Wireless Software and Hardware platforms for Flexible and Unified radio and network control

Project Deliverable D4.2

First Operational network control software platform

Contractual date of delivery:	31-12-2015
Actual date of delivery:	04-01-2016
Beneficiaries:	iMinds, TCD, CNIT, TUB
Lead beneficiary:	TUB
Authors:	Anatolij Zubow (TUB), Mikolaj Chwalisz (TUB), Piotr Gawłowicz (TUB), Sven Zehl (TUB), Adam Wolisz (TUB), Peter Ruckebusch (IMINDS), Spilios Giannoulis (IMINDS), Ingrid Moerman (IMINDS), Pierluigi Gallo (CNIT), Nicolò Facchi (CNIT), Ilenia Tinnirello (CNIT)
Reviewers:	Pierluigi Gallo (CNIT)
Work package:	WP4 – Network Control
Estimated person months:	24
Nature:	R
Dissemination level:	PU
Version:	1.19

Abstract:

This deliverable provides a detailed description on the capabilities and performance of the first network control software platform that follows the Path 1 innovation strategy (black box). It also presents a list of improvements and extensions that will be implemented in Year 2.

Keywords:

Programmable network architecture, software-defined networking, network control

Executive Summary

This deliverable provides a detailed description on the capabilities and performance of the first ***network control*** software platform that follows the Path 1 innovation strategy (black box). It also presents a list of improvements and extensions that will be implemented in Year 2.

In Year 1, we have defined the ***global WiSHFUL architecture*** with its components and interfaces, the provided basic services. Unified Programming Interface (UPI) functionality has been implemented for two different platforms, namely, Linux-based wireless nodes and sensor nodes using the Contiki operating system.

The focus of this deliverable is on a detailed description and implementation the ***Unified Programming Interfaces for network control***, whereas deliverable D3.2 addresses radio control through *UPI_R* focusing on the lower layers, i.e. lower MAC and physical layer.

The following UPIs have been implemented

- *A network control interface, UPI_N,* for controlling the higher layers of the network protocol stack. The UPI_N implementation is based on the development of network connector modules, able to map platform-independent function calls into platform-specific tools and functionalities;
- *a global interface, the UPI_G,* which is required for coordinated (time synchronized) remote execution of configuration and monitoring related functions, UPI_R/N, on a group of nodes;
- *a management interface, UPI_M*, required for managing protocol software modules at any layer of the network protocol stack.

The full documentation of UPI_N, UPI_G and UPI_M together with the code of the implemented software is available in the WiSHFUL GitHub repository.

List of Acronyms and Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
AODV	Ad-hoc On-demand Distance Vector
AP	Access Point
BE	Best Effort
BLIP	Berkeley IP
COAP	Constrained Application Protocol
CPE	Customer Premises Equipment
CREW	Cognitive Radio Experimentation World – EU project
CTP	Collection Tree Protocol
CWMP	CPE WAN Management Protocol
DHCP	Dynamic Host Configuration Protocol
DVB	Digital Video Broadcasting
GITAR	Generic extensions for Internet-of-Things Architectures
GTS	Guaranteed Time Slot
HGI	Home Gateway Initiative
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
MAC	Medium Access Control
MTU	Maximum Transmission Unit
OLSR	Optimized Link State Routing Protocol
OS	Operating System
QoS	Quality of Service
RPL	Routing Protocol for Low-Power and Lossy Networks
SDN	Software-defined networking
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSID	Service Set Identifier
STA	Station
TinyRPL	IPv6 Routing Protocol for Low-power and Lossy Networks (RPL)
ToS	Type of Service
UPI	Unified Programming Interface
UPI_G	Unified Programming Interface global

UPI_M	Unified Programming Interface management
UPI_N	Unified Programming Interface network
UPI_R	Unified Programming Interface radio
VoIP	Voice over IP
VPN	Virtual Private Network
WAN	Wide Area Network
WiMAX	Worldwide Interoperability for Microwave Access
WDS	Wireless Distribution System
ZeroMQ	Embeddable networking library and a concurrency framework

Table of contents

1 Introduction

This deliverable describes the first WiSHFUL operational ***network control*** software platform. It describes the architecture and gives a detailed description of the implemented ***Unified Programming Interfaces for network control***, *UPI_N*, i.e. higher layers of the network protocol stack, whereas document D3.2 is focused on radio control, i.e. lower MAC and physical layer. Full documentation and the code of the implemented software platform is available in the WiSHFUL GitHub repository.

The focus of the network control software platform is on the description of the ***UPI_N*** definition as well as the ***basic services*** provided by the architecture, i.e. time synchronization, time-scheduled and remote execution of UPI functions. Moreover, we give a description of an additional ***global interface***, the ***UPI_G***, which is required for coordinated (time synchronized) remote execution of configuration and monitoring related functions, UPI_R/N, on a group of nodes. Note, that the WiSHFUL control framework provides the capability to run node-local or global control programs, which are devised to reconfigure network and radio behaviour. Finally, also ***management related functions*** required for managing protocol software modules at any layer of the network protocol stack are presented, ***UPI_M***.

We present implementation details for the two networking ***platforms supported***, namely i) systems using Linux Networking Subsystem and ii) Contiki Embedded OS.

2 General description of WiSHFUL architecture for network control

2.1 WiSHFUL components

The WiSHFUL monitoring and configuration engine (MCE), ***has two-tier architecture*** with local MCEs residing on each wireless node and a central global MCE. The ***global MCE*** enables *global control programs* to control the behaviour of each wireless device under test using the well-defined UPI-R/N interfaces provided by each node. Moreover, the global MCE can instantiate local control programs on each wireless node, which are executed by each node-local MCEs independently. Besides the global control there is also the option to control each node independently using a *local control program* on top of a ***node-local MCE***.

