RUBICON
Robotic UBUquitous COgnitive Network

Project No.: 269914

D2.2 – Core Learning Services API and Documentation V.1.0

Editor:	**Davide Bacciu**	**UNIPI**
	Claudio Gallicchio	**UNIPI**
	Alessio Micheli (Supervisor)	**UNIPI**
	Claudio Vairo	**CNR**
Contributor(s):	**Stefano Chessa**	**UNIPI**
	Maurizio Bonuccelli	**UNIPI**
	Mauro Dragone	**UCD**
	Giuseppe Amato	**CNR**

Issue Date	30/09/2012 (M18, MS3)
Deliverable Number	D2.2
WP	WP3 - Learning Layer
Status	☐Draft ☐Working ☒Released ☐Delivered to EC ☐Approved by EC

Document history

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)

V	Date	Author	Description
1.0	08/10/2012	Davide Bacciu, Claudio Gallicchio	First deliverable version for internal revision and quality assurance review.
1.1	15/10/2012	Davide Bacciu, Claudio Gallicchio	Revised version after quality assurance review. Changed status to “Released”.

Disclaimer

Executive Summary

This report describes the release 1.0 of the "Core Learning Service API" software, presented as deliverable D2.2. We focus here on a description of the design and current implementation status of this software, and we outline the future work to be performed as part of tasks 2.4 - 2.5, leading up to the second release in deliverable D2.3 in month 30.

This report includes

- External and internal overviews of the learning layer architecture, with a particular focus on progress since D2.1
- 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, to wit
    - The Core learning service API v1.0
    - The Learning Network (LN)
    - The Learning Network Manager
    - The Training Manager
- A description of the testing carried out and the successful results of all the tests
- 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

Abbreviations

API	Application Programming Interface
CLS	Core Learning Service
ESN	Echo State Network
GUI	Graphical User Interface
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

Tables

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.

1.2 External view of 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 and described in D1.3.1. A Synaptic Connection mechanism is built on the top of the Synaptic Channel API (Sect. 2.2.1, D.3.1.1) 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. For this initial release of the Learning Layer API, we focus on the use of the Synaptic Communication local to the node. Deliverable D2.3 will provide the software implementing the full distributed neural computation functionality. The Learning Layer further uses the "Connectionless Message Passing" mechanism of the Communication Layer (Sect. 2.2.3, D.3.1.1) 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: see Figure 7 in Sect. 5 of D2.1. Such preliminary architecture has been refined and partially modified as a result of the implementation activities (from Task 2.2 and Task 2.3) and of the refinement phase anticipated in the second part of Task 2.1.

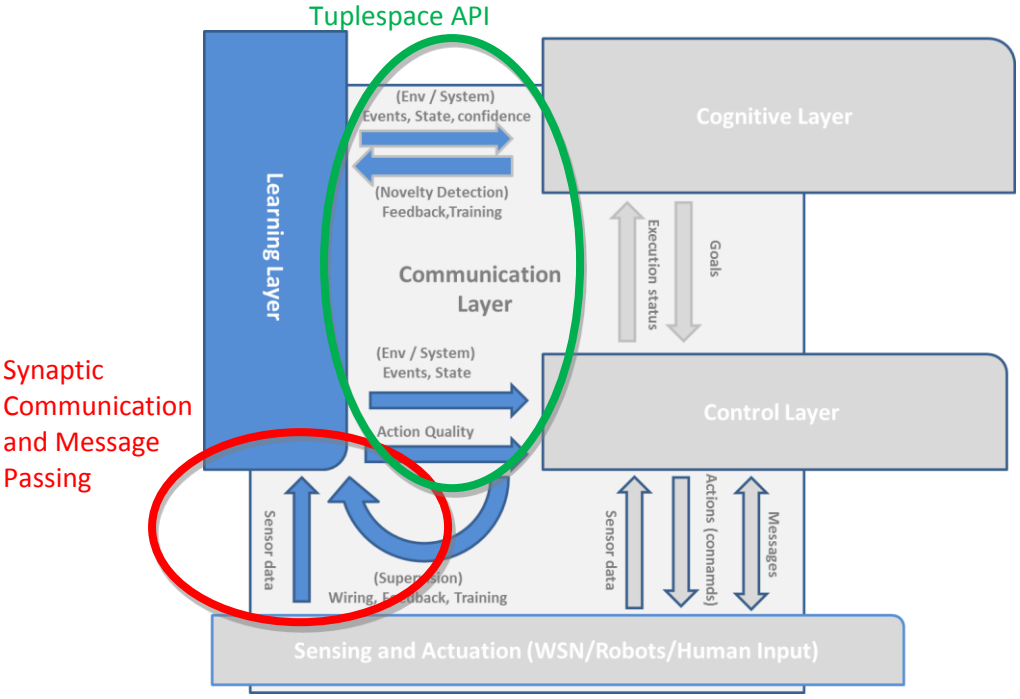

Figure *1 External view of the Learning Layer with an highlighting of the APIs interfacing with other RUBICON layers*.

The up-to-date architecture of the Learning Layer software is depicted in Figure 2. 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 powerful gateways.

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. Currently, these are available as NesC software for TinyOS 2.1 enabled devices.

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.

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. In the current software version, the Repository is implemented also by a Java component, that is likely to be replaced by a database system in future releases. All the TM components are deployed on a PC, as shown in Figure 5.

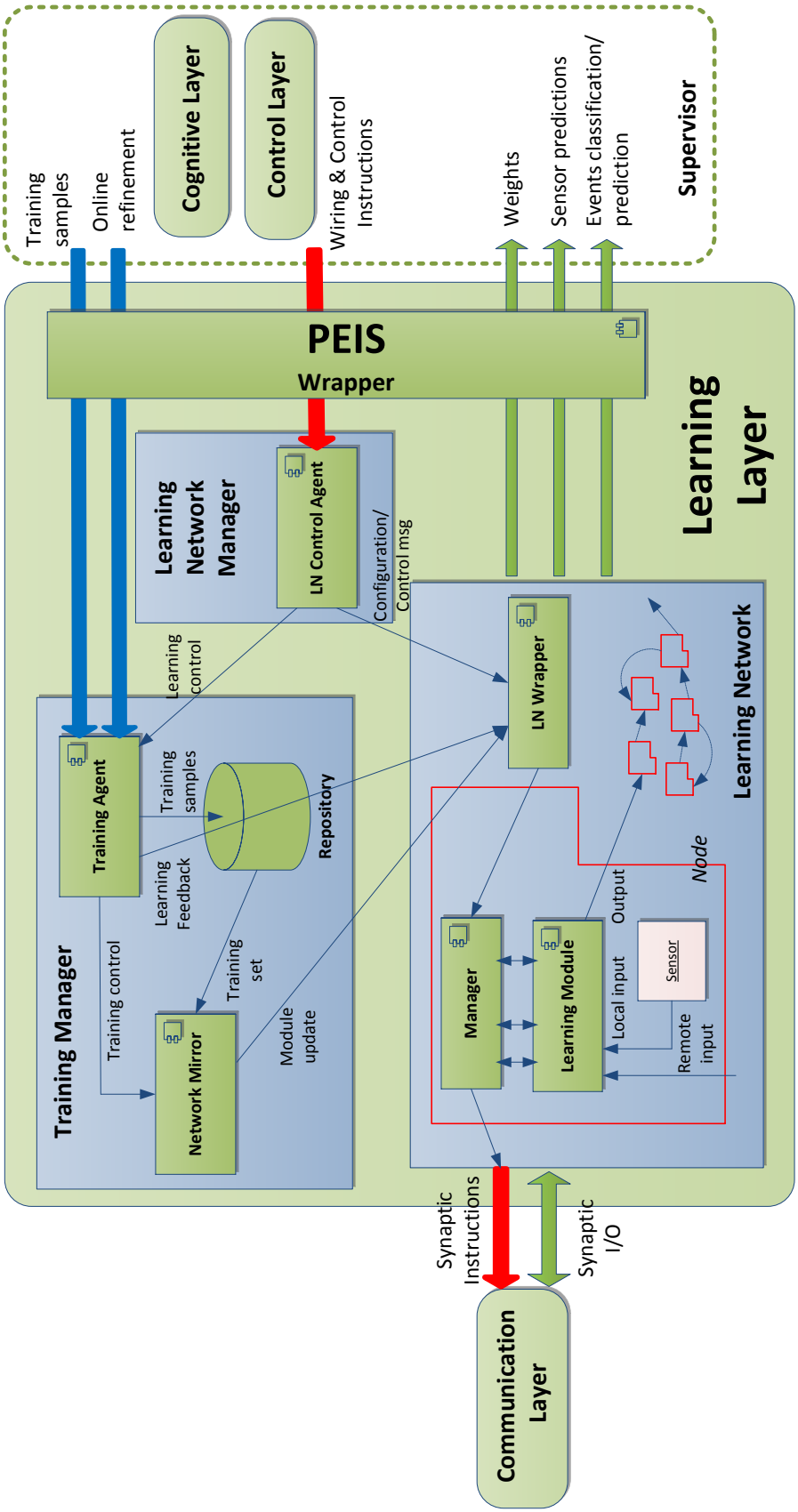

Figure 2 Software architecture of the Learning Layer: derives from the preliminary sketch in D2.1, Section 5, Figure 7. Logical subsystems are represented as simple rectangular boxes, software components are small rectangles with the UML component icon on the top-right corner, while interlayer and intralayer interfaces are denoted as thick and thin arrows.

