RUBICON
RUBICON Robotic UBIquitous COgnitive Network
Project No.: 269914

D1.3.1 – Preliminary version of Communication Layer

Editor: **Mathias Broxvall** **ORU**
Claudio Gennaro **CNR**

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

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

Issue Date	31/03/2012
Deliverable Number	D1.3.1
WP	WP 1 - Communication Layer
Status	☐ Draft ☐ Working ☐ Released ☒ Delivered to EC ☐ Approved by EC

Document history			
V	Date	Author	Description
0.1	24/02/2012	Mathias Broxvall	Initial outline
0.9	22/03/2012	Mathias Broxvall	Draft for internal quality control
0.99	04/04/2012	Mathias Broxvall	Final Draft

Executive Summary

This report describes the software, *Preliminary version of Communication Layer*, presented as deliverable D1.3.1 for RUBICON. We focus here on a description of the design and implementation of this software, the exact scope of the current implementation and outline the future work to be performed as part of task 1.2 - 1.4, leading up to deliverable D1.3.2 in month 24.

In addition to this report with an appendix documenting the communication 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.

This report furthermore summarizes, in brief, the state of workpackage WP1 and the achievement of all tasks scheduled for M1-M12 of RUBICON.

Contents

Figures

Tables

Abbreviations

RUBICON	Robotic UBIquitous COgnitive Network
PEIS	Physically Embedded Intelligent System
BDI	Belief Desire Intention (agent model)
AF	Agent Factory
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actuator Network

Chapter 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.

1.2 External view of communication layer

"Aka. A blackbox view of the communication layer from a hardware external perspective.

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

1.2.1 Services to PC/Robot based programmers

For the PC and robot based application programmers the communication layer provides the tuplespace based communication mechanism described in detail in deliverable D1.1 and extended as detailed in Chapter 2.1.

The most important features of this aspect of the communication layer include:

- **Communication** through publishing/subscribing to tuples
- **Introspection** by associative memory based searches of the tuplespace
- **Dynamic reconfigurability** of running components through the meta-tuple concept

For further details, see Appendix A.

For communication from PC/robots to WSN motes the communication layer provides a tuplespace based communication mechanism provided by the RUBICON gateway. These services are primarily:

- **Introspection** mechanism by providing a list of all available motes and their capabilities in terms of available sensor signals and accepted actuator signals.
- **Proxied sensing and actuation** by accepting subscriptions to sensors – automatically leading to the publications of these sensor data from the motes, and by accepting actuation signals that are transmitted to the motes.

For further details, see Chapter 3.

1.2.2 Services to WSN/WSAN programmers

For the generic WSN and WSAN application programmers the communication layer provides a set of basic WSN network based communication modalities described in deliverable D1.1. Furthermore, in the development of the RUBICON Learning Layer WSN network further requirements are raised on the need for *synaptic channels* with guaranteed quality of service constraints (QoS). For this purpose the services in Section 2.2.1 have been implemented for the developers of the learning layer and the cognitive layer of RUBICON .

Here is a list of features that the communication layer provides to the WSN developers:

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

1.3 Internal architecture of communication layer

Internally the communication layer consists of two separate sides depending on the use case. One for WSN/WSAN mote to mote communication and one for PC to PC communication, where we note that most robots fall into the latter category. The communication layers for each of these sides have been developed separately and have been integrated through a gateway mechanism that enables the PC based functionalities to use WSN/WSAN based devices transparently.

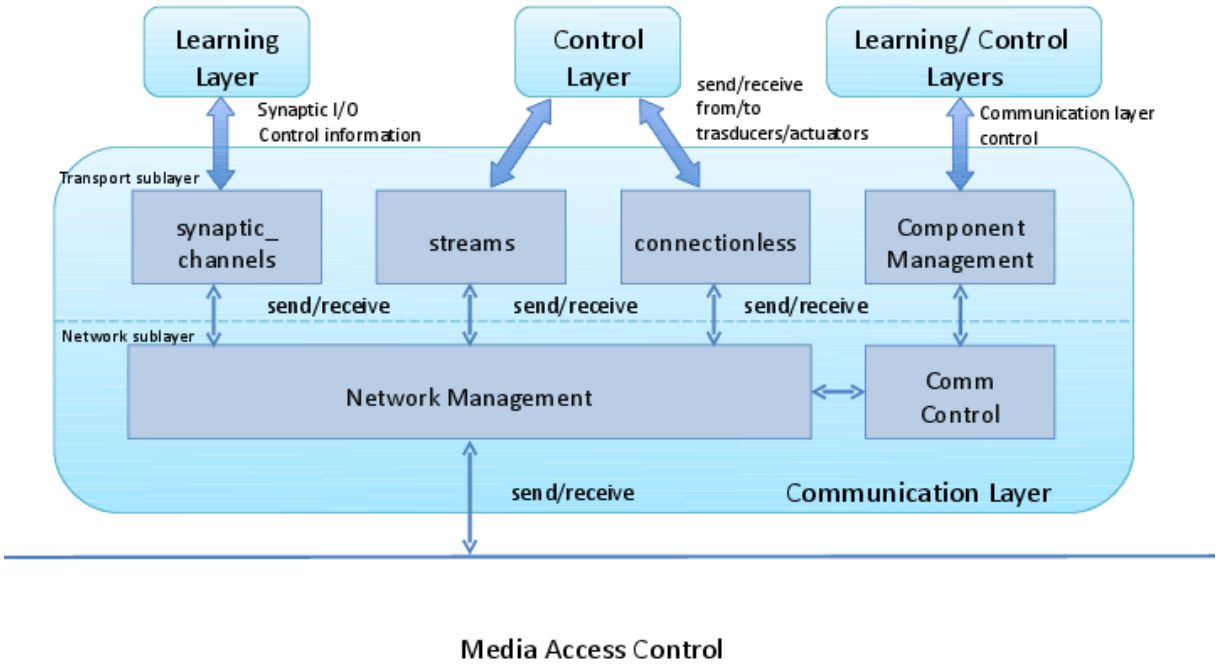

Figure 1.1: Software architecture for the WSAN side of Communication Layer: logical sublayers are separated by a dashed line, software components are small rectangles

1.3.1 Internals of WSN based communication layer

The Communication Layer is a complex software system organized into two logical sub-layers (using the terminology of the OSI reference model), separated by a dashed line in the box representing the communication layer (see Figure 1.1):

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

A sublayer is realized by a variable number of software components, depicted as simple rectangular boxes in Figure 1.1, that are distributed over an heterogeneous networked architecture comprising both resource constrained devices (e.g. sensor nodes) as well as powerful gateways.

The software components realizing the functionalities of the subsystems within the Communication Layer interact through local function calls as well as with remote messages: at this point, we abstract from such distinction and we represent all intralayer interactions as thin arrows in Figure 1.1.

1.3.2 Internals of PC based communication layer

Similarly to the WSN/WSAN based case the PC based communication layer also utilizes a layered approach correspondingly, loosely, to the OSI reference model. The following three layers are implemented in the unix library, the PEIS-kernel, and are used by all participating devices on the PC side of the RUBICON ecology.

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

In addition to the above layers of the PEIS-kernel, another important part of the communication layer is the RUBICON gateway which bridges the WSN/WSAN devices to enable their use as client of any (remote) device in the RUBICON ecology.

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 upon the first accepted project review.

1.4.2 Software requirements and hardware assumptions

Successful operation of this software requires the deployment of the following parts of the software on one or more desktop and/or robot PC's.

- Zero or more *islands* (groups) of WSN motes where each mote is within direct communication range of each other mote.
- Zero or more application specific motes in each such island. The application specific motes must be developed in TinyOS and use the TinyOS based communication layer API.
- One *sink-node* WSN mote *per island* that is connected through a USB serial to a PC running the RUBICON gateway software.
- Zero or more PC softwares linked again the PEIS-kernel unix library, utilizing the unix library based API for tuplespace based communication.

- All participating PC's must have a direct TCP/IP enabled link through wired or wireless ethernet or bluetooth links to some other PC's connected (recursively) to the remainder of the RUBICON ecology.

1.4.2.1 Platform dependencies, PC side

The primary target for the PC side of the Communication Layer is Ubuntu 11.10 based systems running on Intel x86 compatible hardware in 32-bit and 64-bit mode. The software has been tested and guaranteed to work on these systems, but have also to a lesser extent been verified to work an a range of other posix conformant operating systems such as other linux based systems as well as Macintosh based systems. The software is compatible only with a reduced feature set on windows based operating systems.

1.4.2.2 Platform dependencies, WSN side

The primary target for the WSN side of the communication layer are WSN/WSAN based motes based on a microcontroller and bluetooth stack capable of running TinyOS. Furthermore, we assume that the motes have a programming flash memory in addition to the onbord RAM memory. Examples of motes that satisfy these requirements include the Telosb derivatives such as Crossbow.

1.5 Dynamic P2P networks

In the current generation of the WSN/WSAN side of the communication layer we have opted for *single-hop* communication in order to preserve energy consumption and network bandwidth. To facilitate global communication between all participating WSN/WSAN's in a larger area we intend to provide the mechanism of *island-to-island* communication through the gateways devices.

The gateway devices as well as all other participating PC based devices currently utilizes the dynamic P2P network (see deliverable D1.1) of the PEIS-kernel in order to build a communication infrastructure around any series of ad-hoc networks between participating devices. These dynamic P2P networks will be used also by the gateway devices to facilitate the communication between islands as they join/leave the ecology.

This feature of island to island communication has not been implemented for this preliminary version of the communication layer, but will be developed to improve the scalability of the full communication layer for deliverable D1.3.2.

Chapter 2

RUBICON Communication Layer Development

For the development of the RUBICON communication layer we leverage two previous background technologies, the PEIS Ecology middleware and the MaD-WiSe Stream System, developed by ORU and CNR, respectively. These background technologies have been extensively documented in previous deliverables and related publications and we describe here only the major enhancements to these background technologies that have been developed as part of the new communication layer.

2.1 RUBICON customizations and extensions to the PEIS Ecology framework

Based on the previous version of the PEIS-kernel, generation **G5** developed before the commencement of RUBICON , we have developed a new version that satisfies the new requirements and operating assumptions of RUBICON . This version is called generation **G6** of the PEIS-kernel and the main new developments are summarized below.

2.1.1 Overhauled API

Task 1.3 (Implementation and Test of Communication Infrastructure) and Task 3.3 (Documentation and Internal Training) have developed a new overhauled API for all tuplespace manipulations and meta-tuple based communications.

The major API changes can be summarized as follows:

- **Mime-type** extension to all tuples
- **Abstract tuple** based mechanisms for: *subscriptions, callbacks, inserting tuples, tuple lookups and creating and subscribing to meta tuples*.
- **Convenience functions** that accepts simple strings for tuplenames and tuple data for performing the operations above.

