink”, “relax”, “sleep”, “user leaving”, “user arriving”. The dataset for this task was collected in Tecnalia’s Home Lab in 2013.
- **Dimension of the dataset:** A total number of 26 sequences are available, containing data pertaining to the different user activities by different actors.
- **Sampling frequency:** 2 Hz
- **Performance assessment:** ESNs of a suitable dimension for embedding in the motes of a WSN (with reservoir size of 50) were tested on a subset of tasks from the AAL Home Lab Scenario, including “prepare coffee”, “set table”, “eating” and “washing dishes”. Resulting predictive accuracy was close to the 90% on the test set. Figure 27 shows an example of the output of the Learning Layer predictions for the four tasks considered, during a test sequence. Further experimental results considering online learning are illustrated in Section 3.4. This task has been used for the RUBICON live demo of the 2nd year project review.

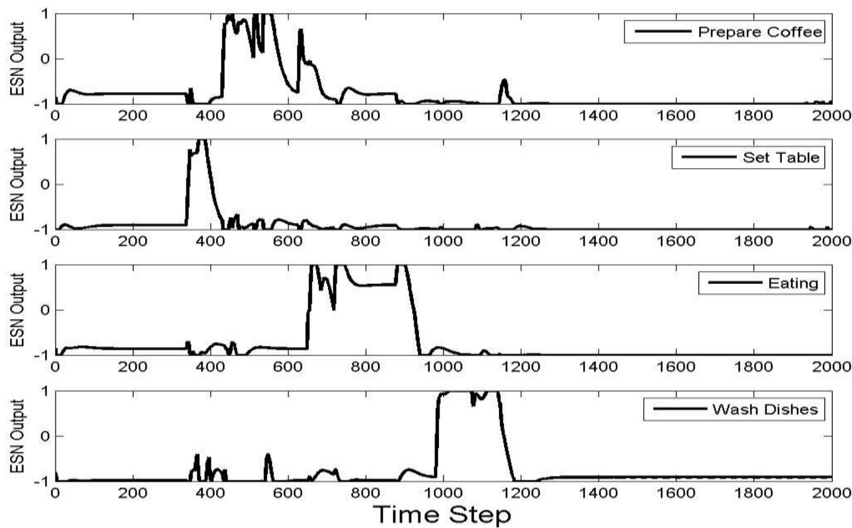



*Figure 27 Learning Layer output predictions for a set of events in the AAL Home Lab scenario.*

**Angen Scenario - Planner Weight Prediction for Localization Systems**

- **Type:** Regression on sequences
- **Input considered:** PIR, light, (x,y) localization information
- **Target:** Real value representing the evaluation of the performance of the robot localization system
- **Ground-truth:** Gathered through empirical algorithms
- **Description:** This dataset contains samples corresponding to two scenarios pertaining to the prediction of the performance of a robot indoor localization system. In the first scenario (Mirror), the target performance of the localization system is influenced by the presence or absence of a mirror in the robot trajectory. In the second scenario (Kitchen), the target performance of the localization system is influenced by the presence or absence of the user in the kitchen, due to privacy issues. More details about this task can be found in Section 3.3.
- **Dimension of the dataset:** A number of 87 sequences are available for the Mirror task, while 69 sequences are available for the Kitchen task.
- **Sampling frequency:** 2 Hz
- **Performance assessment:** ESNs were tested on both the Mirror and the Kitchen tasks, in correspondence of local and global learning settings. Results showed a general better performance of the global learning settings over the local ones. Moreover, small ESNs with 50-units reservoir suitable for being embedded in the motes of a WSN, turned out to represent a good tradeoff between predictive performance and computational feasibility. Figure 28 and Figure 29, show the mean absolute test error achieved by ESNs on the two tasks Mirror and Kitchen, respectively, varying the reservoir dimension and the learning setting. This task has been used for the RUBICON video demo of the 2nd year project review

(ORU – UNIPI collaboration).

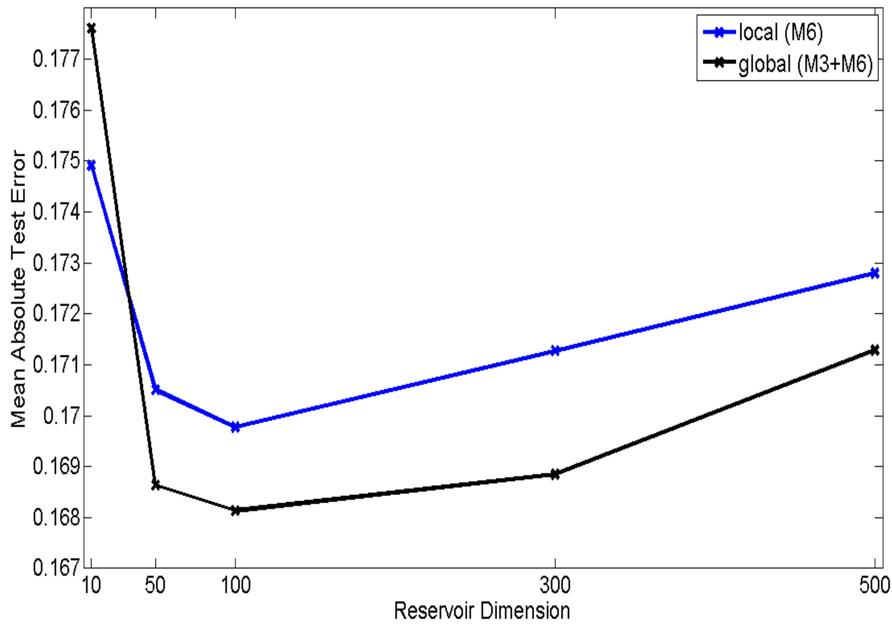



*Figure 28 Mean Absolute Test Error achieved by ESNs on the Planner Weight Prediction for Localization Systems – Mirror task.*

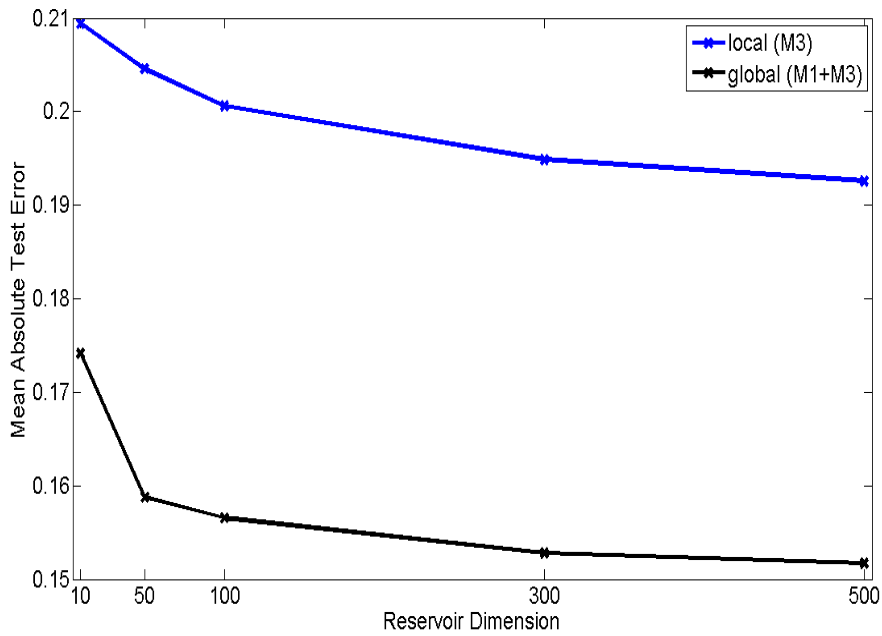



*Figure 29 Mean Absolute Test Error achieved by ESNs on the Planner Weight Prediction for Localization Systems – Kitchen task.*

Further experimental results on these tasks, concerning the feature selection and the online learning functionalities of the Learning Layer can be found in Section 3.3 and 3.4, respectively.

**Hospital Robot Localization**

- **Type:** Regression on sequences
- **Input considered:** RSSI
- **Target:** (x,y) localization coordinates
- **Ground-truth:** Both manual and SLAM/Gmapping/amcl
- **Description:** The Hospital Robot Localization dataset contains data for robot localization in a critical environment, namely a wing of a pediatric hospital building. Different environmental conditions are represented in the dataset, e.g. empty corridors and people walking. The dataset for this task have been collected in Stella Maris, in July 2013 (see Figure 30).
  The dataset also includes data for a set of classification tasks, in which couples of WSN motes represent "flags" and the target is +1 whenever any entity (human or robot) passes through the flag, and -1 otherwise.
- **Dimension of the dataset:** The dataset contains 36 sequences.
- **Sampling frequency:** 4 Hz
- **Performance assessment:** Preliminary experimental investigations were performed on the Hospital Robot Localization task, showing the need for a careful choice of the appropriate tradeoff among the computational cost of the learning model (e.g. reservoir dimension), regularization and localization performance, due to the critical noise/signal ratio in the target measurements. Figure 31 shows a plot of the mean Euclidean test error achieved by ESNs in the first preliminary set of experiments on the Hospital Robot Localization task, using amcl ground-truth data, and considering different reservoir dimensions and different values for the readout regularization parameter $lambda_r$. Such results also put forward the need for a higher precision ground-truth system in such critical environment.