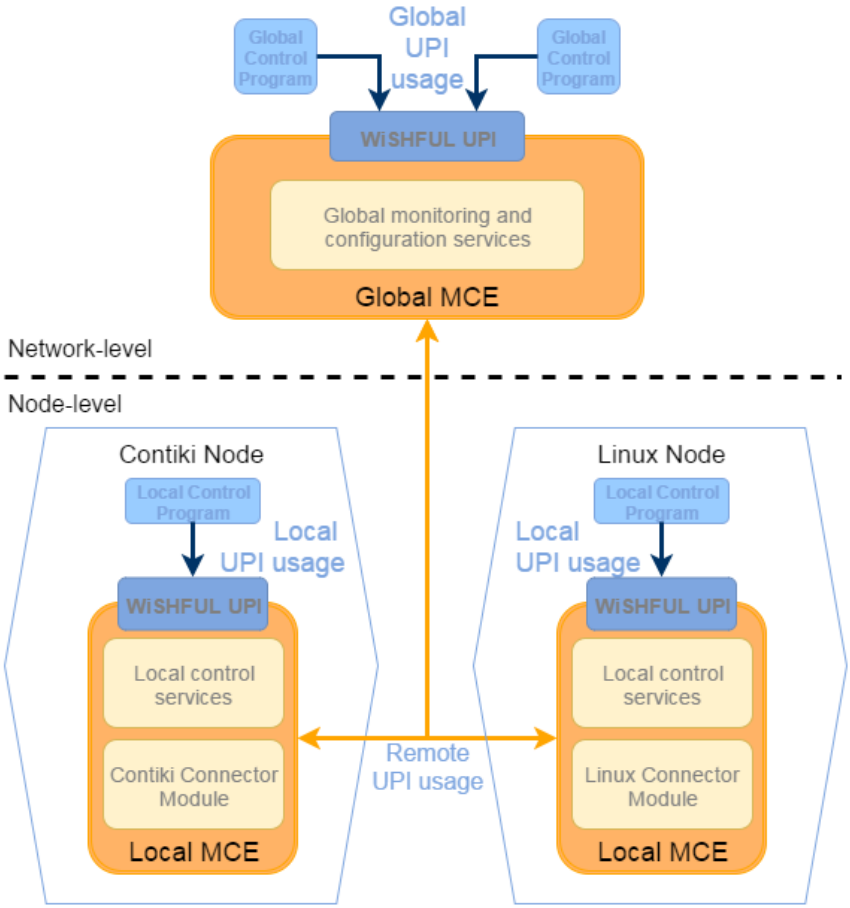

Figure 1 Overview of the components in the general WiSHFUL architecture.

2.2 UPI definition

To enable remote usage of UPI functions using the UPI_G interface, a system is required that supports remote procedure calls. For this purpose the UPI functions must have a generic signature that facilitates serializing function arguments during remote UPI calls. Such types of function definitions are very flexible but error-prone in usage. For this reason, more user-friendly versions are also required that shields end-user from the complexity by offering strongly typed interface descriptions.

The WiSHFUL architecture thus defines two levels of UPIs:

- A generic interface that allows to facilitate serialisation of arguments

```
#C definition
/* Change a higher layer parameter. The following arguments expected nic_t,
param_list_t where param_list is a list of param_t pointers.
**
** @param void* args: pointer to a list of arguments.
** @param int num_args: number of arguments
** @return void*: pointer to return buffer.
*/
void* UPI_N_setParameterHigherLayer(void* args, int num_args)
```

```
#PYTHON definitions:
""" Change a higher layer parameter. The following argument keys are expected
['iface','param_list'] where iface is a NIC_t interface description and param_list
is a dict with {string : Object} pairs.
```

```
        @param dict(string,Object) args: A dictionary containing arg_key arg_value
pairs.
        @return void*: pointer to the return buffer.
"""
def UPI_N_setParameterHigherLayer(args)
```

- A UPI helper interface that wraps the functions in more strongly typed versions:

```
#C definition
/* Change a higher layer parameter
**
** @param NIC_t* iface: the interface on which parameters need to be changed.
** @param param_list_t params: a list of parameters
** @return void*: pointer to a list of error_t elements.
*/
void* UPI_N_setParameterHigherLayer(NIC_t* iface, param_list_t params)
```

```
#PYTHON definitions:
""" Change a higher layer parameter

        @param dict(string,Object) param_keyValues: A dictionary containing param
key:value pairs.
        @return list(string): a list of error messages.
"""
def UPI_N_setParameterHigherLayer(iface, param_keyValues)
```

The second version will be offered in Python by Helper classes, in C by helper functions.

2.3 Basic Services

2.3.1 Node Discovery

A *global control program* requires functionality for ***automatic node discovery***. WiSHFUL provides the protocol developer an easy way to define the set of nodes he want to control. Any wireless node belonging to the same experiment group can be controlled by a *global control program* using the WiSHFUL UPIs. From that set of nodes the user can either select all of them or just a sub-set.

The following example code-snippet shows how a *global control program* defines the experiment group "MyWishFulTest" and waits until the two specified wireless nodes become available for control:

```
# name of the experiment group; only nodes of this group can be controlled
exp_group_name = "MyWishFulTest"
# get reference to global UPI
global_mgr = GlobalManager(exp_group_name)
# nodes under control
nodes = []
node0 = Node("192.168.103.125")  # nuc2
node1 = Node("192.168.103.134")  # nuc3
nodes.append(node0)
nodes.append(node1)
# node discovery: wait until all specified nodes are available
discovered_nodes = global_mgr.waitForNodes(nodes)
```

2.3.2 Control Program Model

The WiSHFUL framework follows a ***proactive approach***. A *local or global control program* has to trigger the execution of UPI functions on the wireless node under control. This polling-based

approach might be not sufficient for each control program application. Therefore, for the future we plan to support also a ***reactive approach***. Here the user can define a trigger where when a certain condition is fulfilled a registered callback function is executed.

2.3.3 Execution Semantics

The WiSHFUL MCE (local and global) supports two ***execution semantics***. The first is a synchronous blocking UPI call where the caller, i.e. the WiSHFUL *control program (local or global)*, is blocked until the callee, i.e. any UPI function, returns. The second option is an asynchronous non-blocking UPI function call. Here any UPI call returns immediately. The caller has the option to register a callback function so that he can receive the return value of the UPI call at a later point in time.

The following example illustrates the use of the two possible execution semantics:

```
if __name__ == '__main__':
    # get reference to local wishful engine
    local_mgr = LocalManager()

    # iface to use
    wlan_iface = 'wlan0'

    # Using the flat UPI
    # UPI_L call: exec remote function on UPI_R/N in 3 seconds
    UPIfunc = UPI_RN.setParameterLowerLayer
    # remote function args
    UPIargs = {'cmd' : UPI_RN.IEEE80211_CHANNEL, 'iface' : wlan_iface, 'channel' :
4}

    """
    Custom callback function used to receive result values from scheduled calls.
    """
    def resultCollector(json_message, funcId):
        log.debug('json: %s' % json_message)

    try:
        # (1) this is a blocking UPI call
        rvalue = local_mgr.runAt(UPIfunc, UPIargs)
        log.debug('Ret value of blocking call is %s' % str(rvalue))

        # create callback for this function call
        callback = partial(resultCollector, funcId=4711)
        # (2) this is a not blocking UPI call
        rvalue = local_mgr.runAt(UPIfunc, UPIargs, None, callback)
        # rvalue is empty
    except Exception as e:
        log.fatal("... An error occurred : %s" % e)

    # teardown engine
    local_mgr.stop()
```

2.3.4 Time-Scheduled Execution of UPI Functions

Beside the possibility of immediate execution of UPI functions either using a blocking or non-blocking scheme the WiSHFUL MCEs provide the possibility for ***time-scheduled execution*** of UPI functions at a particular point in time. This is important if nodes need to coordinate their actions in time, e.g. a set of nodes must perform a time-aligned switching to a new channel. The possibility for time-scheduled execution of UPI functions is especially important for *global control programs* if a non-real-time backbone networking system like Ethernet is used. In such networks we cannot expect that the WiSHFUL control commands are received by all nodes at the same time, e.g. due to network

congestion. Moreover, network congestion and delay are also reasons for providing hierarchical control over UPI_HC between local and global control programs.