- **Consistent naming** using *set* for operations that modify the locally owned tuplespace and *insert* for operations on remote tuples.

The full API is documented extensively in the API header files in a format suitable for doxygen based HTML and `man` page generation. For further details we refer the reader to this documentation as well as to the training material provided as part of the ORU workshop, June 2011, and to the Appendix A of this document.

2.1.2 Appending of tuples

One new major bandwidth saving concept has been introduced to the PEIS based tuplespace and has been implemented in the G6 kernel. This concept is that of operations for appending data to tuples that are continuously updated. The four new API functions `append[String]Tuple[byAbstract]` have been introduced as a mechanism for making an incremental change to an already existing tuple. As the name suggests, the function of these calls serve to append additional data to the end of a tuple, and is used to update incrementally changing streams of data. We envision two common use patterns of this functionality:

Example 1: A component is creating a log of data, each time an additional entry is added to the log the C-string representation of the data is appended with the new value. Consumers that subscribe to this log receive at the first subscription the full value of the log, and at each subsequent update the appended data. By only transmitting these delta changes to data a smaller amount of data is transmitted over the network. From the point of view of the consumer applications the full updates log is instantly available in their local tuplespace cache.

As a second example, consider a component that is creating coded stream of data that utilizes *key frames* and *delta frames* in order to minimize the size of the final coded stream. This behaviour is typically employed in eg. video and sound coding and although the full stream can be reconstructed from only one initial key frame and all subsequent delta frames a common implementation is to insert key frames at regular intervals to simplify searches through the stream and to allow consumers to join the stream at (almost) any point. We see how this can be facilitated using the notion of tuple appends as follows:

Example 2: Component A is a video encoding component that captures data from a camera at eg. 5hz. At the begining of each second (eg. frame 0, 5, 10) , the component publishes the latest camera image as a full tuple using a PNG encoding. At the remaining frames, the encoder computes the *difference* between the last frame and the new frame and encodes this in a compressed format as an *appended* tuple. We remind the reader that each time that a full tuple (key frame) is written to the tuplespace the old values are overwritten, and each time an append is made the sum of the previous keyframe and appends are stored.

Now, assume that a consumer, component B, starts subscribing to the video stream at an odd frame number, eg. at frame 7. It will then receive a tuple containing the keyframe **5** plus the two difference frames **6, 7**. At the next two updates it will receive additional frames and any callbacks in B will be called with the full sequences **5,6,7,8**. At the third update the new tuple value will be only frame **10** and the callbacks will receive only this latest value. Transmissions over the network will be limited to atmost 1 keyframe and 4 difference frames at an initial subscription, and will be limited to only one difference or key frame at each new update to the network.

This implementation of this change relies on the presence of a new (semi-internal) *sequence number* field of the tuple and the semantics outlined above. For further details we refer the reader to the doxygen documentation for the append functionality.

2.1.3 64-bit compatibility

Since RUBICON systems are expected to run on a wide range of platforms including both 32-bit and 64-bit desktop machines and robots we have extended the Communication Layer to operate seamlessly on both forms of platforms.

All PEIS-kernel internal communications have been rewritten to ensure that externally communicated packages use *network byte order* and use a suitable bit-length representation of all external data. Thus both 32 and 64-bit machines perform conversions between the internal memory formats and the network byte format upon transmitting and/or receiving PEIS-kernel packages such as routing packages and all other internal services used in the establishment of the dynamic P2P network.

Due to the use of ASCII representations in the general tuplespace no such conversions have been needed for the application specific components.

Additionally, the internal memory handling in the PEIS-kernel and of the tuplespace have been overhauled to avoid memory related problems with 64-bit architectures - most notably in the generic mechanisms for function callbacks and the internal hashtables.

2.1.4 Bluetooth linklayer

For the G6 version of the PEIS-kernel developed for RUBICON a linklayer module for Bluetooth connections has been re-implemented. This provides an alternative link layer that can be used standalone or in conjunction with the TCP link layer. The bluetooth link layer module is compile time optional, and requires that `libbluetooth-dev` is installed to compile.

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

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

Multiple components can be run using the same bluetooth adaptors. However, only one component can use the adaptor for "broadcasting" that this bluetooth adaptor is connected to a PEIS-kernel. Currently, the first PEIS that opens the device gets this privilege, and hence, when that PEIS is closed no other PEIS will make incoming connections to any PEIS on this computer.

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

By regularly attempting to connect using L2CAP port 7315 (HELLO connections) to detected bluetooth devices, it can be determined if they are PEIS and basic host information data can be transmitted to/from the detected device. The messages used on this L2CAP port also transmit the port numbers of any ports for incoming connections.

By using the signal strengths, and the results of the periodic HELLO connections, it is possible to determine when

a bluetooth connection between two PEIS might be possible. This is reflected by the peisk_bluetoothIsConnectable function, and is used by the connectionManager to determine which devices should connect to which.

2.2 Development of the WSAN Communication Mechanisms

The purpose of the Communication Layer is to implement the stack of communication protocols for the applications that run on top of it (Learning Layer , Control Layer , etc.). The application is the real actor of the communications, and makes the appropriate calls to the communication layer primitives in order to establish the point-to-point communication between two motes. For example if an application wants to make the temperature reading from a mote, the developer must write a small program that reads the sensors and sends the data to a remote node of the RUBICON ecology through the Communication Layer . This is the same as what happens with TCP/IP, in which applications such as telnet and ftp client and server are the ones who actually implement the communication by means of the TPC/IP primitives.

The rest of this section provides the developers of the motes applications an overview of the components of the transport level of the communication layer introduced in Chapter 1. Please refer also to Figure 1.1.

2.2.1 Synaptic Communication

The Learning Layer needs to communicate through appropriate mechanisms implemented by the communication layer. This can be achieved through the abstraction of the *Synaptic Channel*, which provides a point-to-point communication channel between two nodes of the ecology (motes). On the top of the communication mechanism provided by the synaptic channels, the learning layer builds the Synaptic Connections, an abstraction of a connection between two neuron residing on different nodes of the RUBICON ecology.

From the point of view of the QoS, the communication layer provides two communication modalities:

- *Reliable*: in this modality, the source node always transmits the reading of the remote node at each clock tick. At the destination node, the communication layer informs the learning layer of any problems with the delivery of the remote neuron readings. For instance, the unavailability of certain neuron readings is signalled by the Communication Layer to the Learning Layer so as to allow triggering of the appropriate fault handlers.
- *Power save*: in this modality, the reading of a remote node is transmitted, by the source node, only if the reading has changed since the last clock tick. Unchanged readings will not be transmitted and missing data at the destination node will be interpreted as unchanged data.

Note that, the power save modality is transparent to the Learning Layer : an unchanged reading that is not transmitted by the source node should anyway appear in the input interface of the destination node and its availability signalled to the learning layer. It is the responsibility of the communication layer to notify the learning layer at the destination node of the availability of the remote input at the clock tick (even if this input is unchanged since last tick).

The Reliable modality is part of the future work and will be implemented for the next deliverable D1.3.2 in month 24.

In the rest of this section we give some examples of use of the synaptic channels, the complete list of the synaptic channels API can be found in the Appendix B.

The `create_syn_channel_out` sets up of a Synaptic Channel between local node and a specified destination node. Suppose that the Learning Layer wants to setup a synaptic connection between nodes A and B. The learning layer application running on node A must issue a request to the communication layer in A to create a synaptic channel between A and B by calling:

```
create_syn_channel_out (nodeB, n, QoSParams)
```

Where *nodeB* is the destination node of the Synaptic Channel, and n is the number of remote neurons readings. *QoSParams* will be used in the future to setup the QoS of the channel. This primitive returns *FAIL* if creating the channel is impossible because, for example, there is no more available space to store a new channel or the channel already exists. If it returns *SUCCESS* then a `createSynChannelOutDone` event will be signaled once the Synaptic Channel has been opened. This event returns the *channel_id* of the opened Synaptic Channel. This *channel_id* must be preserved by the application to use the channel. In particular, the `start_syn_channel(channel_id)` must be used before starting transmitting the stream of neuron output vectors on the specified channel, and a channel that is activated with this primitive becomes an active synaptic channel.

In order to fill the data vector associated to the Synaptic Channel, you must call the primitive:

```
write_output(channel_id, m, output_data)
```

which sends the data stored in the buffer *output_data* of size m to the remote neuron.

The `stop_syn_channel(channel_id)` can be used to stop transmitting the stream of neuron output vectors over the specified active synaptic channel. Eventually, the `dispose_syn_channel(channel_id)` is used to close the specified Synaptic Channel that resides in the node where the function is invoked. If the channel is still active for transmission, it stops the synaptic communications first and then destroys the channel. Symmetrically, the application that uses the communication layer setups of a remote input synaptic connection on a destination node that matches the activation of an output synaptic connection on the source node. This allows the application to complete the association between neurons residing on two different nodes of the ecology. Correspondingly, we have a `create_syn_channel_in(nodeA, n, QoSParams)` (where *nodeA* is the source node of the Synaptic Channel) and a `read_input(channel_id)` to read the data stored the input of the remote neuron the data stored in a local buffer.

The Learning Layer requires the Communication Layer to set up synchronous communication channels among the learning components residing on different nodes of the RUBICON ecology . For this purposes, all the motes involved in the learning network have an internal clock synchronized. The synchronization will be guaranteed by means of special broadcast message, which will be periodically sent by the sink mote in order to keep the motes clocks aligned. This functionality will be developed in the next deliverable D1.3.2 in month 24.

The primitive `setTimeStep(clock)` sets up the local period of the clock of a mote and starts it. The event `clock_tick(currentClock)` signals the tick of the current clock.

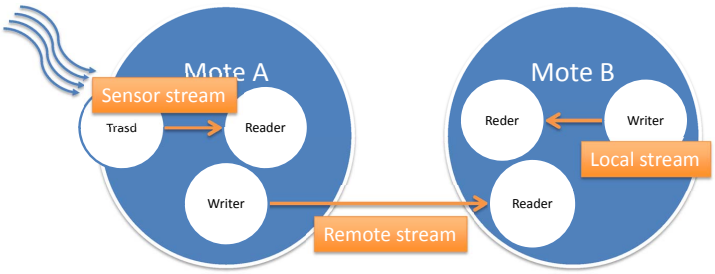

Figure 2.1: The paradigm of the data stream communication

2.2.2 Data Stream Communication

The *streams* component offers a unidirectional data collection and data communication abstraction. The stream represents a generic unidirectional data channel that is able to carry data records. In particular, we have three types of streams:

1. **Local Streams** represent a local data channel where read and write operations must occur on the hosting sensor.
2. **Sensor Streams** are the basic abstraction for collecting readings from transducers. They can only be read by operators since the writing is carried out by the associated transducers (these can be thought of as virtual operators writing to sensor streams).
3. **Remote streams** require cooperation between two nodes since they intend to provide a data channel between two different nodes. Write operations can be carried out on one of them (the stream write-end) and read operations can take place on the other (the read-end).