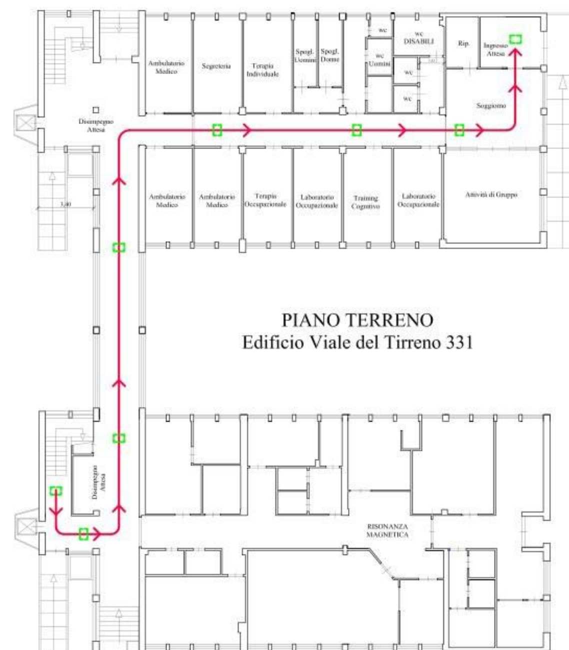



*Figure 30 Map of the corridors involved in the measurement campaign in the wing of the Stella Maris building. The indicative trajectory followed by the robot is illustrated in red.*

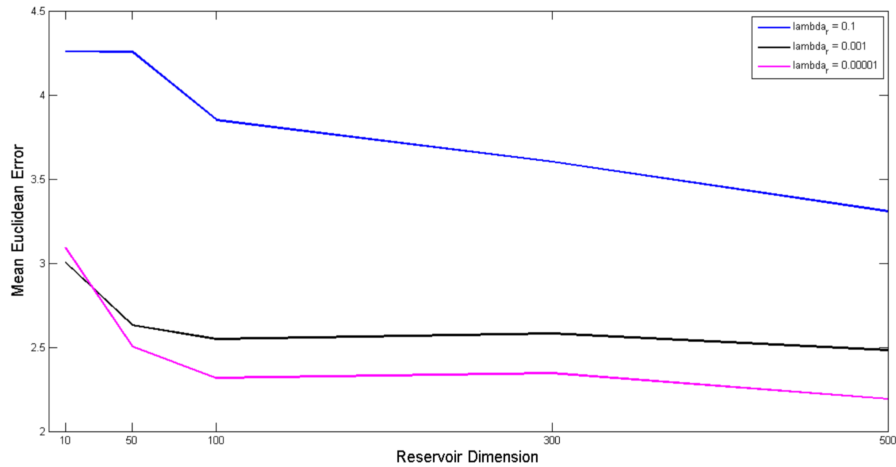



*Figure 31 Mean Euclidean Test Error achieved by ESNs on the preliminary experiments on the Hospital Robot Localization task. Performance results are reported for varying reservoir dimensionality and readout regularization.*

**UNIPI Indoor Robot Localization**

- **Type:** Regression on sequences
- **Input considered:** RSSI
- **Target:** (x,y) localization coordinates
- **Ground-truth:** SLAM/amcl
- **Description:** The UNIPI Indoor Robot Localization task consists in predicting the (x,y) coordinates of a robot moving in an indoor environment. The input data is provided by a WSN composed of 5 anchors and 1 mobile (placed on the robot). The dataset for this task was collected during a measurement campaign in the UNIPI building, in June 2013. Figure 32 shows a map of the area considered in the experimental scenario.
- **Dimension of the dataset:** The dataset contains 12 sequences.
- **Sampling frequency:** 2 Hz
- **Performance assessment:** Preliminary experimental investigations performed on the Indoor Robot Localization task showed a predictive performance of the ESN modules close to the ground-truth precision, with a mean Euclidean error on the test set of approximately 1 m.

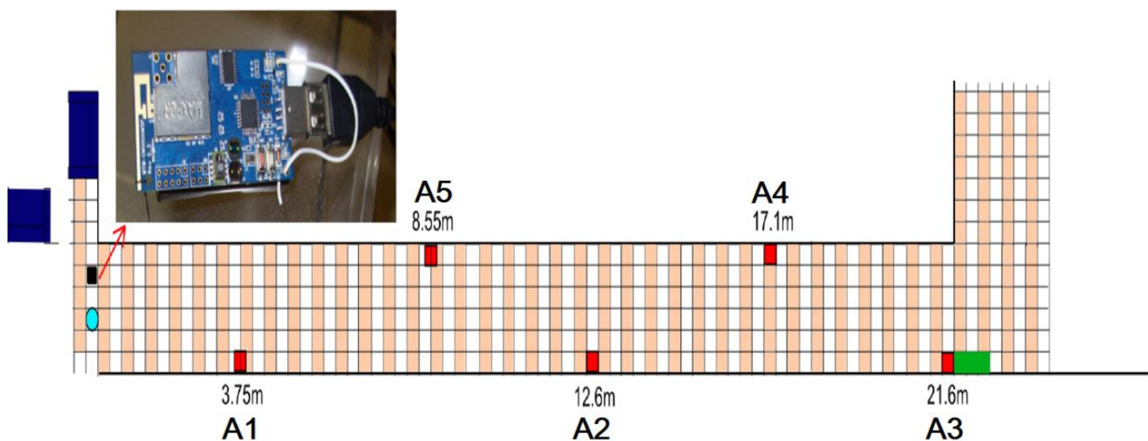



*Figure 32 Map of the area used for the measurement campaign for the UNIPI Indoor Robot Localization task.*

**Waving Activity Recognition**

- **Type:** Step by step classification on sequences
- **Input considered:** light, temperature, humidity, accelerometers
- **Target:** +1/-1 class label
- **Ground-truth:** Manually gathered
- **Description:** The Waving task consist in a binary classification task in which it is requested to discriminate the user activity of waving her hand versus the still situation.
- **Dimension of the dataset:** The dataset contains a total number of 30 sequences (15 of which corresponding to the waving activity, and the other 15 corresponding to the still situation).
- **Sampling frequency:** 4 Hz
- **Performance assessment:** ESNs modules were tested on this task, leading to an extremely good predictive performance, close to 100% of test accuracy. This task has been used for the WP2 live demo of 2nd year RUBICON review. Details concerning the experimental results on this task can be found in Section 3.5.2.

**UNIPI Activity Recognition**

- **Type:** Step by step classification on sequences
- **Input considered:** RSSI, light, temperature, humidity
- **Target:** +1/-1 class label for each event
- **Ground-truth:** Manually gathered
- **Description:** The UNIPI Activity Recognition is a multi-classification task, where each class corresponds to one user activity among “cleaning”, “exercising” and “relaxing”. Input data is collected through a WSN comprising 4 anchors motes and 3 mobile motes, 2 of which are worn by the user and 1 is placed on a table.
- **Dimension of the dataset:** The dataset contains 90 sequences.
- **Sampling frequency:** 4 Hz
- **Performance assessment:** Preliminary experimental investigations performed on the UNIPI Activity Recognition dataset showed very good predictive performance, with test accuracy of 0.81 and F1score of 0.83.

## 3.3 TESTING FEATURE SELECTION

The second round of tests is intended to assess the capability of feature selection mechanism in detecting and removing redundant inputs associated to a novel computational learning task. To this end, we have employed real-world data collected in a task involving prediction of robot navigation preferences in a sensorized home-environment. This task falls under the “planner weight prediction” type of task that can be addressed by the Learning Layer (see the definition of `TaskInformation` in Section 2.4).

The experimental scenario has been designed and put into operation in the Angen senior residence facilities in ORU, through a joint work with UNIPI. The scenario, depicted in Figure 33, comprises a real-world flat sensorized by an RFID floor, a WSN with 6 nodes equipped with light, temperature, humidity and passive infrared (PIR) presence sensors, as well as a mobile robot with range-finder localization. The task is intended to predict a weight evaluating the performance of the localization system on different trajectories. Two trajectory types are considered here, represented as dashed and continuous lines in Figure 33. Performance on the dashed trajectory is expected to be low due to the effect of mirror disturbances which, conversely, should not affect trajectories on the continuous line. Note that the only PIR sensors that are triggered by robot motion are those

onboards motes M3 and M6 in Figure 33. The input data used in the test include all transducers from the 3 motes, plus robot trajectory information under the form of its (x,y) position and orientation θ in time.

A second set of test scripts has also been run on the Exercising event task discussed in Section 3.2, showing how the feature selection mechanisms integrate with the incremental learning process.

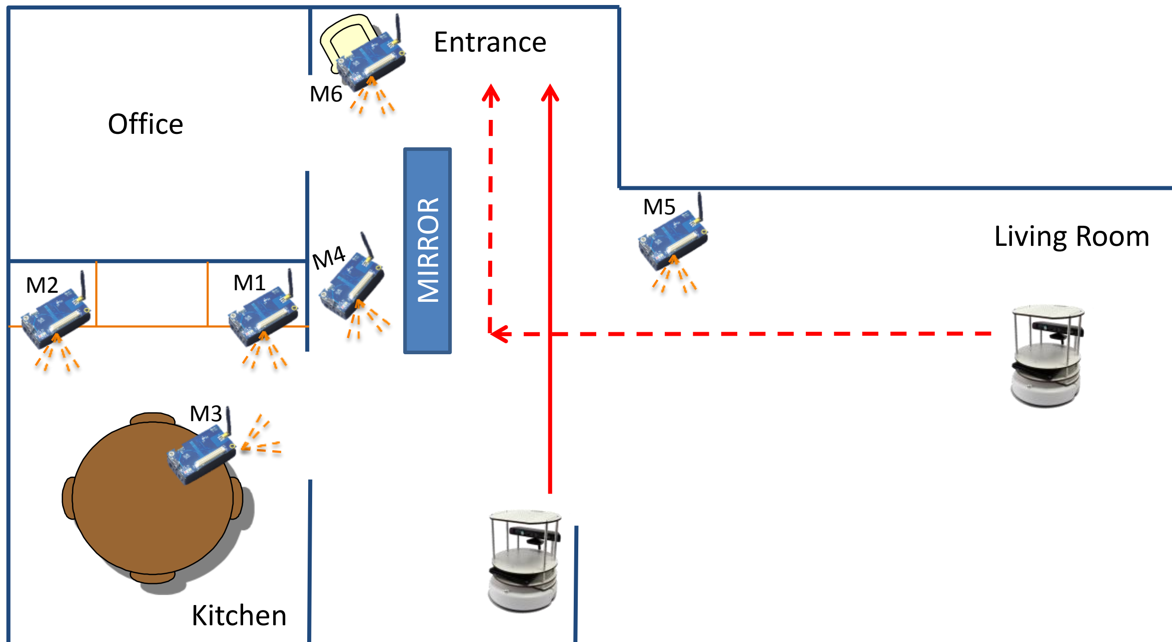



*Figure 33 Angen Experimental Scenario: a mobile robot using laser range-finder localization is performing two types of trajectories ending in the kitchen, represented as continuous and dashed lines. The performance of the localization system is negatively affected by the introduction of a mirror; this effect can only be noticed in the dashed trajectory type. All the sensors are equipped with PIR sensors, but only M3 and M6 are in a line-of-sight that is influenced by the robot moving. The rest of the PIRs is shielded and is not activated by robot motion.*

### Test 3 Description

The third test script, implemented by the `supervisorEntity.set_configuration_test_feature_filter()` method, is organized as follows

1. Load training data from file using the `test.AngenDataParser` helper class to create a dataset.
2. Create a stand-alone `FeatureFilter` object and supply it with the dataset.
3. Run feature filtering and publish information on elapsed time in the different steps of the process.
4. Publish information on selected and deleted features.

### Test 3 Outcome

The test script has been repeated for varying configurations (and number) of the input features, ranging from the worst-case scenario with all transducers from all the 6 motes, to the simplest configuration with only the transducers readings from the task-relevant M3 motes. For each configuration it has been recorded the result of the filtering phase, reported in Table XXI, as well as

the elapsed time spent in the key parts of the filtering algorithm, i.e. the construction of the redundancy mask, the autocorrelation noise filter and the iterative selection-elimination procedure.