The following example illustrates the time-scheduled execution of UPI functions:

```
if __name__ == '__main__':

    # get reference to local wishful engine
    local_mgr = LocalManager()

    # iface to use
    wlan_iface = 'wlan0'

    # Using the flat UPI
    # UPI_L call: exec remote function on UPI_R/N in 3 seconds
    UPIfunc = UPI_RN.setParameterLowerLayer
    # remote function args
    UPIargs = {'cmd' : UPI_RN.IEEE80211_CHANNEL, 'iface' : wlan_iface, 'channel' :
4}

    """
    Custom callback function used to receive result values from scheduled calls.
    """
    def resultCollector(json_message, funcId):
        log.debug('json: %s' % json_message)

    try:
        now = get_now_full_second()
        # set execution time to be in 3 seconds
        exec_time = now + timedelta(seconds=3)

        # create callback for each function call
        callback = partial(resultCollector, funcId=4711)
        # this is a non-blocking time-scheduled UPI call
        rvalue = local_mgr.runAt(UPIfunc, UPIargs, unix_time_as_tuple(exec_time),
callback, 1)
        # rvalue is empty
    except Exception as e:
        log.fatal("... An error occurred : %s" % e)

    # teardown engine
    local_mgr.stop()
```

2.3.5 Remote Execution of UPI Functions

WiSHFUL provides ***full location transparency***. Any UPI function can be executed either locally by a *local control program* or remotely by a *global control program*. In the latter case, the WiSHFUL global MCE transparently serializes (marshalling) all input and output arguments. The calling semantic for both the local and remote calls is ***call-by-value***. This has to be considered when extending the UPIs with additional functionality. Finally, as with the local execution also the execution of remote functions can be time-scheduled. This is especially important if a given UPI function needs to be executed at the same time on a set of wireless nodes.

The following example illustrates how a *global control program* remotely executes a UPI function to control the behaviour of nodes:

```
if __name__ == '__main__':

    # name of the experiment group; only nodes of this group can be controlled
    exp_group_name = "MyWishFulTest"

    # get reference to global UPI
    global_mgr = GlobalManager(exp_group_name)
    radioHelper = RadioHelper(global_mgr)

    nodes = []
    node0 = Node("192.168.103.125") # nuc2
    node1 = Node("192.168.103.134") # nuc3
    nodes.append(node0)
    nodes.append(node1)
    # node discovery: wait until all specified nodes are available
    nodes = global_mgr.waitForNodes(nodes)

    expectedNodeIps = [node.getIpAddress() for node in nodes]
    log.info('All required nodes are available ... %s' % str(expectedNodeIps))

    # start thread for callback, have to be done after some peers are available
    global_mgr.startResultCollector()

    iface = 'mon0'
    ch = 36

    try:
        # exec remote UPI on node0
        rv = radioHelper.setRfChannelRemote(node0, iface, channel)
        # exec remote UPI on node1
        rv = radioHelper.setRfChannelRemote(node1, iface, channel)
    except Exception as e:
        log.fatal("... An error occurred : %s" % e)
    # tear down engine
    global_mgr.stop()
```

To run the above global control program two steps are required. First, on each node under test we have to start the WiSHFUL agent (included in WiSHFUL framework):

```
$ python start_agent.py
```

The global controller itself can be started by calling:

```
$ python ex_global_ctrl.py
```

Note: the global controller passes control commands (UPI calls) to agents, which executes them locally.

2.3.6 Time Synchronization

A wide range of WiSHFUL applications like the centralized control of channel access requires a ***tight time synchronization*** among wireless nodes. The way the wireless nodes are time synchronized is platform and architecture-dependent. Basically, we distinguish between systems where a backbone network exists. Here in order not to harm the performance of the wireless network the nodes are time synchronized using the backbone (e.g. Ethernet) and some time protocol like PTP. Wireless nodes without a backbone have to rely on other techniques for time synchronization (e.g. GPS).

2.3.7 Packet Forgery, Sniffing and Injection

WiSHFUL provides a wide range of functionality for packet forgery, sniffing and injection. A control programs can use this to create and inject network packets into the network stack of a node or to receive copies of packets. All WiSHFUL nodes support the sniffing and injection on IP layer (layer 3).

2.3.8 Deployments of new UPI functions

WiSHFUL provides an open and extensible architecture, which can be easily extended by new UPI functions. Any new introduced UPI function can be implemented in a different way for different platform and architecture. Therefore, in WiSHFUL for each platform there is separate **network connector module**. A connector module maps the general UPI call into platform specific implementations. Since they are mainly tailored for the underlying radio platform, they are described in D3.2. The UPI_N functionality, provided by the connectors, is operating system specific (e.g. Linux, Window, Contiki, TinyOS) and are grouped with the connectors for the radio platforms.

Moreover, some WiSHFUL platforms support the deployment and execution of new UPI functions "on-the-fly" from the *global control program*.