Figure 2.1 illustrates these concepts.

Another important concept in command/event-based systems is that of split-phase operations. The call requesting to start the operation returns immediately, without waiting for the operation to complete.

The main difference with the synaptic channels above described is that the streams provided by this component do not have a global timing. However, they are designed to enable the transmission of data at a certain frequency between remote nodes also of different islands. This component for now does not support messages to motes of remote islands, this is part of the future work and will be implemented for the next deliverable D1.3.2 in month 24.

The primitives of this component offer functionalities for creating, destroying, writing and reading records to/from streams, and their details are given in the following.

2.2.2.1 Local Stream

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

```
open_local()
```

that immediately returns the assigned stream descriptor. The read-end and write-end of a local stream are on the same node.

2.2.2.2 Sensor Stream

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

```
open_sensor(sensor_tid, stream_rate, sampling_type)
```

the *sensor_tid* specifies the identifier of the transducer whose sensor stream has to be opened, *stream_rate* specifies the input rate in milliseconds at which sensor must sample environment. `sampling_type` specifies if sampling is periodic or on demand. With periodic sensor streams the communication layer periodically commands a reading from the associated transducer with a fixed, timer-driven rate. With on-demand sensor streams the communication layer performs a transducer sampling only when explicitly requested to do so with a *read* command. When the transducer supplies its reading, the *streams* component writes a record (with the same content as for periodic sensor streams) to the stream and immediately signals the event *readDone*.

2.2.2.3 Remote Stream

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

```
open_remote(island_addr, mote_addr, symbolic_name, stream_rate, QoS),
```

where the `island_addr` specifies the address of the island destination of the message, *mote_addr* specifies the destination of stream, *symbolic_name* specifies the code univocally identifying the stream to be opened,

`stream_rate` specifies the rate in milliseconds at which source (destination) of the stream is going to write data into the stream, and QoS is the requested quality of service of the stream. The value 65535 for the *mote_addr*, used to indicate a broadcast, means that the application wants to open a stream with all the motes of the island. This kind of stream has important implications to the QoS parameter, which for the moment is not used by the Communication Layer . This parameter will be used to find a tradeoff between the collision avoidance and the throughput of stream transmission in case of streams simultaneously open with all the motes of an island. The user in this case may decide, for example, the throughput with which it will want to open a remote streams with all the motes of the island, and the system will respond *ok* if it is possible, or *no* if it is not possible, and in this case a recommended lower throughput.

2.2.2.4 Read/Write Operations

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

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

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

```
read(desc)
```

where argument *desc* identifies the stream, as usual, and `nbytes` specifies the number of bytes to be read. `read` is implemented with the split-phase paradigm: the call immediately returns. As soon as a record is available in the stream queue (possibly immediately), the communication layer removes and returns the oldest one to the user by signalling event

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

The arguments have the same meaning as for `write`. Finally, when a stream is no longer needed, it must be destroyed with the command

```
close(desc)
```

2.2.3 Connectionless Message Passing

The connectionless component is used to send commands to the motes, actuators, robots. To this aim, we make available to the developers a pair of primitives *send/receive* for exchanging single messages between two motes with or without acknowledgement.

The primitive:

```
send(island_addr, mote_addr, buf, nbytes, reliable, am_type)
```

sends a message to the destination mote in a specified island. Where *island_addr* specifies the address of the island destination of the message, and *mote_addr* specifies the address of the mote destination of the message. *buf* is the pointer to a memory region where the data to be sent are stored. *nbytes* specifies umber of bytes to be sent. The QoS for this primitive is definable with the boolean parameter *reliable*, which must be set to TRUE if an ack is requested for the current message. Finally, *am_type* identifies the interface to be used to send the message.

From the receiver point of view, the connectionless component provides an event, which signals that the data is ready.

```
receive(island_addr, mote_addr, buf, nbytes, am_type)
```

Where the pair *src_island_addr/src_mote_addr* identifies the source of the message, and *buf* the data to be received.

This component for now does not support streams to remote islands and reliable messages, this is part of the future work and will be implemented for the next deliverable D1.3.2 in month 24.

2.2.4 Component Management

The component management is responsible for the initialization the mote (radio and serial interface) and to handle the join to the island of the motes.

The `initCommunications` starts up the communication layer and switches on the radio of the mote. It must be invoked before any other call to the communication layer.

The primitive `joinIsland(description)` allows a mote to request to join the island under the competence of a given sink node. In response, the requesting mote will receive (i) the *island_addr* that identifies its new island, and (ii) its unique *mote_addr* within the island. This operation is important not only to allow the mote to communicate but also to notifies the presence of the mote to the RUBICON ecology by means of the Gateway described in Chapter 3 connected to the the sink. In this way the programmer can send a message containing a description with all the capabilities of the mote in terms of type of mote, transducers, and actuators. This information is sent via the parameter *description* of the *joinIsland* primitive. In Figure 2.2, we show a sequence diagram of the joining protocol.

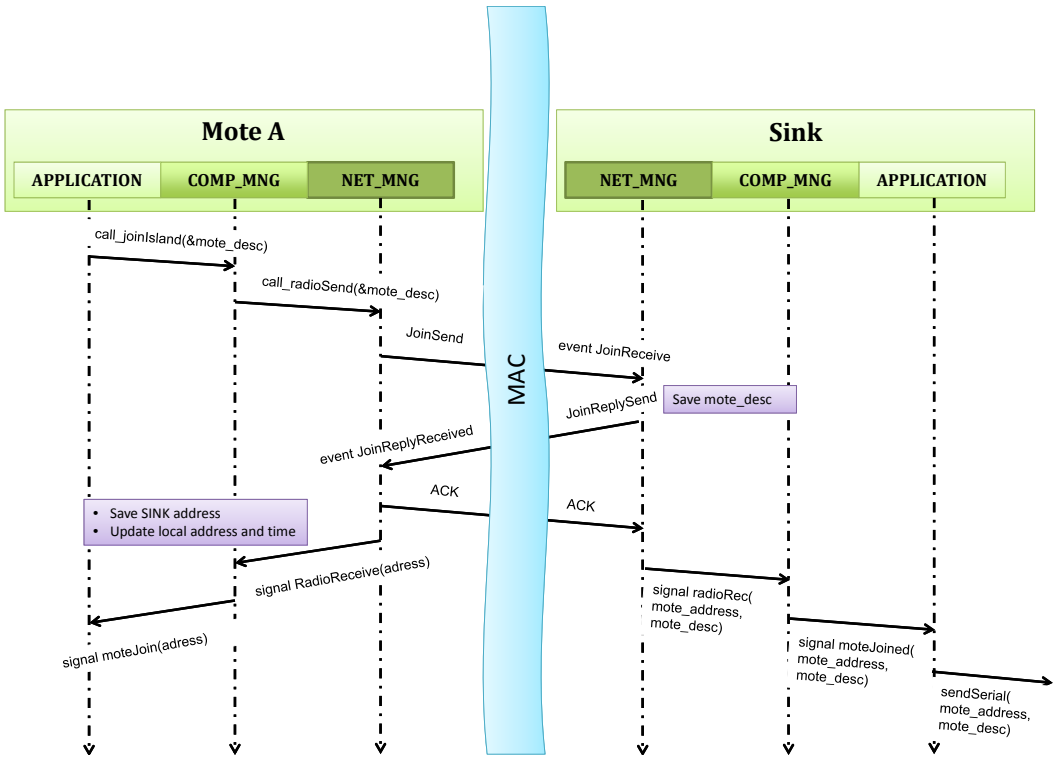

Figure 2.2: Sequence diagram of the joining phase of a mote in an island

Chapter 3

The WSAN/Robotic Gateway

The gateway is a component that talks through a *sink mote* connected to a USB serial port. The gateway uses this mote to communicate with all other WSN motes within range of the sink-mote. The set of all motes reachable in this way is referred to as the *island* under control by the gateway.

There is exactly one gateway per island, but there may be multiple islands and gateways in the whole ecology.

3.1 The WSAN Sink node

The WSAN sink node is a Telosb (or equivalent) mote that is connected through a serial USB connection to a PC participating in the RUBICON ecology. The mote runs the same version of the Communication Layer as the remainder of the motes, but with one notable difference. *The sink node always have id 0* in the current island.

By denoting the sink node with id 0, we are ensured that there cannot exist more than one such sink node on a single island, and all participating other motes can easily recognize messages originating from the gateway as opposed from other peers. The Communication Layer currently includes a hard-coded check for this id and if detected it forwards all island radio messages sent to the island or sent to a destination outside of the current island along the serial connection.

The message format of this serial communication adheres to the *Active Message* (AM-message) protocol of TinyOS with a RUBICON specific payload. For the example of *telosb* motes with the three sensors *temperature, humidity* and *light-level* the transmitted byte stream appears as in Table 3.1. Note that all non-constant occurrences of 0x7E and 0x7D are quoted as 0x7D5E or 0x7D5D respectively, giving these packages a length of *atleast* 26 bytes including the terminating 0x7E end-of-frame byte.

Table 3.1: Example byte format of AM-messages from sink to gateway.

1 byte	1 byte	1 byte	2 bytes	2 bytes	1 byte	1 byte	1 byte
Frame: 0x7E	Protocol: 0x45 or 0x44	Dispatch	AM Destination	AM Source	Length	Group	Type
1 byte	2 bytes	2 bytes	2 bytes	2 bytes	2 byte	2 bytes	2 bytes
Source island	Source mote	Counter	Temperature	Humidity	Light level	RSSI	CRC

3.2 Responsibilities of gateway software

The RUBICON gateway software consists of a unix program `gateway` that communicates with a sink node connected to a local USB port. For each island controlled by a machine[1] a local PEIS-init component is created by instantiating the gateway software with the specific hardware address of the USB port it uses to communicate with its WSN island. This component is configured to start automatically upon boot-up time of the computer. See Figure 3.1 for an example of such a configuration file.

The RUBICON gateway software uses the sink-mote to provide access to the raw sensor readings from the WSN/WSAN nodes reachable by the sink-mote and, in the future, to send actuation commands to these nodes.

- **Introspection** mechanism by providing a list of all available motes and their capabilities in terms of available sensor signals and accepted actuator signals.
- **Proxied sensing and actuation** by accepting subscriptions to sensors – automatically leading to the publications of these sensor data from the motes, and by accepting actuation signals that are transmitted to the motes.

The first most of these services, introspection, is performed by listening to the data messages sent on regular intervals by all motes and forwarded by the sink node. The gateway collects a list of the identifiers of all motes and continuously publishes a list of *reachable* motes that satisfy the criteria that at least one beacon message have been received from the mote during the last 30 seconds.