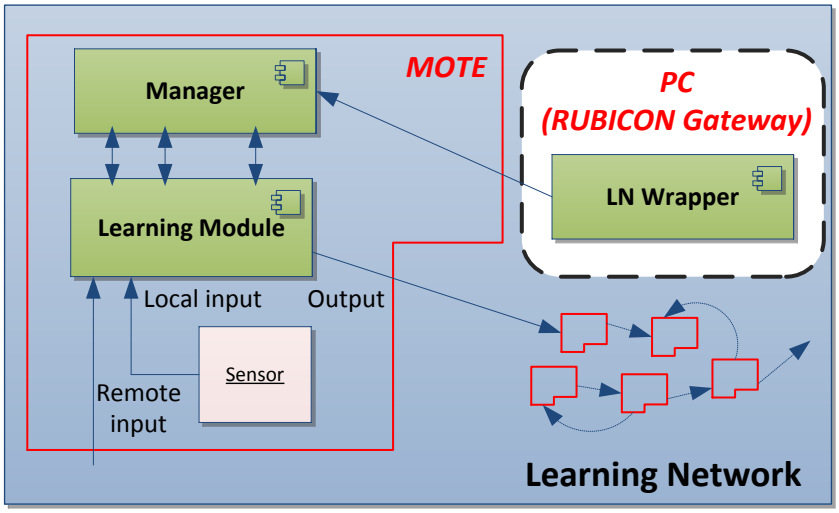

Figure 3 Architectural detail of the LN subsystem: the Manager and Learning Module components run on ecology devices (represented as L-shaped boxes) distributed in the environment (currently only mote devices are supported); the LN Wrapper component is deployed on a PC (currently the RUBICON Gateway), represented as a dashed rectangle.

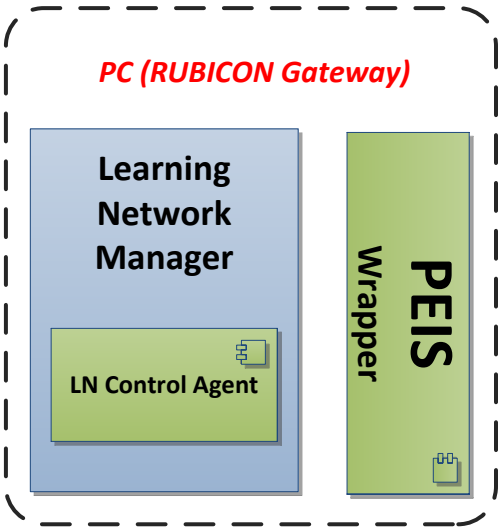

Figure 4 Architectural detail of the LN Manager subsystem: the LN Control Agent component is deployed on a PC (currently the RUBICON Gateway). The LN Manager subsystem interacts with the PEIS Wrapper component, running on the same PC, that interfaces the Learning Layer with other PEIS-enabled components.

1.3.1 Differences from the preliminary architecture in D2.1.

A detailed specification of the role and operations of each subsystem and software component can be found in Section 5 of D2.1, which can be used as a reference for the current deliverable. Here we summarize the major differences and the new components that have been introduced, with respect to the preliminary specification in D2.1, as a result of the implementation and requirement refinement phase.

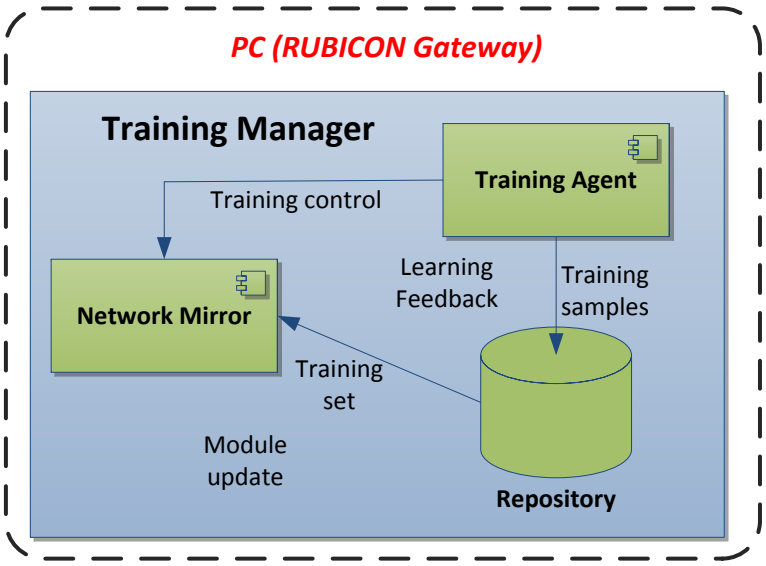

Figure 5 Architectural detail of the Training Manager subsystem: the Repository, the Training Agent and Network Mirror components all run on the same PC (currently the RUBICON Gateway), represented as a dashed rectangle.

Two major software components have been added to the architectural sketch in Figure 2:

1. The **PEIS Wrapper** is a component that explicitly models the fact that the interface between the Learning Layer and the Control and Cognitive Layer (Supervisors) is 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 notifies the reception of messages to the appropriate Java components of the Learning Layer.

2. The **LN Wrapper** is a component that abstracts the distributed nature of the Learning Network. 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.

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 second 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 WSN nodes (also called *motes*) where each mote is within direct communication range of each other mote.

- Zero or more motes in each such island. Each motes must deploy the TinyOS based Communication Layer API (see D1.3.1) and the Learning Layer API.

- One sink-node WSN mote per island deploying the TinyOS based Communication Layer API (see D1.3.1) and the Learning Layer API.

- The sink-mote must be connected through a USB serial to a PC running
    - The RUBICON gateway software (D1.3.1);
    - The gateway wrapper software enclosed within the Learning Layer API;
    - A PEIS-init component (D1.3.1);
    - A working peisjava distribution;
    - The Training Manager and LN Manager subsystems of the Learning Layer API, together with the LN Wrapper component.

1.4.2.1 Platform dependencies, PC side

The primary target for the PC side of the Learning Layer is an Ubuntu 11.10 based systems running on Intel x86 compatible hardware in 32/64-bit modes and with the Oracle Java implementation. The software has been tested and guaranteed to work on these systems, but have also to a lesser extent been verified to work with a range of other Posix conformant operating systems such as other linux based systems as well as Macintosh based systems.

1.4.2.2 Platform dependencies, WSN side

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

2. Learning Layer Software System

2.1 Core Learning Service (CLS) API V1.0

The Core Learning Service (CLS) API V1.0 provides a **preliminary implementation of the RUBICON Learning Layer**, whose specification is discussed in detail in D2.1. The current release provides the management and control functionalities specified in D2.1, Sections 5.2-5.4; the training functionalities are currently limited to the sole Local Learning mechanism (Task 2.3.). As per the RUBICON DoW, the distributed, reinforced and refinement learning mechanisms, as well as feature selection, will be delivered in D2.3.

The CLS API is articulated into two software libraries:

1. A Java API comprising the PC-side software that runs on the RUBICON gateway (see Sect. 1.4.2) 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, targeted to TinyOS devices, that provides the implementation of the distributed LN subsystem, including the Synaptic Connection mechanism.

In the remainder of this section, we discuss the main functionalities implemented as part of the CLS API. Rather than discussing the two 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 Java and NesC libraries are discussed briefly in the following two subsections, whereas a more technical reference manual of the implemented functionalities is available in Appendix A of this document.

2.1.1 The CLS Java API

The CLS Java API 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 CLS Java API requires the Java TinyOS and peisjava libraries to be installed into the system where it is deployed. In its current version, the CLS Java API needs to be deployed on a PC running the RUBICON gateway, which, in turn, should be attached to the sink mote controlling the island.

The CLS Java API comprises 6 packages

- `LearningLayerApi.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.
- `LearningLayerApi.learningnetwork` – Includes the Java wrapper for accessing the distributed LN and an interface to the RUBICON gateway that currently supports communication with TinyOS devices.

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

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

In addition to that, the software includes a Java package LearningLayer.gui implementing a Graphical User Interface (GUI) for the configuration and control of the Learning Layer. The GUI is NOT officially a part of D2.2, as per DoW description, and it is still in a preliminary beta version that has not yet undergone a complete debugging process; however it is a useful ad-hoc development and validation tool.