The following example illustrates this:

```
if __name__ == '__main__':

    # name of the experiment group; only nodes of this group can be controlled
    exp_group_name = "MyWishFulTest"

    # get reference to global UPI
    global_mgr = GlobalManager(exp_group_name)
    nodes = []
    node0 = Node("192.168.103.125") # nuc2
    node1 = Node("192.168.103.134") # nuc3

    nodes.append(node0)
    nodes.append(node1)

    # node discovery: wait until all specified nodes are available
    nodes = global_mgr.waitForNodes(nodes)

    """
    Custom callback function used to receive result values from scheduled calls.
    """
    def resultCollector(json_message, funcId):
        log.debug('json: %s' % json_message)

    """
    Custom function: print out the current configured firewall rules.
    """
    def customFuncIpTables(myargs):

        import iptc
        import logging

        log = logging.getLogger()

        def printIpTable(table):
            for chain in table.chains:
                # […]

        table = iptc.Table(iptc.Table.FILTER)
        printIpTable(table)

    try:
        # deploy a custom control program on each node
        CtrlFuncImpl = customFuncIpTables
        CtrlFuncargs = (123,)

        # exec in 2s
        exec_time = now + timedelta(seconds=2)
        callback = partial(resultCollector, funcId=99)
        # remote execution of a custom function
        global_mgr.runAt(CtrlFuncImpl, CtrlFuncargs, unix_time_as_tuple(exec_time),
callback)

    except Exception as e:
        log.fatal("... An error occurred : %s" % e)

    # teardown engine
    global_mgr.stop()
```

3 Implementation of WiSHFUL architecture

Deploying global control in wireless networks is challenging but also promises rewards in terms of network performance (i.e. increased reliability and QoS). The main problem is that every additional control flow requires a certain amount of bandwidth and should thus be carefully evaluated because in real-life networks it can have a dramatic impact on the overall network performance.

To overcome this, we facilitate both initial research exploration and advanced real-life evaluation by offering two possible communication channels over which the control messages can be exchanged:

1. An out-of-band channel tailored for testbed experimentation that makes use of the testbed backbone to transfer control messages[1].
2. An in-band channel that uses the same physical channel as the normal data flows and can be used during field trials or real-life deployment.

The global MCE is executed in a Linux environment and is common for all platforms. The local MCEs are currently implemented in Linux with specific connector modules for ATH9k (Atheros cards), WMP (Broadcom B43) and Contiki (sensor nodes) using an out-of-band channel for control flows. In Contiki there is also a proof-of-concept implementation for a local MCE that uses an in-band channel.

3.1 Linux Networking Subsystem

Here we have to distinguish whether the devices under test have an additional interface to a common backbone network (Ethernet). So far we have assumed that all wireless nodes have an additional wired interface, which can be used to set-up a low-latency, high capacity out-of-band control channel (Gigabit Ethernet). Such a requirement is fulfilled in Enterprise IEEE 802.11 networks. For the future we plan to provide a solution using an in-band control channel.

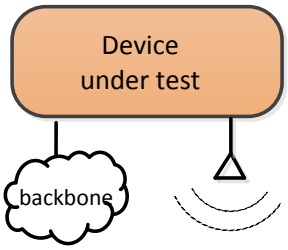

Figure 2. Linux device under test with a dedicated backbone interface.

3.1.1 Node Discovery

For WiSHFUL nodes using Linux as operating system and having a backbone network we use the ZeroMQ Realtime Exchange Protocol (ZRE) (http://rfc.zeromq.org/spec:36) which is a peer-to-peer protocol for node discovery. For the automatic node discovery it is required that all devices under test as well the global control must reside in the same LAN segment. If a wireless node has more than one network interface card (NIC), the NIC to be used by the discovery algorithm must be configured in both the global controller and the agents of the wireless nodes. Please edit the following variable in constants.py:

```
DISCOVERY_HOST_TO_INF = {
    'himalaia' : 'eth0',
    'matrix' : 'eth1'
}
```

[1] If a backbone network is available in real-life deployments, of course, the out-of-band solution can also be used for providing control to the devices under test.

where the key is the hostname and the value is the NIC to be used.

3.1.2 Controller Model

The mandatory proactive approach is implemented.

3.1.3 Time-scheduled execution of functions

Linux-based WiSHFUL nodes use the Advanced Python Scheduler, a Python library, for time scheduling of UPI functions. More information can be found here: https://apscheduler.readthedocs.org/en/latest/

3.1.4 Remote Execution of UPI Functions

The Linux-based WiSHFUL nodes use the ZeroRPC library on top of ZeroMQ as transport layer protocol. ZeroRPC provides the required functionality for serialization (marshalling) all input and output UPI arguments.

For Linux-based WiSHFUL nodes the ZeroRPC library which uses ZeroMQ as transport protocol is used. ZeroRPC provides the required functionality for serialization (marshalling) all input and output UPI arguments.

In order to evaluate the performance of the proposed controller framework, we measured execution time spent by running experiments. Specifically, we are interested in comparing the execution of local and remote UPI calls.

We measured time required to execute 1000 function calls in blocking mode. We chose four functions:

- *getHwAddr() – returns the MAC address of a particular network interface*
- *getIpTable() – return the packet netfilter entries*
- *setRfChannel() – return the radio channel used by a particular WiFi interface*
- *setMarking() – perform flow marking*

Local Control Program

For local execution of UPI functions we tested a local controller on two platforms:

- Intel-NUC-i5 - Intel-NUC mini-PC, Ubuntu 14.04 LTS, processor Intel-i5
- PC-i7 - PC, Ubuntu 14.04 LTS, processor Intel-i7.

Each test was repeated 10 times to calculate average execution time needed to call function 1000 times.

Function	Intel-NUC-i5		PC-i7	
	Avg. Execution Time (s)	Std.	Avg. Execution Time (s)	Std.
getHwAddr	0.024268	0.001697	0.024130	0.003183
getIpTable	0.488779	0.004331	0.450982	0.008743
setRfChannel	6.943339	0.161535	4.478201	0.164465
setMarking	1.837180	0.052953	1.143227	0.036594

Global Control Program

When using a global controller the UPI functions are executed remotely. Hence the performance of the backhaul network plays an important role. The remote WiSHFUL agent was tested on two different platforms:

- Intel-NUC-i5 – Two Intel-NUC mini-PCs, Ubuntu 14.04 LTS, processor Intel-i5
- PC-i7 – Two PCs, Ubuntu 14.04 LTS, processor Intel-i7.

Each test was repeated 10 times to calculate average execution time needed to call function 1000 times.

We examined influence of RTT between two nodes on time needed to execute function 1000 times. We used netem QDisc to artificially change queuing delay on egress interfaces.