Advantages: This simplistic notion of reachability provides a simple yet robust notion of the motes that are reachable – at the cost of a bandwidth overhead growing linearly with the number of motes sharing the same communication space. Any motes leaving the ecology or malfunctioning are detected within the 30 seconds above which is deemed sufficient for this project. In the future we will provide a mechanism to turn on/off the radio of a specific mote in the island depending on the needed sensor information.

The gateway currently provides the following services to the PEIS based components, as tuples:

- Publish a list of all motes in the current island. This list contains the following information for each mote.
  - Mote ID (integer)

[1] Multiple islands can be controlled by one machine by attaching multiple sink nodes attached to USB extension cables connecting to different rooms

```
# gateway.cmp
# RUBICON gateway
Name=gateway
Id=+1
Exec='gateway $PEISINIT'
ExtraArguments='--port /dev/ttyUSB0'
InitState=on
SpawnDelay=3.0
```

Figure 3.1: Example PEIS-init configuration file for a RUBICON gateway component

  - Mote type ID (integer and/or a bitfield)
  - Number of connected analog sensors
  - Number of connected analog actuators
  - Number of connected digital sensors
  - Number of connected digital actuators

- Publish a list of the latest sensor value
  - For each analog sensor, a single floating point value
  - For each digital sensor, an integer value. Note that Digital sensors may represent multiple bits of information, or just individual bits.
  - For each sensor, a parameter tuple specifying the requested sensing rate, represented as a floating point number of samples per second. (Ie, 0.5 means one reading every 2 seconds, while 5.0 means 5 readings per second).

- For each actuator, it should have a meta-tuple configuration mechanism to receive:
  - For analog actuators, a single floating point value
  - For digital actuator, an integer value. Again, with possibly multiple bits of data.

  In the current version of the Communication Layer this forwarding of actuation messages through the sink mote have not been implemented, and this functionality exists only as a place holder. Full actuation of motes will be implemented for the full Communication Layer in deliverable D1.3.2.

For an example view of the tuples published when the gateway is running in an ecology that two motes see Figure 3.2. In this example a Crossbow mote is present as mote 2 and a Mote IV is present as mote 3. Both contains the same three digital sensors (humidity, light and rssi) and a single analog sensor (temperature).

The gateway utilizes the PEIS-kernel to identify when there exists subscribers to the specific sensor tuples. This allows for future optimization by turning on the sampling of sensors only for when there exists valid subscribers – saving both bandwidth and energy consumption in the participating motes. Currently this mechanism exists as a place-holder in the gateway and will be implemented for the full Communication Layer in deliverable D1.3.2.

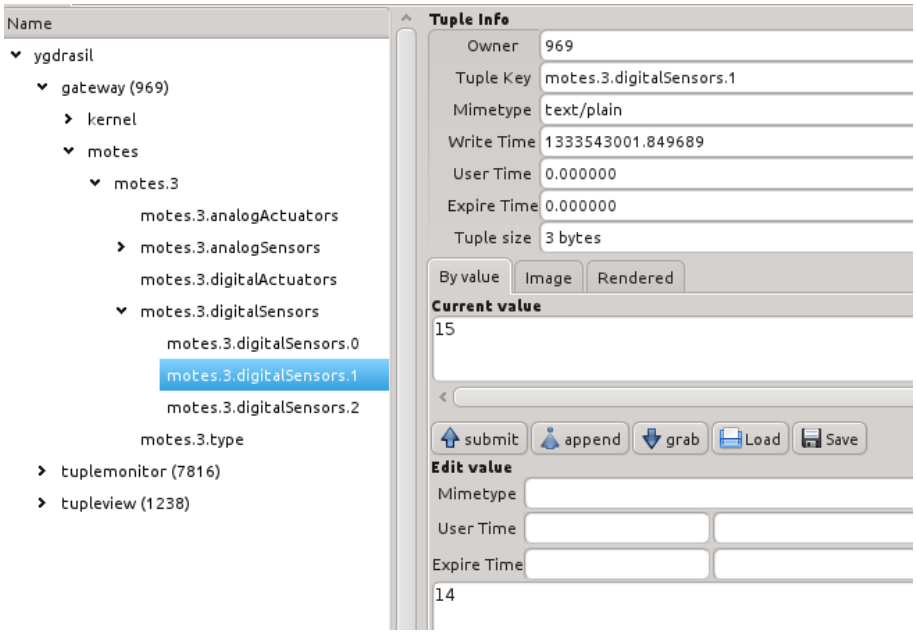

Figure 3.2: Example snapshot of the tuples provided by the gateway when two motes (2, 3) are present in the ecology.

3.3 Exported services to Control Layer

From the point of view of the Control Layer and all other participating components in the RUBICON ecology the gateway provides primarily a *homogeneous* mechanism for accessing motes of all different types and locations in the ecology. Furthermore, this mechanism make it transparent that it is a mote that is accessed by using exactly the same tuple formats and mechanisms as for any other participating component in the network. Specifically,

- The gateway appears as a traditional component in the network, complete with an XML based semantic description containing all of the capabilities of the present motes – thus providing a transparent *introspection* mechanism for the motes identical to the already used introspection in the remainder of the ecology.
- It performs all translations between data formats, including the translation of the compact bit field representation of mote capabilities and the XML descriptions suitable for self-osgi, the translation between binary and ascii representation of sensor values and of actuators.
- It enables configurability of parameters and input sources for each WSAN mote, through the meta-tuple concept used in the remainder of the ecology.
- It enables the (not yet implemented) feature of automatically enabling/disabling mote based sensors based on the current configuration of the network.

3.4 Requirements on Final version of Communication Layer

In later iterations of the gateway and of the sink mote, the different islands will be bridged by having the gateways forward messages through PEIS's dynamic P2P network.

These future versions of the gateway should provide the following services to the islands:

- List of available islands
- Routing of messages from one island to another island

If the next generation of the WSN/WSAN side of the communication network will allow multi-hop communications (yet to be decided) then the final version of the gateway must be able to use multi-hop WSN/WSAN communication to reach motes associated to the same island but currently outside of the direct communication range of the gateway.

Chapter 4

Testing Suites

In order to validate the developed communication layer for the purpose of RUBICON we have in Deliverable D3.1 described a sequence of *testing suites* that the Communication Layer must satisfy. Since parts of the Communication Layer is based on previously existing and tested software (eg. the purely WSAN based communication mechanisms and the purely PEIS based communication mechanisms) these tests focus only on the newly developed features – mainly that of PEIS to WSAN communication through the gateway and the specific features needed for implementing the distributed Learning Layer of the RUBICON.

We present in this chapter the latest results of the test cases and describe briefly the API and tuple interfaces that the performed tests exercise. We note also that a few of the test cases are not (yet) applicable for the current iteration of the Communication Layer . These test cases will be implemented before and presented in the next version of the control layer, released as Deliverable D1.3.2.

4.1 New test cases

Since many of the Communication Layer features were only on an initial design stage when D3.1 was completed (M6), tests for evaluating the synaptic channels were missing. We therefore introduce an additional test case in Table 4.1 and present the result of the evaluation of this test case together with the remaining test cases in Section 4.2.

4.2 Test case evaluations

We summarize below the API functions of the Communication Layer that have been exercised in each of the test cases as well as the result of the performed tests. For further details of the performance of the tests we refer the reader to the relevant sourcecode and associated scripts.

The first two test cases `TDiscoveryMote` and `TSensorX` exercises the basic communication from motes.

Test case	API	Status
TDiscoveryMote	Test case implemented on WSAN motes by using commands and events from the `ComponentManagement` module of Communication Layer . • command `joinIsland(mote_desc_t *description)` • event `moteJoined(uint8_t island_addr, uint16_t mote_addr, mote_desc_t *desc)` Test case also verified on the PEIS Ecology using the gateway tuples by subscribing to • `motes.*` displaying all discovered motes • `motes.X` where X is a specific mote to be discovered	**passed**
TSensorX	Tested on the mote side using: • command `SSToApp.open_s(uint8_t sensor_tid, uint32_t stream_rate, uint8_t buffer_size, uint8_t sampling_type)` • command `SSToApp.read(stream_type stream_desc, uint8_t nbytes)` Tested on PEIS side through gateway using: • `motes.X.analogSensors.Y` where X is a specific mote and Y is a an integer (0 - 3) representation of sensor Hardcoded integer values 0 - 3 used for default sensors of Mockup layer.	**passed**

The execution of the above two test cases have been illustrated by a timeline in Figure 4.1. In this execution the two motes, Mote2 and Mote 3, were present. Mote2 was removed at the timepoint marked in red, which was discovered at the timepoint marked in green and it was re-introduced again at the timepoint marked in yellow. The re-introduction was detected at the immediate next time period after the yellow marker. Capturing and visualization of the events was done with the visualization tools described in Deliverable D3.2.

The next two tests, `TSensorXError` and `TActuatorX` perform more elaborate checks on bi-directional communication with the motes.

Test case	API	Status
TSensorXError	Not tested at time of deliverable. Intended to use • command `SSToApp.open_s(uint8_t sensor_tid, uint32_t stream_rate, uint8_t buffer_size, uint8_t sampling_type)`	**passed**
TActuatorX	Tested on the mote side using: • command `Connectionless.send(uint8_t island_addr, uint16_t mote_addr, int8_t* buf, uint8_t nbytes, bool reliable, uint8_t am_type)` Not yet tested from the gateway side pending next iteration of sink mote.	**50% done**

Since task T1.2 and task T1.4 have not yet commenced there is not yet any specific results in providing a generic semantic description of the motes. As such the sensor and actuator readings provided by the gateway are non-generic and the two test cases above have only been performed for the WSAN side of the network and not from the gateway. On the PEIS ecology side of the network any attempt to read non-published tuples will fail, as such `TSensorXError` is automatically satisfied. For the `TActuatorX` no actuation from the PEIS ecology to WSAN network is possible until the next iteration of the sink mote, and thus does not yet pass the test above. Both of these tests therefore have to still be included in the tests of the next iterations of the gateway.

The last two tests presented in D3.1 relates only communication with WSAN motes from remote motes and PEIS components. Since the inter-island communication with motes were not scheduled for this initial prototype of the Communication Layer these tests have yet to be implemented and passed.

TRemoteSingleIslandStream	Not tested from the gateway side since no direct PEIS-WSN stream connections will be made. Inter-island WSN to WSN connections not implemented for this iteration of Communication Layer .	**not implemented**
TRemoteSingleIslandMsg	Not yet tested from the gateway side pending next iteration of sink mote.	**not implemented**

The final test is that of exercising the synaptic channel operating in the Learning Layer. This actual test as described in Table 4.1 is passed by the latest iteration of the Communication Layer and the exercised API functionalities are presented below.