Figure 34 shows the output of the test script on the worst case configuration, i.e. number 5 in Table XXI. Only 6 features are retained from an initial set of 27: four (out of six) of them are relevant features based on our expert ground truth, as they encode relevant information for the task at hand.

*Table XXI Feature Selection on the Angen data: results are provided for different configurations of the input features, e.g. M3 means that all transducers from mote 3 are provided as inputs. The ground-truth task-relevant features (determined by expert knowledge) are marked in red.*

| ID | Configuration | N. Initial Features | Selected Features |
|---|---|---|---|
| 1 | M3 + (X,Y,θ) | 7 | X, Y, PIR3 |
| 2 | M3+M6+(X,Y,θ) | 11 | X, Y, PIR3 |
| 3 | M4-M6 + (X,Y,θ) | 15 | Y, PIR4, PIR6, LIGHT4 |
| 4 | M3-M6 + (X,Y,θ) | 19 | X, Y, PIR3, PIR6 |
| 5 | M1-M6 + (X,Y,θ) | 27 | X, Y, PIR1, PIR2, PIR3, PIR6 |

Overall, the results in Table XXI show that the algorithm is capable of consistently reducing the number of input features by maintaining the majority (if not all) of the relevant features in all the configurations considered. Figure 34 also shows that the computational complexity is dominated by computation of the redundancy mask, while the effort required for the iterative selection-elimination algorithm is negligible (it is equal to 0 milliseconds for initial configurations comprising less than 6 motes). A snapshot of the relationship between the number of initial features and the time required to complete the feature selection process is provided by the plot in Figure 35.

File Edit Navigate Search Project Run Window Help

Type your command

Problems | Javadoc | Declaration | Console

```
<terminated> LearningLayer (4) [Java Application] /usr/lib/jvm/java-6-openjdk-amd64/bin/java (Sep 30, 2013 12:33:55 PM)
/home/bacciu/workspace/LearningLayerAPI2.0/data/Angen/Entrance/AllInOne/in_exp218_s6_kin.csv
/home/bacciu/workspace/LearningLayerAPI2.0/data/Angen/Entrance/AllInOne/out_exp218_kin.csv
[FEATURE FILTER] TIME -> Redundancy mask computed in 2153 milliSec
[FEATURE FILTER] TIME -> Autocorrelation filetering completed in 0 milliSec
[FEATURE FILTER] TIME -> Feature Filtering completed in 1 milliSec
[FEATURE FILTER] Selected 21: pirS6
[FEATURE FILTER] Selected 25: pos_yS6
[FEATURE FILTER] Selected 5: pirS2
[FEATURE FILTER] Selected 1: pirS1
[FEATURE FILTER] Selected 9: pirS3
[FEATURE FILTER] Selected 24: pos_xS6
[FEATURE FILTER] Deleted 2: temperatureS1
[FEATURE FILTER] Deleted 3: humidityS1
[FEATURE FILTER] Deleted 6: temperatureS2
[FEATURE FILTER] Deleted 7: humidityS2
[FEATURE FILTER] Deleted 8: lightS3
[FEATURE FILTER] Deleted 10: temperatureS3
[FEATURE FILTER] Deleted 11: humidityS3
[FEATURE FILTER] Deleted 14: temperatureS4
[FEATURE FILTER] Deleted 15: humidityS4
[FEATURE FILTER] Deleted 17: pirS5
[FEATURE FILTER] Deleted 18: temperatureS5
[FEATURE FILTER] Deleted 19: humidityS5
[FEATURE FILTER] Deleted 22: temperatureS6
[FEATURE FILTER] Deleted 23: humidityS6
[FEATURE FILTER] Deleted 0: lightS1
[FEATURE FILTER] Deleted 26: orientationS6
[FEATURE FILTER] Deleted 13: pirS4
[FEATURE FILTER] Deleted 4: lightS2
[FEATURE FILTER] Deleted 16: lightS5
[FEATURE FILTER] Deleted 20: lightS6
[FEATURE FILTER] Deleted 12: lightS4
[Supervisor] : Thread terminated
[LN Wrapper] : Thread terminated
[GATEWAY Wrapper]: Thread terminated.
```

*Figure 34 Results of the feature selection algorithm on configuration 5 for the Angen data (see Table XXI), including all transducers from all the involved motes. The majority of the running time is spent on the computation of the redundancy mask, while the effort required for feature filtering is negligible. Only 6 features are retained from an initial set of 27.*

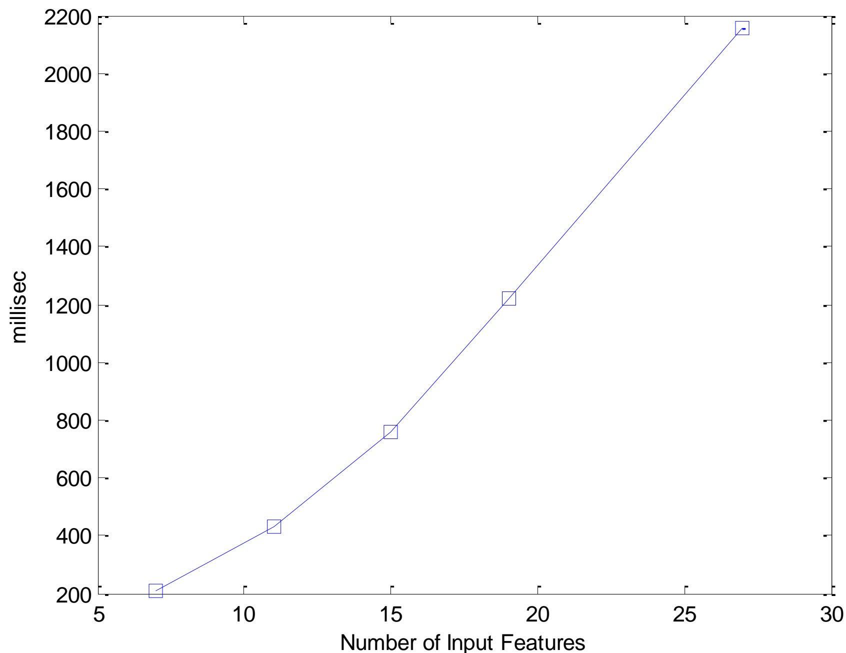



*Figure 35 Plot of the elapsed time(milliseconds) for feature filtering as a function of the number of input features.*

### *Test 4 Description*

The fourth test re-uses the same Exercising event data, device configuration and the same script used in Section 3.2.2 for assessing Java learning modules. Only, the test script `set_configuration_demo_java()` is modified to

- post appropriate PEIS messages to switch-on feature selection on the new task;
- to load an extended set of input information which includes humidity and temperature reading for the ankle and wrist motes, in addition to the 2D accelerometers.

The script runs as follows

1. Creates the main `LearningLayer` object and the associated components and waits for the bring-up message on the `PeisSymbols.SUPERVISOR_INFO` tuple.
2. Triggers learning of a new task by sending wiring and task instructions on the `PeisSymbols.WIRING_CMD` tuple: source transducers include humidity, temperature, 2D accelerometers of the wrist and ankle motes.
3. Activates feature selection on the task by posting the the `PeisSymbols.FEAT_SELECTION_ON` command in the `PeisSymbols.CONTROL_CMD` tuple.
4. Loads training data from file and forwards it to the Training Agent component through the `PeisSymbols.TRAINING_SAMPLE` tuple.
5. When all training data has been sent, a new task is allocated, feature filtering is preliminary run on the dataset and a Java learning module is trained through a cross-validation procedure on the filtered data. The learning module is then deployed on a target device and the synaptic communication is appropriately configured.
6. Starts the synaptic streaming using by posting the `PeisSymbols.ACTIVATE_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple.
7. Collects and publishes LN predictions generated on the Java learning module using inputs from remote WSN devices. Keeps streaming predictions until the user enters the keyboard combination "q+RETURN".
8. Stops the synaptic streaming by posting the `PeisSymbols.STOP_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple, and activates termination signals in the Learning Layer components.

### *Test 4 Outcome*

Execution of this script progresses as discussed in Section 3.2.2 until all the dataset has been received. Then, instead of directly activating the cross-validation model selection process on the full dataset, the latter is provided in input to the feature filtering algorithm. Figure 36 shows the result of the selection process, which maintains the accelerometer and humidity information but deletes the redundant/irrelevant temperature information, requiring slightly less than one second to complete the process.

The feature filtered dataset is then supplied to the cross-validation process for models selection: the results in Figure 36 shows that the selected Java learning module attains a test-set accuracy equivalent to the manually filtered version in Section 3.2.2.

Before deploying the learning module and the associated synaptic connections, the `TrainingAgent` exploits the `FeatureFilter` to clean the original wiring instructions from the irrelevant features, thus obtaining a deployed learning task with reduced computational and communication effort.

The rest of the script progresses as in Section 3.2.2, with the successful deployment of the task and the streaming of the predictions from the Java learning module to the Learning Layer output interface.

```
[FEATURE FILTER] TIME -> Redundancy mask computed in 10452 milliSec
[FEATURE FILTER] Autocorrelation Filter removing features....
[FEATURE FILTER] TIME -> Autocorrelation filetering completed in 0 milliSec
[FEATURE FILTER] Being Protective in RULE 2
[FEATURE FILTER] TIME -> Feature Filtering completed in 0 milliSec
[FEATURE FILTER] Selected 0: T_ACCELX_720897
[FEATURE FILTER] Selected 1: T_ACCELY_720897
[FEATURE FILTER] Selected 3: T_HUMID_720897
[FEATURE FILTER] Selected 4: T_ACCELX_720898
[FEATURE FILTER] Selected 5: T_ACCELY_720898
[FEATURE FILTER] Selected 7: T_HUMID_720898
[FEATURE FILTER] Deleted 2: T_TEMP720897
[FEATURE FILTER] Deleted 6: T_TEMP720898
[NETWORK MIRROR] Starting Cross-Validation on 2 models
[NETWORK MIRROR] Training ESN Configuration 0 (ReservoirSize:100, Regularization:0.01, Leaky:0.2)
[NETWORK MIRROR] Trained ESN n.0 with CV performance 0.82
[NETWORK MIRROR] Trained ESN n.1 with CV performance 0.8650595
[NETWORK MIRROR] Trained ESN n.2 with CV performance 0.81577384
[NETWORK MIRROR] Training ESN Configuration 3 (ReservoirSize:100, Regularization:0.001, Leaky:0.2)
[NETWORK MIRROR] Trained ESN n.3 with CV performance 0.8327976
[NETWORK MIRROR] Trained ESN n.4 with CV performance 0.86863095
[NETWORK MIRROR] Trained ESN n.5 with CV performance 0.8205358
[NETWORK MIRROR] ESN number 4 has the maximum Accuracy
[NETWORK MIRROR] Average accuracy on the test samples = 0.81870383
[NETWORK MIRROR] End of Cross Validation
[TRAINING AGENT] Training completed for Task 1
```



*Figure 36 Result of the feature selection mechanism when running incremental learning on the Exercising event.*

## 3.4 TESTING ONLINE LEARNING

Online learning functionalities have been tested on several benchmark real-world tasks.