Function	Two Intel-NUC-i5, Avg RTT 0.352ms		Two PC-i7, Avg RTT 0.489ms	
	Avg. Execution Time (s)	Std.	Avg. Execution Time (s)	Std.
getHwAddr	10.637399	0.6335301	5.834721	0.5159092
getIpTable	11.324886	0.4444439	6.604602	0.5237034
setRfChannel	20.221329	0.8901996	13.574112	0.6672405
setMarking	12.787135	0.5414053	7.532173	0.5102143

Function	Two Intel-NUC-i5, Avg RTT 2.334ms		Two PC-i7, Avg RTT 2.389ms	
	Avg. Execution Time (s)	Std.	Avg. Execution Time (s)	Std.
getHwAddr	18.615671	0.2515008	14.167905	0.4131221
getIpTable	19.473895	0.2331258	15.349719	0.5968861
setRfChannel	27.982911	0.3199609	23.525305	0.6204532
setMarking	20.979897	0.2283876	16.543255	0.5344376

Function	Two Intel-NUC-i5, Avg RTT 4.562ms		Two PC-i7, Avg RTT 4.326ms	
	Avg. Execution Time (s)	Std.	Avg. Execution Time (s)	Std.
getHwAddr	26.775350	0.2363079	22.852124	0.5229753
getIpTable	27.652234	0.2179322	23.767675	0.6249199
setRfChannel	36.053753	0.2788598	33.188614	0.6600535
setMarking	29.049699	0.2264008	25.075865	0.6886534

3.1.5 Time Synchronization

For WiSHFUL nodes using Linux as operating system and having a backbone network we use the IEEE 1588 Precise Time Protocol (PTP, “IEEE standard for a precision clock synchronization protocol for networked measurement and control systems,”) for time synchronization of the wireless nodes.

To measure the achieved precision of the time synchronization the WiSHFUL framework was used itself (Figure 3). Specifically, we setup an experiment, which consisted of a WiSHFUL global control

program running on a server and three wireless devices under test (802.11 Linux). The WiFi device of one WiSHFUL node was set-up to operate in monitor mode to sniff the transmitted packets by the other two nodes. The received wireless packets in the card are timestamped in µs resolution by the hardware. We fix the time interval of the transmission of layer-2 broadcast packets in the transmitter platform to 1s and measure the inter-packet time spacing as the metrics for the accuracy of the local execution time. The test was measured with 1000 packets.

The following hw/sw setup was used. For the two transmitter nodes we used Ubuntu 14.04, dual-core, x86 with a wired Ethernet NIC from Intel supporting HW timestamping. The wireless interface was an Intel WiFi 5300 802.11n chip. For the sniffing node we used a TP link router (MIPS), OpenWRT and using an Atheros 802.11n chipset.

The experiment set-up is shown Figure 3 whereas Figure 4 shows the achieved timing accuracy of the scheduled transmission. We can see that the maximum timing error is bounded to microseconds level and is not higher than 40µs with a median value of 11µs.

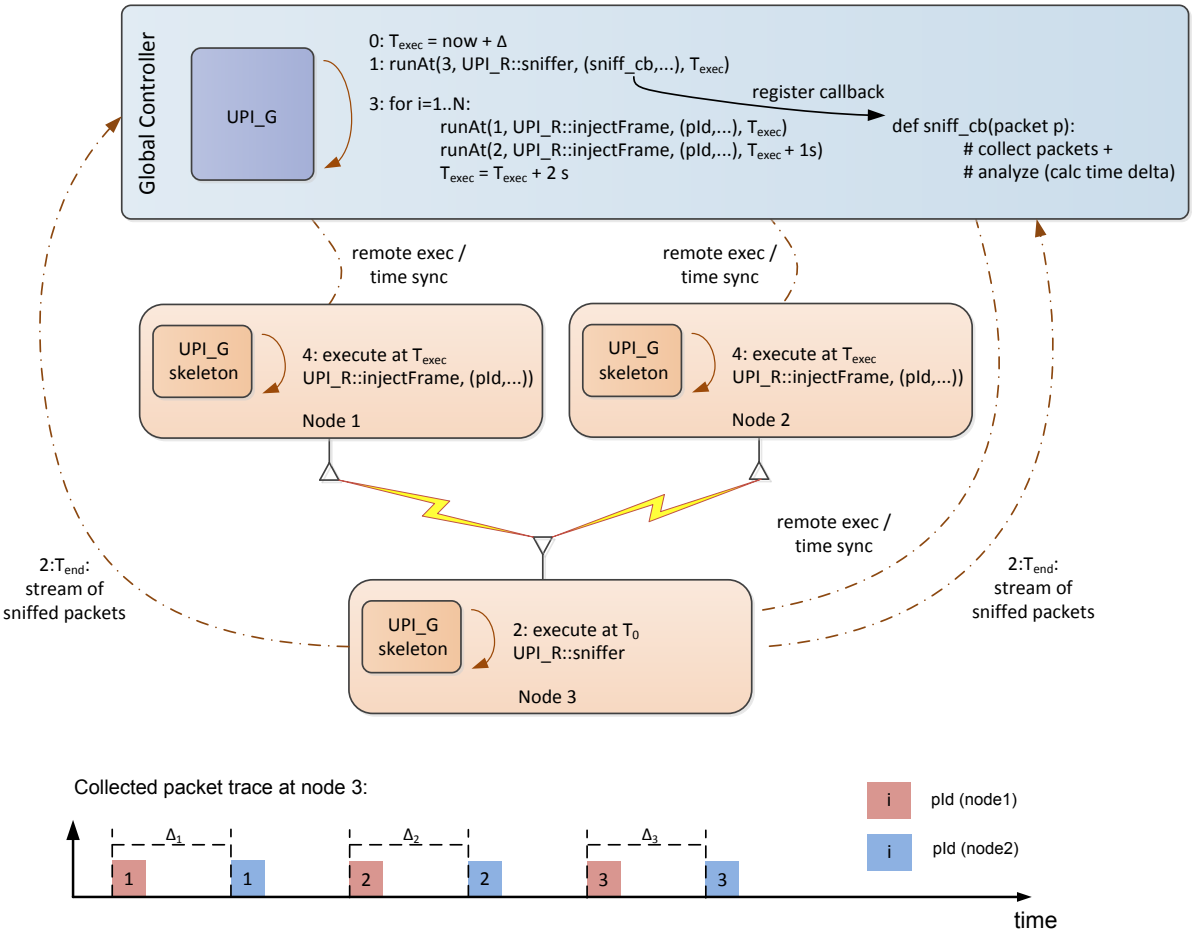

Figure 3. WiSHFUL experiment control setup to measure the precision of the time synchronization between nodes.