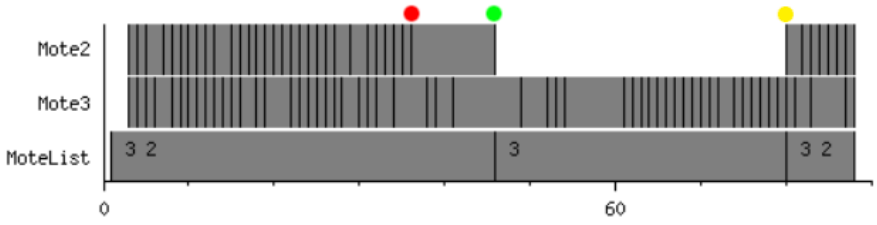

Figure 4.1: Timeline of sensor values and discovered motes during execution of test case `TDiscoveryMote`. Updates of sensor values marked by ticks on the *Mote2* and *Mote3* axises. Detected motes marked on the *MoteList* axis.

TSynapticChannel	Tested on the Mote side using: • command `create_syn_channel_out(uint16_t dest, uint8_t size, int8_t params[DIM_SYNCHANNEL_PARAM]);` • event `createSynChannelOutDone(syn_ch_id_t channel_id, error_t success);` • command `create_syn_channel_in(uint16_t src, uint8_t size, int8_t params[DIM_SYNCHANNEL_PARAM]);` • event `createSynChannelInDone(syn_ch_id_t channel_id, error_t success);` • command `setTimeStep(uint16_t clock);` • command `start_syn_channel(syn_ch_id_t channel_id);` • command `write_output(syn_ch_id_t channel_id, uint8_t size, syn_data_t* output_data);` • event `synData_received(syn_ch_id_t channel_id, uint8_t status);` • command `read_input(syn_ch_id_t channel_id);` Not relevant to test from Gateway side due to nature of test.	**passed**

4.3 Test case conclusions

The tests outline in D3.1 and the new additional test have been performed successfully. Additionally, a number of less formal tests have been performed during the development of the WSAN mockup layer and of the gateway. The software is deemed usable for the next stage of the project.

Table 4.1: Description of the synaptic channel test case

Test Case ID:	TSynapticChannel	**Focus:**	Learning Layer integration	**Type:**	Positive
Purpose:	Test the ability to integrate learning layer synaptic channels in the WSAN				
Environment:	Mote 1, interfaced with a number of sensors. Mote 2, base station connected to a PC via USB. Software Mote 1: Communication Layer, Learning Layer and test program (PTSynChannelMote.WSAN). Software Mote 2: Communication Layer, Learning Layer and test program (PTSynChannelSink.WSAN). Software PC: TinyOS SDK and test program (PTSynChannel.PC).				
Initialization:	Run test program PTSynChannel.PC on PC side and test program PTSynChannel.WSAN on motes 1 and 2.				
Test Process:	**PTSynChannelMote.WSAN** 1. Mote 1 opens an input local synaptic channel (D1.1 /5.3.1/*create_syn_channel_in*) to acquire the sensor values and a remote output synaptic channel (D1.1 /5.3.1/*create_syn_channel_out*) to forward data to Mote 2 2. Read acquired data from the local input synaptic channel (D1.1 /5.3.1/*read_input*) 3. Perform local computation and write output data into the remote synaptic channel (D1.1 /5.3.1/*write_output*) 4. Start the output synaptic channel (D1.1 /5.3.1/*start_syn_channel*) **PTSynChannelSink.WSAN** 1. Mote 2 opens a remote input synaptic channel (D1.1 /5.3.1/*create_syn_channel_in*) to receive data from Mote 1 2. Listen for the synaptic data reception event (D1.1 /5.3.1/*syn_input_update*) 3. Read received data from the remote input synaptic channel (D1.1 /5.3.1/*read_input*) and forward them to the USB port **PTSynChannel.PC** 1. open USB connection and repeat until exit signal 2. read data from USB port				
Test Results:	Test program PTSynChannel.PC will receive periodically the data acquired and computed by Mote 1 and forwarded to Mote 2 by means of Synaptic Channels.				

Chapter 5

Conclusions

5.1 Compliance to workplan

We summarize this report with an overview of the planned tasks for the first year of RUBICON within WP1 as documented in the Description of Work and the performed work as documented in deliverable D1.1, D1.3.1 and the progress reports.

Task 1.1 Specification and design of the network (M0 - M6): have been performed as planned and finished in October 2011. The result of this task is documented in D1.1 and provided the specification used in the implementation of the software described in this report.

Task 1.3 Implementation and test of the communication infrastructure (M4 - M24): is currently ongoing, and have led to the revisions of the PEIS-kernel middleware described in Section 2.1 and to the development of the WSN side of the communication layer described in 2.2. Initial performance tests have been performed on early version of the communication layer (i.e. *the Mockup layer*) in partnership with the control layer and UCD specifically.

Task 1.4 Adaption of existing data-sharing middleware and proxy solutions for WSN/robot integration (M13-M24): have started earlier than scheduled and is still ongoing. The current work in this task has led to the development of the *gateway* software used for representing the WSN/WSAN based components in the tuplespace. The current solution to these problems is not based on proxies, and the gateways appears as a single component with an extensive and dynamically changing set of capabilities. Future work within this task will provide additional modularity, flexibility and transparency for the robotic devices that use WSN/WSAN motes.

We summarize the state of each task in WP1 and the achieved progress by March 2012 in Table 5.1.

Deliverable D1.1 Functional Design and Specification and Mockup layer (M6): has been delivered as scheduled in the form of a report documenting the overall architecture of the RUBICON communication layer and the software design of the developed components.

Deliverable D1.3.1 Preliminary version of the communication layer (M12): has been delivered in the form of access to the (timestamped) software repository containing all the code and examples needed for the execution of the communication layer, and to this written report describing the developed software. The deliverable appears as following the Description of Work:

Table 5.1: Degree of completion for each task in WP1

Task	Start – End	Achievement	Notes
T1.1	M0 – M6	100%	Documented in Deliverable D1.1
T1.2	M13 – M24	0%	Not yet started
T1.3	M4 – M24	60%	Sections 2.1, 2.2
T1.4	M13 – M24	20%	Chapter 3

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

These points are addressed as follows:

- The developed solutions for WSAN to WSAN communication are described in Section 2.2 and provide all needed communication infrastructure through the use of streams.
- The developed solutions for PC to PC communication (or robot-robot) are based on the tuplespace concept and implemented by the PEIS-kernel and by the additional developments in Section 2.1 and the redesign API interface documented in Appendix A.
- The integration of WSAN and Robotic devices are based on the gateway mechanism described in Chapter 3.
- The solutions for data and synaptic channels are implemented in the WSAN side of the communication layer as described in deliverable D1.1 and in Section 2.2.

5.2 Impact on project

The communication layer has progressed as expected and made available to the rest of the RUBICON project partners on a continuous basis through the use of the shared code repository. Use of the communication layer has been facilitated by the initial workshop in Örebro on using the PEIS-kernel for the PC side of the communication layer and use of the WSAN side of the communication layer have been facilitated by the initial (early) mockup layer.

The software is currently present at and in active use by all of the partners. Based on the feedback of the partners, additional requirements are expected to emerge on the communication layer and these requirements will be addressed in the continuation of task 1.3 during months M12-M24, culminating in the release of the final communication layer in deliverable D1.3.2.

5.3 New developments

Task 1.4 has started earlier than scheduled due to the need for integrating the WSAN and tuplespace sides of the communication layer. This should be considered a natural change to the schedule and has not caused significant

changes in the expended efforts.

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

Appendix A

Appendix A - PC based API to communication layer

The interface to the communication layer in RUBICON is from the point of view of PC based applications the unix library PEIS-kernel. This library provides the following API for initialization and for manipulating the tuples of any participating RUBICON device.

A.1 Non-threaded applications

We present in this appendix only the multithreaded wrappers around the API functions, and refer the reader to the documentation provided with the PEIS-kernel library for the non-multithreaded functions – and note that for each multithreaded function `peiskmt_X` there exists a similar non-threaded implementation `peisk_X`.

A.2 Threaded applications

Most (but not all) operating systems and programming languages have an implementation of posix threads, for these systems and languages there exists a multithreaded wrapper around the peiskernel.

This library is required when the application nativly uses threads and may call kernel functions from any threads.

When using callbacks some caution should be used. In this situation, callbacks etc. can be registered from any thread using the peiskmt_ functions. However, since the callbacks will be called by an internal peiskernel thread, they must make sure to terminate quickly (within approx 10-100ms), OR use the peisk_step function internally. Note that inside these callback functions it is an error to use any of the multithreaded wrappers, ie. do not use the peiskmt_$*$ functions, only the peisk_$*$ functions.

A.2.1 The Tuple data structure

The basic data structure for reading and manipulating tuples throught the RUBICON communication layer is the `PeisTuple` structure below. This structure represents a tuple in the local tuplespace and is also used for (partial)

tuple as used when doing searches in the local and in the distributed tuple space. Wildcards on specific fields are specified using NULL values (for pointers) or -1 for integers.

- int **owner**
- int **creator**
- int **ts_write** [2]
- int **ts_user** [2]
- int **ts_expire** [2]
- int **encoding**
- char ∗ **mimetype**
- char ∗ **data**
- int **datalen**
- char ∗ **keys** [7]
- int **keyDepth**
- int **alloclen**
- char **keybuffer** [PEISK_KEYLENGTH]
- int **isNew**
- int **seqno**
- int **appendSeqNo**

A.2.1.1 Data fields

A.2.1.1.1 int PeisTuple::alloclen

Size of allocated data buffer. Usually only used by the tuple space privatly. Should be zero (uses datalen size instead) or a value >= datalen

A.2.1.1.2 int PeisTuple::appendSeqNo

Sub-sequence number used to arrange the order of append messages

A.2.1.1.3 int PeisTuple::creator

ID of Peis last writing to this tuple, or -1 as wildcard

A.2.1.1.4 char∗ PeisTuple::data

Data of tuple or NULL if wildcard.

A.2.1.1.5 int PeisTuple::datalen

Used length of data for this tuple if data is non null, otherwise -1

A.2.1.1.6 int PeisTuple::encoding

Type of encoding of tuple data. Should be one of PEISK_ENCODING_{WILDCARD,PLAIN,BINARY}

A.2.1.1.7 int PeisTuple::isNew

True if tuple has not been read (locally) since last received write. If used in a prototype, then any nonzero value means that only new tuples will be accepted. (findTuple returns zero if not new).

A.2.1.1.8 char PeisTuple::keybuffer[PEISK_KEYLENGTH]

Memory for storing the (sub)keys belonging to this tuple. Only for internal use.

A.2.1.1.9 int PeisTuple::keyDepth

Depth of key or -1 as wildcard.