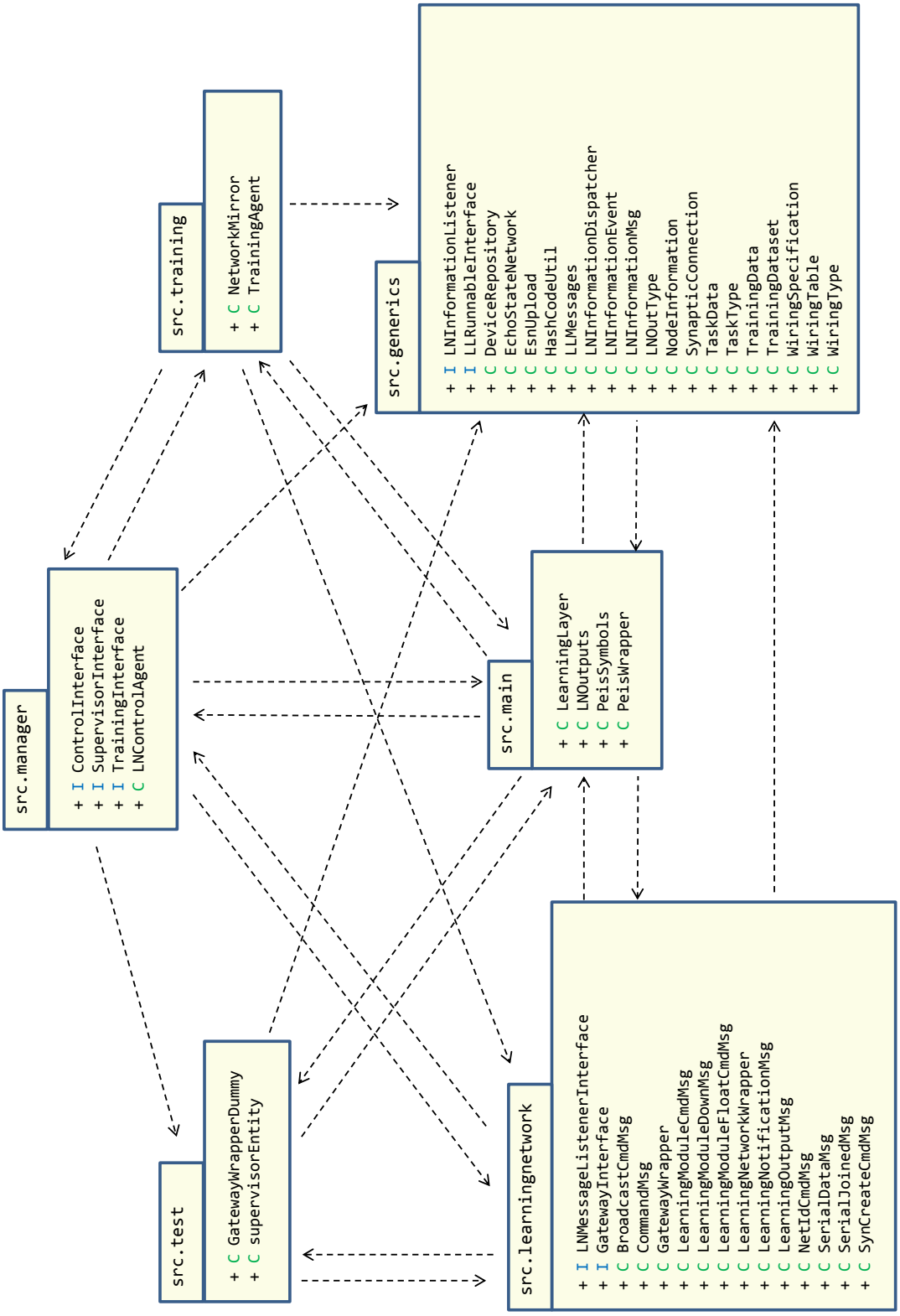

Figure 6 The package diagram of the Learning Layer Java API.

2.1.2 The CLS NesC API

The CLS NesC API provides the **mote-side** functionalities of the Learning Layer. These are implemented as ESN learning networks for TinyOS devices, including all the data structures and functions used for the feed-forward computation, implementation of the synaptic connections, upload/download and activation/stop of the ESN modules. The CLS NesC 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 CLS NesC API requires TinyOS 2.1.1 to be installed into the devices where it is deployed, supporting both mobile and sink devices.

The CLS NesC 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 CLS NesC API, showing the wiring among the different NesC components involved.

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, 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 CLS Java API and partly by the CLS NesC API. The `LearningLayerApi.learningnetwork` package of the CLS Java API 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

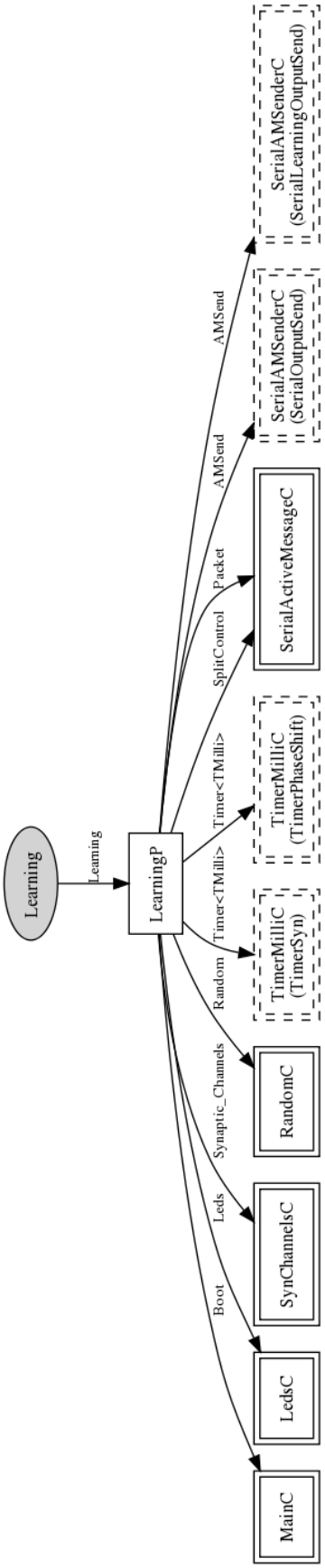

Figure 7 Schematic illustration of the wiring among software components involved in the CLS NesC API. Learning Network (LN)

well as a stand-alone component implementing the interface (see `learningnetwork.GatewayWrapper`).

The CLS NesC API implements the learning modules embedded in the TinyOS ecology devices as well as the mechanisms supporting the Synaptic Connection abstraction. In particular, in the NesC file `LearningP.nc`, a complete implementation of the Leaky-Integrator ESN model is provided, including functions for initialization, forward computation, upload/download and activation/stop of the learning network. In the same NesC file, the mechanisms for the realization of Synaptic Channels are implemented.

The CLS Java API implements the learning modules in the Network Mirror component. Such implementation corresponds to the embedded implementation in the NesC API, but can also be used to train off-line the parameters of the ESNs.

2.2.2 Learning Modules

The implementation of the ESN module in the CLS NesC API is based on several variables and structures that are used to represent the main 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 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.

The implementation of the ESN module in the CLS Java API is based on the class `EchoStateNetwork`. 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` represent the input, the state and the output of the neural network. The class also includes the ID of the computational task associated. The most relevant members of the class `EchoStateNetwork` are illustrated in Table I.

2.2.3 Synaptic Connections

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 II.

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 III.