A first point to show is that ESN modules trained using the online mechanisms implemented in the Learning Layer APIs can reach comparable performances to ESN modules trained with standard offline methods, starting from the same conditions. To this aim, we designed experiments in which the same ESNs undergo an offline training process and an online training process on the same training data, starting from the same initial conditions. Figure 37 and Figure 38 compare the mean accuracy on the test set for the tasks AAL Home Lab-Preparing Coffee and Indoor User Movement Forecasting, respectively, achieved by ESNs with increasing reservoir dimension, trained with (1 epoch of) online learning, for an increasing number of training patterns used for the online learning algorithm, and with offline learning. It can be observed that the test set accuracy of the networks trained using online learning approaches, or even overcomes, that achieved by the networks trained using offline learning, as the number of online training samples approaches the entirety of the training set. Analogously, Figure 39 shows a similar comparison between the mean absolute test

error of ESNs trained using online learning and the same networks trained using offline learning on the Planner Weight Prediction for Localization Systems-Mirror task. In this case it can be seen that, for each choice of the reservoir dimensionality, after a number of online learning samples corresponding to the 25% (or less) of the whole training set the performance of the networks trained using online learning overcomes that achieved by the same networks trained with offline learning. We can therefore conclude that the online learning mechanisms designed and implemented in the Learning Layer APIs can be used to reach analogous performances as the standard offline learning algorithms, starting from the same conditions. For the sake of completeness, Figure 40 presents results analogous to Figure 39, but corresponding to a standard RLMS algorithm setting for regression tasks, in which the target output is known at each time step, and can be used directly for the reservoir-to-readout weights values adjustment.

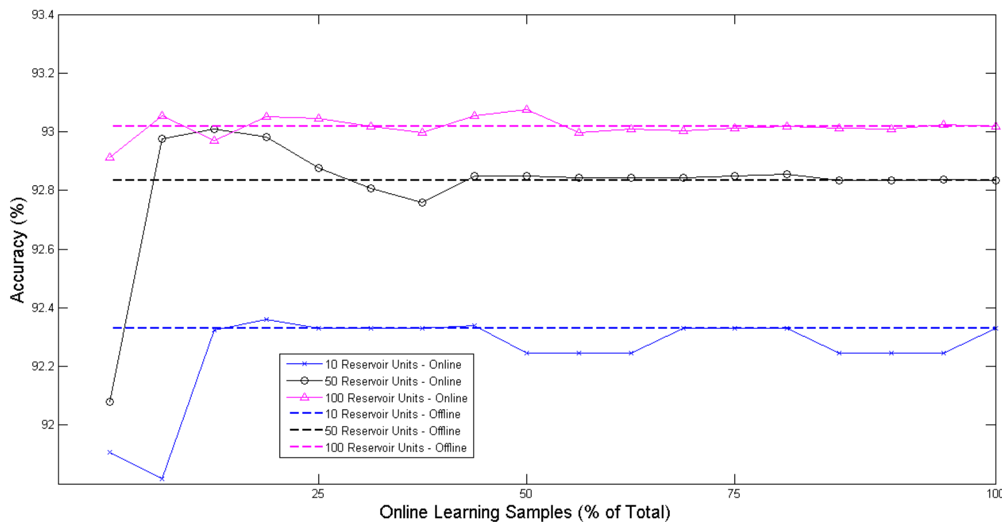



*Figure 37 Mean test accuracy on the AAL Home Lab-Preparing Coffee task, achieved by ESNs with different reservoir dimensions, trained using offline learning (dashed lines) and online learning (continuous lines), for increasing number of training samples considered.*

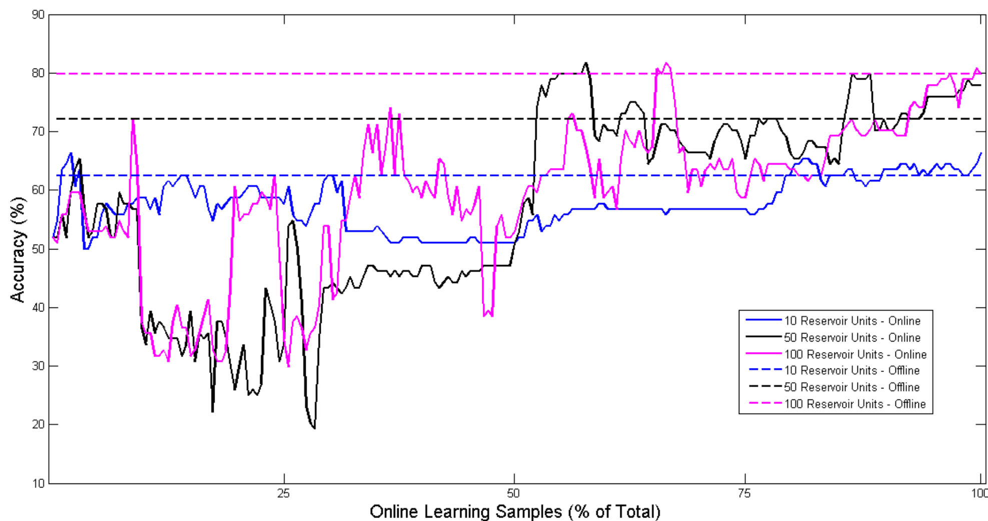



*Figure 38 Mean test accuracy on the Indoor User Movement forecasting task, achieved by ESNs with different reservoir dimensions, trained using offline learning (dashed lines) and online learning (continuous lines), for increasing number of training samples considered*

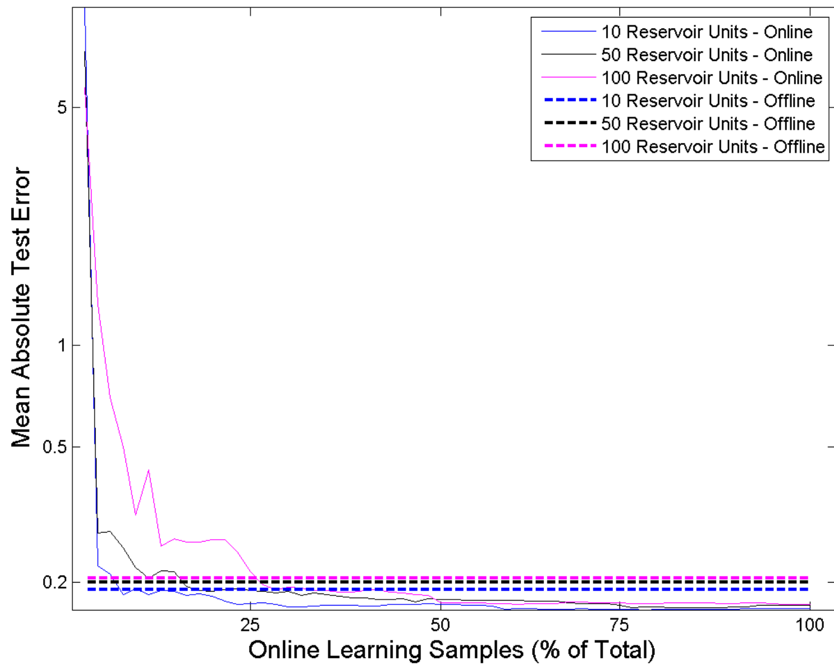



*Figure 39 Mean absolute test error on the Angen Scenario-Mirror task, achieved by ESNs with different reservoir dimensions, trained using offline learning (dashed lines) and online learning (continuous lines), for increasing number of training samples considered. Y-axis is in logarithmic scale.*

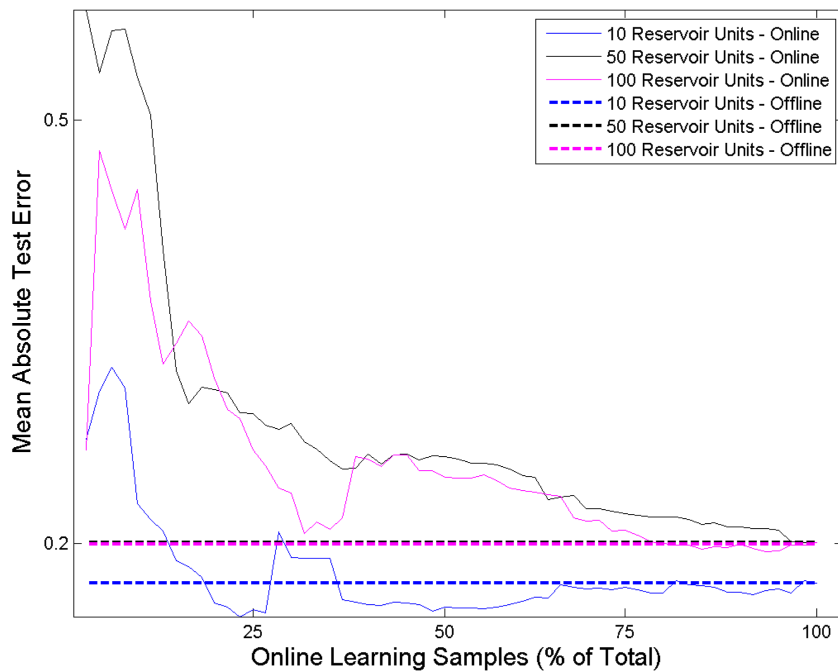



*Figure 40 Mean absolute test error on the Angen Scenario-Mirror task, achieved by ESNs with different reservoir dimensions, trained using offline learning (dashed lines) and online learning (continuous lines), for increasing number of training samples considered. Standard RLMS with known target is used. Y-axis is in logarithmic scale.*

A second important point to show is that online learning can be useful to improve the prediction performance of already trained ESN modules, whenever new training samples became available. To this aim, we have designed experiments in which ESNs trained offline on a training set, undergo a process of online learning on a separate test set. Figure 41 shows the comparison between the test accuracy achieved by ESNs on the Indoor User Movement Forecasting task, with different reservoir dimensions, before and after online learning (for increasing number of online test samples used). It can be seen that the online learning process has a great impact on the ESN models performance, leading to an increase of more than 20% of test accuracy. The effectiveness of the online learning mechanisms in this context is clearly apparent.

Online learning mechanisms have also been tested in the Java API implementation.