A.2.1.1.10 char* PeisTuple::keys[7]

Key of tuple split up into subparts and represented as C-strings. Eg. "foo.boo.moo" becomes the three strings "foo" "boo" and "moo", representing a tuple of key depth 3. Use NULL pointers as wildcards.

May not be modified directly by the user, use peisk_setTupleName function instead.

A.2.1.1.11 char* PeisTuple::mimetype

Pointer to MIMETYPE or NULL for wildcard

A.2.1.1.12 int PeisTuple::owner

The owner (controlling peis) of the data represented by this tuple, or -1 as wildcard

A.2.1.1.13 int PeisTuple::seqno

Sequence number incremented by owner when setting the tuple

A.2.1.1.14 int PeisTuple::ts_expire[2]

Timestamp when tuple expires. Zero expires never, -1 as wildcard.

A.2.1.1.15 int PeisTuple::ts_user[2]

Timestamp as assigned by user or -1 as wildcard

A.2.1.1.16 int PeisTuple::ts_write[2]

Timestamp assigned when first written into owner's tuplespace or -1 as wildcard.

A.2.2 API functions

A.2.2.0.17 void peiskmt_appendStringTuple (int *owner,* const char * *key,* const char * *value*)

Appends the given string to the tuple(s) specified by the given owner and key.

This function disregard any checks to see that the appended tuple is as we expected since previous (ie. no sequence numbers can be specified). Note that due to the semantics of appending tuples, attempting to perform an append on a tuple that does not yet exist will not result in any append operation on it.

Definition at line 682 of file peiskernel_mt.c.

A.2.2.0.18 void peiskmt_appendStringTupleByAbstract (PeisTuple * *abstractTuple,* const char * *value*)

Appends the given string to the matching tuple in the tuple space.

The argument tuple is here an ABSTRACT TUPLE that identifies which tuple that should be appended. You would normally have an empty data field in the given abstractTuple. If there is wildcards in the owner and or key parts of the abstract tuple, the data will be appended to each such tuple. If there is a concrete sequence number in the given tuple, then the append will only be performed if the two tuples have the same sequence number.

Note that due to the semantics of appending tuples, attempting to perform an append on a tuple that does not yet exist will not result in any append operation on it.

A.2.2.0.19 void peiskmt_appendTuple (int *owner,* const char * *key,* int *difflen,* const void * *diff*)

Appends the given data to the tuple(s) specified by the given owner and key.

This function disregard any checks to see that the appended tuple is as we expected since previous (ie. no sequence numbers can be specified). Note that due to the semantics of appending tuples, attempting to perform an append on a tuple that does not yet exist will not result in any append operation on it.

A.2.2.0.20 void peiskmt_appendTupleByAbstract (PeisTuple * *abstractTuple,* int *difflen,* const void * *diff*)

Appends the given data to the matching tuple in the tuple space. The argument tuple is here an ABSTRACT TUPLE that identifies which tuple that should be appended. Thus, you would normally have an empty data field in the given abstractTuple. If there is wildcards in the owner and or key parts of the abstract tuple, the data will be appended to each such tuple. If there is a concrete sequence number in the given tuple, then the append will only be performed if the two tuples have the same sequence number.

Note that due to the semantics of appending tuples, attempting to perform an append on a tuple that does not yet exist will not result in any append operation on it.

A.2.2.0.21 void peiskmt_autoConnect (char * *url*)

Add to list of hosts that will repeatedly be attempted to be connected to

A.2.2.0.22 PeisTuple* peiskmt_cloneTuple (PeisTuple * *tuple*)

Clone a tuple (including the data field). The user of this function is responsible for freeing the memory eventually, see peisk_freeTuple. Updates peisk_tuples_errno if failed. Multithreaded version of peisk_cloneTuple

A.2.2.0.23 int peiskmt_compareTuples (PeisTuple * *tuple1,* PeisTuple * *tuple2*)

See if two tuples are compatible, ie. could be unified. Returns zero if compatible, less then zero if tuple1 is $<$ tuple2 and greater than zero if tuple1 is $>$ tuple2.

This function is suitable to be used when sorting lists of tuples or getting the maxima/mini of tuples as well as in the search routines.

Note that the data field of tuples are compared using non case sensitive string compares. In the future this might be modifiable with tuple specific flags.

A.2.2.0.24 int peiskmt_connect (char * *url,* int *flags*)

Force a connection to specific url. Returns -1 if failed

A.2.2.0.25 PeisTupleResultSet* peiskmt_createResultSet ( )

Creates a fresh resultSet that can be used for iterating over tuples, you must call peiskmt_resultSetNext before getting first value.

A.2.2.0.26 void peiskmt_declareMetaTuple (int *metaOwner,* const char * *metaKey*)

Creates a meta tuple as the given meta-owner/meta-key. Initially points to no concrete data.

A.2.2.0.27 void peiskmt_deleteResultSet (PeisTupleResultSet *)

Deletes the given resultSet and frees memory.

A.2.2.0.28 void peiskmt_deleteTuple (int *owner,* const char * *key*)

Convinience function for quickly deleting a tuple given it's name and owner. This works by intersting a tuple which expires immediatly. A callback functions can be aware of this by noticing the expiry field of the tuple which recives the value PEISK_TUPLE_EXPIRE_NOW in field 0

A.2.2.0.29 int peiskmt_findOwner (char * *key*)

Blocks until an owner for the given key (in any possible owners namespace) have been found. Returns the ID of the found owner.

A.2.2.0.30 void peiskmt_freeTuple (PeisTuple * *tuple*)

Free a tuple (including the data field).

A.2.2.0.31 int peiskmt_gettime ( )

Gives the current time in seconds since the EPOCH with integer precision.

A.2.2.0.32 void peiskmt_gettime2 (int * *t0,* int * *t1*)

Gives the current time in seconds and microseconds since the EPOCH with two long int pointer arguments where the result is stored.

A.2.2.0.33 double peiskmt_gettimef ( )

Gives the current time in seconds since the EPOCH with floatingpoint precision.

A.2.2.0.34 struct PeisTuple* peiskmt_getTuple (int *owner,* const char * *key,* int *flags*) [read]

Wrapper for getting a value from the local tuplespace for the fully qualitified given key and owner.

Parameters

key	The keyname of the tuple.
owner	The owner ID for the namespace in which tuple belong Returns nonzero if successfull (returns pointer to tuple structure, use with care!)
flags	Flags modifiying the behaviour of getTuple. Bitwise OR of the flags PEISK_KEEP_OLD, PEISK_-FILTER_OLD, PEISK_BLOCKING and PEISK_NON_BLOCKING - respectively.

A.2.2.0.35 PeisTuple* peiskmt_getTupleByAbstract (PeisTuple * *prototype*)

Looks up a tuple in the local tuplespace and locally cached tuples using the fully qualified owner and key only (no wildcards permitted). Returns pointer to tuple if successfull.

A.2.2.0.36 int peiskmt_getTupleErrno ( )

Returns the error code for the last user called function.

A.2.2.0.37 PeisTuple* peiskmt_getTupleIndirectly (int *metaOwner,* const char * *metaKey,* int *flags*)

Use the meta tuple given by (metaOwner,metaKey) to find a reference to a specific tuple. Returns this referenced tuple [metaOwner,metaKey] if found. Must previously have created a metaSubscription to metaKey and metaOwner.

A.2.2.0.38 PeisTuple* peiskmt_getTupleIndirectlyByAbstract (PeisTuple * *M*)

Makes an indirect lookup and returns the tuple [M] referenced by meta tuple M. Meta tuples always point only to one specific tuple.

A.2.2.0.39 int peiskmt_getTupleName (PeisTuple * , char * *buffer,* int *buflen*)

Returns the key name of given tuple, eg. "foo.*.a" if the tuple has a three key parts where the second is a wildcard. Returns zero on success.

A.2.2.0.40 int peiskmt_getTuples (int *owner,* const char * *key,* PeisTupleResultSet *)

Looks up a tuple in the local tuplespace and locally cached tuples. May contain wildcards and may give multiple results. A previously created resultSet must be passed as argument, freeing this resultSet (using the deleteResultSet command) is the responsibility of the caller. The function precomputes all matching tuples and sets the given PeisTupleResultSet to point to first found tuple and returns nonzero if successfull. Note that the resultSet is not reset by a call to this function, thus multiple getTuplesByAbstract calls in a row gives a tuple resultSet with all the results concatenated. Returns number of new tuples added to the resultSet.

A.2.2.0.41 int peiskmt_getTuplesByAbstract (PeisTuple * , PeisTupleResultSet *)

Looks up a tuple in the local tuplespace and locally cached tuples. May contain wildcards and may give multiple results. A previously created resultSet must be passed as argument, freeing this resultSet is the responsibility of the caller. The function precomputes all matching tuples and sets the given PeisTupleResultSet to point to first found tuple and returns nonzero if successfull. Note that the resultSet is not reset by a call to this function, thus multiple getTuplesByAbstract calls in a row gives a tuple resultSet with all the results concatenated. Returns number of new tuples added to the resultSet.

A.2.2.0.42 int peiskmt_hasSubscriber (const char * *key*)

Wrapper to tests if anyone is currently subscribed to the given (full) key in our local tuplespace. Wrapper for backwards compatability.

A.2.2.0.43 int peiskmt_hasSubscriberByAbstract (PeisTuple * *prototype*)

Tests if anyone is currently subscribed to tuples comparable to the given prototype. Ie. if the prototype is concrete then this is a test if any subscriber would be triggered by it. If the prototype is abstract, then tests if it partially overlaps any subscription. For instance, if we have a subscription for A.*.C and we test with prototype A.B.* then the result is _true_.

Returns nonzero if test is true.

A.2.2.0.44 void peiskmt_initAbstractTuple (PeisTuple *)

Initializes a tuple to reasonable default values for searching in local tuplespace. Modify remaining fields before calling insertTuple or findTuple. Differs from peiskmt_initTuple in what the default values are.

A.2.2.0.45 void peiskmt_initialize (int * *argc,* char ** *args*)

Initializes peiskernel using any appropriate commandline options and start a new running thread for it.

A.2.2.0.46 int peiskmt_initIndirectTuple (PeisTuple * *M,* PeisTuple * *A*)

Makes a lookup for latest value of meta tuple M and initializes abstract tuple A to point to tuple referenced by M. Returns zero on success, error code otherwise

A.2.2.0.47 void peiskmt_initTuple (PeisTuple *)

Initializes a tuple to reasonable default values for inserting into some tuplespace. Modify remaining fields before calling insertTuple or findTuple. Differs from peiskmt_initAbstractTuple in what the default values are.