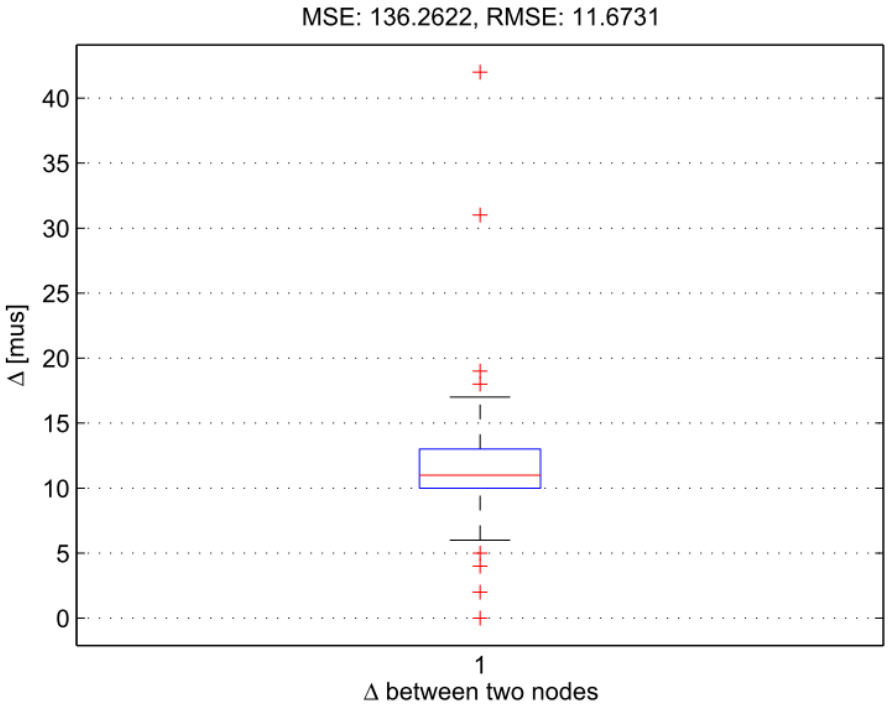

Figure 4. Achieved time precision.

3.1.6 Packet Forgery, Sniffing and Injection

For the Linux implementation we use the Python Scapy library for packet forgery, injection and sniffing. Besides sniffing and injection on layer 3 we support also layer-2 operation (e.g. 802.11 frames). For more details please take a closer look at the implementation of the following two UPI_R functions:

```
# send out 802.11 broadcast link probes
IEEE80211_L2_GEN_LINK_PROBING = "IEEE80211_L2_GEN_LINK_PROBING"
# receive 802.11 broadcast link probes
IEEE80211_L2_SNIFF_LINK_PROBING = "IEEE80211_L2_SNIFF_LINK_PROBING"
```

3.1.7 Deployments of new UPI functions

Linux-based WiSHFUL nodes support the deployment and execution of new UPI functions "on-the-fly" from the global controller.

3.2 Contiki Embedded OS

Deploying global control in constrained networks such as WSNs is challenging but also promises rewards in terms of energy usage (i.e. lifetime) and network performance (i.e. increased reliability and QoS). The main problem is that every additional control flow requires already limited resources (e.g. energy, bandwidth) and should thus be carefully evaluated. In research focusing on constrained devices, this puts an extra burden on the experimenter. On the other hand, in real-life networks it can have a dramatic impact on the lifetime and availability of the network.

By implementing an out-of-band and an in-band control channel, both initial research exploration and advanced real-life evaluation can be supported.

1. Out-of-band control is achieved by running a local MCE on the Linux Host PC of each sensor node and one on the sensor itself. The local MCE in Linux provides a connector module specific for Contiki that uses serial communication to relay UPI flows to the local MCE on the sensor. The local control program is executed on the Linux host in this case.
2. In-band control is achieved by running a local MCE on each sensor that uses CoAP for enabling interactions with a global MCE. It must however share the same physical channel with the normal data flows. The local control program is executed on the Contiki sensor node in this case. All sensors are decoupled except one border router that acts as a gateway for the WSN.

Both cases however make use of the same implementation of the local MCE on the sensor node provided by the GITAR framework. GITAR (Generic extensions for Internet-of-Things Architecture) extends the Contiki OS with advanced control and management capabilities such as on-the-fly software upgrades, protocol reconfiguration and monitoring. As depicted on right side of Figure 5, GITAR offers generic cross-layer monitoring, configuration and management services. The left side of the figure illustrates the typical IPv6 protocol stack, as used in WSNs.

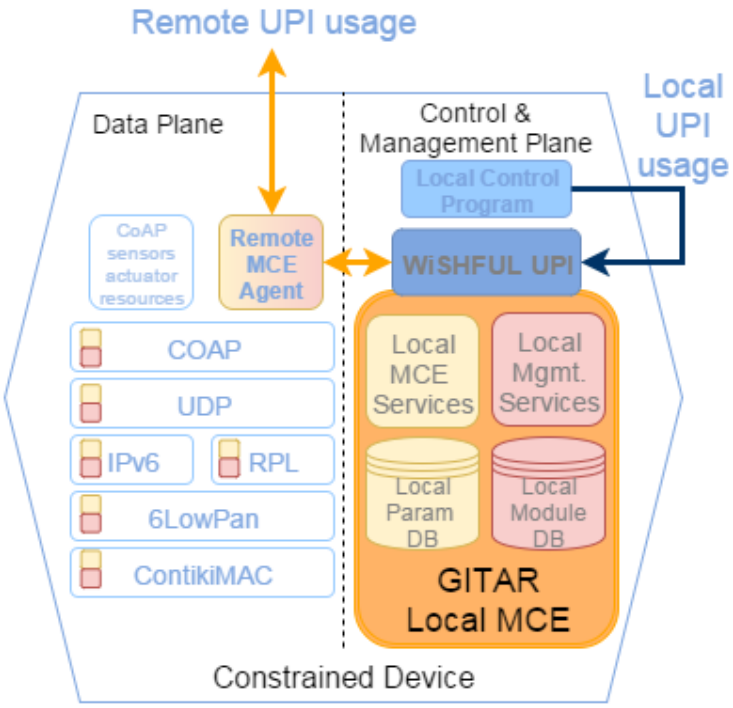

Figure 5 GITAR local MCE for use on Contiki sensor nodes.

The GITAR services can be used by protocol developers to make their protocols upgradeable, reconfigurable and observable with only minor modifications required to the protocols itself. On the other hand the same generic services can be used to implement the WiSHFUL UPIs. These can then be used by both local control programs and Remote MCE agents, enabling both local and remote UPI usage.

The GITAR services heavily rely on two databases tailored for sensor nodes with limited memory. The local parameter DB stores references to configuration settings and monitoring values. It also allows publishing of and subscribing to monitoring events. The local module DB allows to dynamically link new or updated software modules.

3.2.1 Controlling Contiki sensor nodes over Ethernet

In the former case, a local MCE is executed on the Linux host PC and all configuration and monitoring flows are supported as described in Section 3.1 and depicted in Figure 6. A connector module for Contiki transforms UPI requests into configuration commands that are injected on the node over serial. The serial MCE agent uses the UPI interfaces provides by GITAR to invoke the local UPI functions.