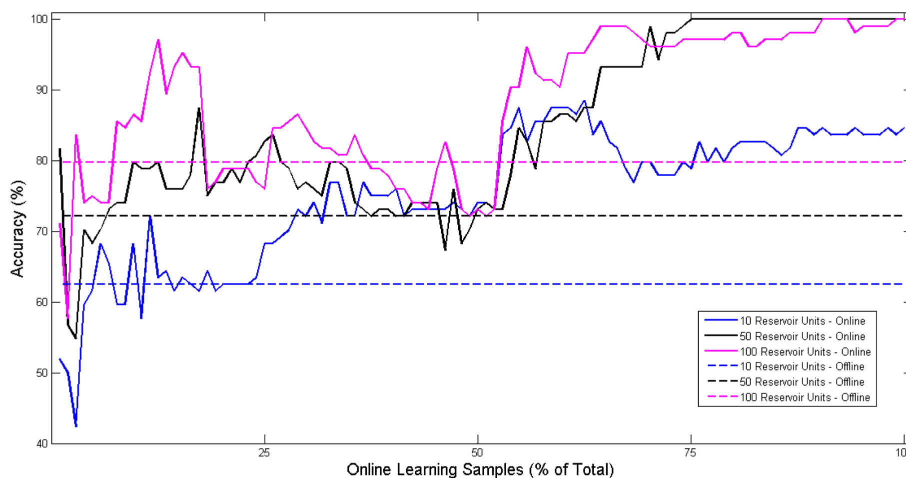



*Figure 41 Mean test accuracy on the Indoor User Movement forecasting task, achieved by ESNs with different reservoir dimensions, trained using offline learning, before (dashed lines) and after (continuous lines) online learning, for increasing number of training samples considered*

### Test 5 Description

This test is used to assess the online learning mechanism, showing a process of online training of an ESN module and subsequent predictions. To this end, we use the Waving Activity task and train an ESN model with a very small subset of the training data. Then we simulate an online process in which the input samples in the Waving Activity dataset are provided one-by-one to the learning module, while the online learning mode is active. The script is implemented in the SupervisorEntity method set_configuration_demo_online(), and it runs as follows:

1. Creates the main `LearningLayer` object and waits for the bringing message on the `PeisSymbols.SUPERVISOR_INFO` tuple.
2. Triggers learning of a new task by sending appropriate wiring and task instructions on the `PeisSymbols.WIRING_CMD` tuple.
3. Loads training data from file and forwards a small subset of it to the Training Agent component through the `PeisSymbols.TRAINING_SAMPLE` tuple.
4. The learning module is then deployed on a target device and the synaptic communication is appropriately configured.
5. Starts and proceed with the online learning process by posting the `PeisSymbols.ONLINE_EVENT`, or `PeisSymbols.ONLINE_NOEVENT` command (depending on the target for the different online training samples) in the `PeisSymbols.ONLINE_REFINE` tuple. In this phase the input for the ESN module is forced to be as in the training dataset.

6. When all the training samples have been presented, the online learning is stopped by posting the command PeisSymbols.STOP_ONLINE in the PeisSymbols.ONLINE_REFINE tuple.
7. Starts the synaptic streaming using by posting the `PeisSymbols.ACTIVATE_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple.
8. Collects and publishes LN predictions generated on the Java learning module using inputs from remote WSN devices. Keeps streaming predictions until the user enters the keyboard combination “q+RETURN”.
9. Stops the synaptic streaming by posting the `PeisSymbols.STOP_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple, and activates termination signals in the Learning Layer components.

### *Test 5 Outcome*

The execution of this script proceeds as described in Section 2.5.2, with the exception of the preliminary reduced offline training phase (see Figure 42). When the online learning phase terminates, the ESN starts providing predictions for the Waving Activity task (see Figure 43).

```
Problems  @ Javadoc  Declaration  Console  Call Hierarchy
sequence-to-sequence
[PEIS Wrapper] : Got Wiring and Task Information <T_LIGHT_720897,IN_LIGHT_720897><T_ACCELX_720897,IN_ACCELX_720897><T_ACCELY_
{classification}{sequence-to-sequence}{sensory}{event}
[Training Agent] : Allocated new task 1
======> Ready to accept TRAINING DATA for new task
[PEIS Wrapper] : Got sample(63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63.0,63
[PEIS Wrapper] : Got sample (29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,2
======> Train and deploy task when enough data is available

[TRAINING AGENT] Training completed for Task 1
                                                    Online Learning Active
======> Start online learning phase
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
[PEIS WRAPPER] : Online event
```

*Figure 42 Outcome of the online learning script after the activation of the online learning process.*

```
Problems  @ Javadoc  Declaration  Console  Call Hierarchy
[DEMO] Activate Forward Computation
[GATEWAY Wrapper]: Sending a Broadcast message
Mote 0 WAVING -> Confidence 1.4E-45
Mote 0 STILL -> Confidence 0.0
Mote 0 WAVING -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 STILL -> Confidence 1.0
Mote 0 WAVING -> Confidence 1.0
Mote 0 WAVING -> Confidence 1.0
Mote 0 WAVING -> Confidence 1.0
Mote 0 WAVING -> Confidence 1.0
Mote 0 WAVING -> Confidence 1.0
```

*Figure 43 Outcome of the online learning script during the phase of output prediction.*

## 3.5 TESTING RECOVERY

### 3.5.1 Test Configuration

The assessment of the recovery procedure in the CLS API V2.0. entails testing

- the triggering of the watchdog detecting mote disconnection ;
- the identification of a suitable replacement device to host the learning module of the failing device;
- the over-the-air deployment of the learning module from the `NetworkMirror` copy to the replacement mote, together with its associated synaptic connections.

The hardware configuration used in the tests comprises:

- 1 PC (coherent with the platform requirements in 1.4.2.1) running the JLN API.
- 1 sink mote (coherent with the platform requirements in 1.4.2.2) attached to the PC through USB and running the sink-specific code of the NesC LM API.
- 2 motes (coherent with the platform requirements in 1.4.2.2) equipped with light, temperature, accelerometer and humidity transducers and running the NesC LM API.

### 3.5.2 Testing Mote Watchdog and Recovery

This test is intended to assess the correct triggering of the disconnection-watchdog for mote devices and the subsequent execution of the recovery process enabling the re-deployment on a replacement device of the computational learning task hosted by the failing mote. To this end, we have employed real-world data collected on a simple task comprising the recognition of the user performing a waving movement with his/her hand, while holding a mote equipped with 2D accelerometers. The Waving task is first learned and deployed to the initial mote using the incremental learning mechanism as in Section 3.2.2. Then, mote failure is simulated by powering off of the device.

#### *Test 6 Description*

The test script, implemented by the `supervisorEntity.set_configuration_demo_recovery()` method, is organized as follows

1. Creates the main `LearningLayer` object and the associated components; activates the recovery procedure by setting a watchdog threshold > 0 (number of clock tick before considering a device disconnected).
2. Waits for the bring-up message on the `PeisSymbols.SUPERVISOR_INFO` tuple.
3. Triggers learning of a new task by sending wiring and task instructions on the `PeisSymbols.WIRING_CMD` tuple.
4. Loads training data from file and forwards it to the Training Agent component through the `PeisSymbols.TRAINING_SAMPLE` tuple.
5. When all training data has been sent, a new task is allocated, a TinyOS learning module is trained through a cross-validation procedure and deployed on a mote and the synaptic communication is appropriately configured.
6. Starts the synaptic streaming using by posting the `PeisSymbols.ACTIVATE_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple.
7. Collects and publishes LN predictions generated by the learning module on-board the mote.
8. A mote failure is produced by powering off the active device.

9. Stops the synaptic streaming by posting the `PeisSymbols.STOP_FORWARD` broadcast command in the `PeisSymbols.CONTROL_CMD` tuple, and activates termination signals in the Learning Layer components.

### Test 6 Outcome

At start-up, the Learning Layer activates its PEIS and Gateway interface to receive messages from the other RUBICON layers: it waits for one mote joining the ecology (see Figure 44), that is assigned the nodeID = 720897. The script progresses as in Section 3.2.2, with the Learning Layer receiving, through its PEIS interface, the wiring and task information together with the training data necessary to allocate the new Waving task. When enough data is received, the TrainingAgent triggers model selection and training and finally deploys the trained learning module on the device with nodeID = 720897 (see Figure 45).

As soon as the deployment of the learning module and synaptic connections is completed, the ESN located on the mote 720897 starts providing predictions for the waving task (see Figure 46).

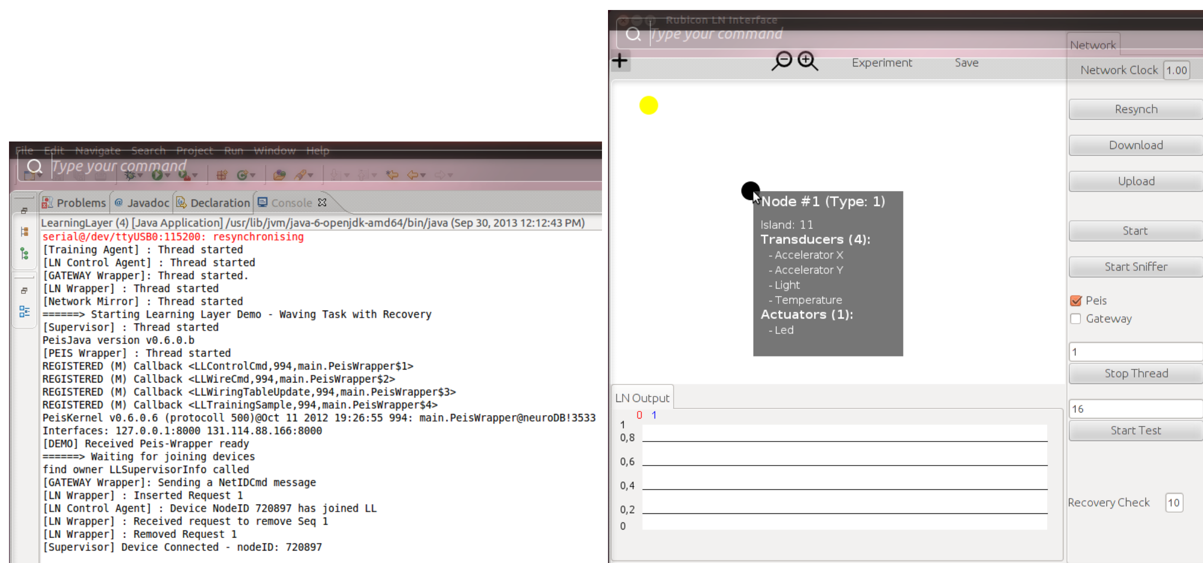



*Figure 44 The script awaits joining of at least one device, that is assigned nodeID=720897 and will initially host the Waving learning task (console and GUI output on the left and right, respectively).*