Table I The main members of the class 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;
```

Table II 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 III 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`.

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.

2.2.4 Forward Computation

The forward computation phase on-board the RUBICON nodes is realized in the `LearningP.nc` file within the NesC API. Each time the RUBICON clock fires, the input for the ESN module is read from the input Synaptic Connections using the function `read_syn_connection(inConnection)`, then the routine `do_feedforward_step()` is called to compute the actual output of the ESN, and finally the output of the ESN module is propagated through a write operation `write_syn_connection(outConnection)` on the Output Synaptic connections. These passes are schematically shown in Figure 8.

The `LNOutputs` component, implemented through a java class in the `main` package of the CLS Java API, maintains information concerning the outputs predicted by the LN, including their symbolic names and their current values. The `LNOutputs` component receives the updated LN predictions from the `LearningNetworkWrapper` (see Sect. 2.2.5): 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 PEIS tuples. To receive information concerning the current LN predictions and their symbolic names, a PEIS-enabled participant need only to subscribe to the tuples with key-string “LLoutputIDValue” and “LLoutputID”, respectively. Table IV and Table V 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 IV). Symbolic names are concatenated into a unique string, using the placeholder “#” to separate between names (see Table V). Publishing of the output-related tuples is achieved by the `main.PeisWrapper` component, which takes care of creating the OUTPUT_ID and OUTPUT_ID_VALUE tuples and updates their value at each tick of the RUBICON clock.

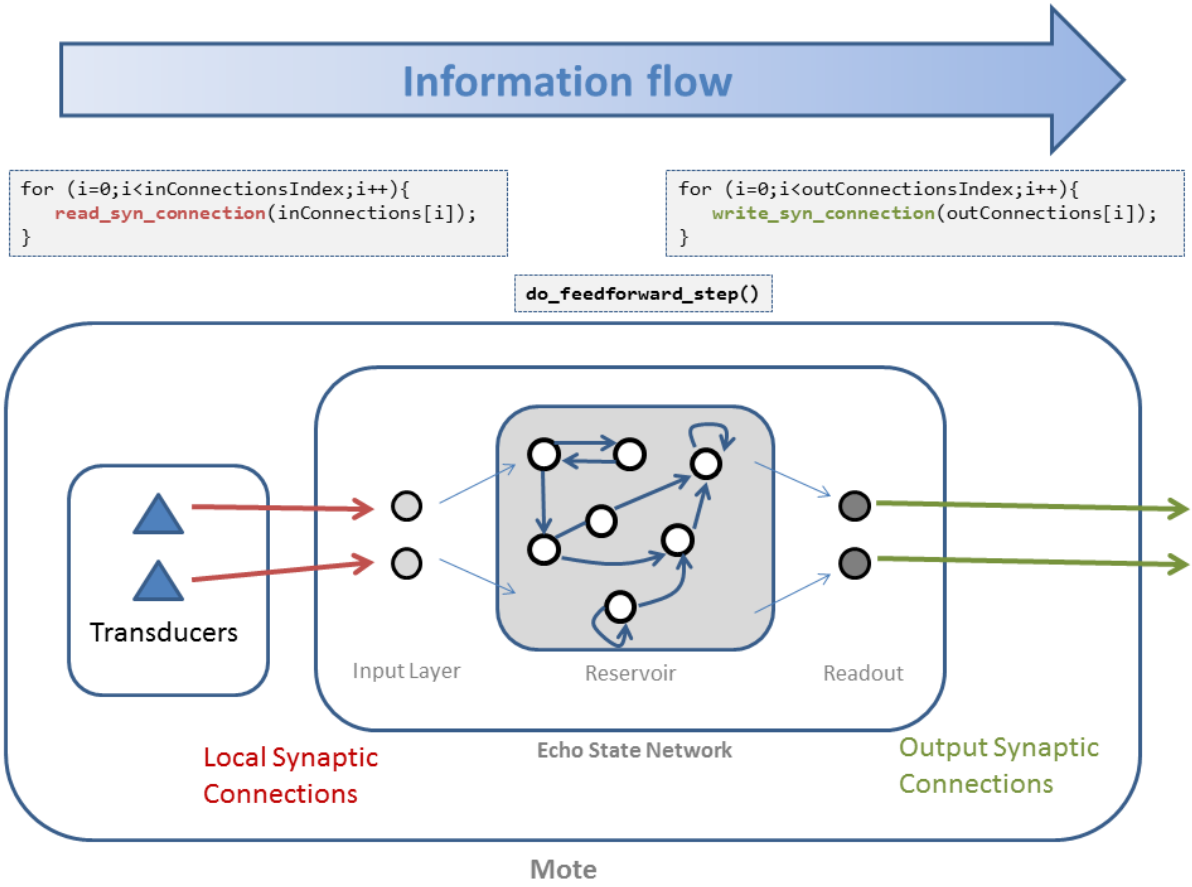

Figure 8 A schematic illustration of the information flow in one step of the feedforward computation on-board a mote.

Table IV 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	OUT_1,OUT_2, …,OUT_K

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

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

2.2.5 Learning Network Wrapper

The `LearningNetworkWrapper` class is defined in the `learningnetwork` package of the CLS Java API: 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 communication primitives defined by the RUBICON Gateway. Further, it 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.

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 learning modules, as well as a list of command types. To receive information from the LN (through the RUBICON gateway), the LearningNetworkWrapper implements the LNMessageListenerInterface method messageReceived().

The LearningNetworkWrapper agent 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.

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 CLS Java API. 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.

The LNControlAgent maintains a repository of deployed synaptic connections (synConList) and a repository of available ecology nodes (deviceRepository) for the purpose of LN management.

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 VI and Table VIII show the expected tuple format as well as the "PEIS Name" macro defined in `main.PeisSymbols.java`.

The `CONTROL_CMD` tuple (Table VI) 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 VII 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 VI 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 VIII) 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 VIII. 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 VII 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

Table VIII 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 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` repository 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 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.

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 sensors, transducers, etc. Such information is communicated to the LNControlAgent through the SupervisorInterface method

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

In the current implementation, such a notification is automatically provided by the device itself when it joins the ecology. A joining message containing the nodeSpec information is forwarded by the RUBICON gateway to the LearningNetworkWrapper which, in turns, activates the connect_node() method above. The deviceRepository also provides a getCompatibleDevice(capability) method, that determines which devices in the repository possess the transducer capabilities passed as argument.

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.

2.4 Training Manager

2.4.1 Overview

The Training Manager comprises the Training Agent and the Network Mirror software components. The Training Manager receives wiring and task information for the creation of new computational tasks. In order to manage the wiring instructions, the Training Manager manages a Wiring Table (see an example in Table IX) 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. The class WiringTable manages an ArrayList of WiringSpecification objects, allowing to add and to remove entries from the table.

Table IX 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)
LIGHT_SENSOR1	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)`.

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 Training of a Learning Module

Training examples are provided to the Training Agent by the new_training_data method. Learning modules can be trained by invoking the method do_training(int taskID) in the TrainingAgent, specifying the ID of the task on which training should be performed. 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.

2.4.3 Deployment of a Learning Module

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. The TrainingAgent then 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). Finally, the TrainingAgent requests the deployment of the Synaptic Connections by a call to the function deploy_synaptic_connections(taskID, sc) in the LNControlAgent object, where sc is a List of SynapticConnection objects (corresponding to the wiring instructions).

2.4.4 Summary of the Steps for the Creation of a New Task

This section summarizes the steps to be used at the Training Agent level for the creation of a new computational task:

1. **Update the wiring table**
   Add an entry in the wiring table for each new entity which is involved in the task, using the TrainingAgent method add_wiring_specification(w), where w is a WiringSpecification object.
2. **Construct the wiring instructions**
   Construct a WiringType object. This can be done by using the constructor WiringType(source, destination, symbolicNames), where source, destination and symbolicNames are ArrayList of String. Another possibility consists in using a static method implemented in the class WiringType, i.e. fromPeisTuple(s), which returns a WiringType object, and where s is a String specifying the wiring instructions according to the format "<source1,destination1>…<sourceN,destinationN>[output1]…[outputM]".
3. **Construct the task information**
   Construct a TaskType object. This can be done by using the constructor TaskType(type, granularity, dataType, outputType), where type, granularity, dataType and outputType are String objects. Another possibility is to use a static method implemented