A.2.2.0.48 int peiskmt_insertTuple (PeisTuple *)

Inserts a tuple into local or remote tuplespace. The tuple is copied into private memory prior to insertion (thus it is safe to allocate tuples and their data on the stack). Tuples must be fully qualified and contain no wildcards. Returns zero if successfull (note. failure may be delayed if modifying tuples inside other components, such failures are not reported here)

A.2.2.0.49 int peiskmt_isEqual (PeisTuple * *tuple1,* PeisTuple * *tuple2*)

Basic test for equality, wildcards match only if both are wildcards. Note that the data field of tuples are compared using non case sensitive string compares. In the future this might be modifiable with tuple specific flags.

A.2.2.0.50 int peiskmt_isGeneralization (PeisTuple * *tuple1,* PeisTuple * *tuple2*)

Similar to peiskmt_comparTuples but treats wildcards differently. Returns true if tuple1 is a generlization or equal to tuple2 (eg. tuple1 has a wildcard or the same data as tuple2 in everyfield.)

A.2.2.0.51 int peiskmt_isMetaTuple (int *metaOwner,* const char * *metaKey*)

Returns true if the given meta tuple is currently pointing to a valid tuple.

A.2.2.0.52 int peiskmt_isRoutable (int *id*)

True if host is known and reachable according to routing tables.

A.2.2.0.53 int peiskmt_isRunning ( )

If the peiskernel is currently supposed to be running or is aborted.

A.2.2.0.54 void peiskmt_lock ( )

Locks the semaphore used by the peiskernel thread to avoid conflicts when calling peisk_* functions directly.

A.2.2.0.55 int peiskmt_parseMetaTuple (PeisTuple * *meta,* int * *owner,* char * *key*)

Picks apart the syntax of a meta tuple to give the owner, key of it.

A.2.2.0.56 int peiskmt_peisid ( )

Gives the ID this peis is currently running as.

A.2.2.0.57 void peiskmt_printTuple (PeisTuple *)

Prints a tuple in human readable format to stdout, used for debugging. Multithreaded version of peisk_printTuple

A.2.2.0.58 void peiskmt_printUsage (FILE * *stream,* short *argc,* char ** *args*)

Prints all peiskernel specific options

A.2.2.0.59 PeisCallbackHandle peiskmt_registerTupleCallback (int *owner,* const char * *key,* void * *userdata,* PeisTupleCallback * *fn*)

Wrapper to add a callback function to tuples with given fully qualified key and owner; which is called when tuple is changed in the local tuplespace. Provides backwards compatability with kernel G3 and earlier. Returns callback handle if successfull, 0 (or NULL) if unsuccessfull. Multithread version of **peisk_registerTupleCallback()**

A.2.2.0.60 PeisCallbackHandle peiskmt_registerTupleCallbackByAbstract (PeisTuple * , void * *userdata,* PeisTupleCallback * *fn*)

Callback function using given abstract tuple as prototype. Returns callback handle if successfull, 0 (or NULL) if unsuccessfull.

A.2.2.0.61 PeisCallbackHandle peiskmt_registerTupleDeletedCallback (int *owner,* const char * *key,* void * *userdata,* PeisTupleCallback * *fn*)

Wrapper to add a callback function to tuples when they are deleted. Uses a given fully qualified key and owner.Returns callback handle if successfull, 0 (or NULL) if unsuccessfull.

A.2.2.0.62 PeisCallbackHandle peiskmt_registerTupleDeletedCallbackByAbstract (PeisTuple * , void * *userdata,* PeisTupleCallback * *fn*)

Callback function using given abstract tuple as prototype. Invoked when matching tuples are deleted. Returns callback handle if successfull, 0 (or NULL) if unsuccessfull.

A.2.2.0.63 int peiskmt_reloadSubscription (PeisSubscriberHandle *subscriber*)

Forces a resend of subscription and reload all subscribed tuples.

A.2.2.0.64 void peiskmt_resultSetFirst (PeisTupleResultSet *)

Resets the resultSet cursor to point back to _before_ first entry, you must call peisk_resultSetNext before getting first value.

A.2.2.0.65 int peiskmt_resultSetIsEmpty (PeisTupleResultSet *)

Returns false iff there are more values available in the result set AFTER the current cursor position. Does not step the cursor value.

A.2.2.0.66 int peiskmt_resultSetNext (PeisTupleResultSet *)

Steps the resultSet cursor to the next value and returns nonzero unless no more values exists.

A.2.2.0.67 void peiskmt_resultSetReset (PeisTupleResultSet *)

Resets the resultSet to empty. Does not modify the cursor pointer.

A.2.2.0.68 PeisTuple* peiskmt_resultSetValue (PeisTupleResultSet *)

Returns the tuple currently pointed to by the resultSet cursor

A.2.2.0.69 void peiskmt_setDefaultMetaStringTuple (const char * *key,* const char * *value*)

See peisk_defaultMetaTuple

A.2.2.0.70 void peiskmt_setDefaultMetaTuple (const char * *key,* int *datalen,* void * *data,* const char * *mimetype,* int *encoding*)

Sets a default value to a tuple in our name space it it does not yet exists. Also creates a corresponding meta tuple that points to it. This allows for this meta tuple to be re-configured to point somewhere else at a later timepoint.

A.2.2.0.71 void peiskmt_setDefaultStringTuple (const char * *key,* const char * *value*)

Sets a value to a tuple in our own namespace if it has not yet been given a value. Usefull for providing default values to tuples which may be configured via the command line --set-tuple option

A.2.2.0.72 void peiskmt_setDefaultTuple (const char * *key,* int *datalen,* void * *data,* const char * *mimetype,* int *encoding*)

Sets a value to a tuple in our own namespace if it has not yet been given a value. Usefull for providing default values to tuples which may be configured via the command line --set-tuple option

A.2.2.0.73 void peiskmt_setMetaTuple (int *metaOwner,* const char * *metaKey,* int *realOwner,* const char * *realKey*)

Sets the meta tuple to point to given real tuple

A.2.2.0.74 void peiskmt_setRemoteStringTuple (int *owner,* const char * *key,* const char * *value*)

Wrapper for creating and setting a tuple in a tuplespace belonging to some other peis. Propagation is done by the other peis if successfull.

A.2.2.0.75 void peiskmt_setRemoteTuple (int *owner,* const char * *key,* int *len,* const void * *data,* const char * *mimetype,* int *encoding*)

Wrapper for creating and setting a tuple in a tuplespace belonging to some other peis. Propagation is done by the other peis if successfull. Provides backwards compatability with kernel G3 and earlier. Multithread version of **peisk_setRemoteTuple()**

A.2.2.0.76 void peiskmt_setStringTuple (const char * *key,* const char * *data*)

A wrapper function for setting tuples whose value are simple C-string. Provides backwards compatability with kernel G3 and earlier. multithread version of **peisk_setStringTuple()**

A.2.2.0.77 int peiskmt_setStringTupleIndirectly (int *metaOwner,* const char * *metaKey,* const char * *value*)

Inserts value into the tuple referenced by the tuple (metaKey,metaOwner). Must already be subscribed to (metaKey,metaOwner) using atleast a normal subscribe but also works with a metaSubscription or if metaOwner is ourselves. Returns zero on success, error code otherwise

A.2.2.0.78 void peiskmt_setTuple (const char * *key,* int *len,* const void * *data,* const char * *mimetype,* int *encoding*)

Simplifying wrapper for creating and setting a tuple in local tuplespace, propagating it to all subscribers. Provides backwards compatability with kernel G3 and earlier. Multithread version of **peisk_setTuple()**

A.2.2.0.79 int peiskmt_setTupleName (PeisTuple * , const char * *fullname*)

Decomposes a name into dot separated substrings and puts into the tuplefield. Returns zero on success.

A.2.2.0.80 void peiskmt_shutdown ( )

Shutdowns the peiskernel, stops the peiskernel thread and frees all related datastructures.

A.2.2.0.81 void peiskmt_snprintTuple (char * *buf,* int *len,* PeisTuple *)

Prints a tuple in human readable format to given buffer, used for debugging. Multithreaded version of peisk_-snprintTuple

A.2.2.0.82 PeisSubscriberHandle peiskmt_subscribe (int *owner,* const char * *key*)

Wrapper for subscribing to tuples with given key from given owner (or -1 for wildcard on owner). Multithread version of **peisk_subscribe()**

A.2.2.0.83 PeisSubscriberHandle peiskmt_subscribeByAbstract (PeisTuple *)

Subscribe using given abstract tuple as prototype

A.2.2.0.84 PeisSubscriberHandle peiskmt_subscribeIndirectly (int *metaOwner,* const char * *metaKey*)

Creates a subscription to the meta tuple given by metaKey and metaOwner. Neither metaKey nor metaOwner may contain wildcard.

A.2.2.0.85 PeisSubscriberHandle peiskmt_subscribeIndirectlyByAbstract (PeisTuple * *M*)

Creates a subscription to the tuple M and to the tuple [M] referenced by the latest content of T continously. Whenever T is changed the subscription to [M] changes also. Tuple M must have a fully qualified key and owner (ie. only point to exactly one meta tuple)

A.2.2.0.86 void peiskmt_tsUser (int *ts_user_sec,* int *ts_user_usec*)

Sets the DEFAULT user defined timestamp (ts_user) for all comming tuples that are created.

A.2.2.0.87 const char* peiskmt_tuple_strerror (int *error*)

Converts a tuple error code to a printable string

A.2.2.0.88 int peiskmt_tupleExists (const char * *key*)

Test if the named tuple exists in our own namespace. Returns true if existing.

A.2.2.0.89 int peiskmt_tupleIsAbstract (PeisTuple *)

Test if a tuple is abstract (contain any wildcards). Returns nonzero if it is abstract.

A.2.2.0.90 void peiskmt_unlock ( )

Unlocks the semaphore used by the peiskernel thread to avoid conflicts when calling peisk_* functions directly.

A.2.2.0.91 void peiskmt_unregisterTupleCallback (PeisCallbackHandle *callback*)

Remove a given callback. Returns zero on success, otherwise error code. multithread version of **peisk_unregisterTupleCallback**()

A.2.2.0.92 void peiskmt_unsubscribe (PeisSubscriberHandle)

Unsubscribe to tuples. Returns zero on success, error number otherwise. Multithread version of **peisk_unsubscribe**()

A.2.3 Callbacks and threading context

We remind the reader that the context of all callback functions are in general executed outside of the threaded application. As such these callbacks should only use the non-threaded functionalities and return within a very short time span. For more advanced computations we recommend the same mechanism as for interupt routine handlers in realtime operating systems – eg. pushing the sensor data (tuples) onto a stack for processing by the main thread. This can typically be implemented by the `peisk_cloneTuple` functionality and related functionalities.

Appendix B

Appendix B - WSAN based API to communication layer

The interface to the Communication Layer in RUBICON is from the point of view of WSAN. This library provides the following API for initialization and for manipulating the communications of any participating RUBICON mote device.