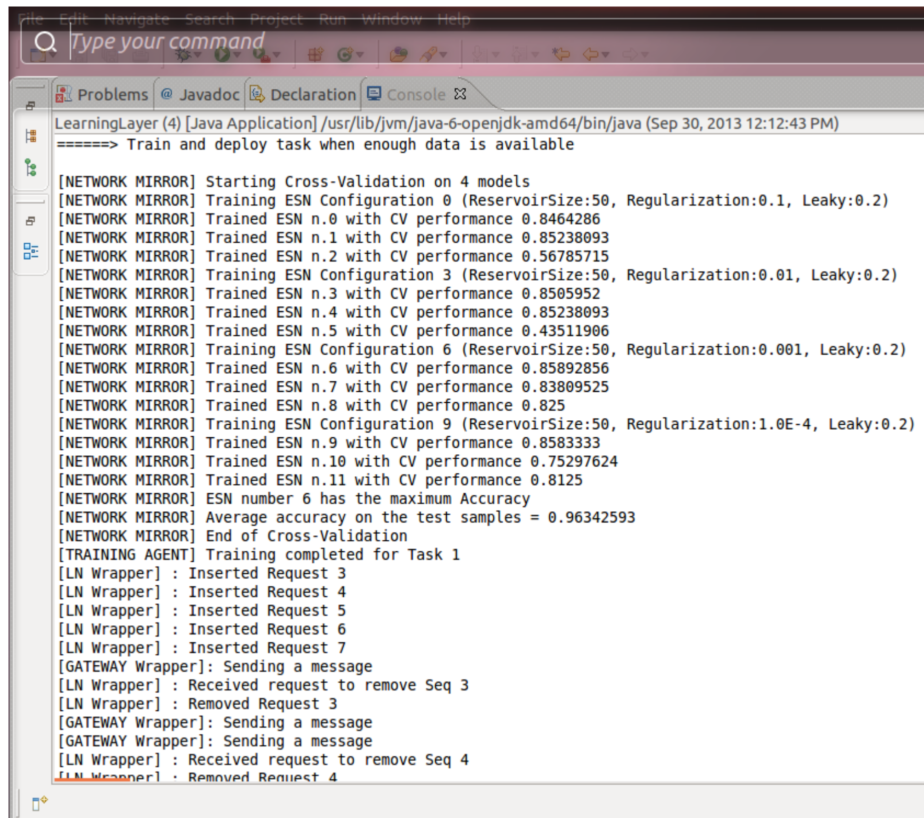



*Figure 45 Training and deployment of the Waving task on the device with nodeID = 720897*

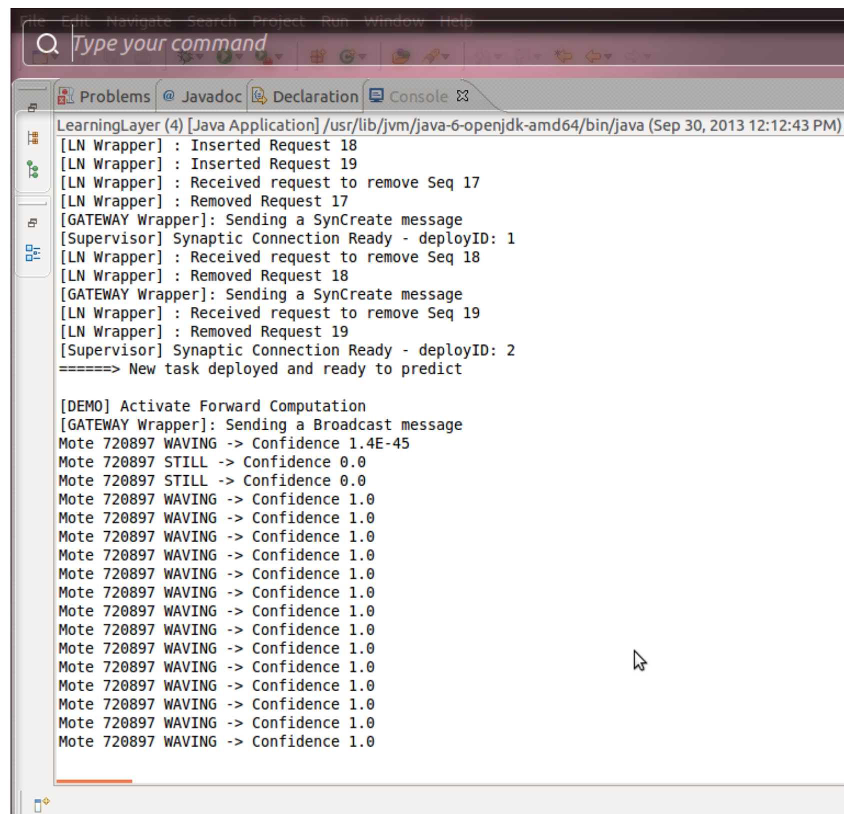



*Figure 46 The Learning Layer providing predictions on the Waving event using the output generated by the ESN on-board the mote with nodeID = 720897.*

While the Learning Layer keeps providing its prediction a new mote joins the ecology (with nodeID 720898) and is kept as a cold spare, e.g. waiting for a new learning task (Figure 47).

At some point in time, the first device (nodeID = 720897) is powered off to simulate failure. After a number of RUBICON cycles equal to the value of the disconnection threshold set by the script (i.e. 10), the mote watchdog detects the failure on mote 720897 and triggers the recovery procedure.

Figure 48 shows the output produced by the deployment of the Waving task on the cold spare 720898. This includes messages transmitting the ESN parameters as well as the appropriate configuration commands to setup the new synaptic communication where mote 720898 replaces the failed 720897 device. The recovery procedure successfully concludes by activating the learning module, that starts providing (again) the prediction associated with the Waving event (see bottom-right of Figure 48).

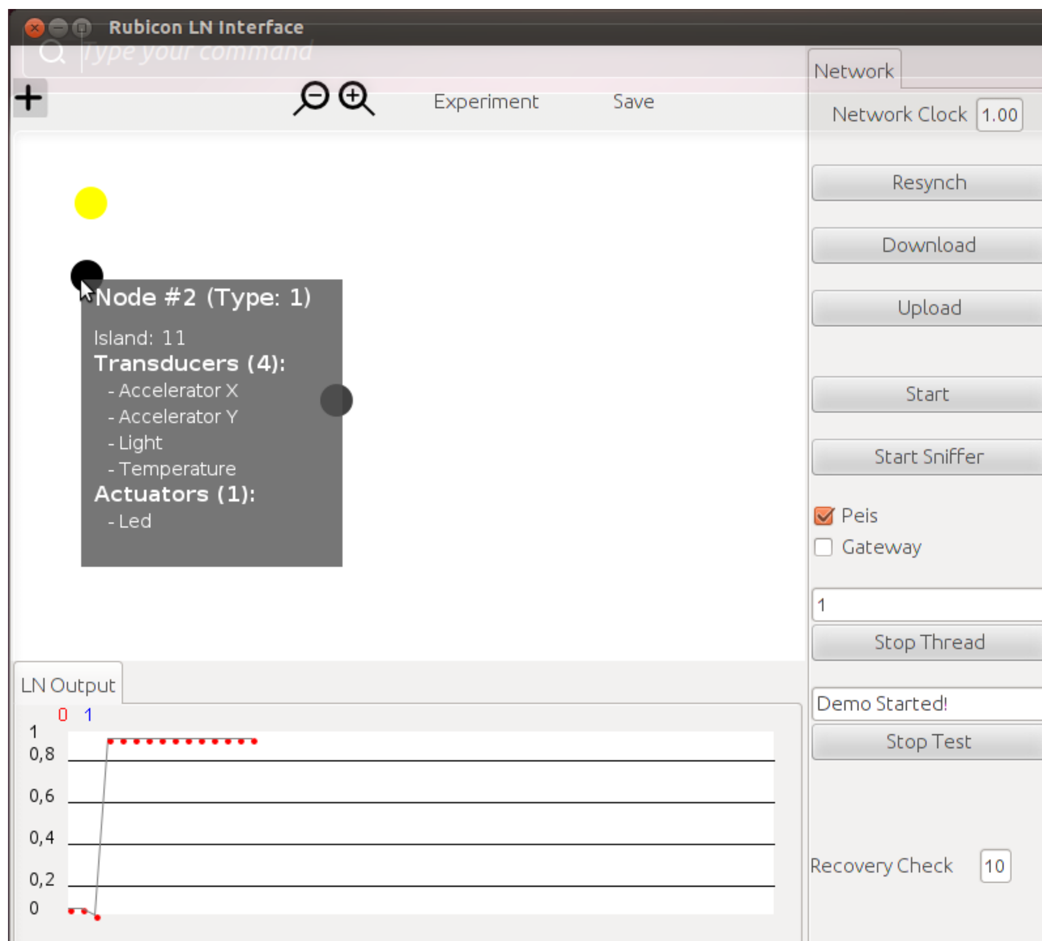



*Figure 47 Another device, with nodeID = 720898, joins the ecology while the other mote keeps providing predictions.*