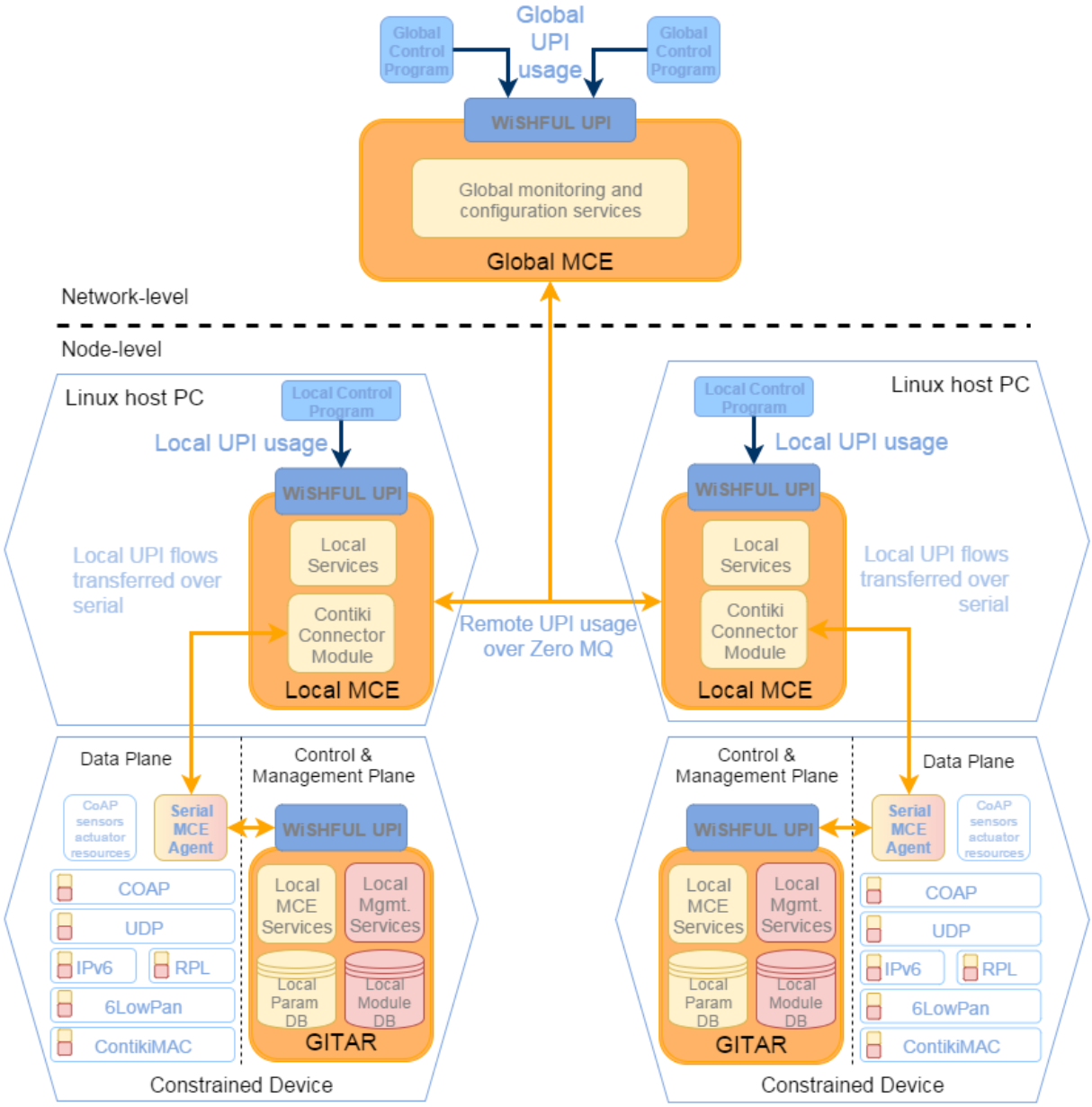

Figure 6 Implementation overview of the WiSHFUL architecture applied on Contiki sensors.

a. Node Discovery

The local MCEs in Linux use the ZeroMQ Realtime Exchange Protocol (ZRE) (http://rfc.zeromq.org/spec:36) as discussed in Section 3.1. The sensor nodes are automatically discovered by adding a reference to an extra 'wpan0' network interface card in constants.py:

```
DISCOVERY_HOST_TO_INF = {
        'himalaia' : 'eth0',
        'matrix' : 'eth1',
        'sensorX' : 'wpan0'
```

```
}
```

, where the key is a unique hostname for the sensor and the value ‘wpan0’.

b. Controller Model

The mandatory proactive approach is implemented. For this purpose the Contiki connector module must emulate the blocking behaviour of the UPI functions on top of the asynchronous serial interface. This is done by using a request response mechanism that must be fully completed before returning the UPI function calls.

c. Time-scheduled execution of functions

The same solution is used as described in Section 3.1.

d. Remote Execution of UPI Functions

As discussed in Section 3.1. ZeroRPC on top of ZeroMQ is used to allow remote execution of UPI functions.

e. Time Synchronization

The sensor nodes rely on the synchronization of their respective Linux host PCs (see Section 3.1).

f. Packet Forgery, Sniffing and Injection

Only packet injection is supported by activating/de-activating packet generator applications that can operate on L2 (MAC) and above.

g. Deployments of new UPI functions

Deployment of UPI functions is foreseen in Year 2 and will use the management extensions provided by GITAR [1].

3.2.2 Controlling Contiki sensor nodes over IEEE-802.15.4

In the decoupled scenario, without Ethernet backbone, the local MCE and control program are both executed on the Contiki sensor device. As depicted in Figure 7, the global MCE still resides on a Linux machine. Via the GITAR connector module, it can remotely configure, monitor and manage the protocol stack of each sensor node. Remote access to the nodes is currently implemented using the CoAP application layer protocol. The CoAP implementation consists of a control and management proxy for each separate sensor network and a COAP server acting as a MCE agent for each sensor node inside the WSN.