B.1 API functions

B.1.1 The Synaptic Channel Descriptor

The Synaptic Channel Descriptor contains the following information:

- **channel_id**. The *id* of the channel.
- **status**. The status of the synaptic channel (active, etc).
- **size**. the actual size of the synaptic channel, i.e., the number of synaptic connections.
- **params**. Channel parameters like type, QoS.
- **data**. The data to be sent by the synaptic channel.

B.1.2 The Structure of the Joining Message

When a mote join the island it sends the a message to sink that contains the following information:

- **mote_type**. An integer that identifies the type of mote (e.g, Iris, Micaz).
- **transducers**. A bitmask specifying the transducers present in the mote.
- **actuators**. A bitmask specifying the actuators present in the mote.

B.1.3 Synaptic_Channels Component

B.1.3.1 command void init()

It initializes the Synaptic component.

B.1.3.2 command error_t create_syn_channel_out(uint16_t dest, uint8_t size, int8_t params[DIM_SYNCHANNEL_PARAM])

It sets up of a Synaptic Channel between local node and a specified destination node. If the Synaptic Channel already exist, the size of the channel is increased by size parameter. *dest* (input) Mote address of destination end of the Synaptic Channel. *size* (input) Number of neuron readings whose values have to be transmitted to dest (i.e. the channel size). *params[]* (input) Channel parameters like type, QoS. It returns *FAIL* if creating the channel is impossible because, for example, there is no more available space to store a new channel. *SUCCESS* means a createSynChannelOutDone() event will be signaled once the Synaptic Channel has been opened.

B.1.3.3 event void createSynChannelOutDone(syn_ch_id_t channel_id, error_t success)

It signals the outcome of the output Synaptic Channel creation request. *channel_id* (output) Channel ID of the opened Synaptic Channel. *success* (output) Whether the Synaptic Channel creation was successful.

B.1.3.4 command error_t create_syn_channel_in(uint16_t src, uint8_t size, int8_t params[DIM_SYNCHANNEL_PARAM])

It sets up of a Synaptic Channel between a remote source node and the local node. *src* (input) Mote address of the source end of the Synaptic Channel. *size* (input) Number of neuron readings (in *src*) whose values have to be received (i.e. the channel size). *params* (input) Channel parameters like type, QoS. return *FAIL* if creating the channel is impossible because, for example, there is no more available space to store a new channel. *SUCCESS* means a *createSynChannelInDone()* event will be signaled once the Synaptic Channel has been opened.

B.1.3.5 event void createSynChannelInDone(syn_ch_id_t channel_id, error_t success)

It signals the outcome of the input Synaptic Channel creation request. *channel_id* (output) Channel ID of the opened Synaptic Channel. *success* (output) Whether the Synaptic Channel creation was successful.

B.1.3.6 command error_t dispose_syn_channel(syn_ch_id_t channel_id)

It closes the specified Synaptic Channel that resides in the node where the function is invoked. If the channel is still active for transmission, it stops the synaptic communications first and then destroys the channel. *channel* (input) The ID of the Synaptic Channel to be closed. It returns *FAIL* if the operation cannot be completed; *SUCCESS* otherwise.

B.1.3.7 command error_t start_syn_channel(syn_ch_id_t channel_id)

It starts transmitting the stream of neuron output vectors on the specified channel. A channel that is activated with this primitive becomes an active synaptic channel. *channel_id* (input) The ID of the Synaptic Channel whose stream of neuron output has to be started. It returns *FAIL* if the operation cannot be completed (for example the channel does not exists); *SUCCESS* otherwise.

B.1.3.8 command error_t stop_syn_channel(syn_ch_id_t channel_id)

It stops transmitting the stream of neuron output vectors over the specified active synaptic channel. *channel* (input) The ID of the Synaptic Channel whose stream of neuron output has to be stopped. It returns *FAIL* if the operation cannot be completed (for example the channel is not active, or does not exists); *SUCCESS* otherwise.

B.1.3.9 command error_t write_output(syn_ch_id_t channel_id, uint8_t size, syn_data_t output_data)

It fills the *ChOut* vector associated to the specified Synaptic Channel with the output data to be sent. *channel_id* The ID of the Synaptic Channel that uniquely identifies a *ChOut* vector structure in the output interface of the local mote. *size* Number of elements to be written in *ChOut*. *output_data* Pointer to the array of values to be written into the ChOut vector. It returns *SUCCESS* if the write operation is completed successfully, *FAIL* otherwise.

B.1.3.10 event void syn_input_update(syn_ch_id_t channel_id, uint8_t index, synStatus status)

It notifies the status update of an input synaptic connection. *channel* (output) The ID of the Synaptic Channel that uniquely identifies a *ChIn* vector structure in the input interface of the local mote. *index* (output) The index of the ChIn element whose status update has to be notified. *status* (output) The status of the synaptic connection. *UPDATED* means that the remote neuron reading has been successfully updated, *MISS* means a missing remote neuron reading.

B.1.3.11 event void synData_received(syn_ch_id_t channel_id, uint8_t status)

It signals the reception of synaptic data from a synaptic channel. *channel_id* The ID of the Synaptic Channel from which the data has been received. *status* Information about the received data (for example: new data, not changed data, etc.)

B.1.3.12 command syn_data_t read_input(syn_ch_id_t channel_id)

It reads the data stored into the specified *ChIn* vector of the input interface of the ESN running on the local mote. *channel_id* The ID of the Synaptic Channel whose *ChIn* vector has to be read. It returns the pointer to the data field of the requested input Synaptic Channel

B.1.3.13 event void clock_tick(uint8_t currentClock)

It signals the tick of the current distributed RUBICON clock T. *currentClock* The current value of the RUBICON clock.

B.1.3.14 command void setTimeStep(uint16_t clock)

It sets the RUBICON clock, that is the frequency at which the forward computation is executed, i.e. the output synaptic data are transmitted to the input buffer of a ESN through the synaptic channel. This command must be called before starting the neural computation. *clock* The new RUBICON clock.

B.1.4 Streams Component

B.1.4.1 command void init()

It starts the Streams Component.

B.1.4.2 command error_t open_remote(uint8_t island_addr, uint16_t mote_addr, uint8_t symbolic_name, uint32_t stream_rate, uint16_t qos)

It requests that a remote stream is opened. *island_addr* (input) The address of the island destination of the message. *mote_addr* (input) Destination of stream (if open_r is invoked by destination, *dest* is the node itself). *symbolic_name* (input) Code univocally identifying the stream to be opened. *stream_rate* (input) Rate at which source (destination) of the stream is going to write (read) data into (from) the stream. It is expressed in milliseconds. *qos* (input) Quality of service of the stream. It returns FAIL if opening the stream is impossible because, for example, Stream System layer data structures have no space to store a new stream or the stream is already opened. *SUCCESS* means a openRemoteDone() event will be signaled once the stream has been opened.

B.1.4.3 event void openRemoteDone(stream_desc desc, uint8_t symbolic_name, error_t success)

It signals that the remote stream has been opened. *desc* (output) Descriptor of the opened stream. *symbolic_name* (output) Code univocally identifying the opened stream. *success* (output) Whether the *open_r* was successful.

B.1.4.4 command stream_desc open_sensor(uint8_t sensor_tid, uint32_t stream_rate, uint8_t sampling_type)

It opens a sensor stream. *sensor_tid* (input) Identifier of the transducer whose sensor stream has to be opened. *stream_rate* (input) Rate at which sensor must sample environment. It is expressed in milliseconds. *sampling_type* (input) If sampling is periodic or on demand. It returns stream descriptor if opening has been completed successfully; -1 *otherwise* (for example no available space to store a new stream or the stream is already opened)

B.1.4.5 command stream_desc open_local()

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

B.1.4.6 command error_t close(stream_desc desc)

It closes the selected stream. *desc* Descriptor of the stream to be closed. It returns FAIL if the operation cannot be completed; *SUCCESS* otherwise.

B.1.4.7 command error_t read(stream_desc desc, uint8_t nbytes)

It requests that *nbytes* bytes are read from the selected stream. *desc* Descriptor of the stream to be read. *nbytes* Number of bytes to be read. It returns FAIL if the stream is not open or the read cannot be performed (for example, attempt to read from a remote stream opened in write mode).

B.1.4.8 event void readDone(stream_desc desc, int8_t buf, uint8_t nbytes, error_t success)

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

B.1.4.9 command int8_t write(stream_desc desc, int8_t buf, uint8_t nbytes)

It writes the input buffer into the selected stream. *desc* Descriptor of the stream to be written. *buf* Pointer to the buffer to be written. *nbytes* Number of bytes to be written in the stream. It returns the number of actually written bytes; -1 if the write fails.

B.1.5 Connectionless Component

B.1.5.1 command error_t send(uint8_t island_addr, uint16_t mote_addr, int8_t buf, uint8_t nbytes, bool reliable, uint8_t am_type)

It sends a message to the destination mote in a specified island. *island_addr* (input) The address of the island destination of the message. *mote_addr* (input) The address of the mote destination of the message. *buf* (input) Pointer to a region where the data to be sent are stored. *nbytes* (input) Number of bytes to be sent. *reliable* (input) TRUE if an ack is requested for the current message. *am_type* Type of the message. It identifies the interface to be used to send the message. It returns *FAIL* if it is not possible to send the message, *SUCCESS* otherwise.

B.1.5.2 event void ack(error_t success)

It signals that the acknowledgment of a previously sent message. success Whether the ack was successful.

B.1.5.3 event void receive(uint8_t src_island_addr, uint16_t src_mote_addr, int8_t buf, uint8_t nbytes, uint8_t am_type)

It signals that the data is ready. *src_island_addr* The address of the island source of the message. *src_mote_addr* The address of the mote source of the message. *buf* Pointer to a region where the received data is stored; meaningful only if *success* = *SUCCESS*. *nbytes* Number of actually received bytes; meaningful only if *success* = *SUCCESS*. *m_type* Type of the message. It identifies the interface from which the message is received.

B.1.6 Component Management

B.1.6.1 command void initCommunications()

It starts both radio and serial interface.

B.1.6.2 event void initCommunicationsDone(error_t success)

It signals the completion of both radio and serial component activation. *Success* Whether the radio and serial activation was successful.

B.1.6.3 command void joinIsland(mote_desc_t description)

It requests the sink of the current island to be joined. As consequence the mote will receive the *mote_addr* and the *island_addr. description* Pointer to the description of the capabilities (type, transducers and actuators) installed on the mote.

B.1.6.4 event void moteJoined(uint8_t island_addr, uint16_t mote_addr, mote_desc_t description)

It signals that a new mote joined the RUBICON system. *island_addr* Address of the joined island. *mote_addr* Identifier of the joined mote. *description* Pointer to the description of the capabilities (type, transducers and actuators) installed on the mote.