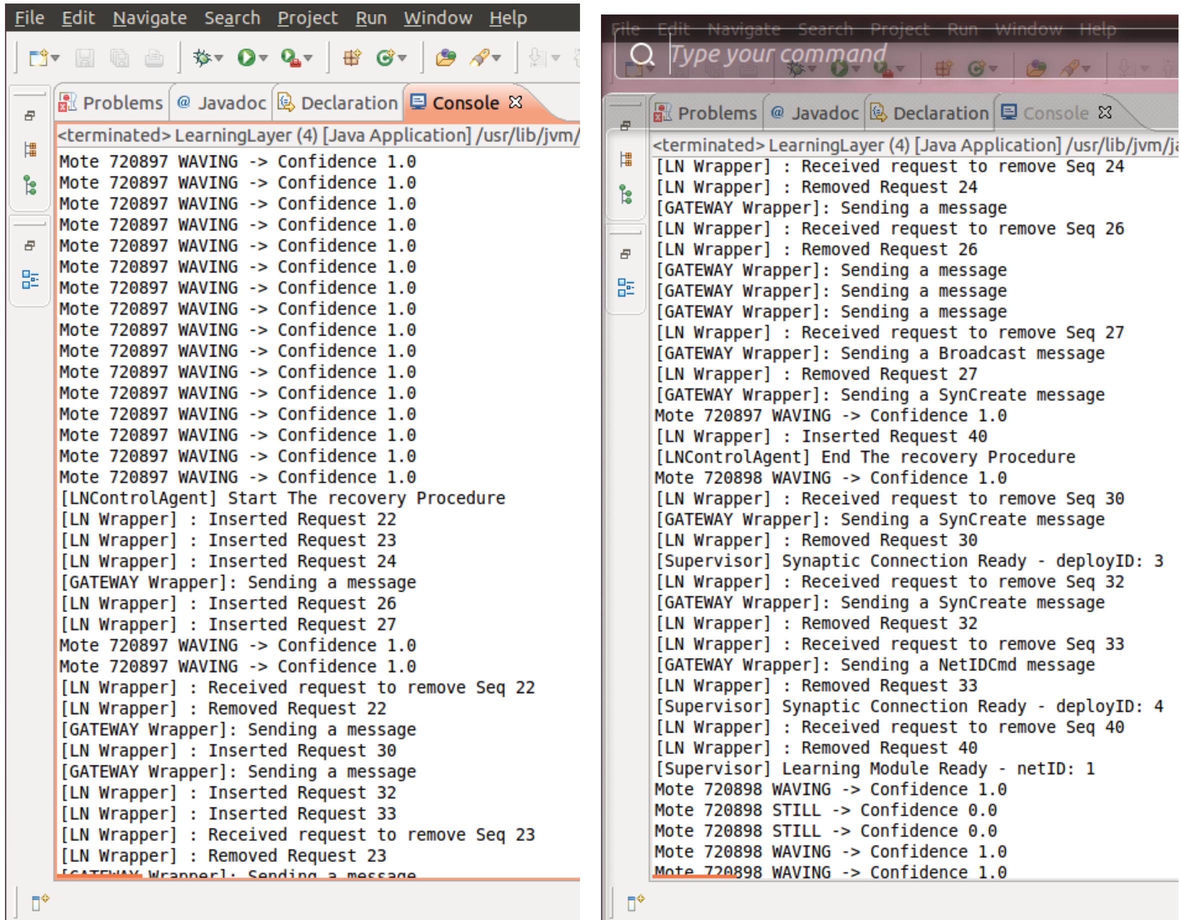



*Figure 48 Recovery procedure triggered by the failure of device 720897 (left): the cold spare with nodeID 720898 is found to have compatible capabilities and is therefore used as a replacement. The learning module implementing the Waving task is deployed on 720898 and starts providing again its predictions (right).*

# 4. Conclusions

## 4.1 Compliance to Workplan

We conclude this report with an overview of the status of the WP2 tasks and Deliverables at Month 30, discussing their compliance with the workplan detailed in the RUBICON DoW.

*Task 2.1 - Learning Layer Specification* (M1-M6 + M12-M16): the activities have been performed as planned and the task is completed. The results of the first part of this task (M1-M6) are documented in D2.1, which provides the Learning Layer specification used to guide the implementation activities documented in this report. The result of the specification refinement phase (M12-M16) are documented in D2.2.

*Task 2.2 - Core Learning Services* (M6-M26): the activities have been performed as planned and the task is completed. The preliminary version of the Learning Layer infrastructure has been delivered with the CLS API V1.0 in D2.2. The final version of the infrastructure is delivered now with D2.3. A summary of the key functionalities is documented in this report as follows:

- Section 2.2.3 describes the Synaptic Communication mechanism ;
- Section 2.2.5 describes the software component wrapping the distributed learning network functionalities ;
- Section 2.3 describes the implementation of the manager component of the Learning Layer ;
- Section 2.4 describes the implementation of the Training Manager subsystem;

Additionally, Sections 2.3.4, 2.3.4 and 2.4.2 describe the implementation of wiring-table utilities, of a recovery mechanism and of a feature-selection procedure supporting the self-sustaining and self-configuration capabilities of the Learning Layer, through automated methods for device and learning module management and control.

*Task 2.3 - Local Learning* (M8-18): the activities have been performed as planned and the task is now completed. The results of the task are documented in the CLS API V1.0 and in its associated report.

*Task 2.4 - Distributed Adaptive Memory* (M16-24): ): the activities have been performed as planned and the task is completed. The results of the task are documented in the CLS API V2.0 and in this associated report. In particular:

- Section 2.2.2 describes the NesC and Java components implementing the local learning modules ;
- Section 2.2.4 describes the implementation of the distributed forward computation producing the LN predictions ;
- Sections 2.4.3.1 and 2.4.4 describe the training of a distributed learning task , its deployment to the LN device and a summary of the required instructions for the allocation and training of a new task, respectively.
- Section 2.4.2 describes the feature selection functionalities and the interaction of the mechanism with the incremental learning of a new task.

*Task 2.5 - Cooperative Cognitive Process* (M23-30): ): the activities have been performed as planned and the task is completed. The results of the task are documented in the CLS API V2.0 and in this associated report. In particular:

- Section 2.4.3.2 describes the implementation of the online learning mechanism for the on-board adaptation/refinement of deployed learning task using instantaneous teaching signals (behavioural rewards).

*Deliverable D2.1 - Functional Design & Specification document* (M6): it has been delivered as scheduled in the form of a report documenting the detailed architecture of the RUBICON Learning Layer and the specification of its software components.

*Deliverable D2.2 - Core Learning Services API and Documentation, version 1.0* (M18): ): it has been delivered as scheduled in the form of a the CLS V1.0 software, the accompanying report, providing an high-level description of the developed software, and the Javadoc documentation, whose full HTML version is available in the RUBICON repository.

*Deliverable D2.3 - Core Learning Services API and Documentation, version 2.0* (M30): delivered in the form of the Core Learning Service (CLS) API V2.0 available in the (timestamped) RUBICON software repository, along with this report, providing an high-level description of the developed software, and the Javadoc documentation, whose full HTML version is available in the RUBICON repository. The CLS API V2.0 contains all the code needed for the execution of the Learning Layer and the example scripts needed to replicate the testing described in Section 3.

The deliverable description, as per RUBICON DoW, is the following:

*"CLS API V2.0 will extend release 1.0 (code, documentation and testing) to include the following features:*

*A1. Implementation of the Distributed Adaptive Memory Model: Distributed learning mechanism*

*A2. Implementation of the Distributed Adaptive Memory Model: Feature selection mechanism*

*A3. Implementation of the Cooperative Cognitive Process"*

These points are addressed in this report as follows:

- The implementation of the distributed learning functionalities (A1) is described in:
  - Section 2.2: implementation of distributed learning models and synaptic communication mechanisms for TinyOS and Java-enabled devices;
  - Section 2.4.3.1: implementation of incremental learning of a distributed task through a mirror-based batch training and its deployment to the LN;
  - Section 3.2: testing of distributed incremental learning with TinyOS and Java-enabled devices.
- The implementation of the feature selection mechanism (A2) is described in:
  - Section 2.4.2: implementation of the feature selection mechanism and its interaction with the distributed learning functionalities;
  - Section 3.3: testing of the feature selection mechanism.
- The implementation of the refinement learning functionalities of the cooperative cognitive process (A3) is described in:
  - Section 2.4.3.2: implementation of online learning for the on-board refinement of deployed learning task both with TinyOS and Java-based learning modules;
  - Section 3.4: testing refinement online learning on-board TinyOS and Java learning modules.

The increment of D2.3 with respect to the functionalities in D2.2 has been summarized in Section 1.5.

## 4.2 Impact on project

The Learning Layer development has progressed as expected and has been made available to the rest of the RUBICON project partners through the shared code repository.

The final version of the software is currently in use within the Consortium. Based on the feedback of the partners and on the preliminary results on the final RUBICON application scenario, some minor modifications and code upgrades can be expected. Such minor modifications will be addressed until the conclusion of the project and the updated software will be made available throughout the project repository.

## 4.3 New developments and unforeseen issues

The RUBICON Learning Layer interfaces with the RUBICON gateway that provides the enabling mechanisms for interacting and configuring the distributed learning modules and the synaptic communication. The implementation of the RUBICON gateway functionalities is still in progress, so further updates to the CLS are expected to be delivered in case of key changes to the RUBICON gateway interface.

No other significant unexpected developments have occurred during WP2 research and development activities interested by the current deliverable.

## *References*

[1] M. Lukosevicius, H. Jaeger, Reservoir computing approaches to recurrent neural network training, Computer Science Review, 3(3), pages 127–149, 2009.

[2] H. Jaeger, Adaptive nonlinear system identification with echo state networks, In NIPS, Advances in Neural Information Processing Systems 15, pages 593–600, MIT Press, 2002.

[3] D. Bacciu, P. Barsocchi, S. Chessa, C. Gallicchio, A. Micheli, An Experimental Characterization of Reservoir Computing in Ambient Assisted Living Applications, Neural Computing and Applications, Springer-Verlag (In Press), doi: 10.1007/s00521-013-1364-4, 2013

[4] C. Gallicchio, A. Micheli, P. Barsocchi, S. Chessa, User movements forecasting by reservoir computing using signal streams produced by mote-class sensors, In: Mobile Lightweight Wireless Systems (Mobilight 2011), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 81, Springer Berlin Heidelberg, pages 151–168, 2012.

[5] D. Bacciu, C. Gallicchio, A. Micheli, S. Chessa, P. Barsocchi, Predicting user movements in heterogeneous indoor environments by reservoir computing, In: M. Bhatt, H.W. Guesgen, J.C. Augusto Editors, Proceedings of the IJCAI Workshop on Space, Time and Ambient Intelligence (STAMI), pages 1–6, 2011.

[6] D. Bacciu, S. Chessa, C. Gallicchio, A. Micheli, P. Barsocchi, An Experimental Evaluation of Reservoir Computation for Ambient Assisted Living, 22nd Italian Workshop on Neural Networks, Vietri sul Mare, Salerno, Italy, 17-19 May 2012, In: Neural Networks and Surroundings, Springer Smart Innovation, Systems and Technologies series, vol. 19, pp. 41-50, ISBN: 978-3-642-35466-3.

[7] S. Chessa, C. Gallicchio, R. Guzman, A. Micheli, Robot Localization by Echo State Networks using RSS, 23rd Italian Workshop on Neural Networks (WIRN) 2013 (To Appear).

[8] C. Gallicchio, A. Micheli, P. Barsocchi, S. Chessa, Reservoir computing forecasting of user movements from RSS mote-class sensors measurements, Technical Report TR-11-03, University of Pisa, 2011.

# 5. Appendix A – Reference Manual

In the following, we provide a snapshot of the full Javadoc reference manual for the Learning Gateway API, including only the package-level documentation. A full HTML version of the manual is available on the RUBICON SVN repository.