in the class `TaskType`, i.e. `fromPeisTuple(s)`, which returns a `TaskType` object, and where `s` is a `String` specifying the task information according to the format "{type}{granularity}{data_type}{output_type}"

4. **Allocate the task**
   Allocate the new task by calling the `TrainingAgent` method `allocate_task(wiring, taskInfo, nodeID)`, which returns the ID of the allocated task.
5. **Provide the training data**
   Provide training examples for the computational task by using the `TrainingAgent` method `new_training_data(sample)`, where sample is a `TrainingData` object specifying the input, the desired output and a list of task ID to which the sample should be associated.
6. **Train the Learning Modules**
   Train the ESNs corresponding to the new task, by using the method do_training(taskID), where taskID is the integer ID of the task. When training is completed, the learning modules are automatically deployed (see section 2.4.3), and the Forward Computation is activated (see section 2.2.4).

2.5 Graphical User Interface

A GUI for configuration and control of the Learning Layer is delivered together with the CLS Java API 1.0: the Java package `gui`, available in the WP2 folder of the RUBICON repository, contains all the classes needed to instantiate the GUI. We reiterate that the GUI is not officially part of this deliverable D2.2. and it is provided in a beta version for a preview of its functionalities.

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`. Further, the GUI interfaces with a (optional) sniffer to receive and publish, at runtime, the values circulating in the synaptic connections as well as the current transducer readings. Finally, the GUI implements a JDBC interface (package `gui.database`) that allows connecting to a PostgreSQL database storing information concerning experiments performed with the LN. This information includes the layout of the LN, including the composing devices, the learning modules and the associated Synaptic Connections. By this means, experiments can be retrieved for future execution/analysis.

A screen-shot of the typical appearance of an experiment in the GUI is shown in Figure 9. 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 9, 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.

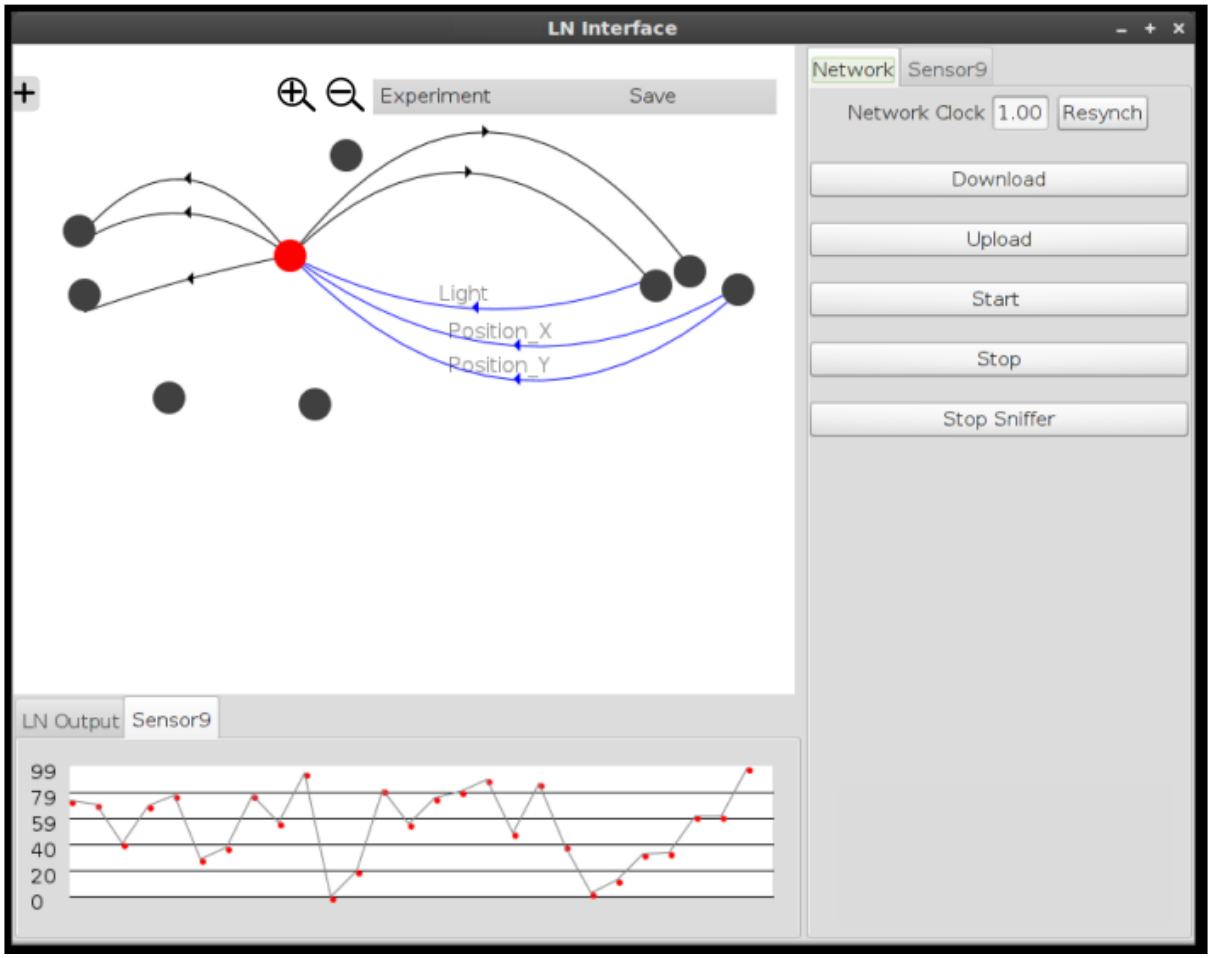

Figure 9 Screenshot of the Learning Layer GUI showing the LN layout.

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 10). 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 10 as well as graphically by drawing a connection in the canvas displaying the LN network, as in Figure 9. In both cases, the user needs to specify the identifiers of the input and output neurons to be connected. A new device can be added to the LN using the plus (+) button on the top-left corner of the GUI (see Figure 10).

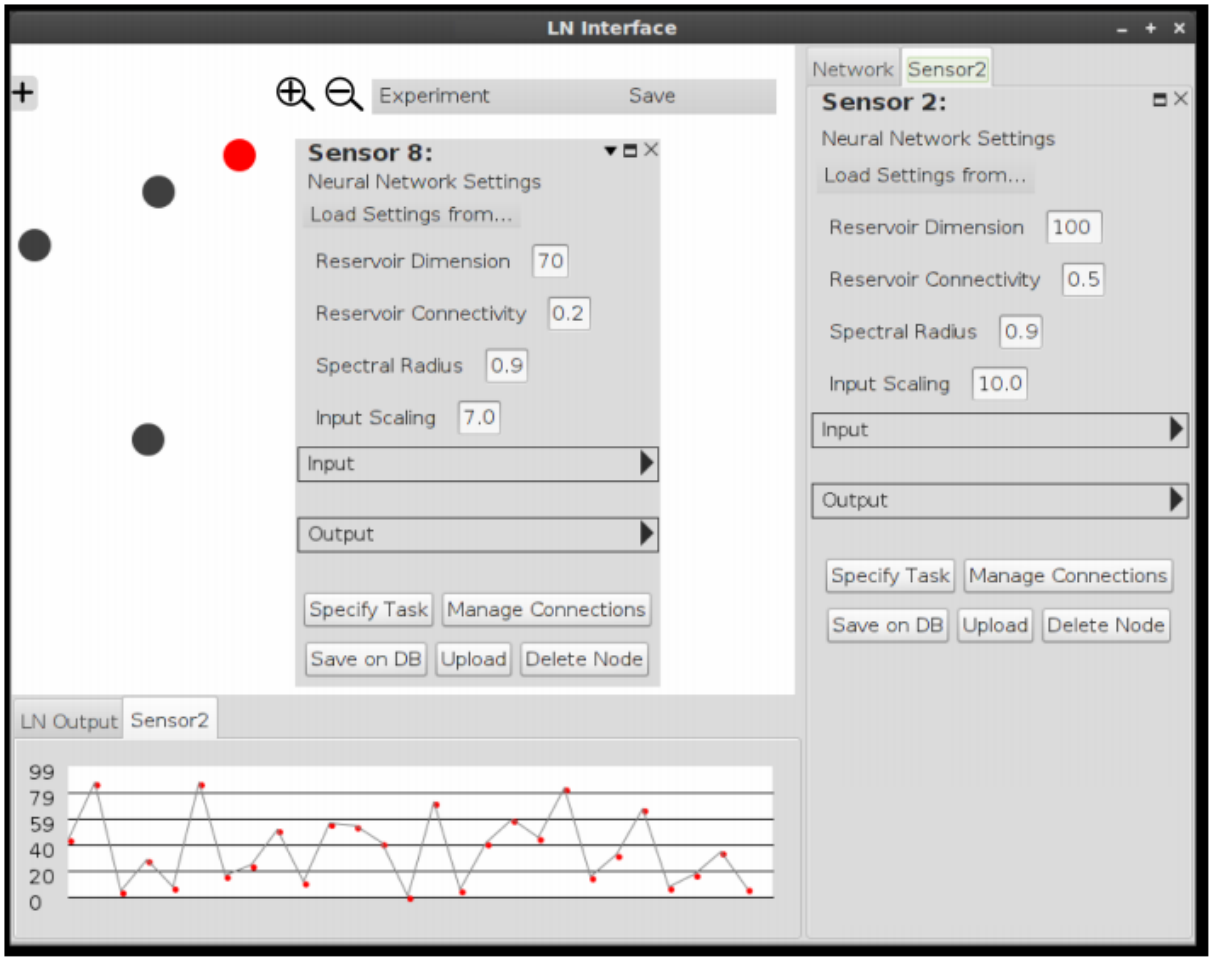

Figure 10 Screenshot of the Learning Layer GUI showing the Device Manager.

Commands affecting the whole network can be issued from the "Network" tab menu on the right side of the GUI (see Figure 9): 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 last button allows to enable or disable the optional sniffer. If the sniffer is active, it is possible to display the learning network outputs in the lower part of the screen. The LN output menu allows to select which neuron output is shown at a given time.

3. Testing

3.1 Outline

Testing of a distributed learning system, such as the Learning Layer, entails assessing several mechanisms that are pipelined to realize a distributed neural computation. The first step of the chain requires training a set of learning modules on some computational learning tasks (i.e. in the TM subsystem). These are then deployed to the devices constituting the LN through appropriate communication mechanisms. After that, the pipeline requires the creation of the synaptic connections linking the single learning modules into a distributed system (the LN). Only then, we are allowed to start the distributed computation, which we refer to as the *feedforward computation*, in the following. The result of such feedforward step are the LN predictions (or outputs), that are published to the higher level components of the RUBICON stack, thus concluding the deployment pipeline.

The debugging and assessment process described in this section focuses on testing the CLS API V1.0 functionalities with respect to the 3 characterizing aspects of D2.2, as per RUBICON DoW description, that are

1. Core Learning Services: this pertains testing the instantiation of the Learning Layer infrastructure, including the successful activation of all the software components and their inter-thread communication mechanisms, as well as testing the deployment of local and remote synaptic connections.
2. Local Learning: this includes testing forward computation, with only local input sources, and the mechanisms to upload/download learning modules to/from the LN.
3. Integration with Control Layer: this involves the integration with the PEIS tuplespace to receive control and configuration commands from the Control Layer (or any Layer that is integrated in PEIS, e.g. the Cognitive Layer) as well as to publish the LN predictions.

These 3 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.

3.2 Testing Core Learning Services

3.2.1 Test configuration

The assessment of the Core Learning Services in the CLS API V1.0. entails testing

1. Creation, runtime execution and destruction of the software components composing the Learning Layer (see Figure 2).
2. Inter-thread communication mechanisms, as well as handling of control and status messages.

3. Deployment and management of the synaptic connections, including activation and halting of synaptic streaming.
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 CLS Java 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 TinyOS CLS API.
- 2 motes (coherent with the platform requirements in 1.4.2.2) equipped with light and temperature transducers and running the TinyOS CLS API.

3.2.2 Offline Test

The first round of tests is intended to assess the Java-based components offline with respect to the LN. To this end, we have employed the stub `GatewayWrapperDummy` component, that implements the same interface of the actual `GatewayWrapper` but does not interact with an actual LN. Instead, it receives all control and configuration commands from the Learning Layer, queues them and responds with appropriate acknowledgement messages after some delay, simulating interaction with the network.

Test 1 Description

The first test script, implemented by the `supervisorEntity.set_configuration_1()` method, is organized as follows

1. Creates the main `LearningLayer` object and the associated components.
2. Allocates 2 local synaptic connections per mote, whose sources are the light and temperature transducers, by calling `create_synaptic_connections()`.
3. Implements a stub forward computation that copies the value red by the local synaptic connection into 2 output neurons.
4. Allocates 2 remote synaptic connections linking the output neurons on each mote to the sink, to simulate the LN predictions.
5. Set the RUBICON clock to 500msec by `set_rubicon_clock()`.
6. Starts the synaptic streaming (and stub forward computation) using both the single broadcast command `activate_forward_computation()` and multiple point-to-point commands `activate_learn_module()`.
7. Collects and publishes LN predictions for 30seconds.
8. Stops the synaptic streaming (`stop_forward_computation()` or `stop_learn_module()`) and sends termination signals to the Learning Layer components.