## Learning Gateway API Documentation

| **Package Summary** | | ***Page*** |
|---|---|---|
| **generics** | Provides the classes implementing the data structures shared across the Learning Layer subsystems, the standard interface of the Learning Layer threads, the inter-thread communication mechanisms as well as some general macros. | |
| **learningnetwork** | Provides the classes for accessing the distributed Learning Network, implementing methods to send command and configuration messages to the single learning modules and to receive status updates from the Learning Network. | |
| **main** | Provides the definition of the main Learning Layer object, as well as the classes necessary to create the output interface of the Learning Layer and the wrapper to access the PEIS functionalities. | |
| **manager** | Provides the classes implementing the Learning Network Manager (LNM) subsystem, that is responsible for the configuration and control of the Learning Layer. | |
| **test** | Includes example scripts and stub implementations for testing and experimenting with the Learning Layer API. | |
| **tinyos** | Provides the classes implementing the TinyOS messages used to deliver the Learning Layer control and configuration messages, as well as the synaptic communication between mote-devices. | |
| **training** | Provides the classes for accessing the Training Manager subsystem, implementing methods for instantiating and managing the datasets, training the learning modules and deploying them to the devices in the LN. | |

# Package generics

Provides the classes implementing the data structures shared across the Learning Layer subsystems, the standard interface of the Learning Layer threads, the inter-thread communication mechanisms as well as some general macros.

| Interface Summary | | *Page* |
|---|---|---|
| *LLRunnableInterface* | The interface provides methods for controlling the activation and termination of the Learning Layer threads. | |
| *LNInformationListener* | Interface that must be implemented by any object interested in receiving LL status messages through the LL information dispatchers. | |

| Class Summary | | *Page* |
|---|---|---|
| **DeviceRepository** | Repository of devices connected to the LL through the joining command `SupervisorInterface.connect_node(int, NodeInformation)` defined in the Supervisor Interface. | |
| **EchoStateNetwork** | This class implements the Echo State Network | |
| **EsnUpload** | Class encapsulating the readout weights of an ESN for transferring them from the LN Wrapper to the Network Mirror via message queues. | |
| **HashCodeUtil** | Collected methods which allow easy implementation of `hashCode`. | |
| **LNInformationDispatcher** | Defines the information dispatcher of the Learning Layer status messages defined in `LLMessages`. | |
| **LNInformationEvent<T>** | Defines a specific event object for the notification of LL status messages generated by the Learning Manager and directed towards its Supervisor, Control and Training interfaces. | |
| **LNInformationMsg<T>** | Class encapsulating the type and payload of LL status messages. | |
| **LNOutType** | **Deprecated.** | |
| **NodeInformation** | Specification of the devices sensing and actuator capabilities, as well as their basic network information, including island and node address. | |
| **PCLMRepository** | Repository for the PC LearnignModules. | |
| **Recover** | Implements the monitor of the deployed synaptic connections, that is used to track device behavior and determine the presence of failures. | |
| **SynapticChannel** | Class that encodes a SynapticChannel between PC devices. | |
| **SynapticConnection** | Models the synaptic connection abstraction: stores information on the source and destination peers of the connection, as well as its deployment status. | |
| **TaskData** | Implements the task abstraction used in the Network Mirror. | |
| **TaskType** | Contains the information associated to a computational task. | |
| **TrainingData** | Implements the Training Agent software component within the Training Manager in the RUBICON Learning Layer | |

| | | |
|---|---|---|
| **TrainingDataset** | Implements a training dataset, i.e. a collection of examples to be used for neural network training. | |
| **WiringSpecification** | This class implements an item in the Wiring Table described in Section 5.4.2 of Deliverable 2.1 | |
| **WiringTable** | Implements the Wiring Table as described in Section 5.4.2 of Deliverable 2.1 | |
| **WiringType** | Contains the information associated to a wiring instruction. | |

| **Enum Summary** | | ***Page*** |
|---|---|---|
| **DeviceRepository.DeviceNature** | Service class describing the nature of a device (PC or MOTE) | |
| **DeviceRepository.DeviceState** | Service class describing the current state of the device | |
| **LLMessages** | List of messages exchanged by the Learning Layer components either through message queues or by the information dispatchers of the `LNControlAgent`. | |
| **SynapticConnection.synapticState** | Enumerated type defining the deployment states of a synaptic connection | |
| **SynapticConnection.synapticType** | Enumerated type of synaptic connections | |
| **TaskData.taskState** | Service class describing the current state of a task | |

| **Exception Summary** | | ***Page*** |
|---|---|---|
| **SynapticConnection.SynapseException** | Exception thrown when creating a local synapse with improper parameters. | |

# Package learningnetwork

Provides the classes for accessing the distributed Learning Network, implementing methods to send command and configuration messages to the single learning modules and to receive status updates from the Learning Network.

The **LearningNetworkWrapper** class defines a unified point of access to the Learning Network and interfaces with the RUBICON gateway, that offers the communication facilities for the TinyOS-enabled devices, and with the PEIS messaging system for Java-enabled devices. The package includes a **GatewayWrapper** class that wraps the RUBICON Gateway functionalities needed by the Learning Layer to communicate with TinyOS devices. The classes **PCLMmanager** and **LearningModulePC** realize the Learning Modules for Java-enabled devices, together with the associated manager component.

| **Interface Summary** | | ***Page*** |
|---|---|---|
| ***GatewayInterface*** | Interface for sending LL configuration and control commands to the modules of the distributed Learning Network. | |
| ***LNMessageListenerInterface*** | Interface that must be implemented by objects interested in receiving messages from the Learning Network through the Gateway component. | |

| **Class Summary** | | ***Page*** |
|---|---|---|
| **GatewayWrapper** | Implementation of the `GatewayInterface` wrapping the Rubicon Gateway functionalities needed by the Learning Layer. | |
| **LearningModuleInputMsg** | Class used for delivering inputs from the RUBICON gateway to a Java Learning Module executed on a PC. | |
| **LearningModuleOutputMsg** | Class used for delivering inputs to the RUBICON Gateway from a Java Learning Module executed on a PC. | |
| **LearningModulePC** | Class implementing a Java Learning Module targeted to PC-like (non-motes) devices. | |
| **LearningNetworkWrapper** | Implements a component that abstracts the distributed nature of the Learning Network. | |
| **PCLMmanager** | Implements a stand-alone management component that runs on Java Enabled devices. | |

# Package main

Provides the definition of the main Learning Layer object, as well as the classes necessary to create the output interface of the Learning Layer and the wrapper to access the PEIS functionalities.

The constructor of the `LearningLayer` class instantiate all necessary objects of the Learning Layer. A component possessing the reference to a LearningLayer object can control the Learning Layer. Alternatively, the `LearningLayer` class implements a main method to activate a stand-alone instance of the Learning Layer from the command line.

| **Class Summary** | | ***Page*** |
|---|---|---|
| **LearningLayer** | Creates the main Learning Layer object. | |
| **LNOutputs** | Provides access to the LL predictions generated by the Distributed Learning Network. | |
| **PeisSymbols** | Provides the vocabulary of strings used as tuple-keys to implement the PEIS interface of the Learning Layer, as well as the `int` encoding of the commands written in the tuples. | |
| **PeisWrapper** | Realizes the PEIS interface between the Learning Layer and the Control and Cognitive Layer (Supervisors) Provides mechanisms for writing into the appropriate interface tuples and notifies the reception of messages to the appropriate Java components of the Learning Layer. | |

# Package manager

Provides the classes implementing the Learning Network Manager (LNM) subsystem, that is responsible for the configuration and control of the Learning Layer.

The LNM is implemented by a single software component implemented by the `LNControlAgent` class. The package defines three interfaces that serve to interact with the Learning Layer. The `SupervisorInterface` regulates the interaction with Supervisor components by providing methods for controlling the single devices and learning modules of the LN, as well as the activation of the forward computation phase. The `ControlInterface` complements the control and configuration functionalities advertised by the `SupervisorInterface`. The `TrainingInterface` regulates the interaction with the Training Manager subsystem by providing methods for controlling the deployment of synaptic connection bundles associated to computational learning tasks.

| **Interface Summary** | | ***Page*** |
|---|---|---|
| ***ControlInterface*** | Defines the LN Manager interface towards an high level controlling component such as the LL Graphical User Interface. | |
| ***SupervisorInterface*** | Defines the LN Manager interface towards an high level component acting as Supervisor, e.g. the Control and Cognitive Layer. | |
| ***TrainingInterface*** | Defines the LN Manager interface towards the Training Manager subsystem. | |

| **Class Summary** | | ***Page*** |
|---|---|---|
| **LNControlAgent** | Implement the LNM agent that collects configuration and control messages from the Supervisor and routes them to the components of the LL. | |

# Package test

Includes example scripts and stub implementations for testing and experimenting with the Learning Layer API.

The `supervisorEntity` class provides a test Supervisor component with example scripts that exercise the various learning layer functionalities. The `GatewayWrapperDummy` class provides a stub implementation of the RUBICON gateway that does not require the availability of a fully deployed Learning Network: it receives Learning Layer commands and respond with appropriate acknowledgement messages. The `AngenDataParser` and `TaskDataParser` provide helper functions to read and parse data files with training samples collected in real world ecology instantiations.

| **Class Summary** | | ***Page*** |
|---|---|---|
| **AngenDataParser** | Parser class to read files from the Angen Experimental Scenario | |
| **GatewayWrapperDummy** | Stub implementation of the `GatewayInterface` wrapping the Rubicon Gateway functionalities needed by the Learning Layer. | |
| **supervisorEntity** | Stand-alone implementation of the Supervisor component for testing and experimentation purposes. | |
| **TaskDataParser** | Parser class to read files from UNIPI's in-house experimental scenarios | |

# Package training

Provides the classes for accessing the Training Manager subsystem, implementing methods for instantiating and managing the datasets, training the learning modules and deploying them to the devices in the LN.

The `TrainingAgent` class defines the agent controlling the Training Manager subsystem: it receives training samples and requests for the instantiation of new computational learning tasks. The `NetworkMirror` component maintains an up-to-date mirror of the modules in the LN and performs the training routines. The `FeatureFilter` class implements the redundancy reduction algorithm for the filter phase of the feature selection mechanism (similarly, `FeatureWrapper` implements the relevance-guided mechanism). The Trainer component provides all the routines implementing model-selection and the cross-validation schemes used by the `NetworkMirror`.

| **Class Summary** | | ***Page*** |
|---|---|---|
| **FeatureFilter** | Implements the redundancy reduction step of the feature selection mechanisms. | |
| **FeatureWrapper** | Implements the relevance-guided step of the feature selection mechanisms. | |
| **NetworkMirror** | Implements the Network Mirror component: it maintains an up-to-date mirror of the modules in the LN and performs the training routines. | |
| **TaskDataParser** | A variant of the parser class in the test package, to read files from UNIPI's in-house experimental scenarios | |
| **Trainer** | Class that implements the CrossFold-Validation procedure on a learning task. | |
| **TrainingAgent** | Implements the Training Agent software component within the Training Manager in the RUBICON Learning Layer. | |

Java API documentation generated with DocFlex/Doclet v1.6.1

DocFlex/Doclet is both a multi-format Javadoc doclet and a free edition of DocFlex/Javadoc. If you need to customize your Javadoc without writing a full-blown doclet from scratch, DocFlex/Javadoc may be the only tool able to help you! Find out more at www.docflex.com