Test 1 Outcome

The offline execution of this script produces the correct activation of all the Java objects and threads composing the Learning Layer. The invocation of the synaptic connection create command, correctly allocates new synapses in the `LNControlAgent.synConList` repository: the state of the synapses in the repository is maintained coherent with the deployment status advertised for the connections (e.g. the remote connections' state is set to `READY` only when the `LNControlAgent` receives acknowledgement for both connection' ends).

The request queue `LearningNetworkWrapper.requestState` correctly handles the pending messages and the associated acknowledgments. The LN status messages are correctly delivered by the `LearningNetworkWrapper` to the other components. The stub forward computation progresses

without problems for 30seconds and a default prediction value is published by the `LNOutputs` component. Termination of the synaptic streaming is correctly propagated to the stub gateway and all the Learning Layer thread terminate gracefully without errors.

Test 2 Description

The second test script, implemented by the `supervisorEntity.set_configuration_2()` method, aims at assessing the mechanisms for connecting and disconnecting devices from the LN. The script:

1. Creates a capability specification object `nodeInfo` for each mote ;
2. Instructs the Learning Layer to connect the associated devices ;
3. After a delay of 5 seconds, requests the disconnection of the devices and terminates.

Test 2 Outcome

Execution of the script correctly places the new devices in the `LNControlAgent.deviceRepository` with the appropriate `INITING` status, that is changed to `CONNECTED` upon reception of the acknowledgment message from the gateway. When the disconnection message is received, the device state is changed to `DISCONNECTED` and the Learning Layer terminates without errors.

Test 3 Description

The third test script, implemented by the `supervisorEntity.set_configuration_3()` method, aims at assessing the `LNControlAgent.deploy_synaptic_connections()` mechanism for the deployment of a bundle of synaptic connections associated to a computational learning task.

The test configuration comprises the deployment of the same local and remote connections in `supervisorEntity.set_configuration_1()`.

Test 3 Outcome

Execution of the script produces the correct instantiation of the synaptic connections, as for the first test.

3.2.3 Online Testing

The second round of tests assesses the full CLS API, including both the Java components on the gateway and the TinyOS components onboard the motes. The stub `GatewayWrapperDummy` is replaced with the working `GatewayWrapper` object that interacts with the LN through the sink mote connected on the PC USB port.

Test 4 Description

The first online test uses the `supervisorEntity.set_configuration_1()` script with actual motes. The execution of the script replicates the steps of the successful offline execution.

Test 4 Outcome

All the synaptic connections are correctly deployed and acknowledged. The activation of synaptic streaming delivers the transducer readings (light and temperature) from the motes to the LNOutput interface, that publishes them to the standard out. Streaming of the transducer readings is maintained for 30 seconds, after which it is terminated by the supervisorEntity with all threads exiting without errors and the motes acknowledging the stopping of the transmission.

Test 5 Description

The second online test assesses the mechanisms for connecting and disconnecting devices from the LN using self-joining messages sent by the motes. Differently from the script in `supervisorEntity.set_configuration_2()` the availability of a device is not signalled by the supervisor. Rather, the device itself advertises its availability through a `SerialJoinedMsg` that is notified to the `LearningNetworkWrapper`.

Test 5 Outcome

The test script `supervisorEntity.set_configuration_2online()` starts the Learning Layer and waits for joining messages. Execution of the script shows that the two motes send their joining messages, that are received by the `LearningNetworkWrapper` which notifies the `LNControlAgent` through the `connect_node()` method. The device repository `LNControlAgent.deviceRepository` is correctly updated to reflect the availability of the motes, whose state is assigned to `CONNECTED` as soon as the acknowledgment message is received.

Testing of the online deployment of a bundle of synaptic connections (e.g. as in the `supervisorEntity.set_configuration_3()` script) is shown in the next section together with the deployment of a local learning task.

3.3 Testing Local Learning

These rounds of tests focus on the deployment of learning modules, the creation of a new computational task and training of the learning modules. Moreover, in this section, we show in Figure 11 an example of the standard out produced by the Learning Layer Java API during a typical script execution.

Test 6 Description

The first test uses the script `supervisorEntity.set_configuration_6()` script, using one mote (whose information are added to the device repository by using the same protocol described in the previous section 3.2.2). In this script:

1. An `EchoStateNetwork` object is created adopting default values for the reservoir parameters and explicitly specifying only the network ID and the input and output dimensions.
2. The upload of the ESN is requested by invoking the method `upload_module` in the `LNControlAgent` object.

Test 6 Outcome

The ESN relevant parameters, i.e. the weight matrices for the input-to-reservoir, recurrent reservoir and reservoir-to-readout connections, are correctly uploaded to the mote.

Test 7 Description

A second test concerns all the steps which are necessary for the creation of a new learning task, training of the corresponding learning network and deployment of the trained network onboard the mote. The script used for this test is `supervisorEntity.set_configuration_7()` which in turn calls the script `new_task_example()` implemented in the `TrainingAgent` class. This script contains an example of the necessary instructions for the allocation and training of a new task (see

section 2.4.4), i.e. construction of the wiring table, construction of the task information, construction of the wiring instructions, allocation of the new task and training of the associated learning network. More in detail:

1. A wiring table with six entries is created, containing the wiring information related to light and temperature sensors, two input neurons and two output neurons where the prediction output will be produced.
2. A `TaskType` object is created, specifying a sequence-to-sequence regression task, which deals with sensory information and produces a sensor refined prediction.
3. A `WiringType` object is created, to associate the input sensory readings to the input neurons
4. The new task is allocated; this step also instantiate and initializes a new `EchoStateNetwork` object in the Network Mirror, based on the information provided on the new task.
5. A dummy training set containing 100 training samples is created and its elements are passed to the `TrainingAgent` one by one.
6. The ESN module is trained.
7. The ESN module is deployed on the mote. This last step included the deployment of the necessary synaptic connections, which are obtained by the Training Agent based on the wiring instructions and the wiring table, and passed to the `LNControlAgent` object (through a call to the `deploy_synaptic_connections` method).

Test 7 Outcome

The wiring table, the task information and the wiring instructions are correctly created. The new task is successfully instantiated, together with the `EchoStateNetwork` object in the `NetworkMirror`. The training samples in the dummy training set are correctly generated and inserted in the `TaskData` object associated to the newly created task. The ESN module associated to the task is successfully trained and deployed on the mote.

Figure 11 shows the standard out produced by the Java API for this script, grouping all the output lines based on the main steps in the script. In particular, the output concerning the local learning functionalities (i.e. the mean absolute error on the training set before and after learning) is highlighted in Figure 11.

3.4 Integration with Control Layer

The last round of tests focuses on assessing the mechanisms needed by the Control Layer (or any other PEIS-enabled component, e.g. the Cognitive Layer) to act as a Supervisor of the Learning Layer. This involves testing the invocation of control and configuration commands from 3 PEIS tuples

- `PeisSymbols.CONTROL_CMD` - Publishes commands from `manager.SupervisorInterface` that serve to control the activation/stop of forward computation, set the clock, etc.
- `PeisSymbols.WIRING_CMD` – Publishes a single command requesting the creation of a new computational learning task.
- `PeisSymbols.TRAINING_SAMPLE` – Serves to provide training data to the TM subsystem.

In addition to the 3 tuples above, we test the publication of the LN predictions in PEIS through the `PeisSymbols.OUTPUT_ID_VALUE` tuple.

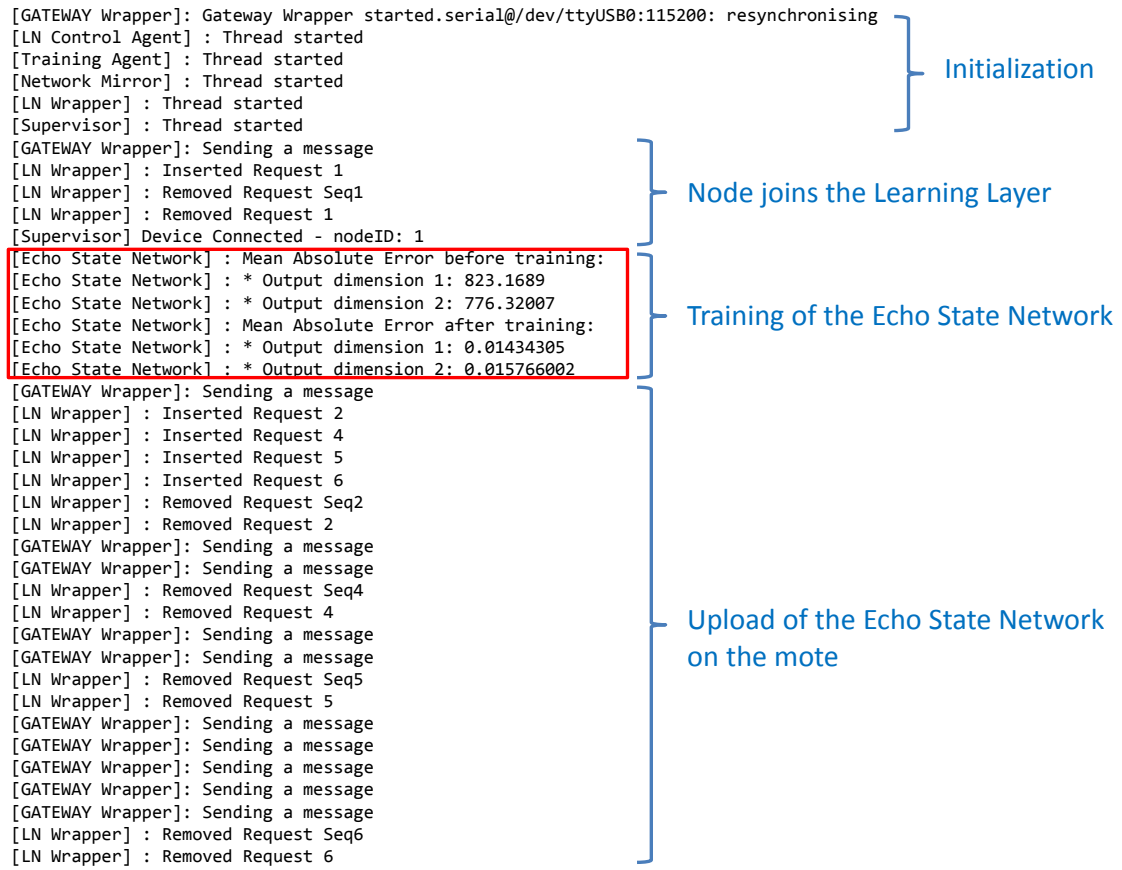

Figure 11 The standard out produced by the script supervisorEntity.set_configuration_7() used in Test 7. The output lines are grouped according to the corresponding steps in the script. The output concerning the training functionalities is highlighted.

Test 8 Description

The `supervisorEntity.set_configuration_peis1()` script tests the Control Layer interface through the `PeisSymbols.CONTROL_CMD` and the `PeisSymbols.OUTPUT_ID_VALUE` tuples. It replicates the same LN configuration described in `supervisorEntity.set_configuration_peis1()`, however, control and configuration commands are sent through PEIS tuples instead of using local method calls. This script allows to test the full feedforward chain that leads to the computation, propagation and publishing of the LN outputs as depicted in Figure 12:

1. The LN predictions are computed by the output neurons of learning modules residing on the single motes.
2. Such predictions are transferred to the island sink by means of remote synapses.
3. The sink assembles the predictions generated by the motes and sends the resulting output vector to the `LNWrapper` component on the attached RUBICON gateway (i.e. through its serial interface).
4. The `LNWrapper` propagates the predicted values to the `PEISWrapper` component, which publishes them in PEIS, in the `PeisSymbols.OUTPUT_ID_VALUE` tuple.
5. The Control Layer (or any other PEIS-enabled component, e.g. the Cognitive Layer) reads the LN prediction from the tuple.

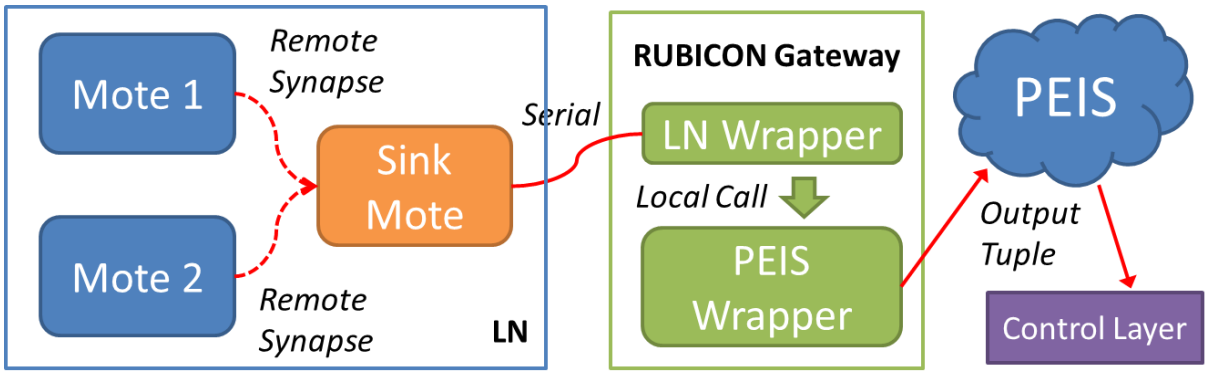

Figure 12 Information flow in the feedforward chain leading to the delivery of the LN predictions to the Control (or Cognitive) Layer.

Test 8 Outcome

Execution of the `supervisorEntity.set_configuration_peis1()` script replicates the results of the `supervisorEntity.set_configuration_peis1()` procedure, while using the `PeisSymbols.CONTROL_CMD` tuple to send control commands to:

1. set the RUBICON clock ;
2. activate forward computation , and
3. stop the single learning modules.

The first command requires a single argument (i.e. the clock value); the second command has no arguments, while the third command type has two arguments. By these means we have tested all possible command configurations of the `PeisSymbols.CONTROL_CMD` tuple. Figure 13 shows the content of Tuplespace during the execution of the script using the Tupleview utility: on the left, we see an active instance of the `PeisWrapper` component that has created 5 interface tuples. The current value of the `PeisSymbols.CONTROL_CMD` tuple is shown on the right, and encodes the stop learning module command directed to the learning module with `netd=3` on-board the device with `nodeID=2`.

Similarly, can inspect the content of the output interface towards the Control Layer, realized through the `PeisSymbols.OUTPUT_ID` and `PeisSymbols.OUTPUT_ID_VALUE` tuples, that provide the current symbolic name and value of the LN predictions, respectively. Figure 14 and Figure 15 show a snapshot of the values assigned to the tuples during the execution of the script. These values represent the temperature and light readings of the two motes involved in the test, and are fully coherent with those captured by the `supervisorEntity.set_configuration_peis1()` script.

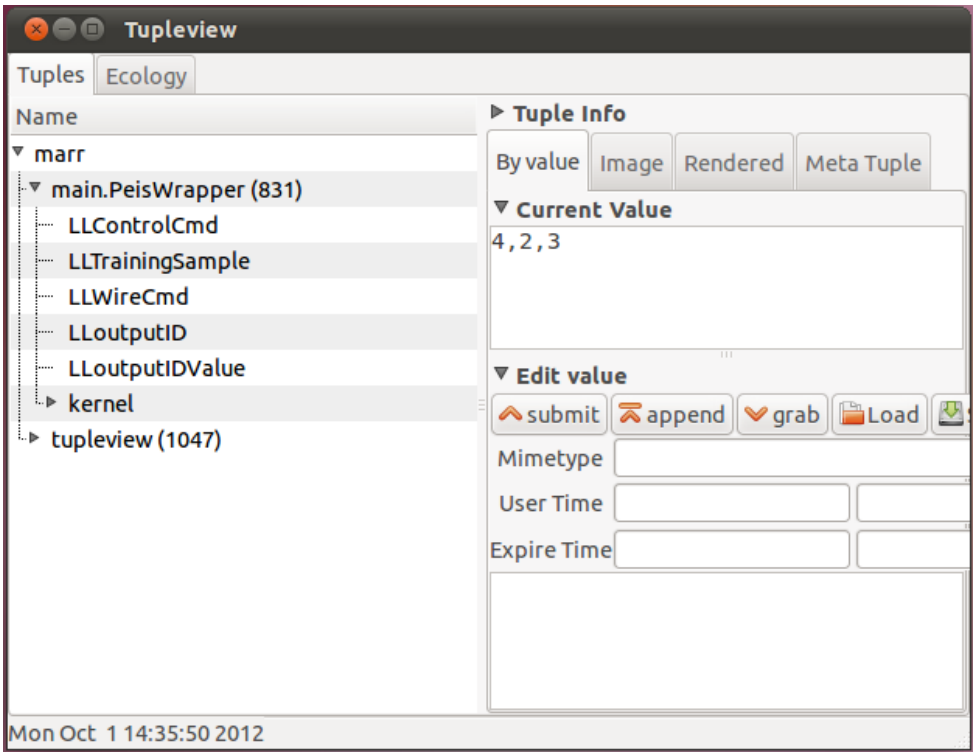

Figure 13 Screenshot of the content of the `PeisSymbols.CONTROL_CMD` *tuple during the execution of the* `supervisorEntity.set_configuration_peis1()` *script.*

Test 9 Description

The `supervisorEntity.set_configuration_peis2()` script completes the test of the Control Layer integration by exercising the training interface provided by the `PeisSymbols.WIRING_CMD` and the `PeisSymbols.TRAINING_SAMPLE` tuples.

Test 9 Outcome

Execution of the script produces the instantiation of a new computational learning task and the setup of a bundle of synaptic connections in response to posting wiring and task information in the `PeisSymbols.WIRING_CMD` tuple. After that, the test script starts posting training data through the `PeisSymbols.TRAINING_SAMPLE` tuple. The training data is received by the `PeisWrapper` component and forwarded to the `TrainingAgent` for storage into the training repository. The script can be complemented with the instructions for the deployment of the synaptic connection bundle and of the trained learning modules to reproduce the results of the tests in Section 3.3.

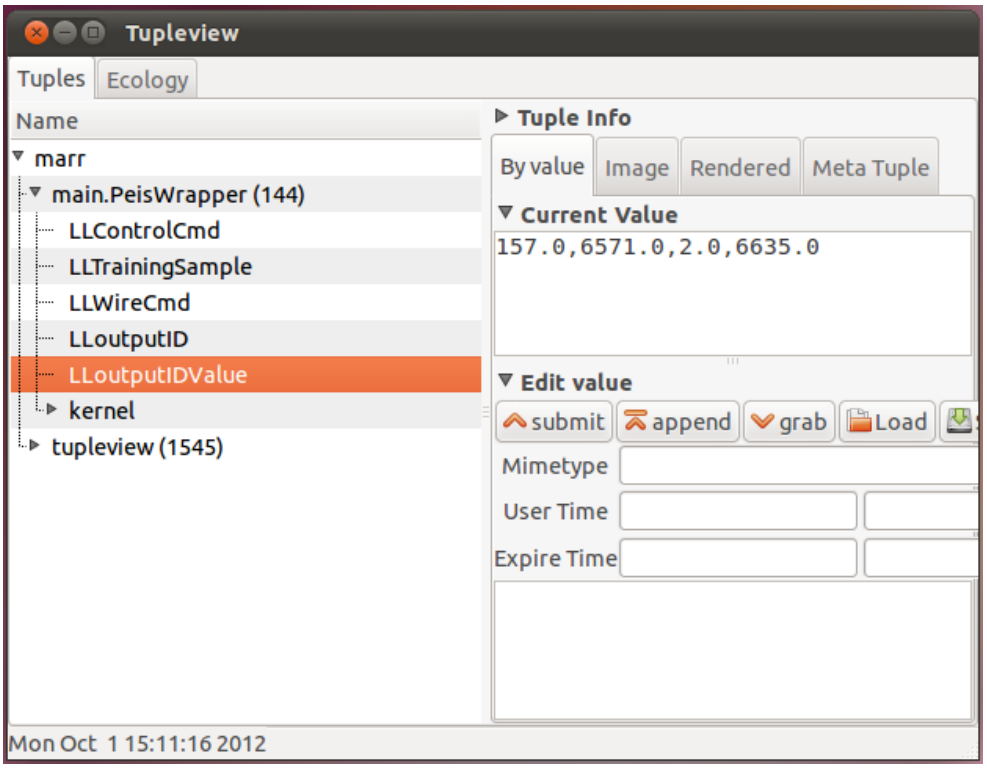

Figure 14 Screenshot of the content of the `PeisSymbols.OUTPUT_ID_VALUE` *tuple during the execution of the* `supervisorEntity.set_configuration_peis1()` *script. The current values represent the light and temperature readings of the transducers on-board the 2 motes. The sharp difference in the readings of the 2 light sensors (i.e. the first and third values) are due to a malfunctioning transducer on the second mote.*

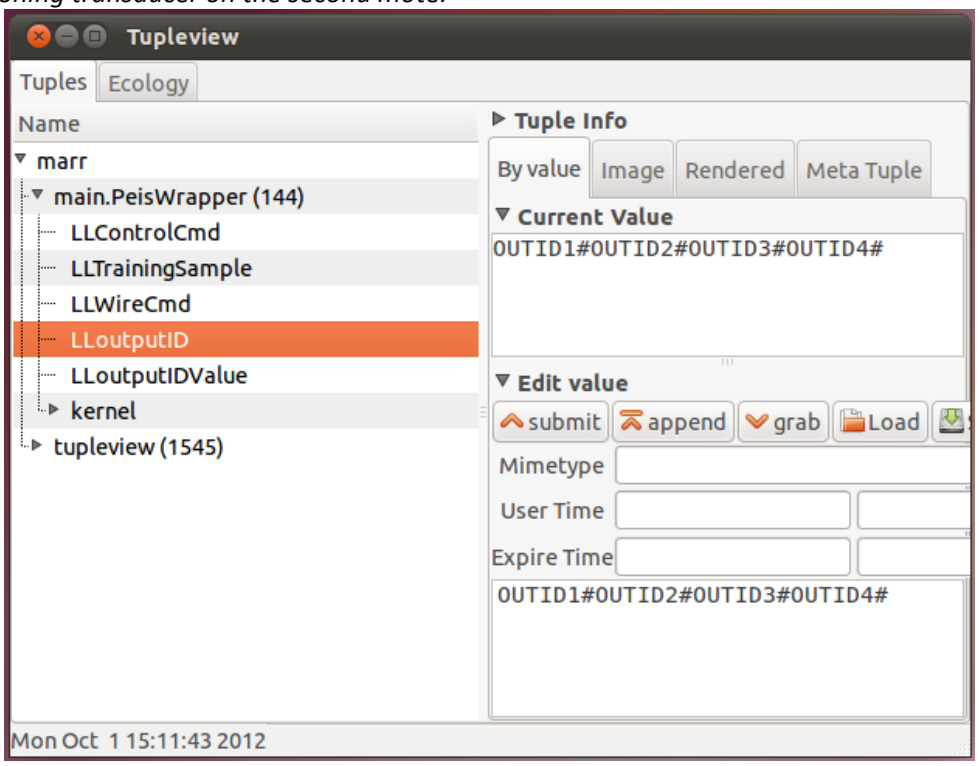

Figure 15 Screenshot of the content of the `PeisSymbols.OUTPUT_ID` *tuple during the execution of the* `supervisorEntity.set_configuration_peis1()` *script.*

4. Conclusions

4.1 Compliance to Workplan

We summarize this report with an overview of the status of the WP2 tasks and Deliverables at Month 18, 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 now 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 included in this deliverable D2.2: in particular, Section 1.3.1 reports the architectural changes with respect to the preliminary specification in D2.1.

Task 2.2 - Core Learning Services (M6-M26): the activities are progressing as planned and have contributed to the implementation of the Learning Layer infrastructure in the CLS API V1.0. These contributions are 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 to the Training Manager subsystem.

The T2.2 activities will continue until M26 progressing the development of the CLS API, with the incremental addition of new functionalities and a continuous activity of code refinement.

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 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 local forward computation producing the local LN predictions ;
- Sections 2.4.2, 2.4.3 and 2.4.4 describe the training of a local learning module , its deployment to a LN device and a summary of the required instructions for the allocation and training of a new task, respectively.

Task 2.4 - Distributed Adaptive Memory (M16-24): the task is currently on-going as planned and will implement a distributed neural computation on the top of the services released in the CLS API V1.0. The results of this task will be reported in D2.3 at M30, with the release 2.0 of the CLS API.

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): delivered in the form of the CLS Java and NesC API V1.0 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 V1.0 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:

> *"The release (code, reference manual/specifications, testing & benchmarks) will include the following features:*
>
> *A1. Implementation of the core learning service platform on top of the communication layer (data-structures, learning layer skeleton, remote synapses)*
> *A2. Implementation of local learning functionalities*
> *A3. Integration with Control Layer"*

These points are addressed in this report as follows:

- The implementation of the core learning service platform (A1) is described in:
  - Section 2.2.3: implementation of local and remote synapses;
  - Sections 2.2, 2.3 and 2.4: implementation of main data-structures and Learning Layer skeleton (all 3 subsystems) ;
  - Section 3.2 : testing of the core learning services.
- The implementation of the local learning functionalities (A2) is described in:
  - Section 2.2.2 : implementation of NesC and Java components realizing the local learning modules ;
  - Section 2.2.4 : implementation of the local forward computation ;
  - Sections 2.4.2 and 2.4.3: implementation of mechanisms for training a local learning module and for its deployment to a LN device ;
  - Section 3.3 : testing of the local learning functionalities.
- The integration with Control Layer (A3) is described in:
  - Section 2.2.4 : implementation of the output interface towards the Control and Cognitive layer using PEIS tuplespace ;
  - Section 2.3.2 implementation of the control and configuration interface from the Control and Cognitive layer using PEIS tuplespace ;
  - Section 3.4 : testing of the integration between the Learning and Control Layer.

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 software is currently present at and in active use by the project partners. Based on the feedback of the partners, additional requirements are expected to emerge on the Learning Layer and these requirements will be addressed in the continuation of task 2.2 during months M18-M26, culminating in the release of the final Learning Layer in deliverable D2.3 at M30, which will also integrate the outcome of the research activities in Task 2.4 and 2.5.

4.3 New developments and unforeseen issues

The RUBICON Learning Layer depends on a number of enabling mechanisms provided by the Communication Layer, including the Synaptic communication support and the RUBICON gateway. Currently, such support is limited to single independent islands of TinyOS devices: therefore, the current CLS release is limited to such architecture. Updates of the CLS are expected to be delivered as soon as the corresponding enabling mechanisms are made available.

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

5. Appendix A – Reference Manual

In the following, we provide a snapshot of the full Javadoc reference manual for the Learning Layer Java API V1.0, including only the package-level documentation. A full HTML version of the manual is available on the RUBICON SVN repository.

Learning Layer API V1.0 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.	
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.

See:

Description

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.	
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.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		

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

Package generics Description

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.

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.

See:

Description

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
BroadcastCmdMsg	This class is automatically generated by mig.	
CommandMsg	This class is automatically generated by mig.	
GatewayWrapper	Preliminary implementation of the `GatewayInterface` wrapping the Rubicon Gateway functionalities needed by the Learning Layer.	
GatewayWrapper.LearningModuleFloatQueue	Helper class for maintaining a queue of `float` weight matrices, needed to complete the upload of a learning module in a target node of the LN.	
GatewayWrapper.LearningModuleShortQueue	Helper class for maintaining a queue of `short` weight matrices, needed to complete the upload of a learning module in a target node of the LN.	
LearningModuleCmdMsg	This class is automatically generated by mig.	
LearningModuleDownMsg	This class is automatically generated by mig.	
LearningModuleFloatCmdMsg	This class is automatically generated by mig.	
LearningNetworkWrapper	Implements a component that abstracts the distributed nature of the Learning Network.	
LearningNotificationMsg	This class is automatically generated by mig.	
LearningOutputMsg	This class is automatically generated by mig.	
NetIdCmdMsg	This class is automatically generated by mig.	
SerialDataMsg	This class is automatically generated by mig.	
SerialJoinedMsg	This class is automatically generated by mig.	
SynCreateCmdMsg	This class is automatically generated by mig.	

Package learningnetwork Description

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 to the Learning Layer. The package includes a `GatewayWrapper` class that wraps the RUBICON Gateway functionalities needed by the Learning Layer and that currently supports communication with TinyOS 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.

See:

Description

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 main Description

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.

See Also:

peisjava

Package manager

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

See:

Description

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 manager Description

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 Trainin Manager subsystem by providing methods for controlling the deployment of synaptic connection bundles associated to computational learning tasks.

See Also:

main

Package test

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

See:

Description

Class Summary		*Page*
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.	

Package test Description

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.

See Also:

learningnetwork, main

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.

See:

Description

Class Summary		Page
NetworkMirror	Implements the Network Mirror component: it maintains an up-to-date mirror of the modules in the LN and performs the training routines.	
TrainingAgent	Implements the Training Agent software component within the Training Manager in the RUBICON Learning Layer.	

Package training Description

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.