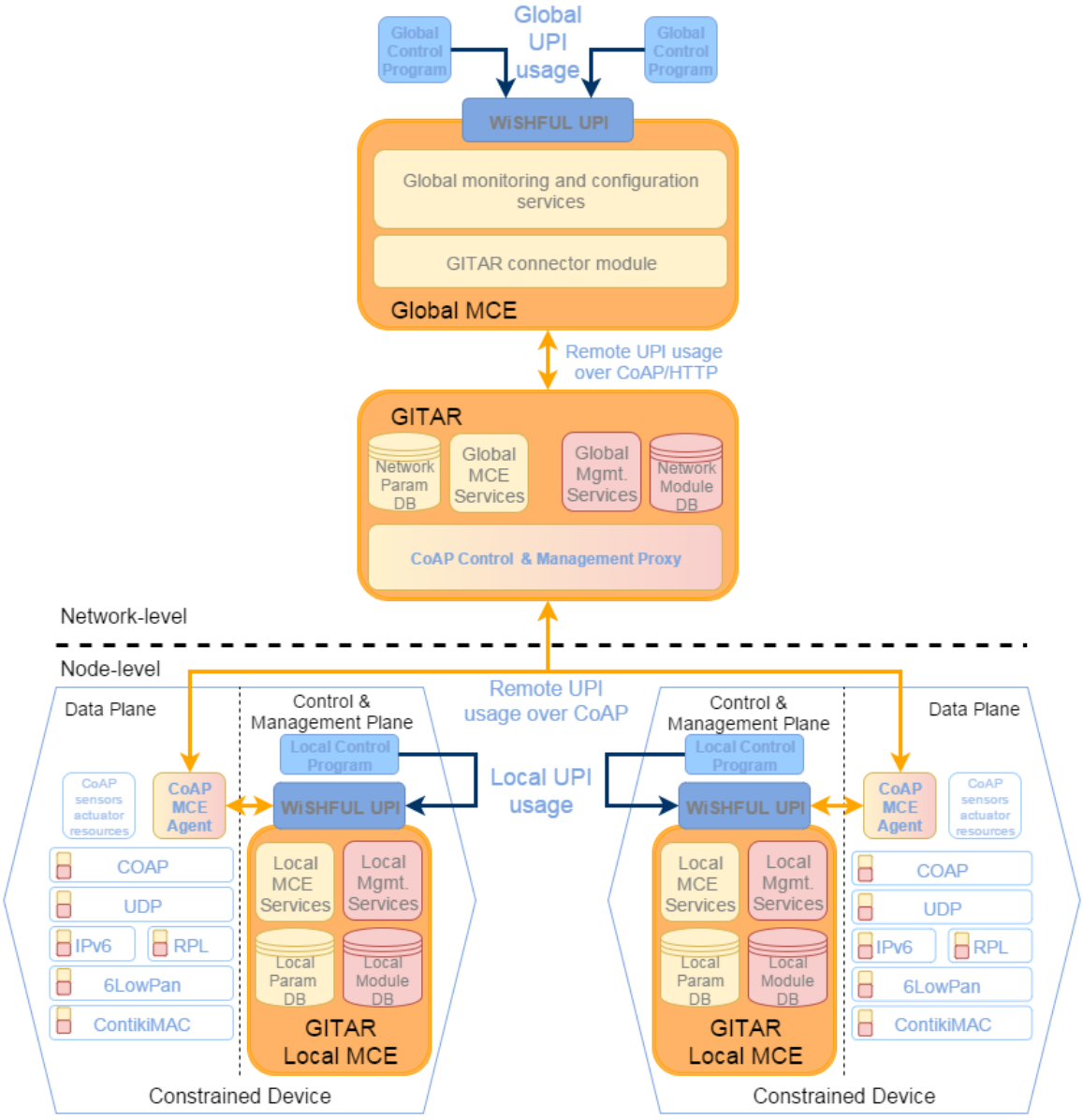

Figure 7 Implementation of the WiSHFUL architecture on Contiki sensor devices in the decoupled scenario.

The Control and Management proxy enables to implement global monitoring, configuration and management services by maintaining a network-wide view of the current settings and state in the network parameter DB, and by offering a central software repository. The proxy also transforms UPI requests into a compact format to minimize the overhead in the WSN. For this purpose it can also use the network parameter DB as a cache. Global MCE services such as synchronised execution and node discovery can also be added on top of the proxy.

The CoAP MCE agent enables remote UPI usage over CoAP by implementing a CoAP server. Currently only getting and setting of parameter is supported, enabling already the basic UPI interactions. For this purpose, the CoAP MCE agent makes configuration parameters discoverable and implements a generic interface that allows get/set a specific parameter or a group of parameters. To enable all these interactions, protocols must store references to its parameters in the local parameter DB as discussed in Section 3.2. More advanced monitoring and management interactions will be included in Year 2.

Local UPI usage by local control programs uses the same UPI interface as the CoAP MCE agent. The location of the UPI function caller is thus transparent the local MCE.

a. Node Discovery

The CoAP proxy creates a single container resource that makes the entire network discoverable. The network resource contains a sub resource for each node in the network. The node resource also contains sub resources for the required remote services (configuration, monitoring, management, synchronization, and discovery). Nodes are discovered by regularly sending GET request to the CoAP discovery service, which is an observable resource.

b. Controller Model

The mandatory proactive approach is implemented for local UPI usage. For remote UPI usage, only a reactive model can be use by the global MCE due to the high communication delays. The global control program must thus register callback functions when invoking global UPI functions.

c. Time-scheduled execution of functions

Time-scheduled execution of functions will be implemented in Year 2 and depends on the synchronisation services discussed in a subsequent section.

d. Remote Execution of UPI Functions

As discussed in the previous sections CoAP is used to allow remote execution of UPI functions. Currently only get/set parameter is supported. The system can be easily extended to support more UPI functions in Year 2.

e. Time Synchronization

Embedded sensor network nodes have in general no backbone network. This complicates the task of time synchronization significantly as protocols like PTP cannot be executed. Moreover, it is not possible to synchronize using external clocks (e.g. GPS) as those devices are resource constraint.
On the other hand, sensor nodes often synchronize own network clock with its neighbours over the air.
Tight time synchronization between nodes is especially important in TSCH network, due to its slotted nature. “TSCH adds timing information in all packets that are exchanged. This means that neighbour nodes can resynchronize to one another whenever they exchange data” [2]. There are two methods of synchronization in IEEE 802.15.4e: acknowledgement- and frame-based. In both cases the difference between expected and actual time of arrival times is calculated. Detailed synchronization policies are not defined by the standard and leave room for adaptation depending on the network requirements. Also assignment of adequate “time source neighbour” is left open as part of network scheduling.

f. Packet Forgery, Sniffing and Injection

Only packet injection is supported by activating/de-activating packet generator applications that can operate on L2 (MAC) and above. For this purpose, the applications need to register a configuration parameter that allows this.

g. Deployments of new UPI functions

Deployment of UPI functions is foreseen in Year 2 and will use the management extensions provided by GITAR [1].

4 UPI_N implementation

This section introduces ***UPI_N***, its functions and how the implementation is done for each software platform. Note, the UPI_N covers the reconfiguration of the higher layers of the network protocol stack of a particular wireless node whereas the UPI for radio control, UPI_R, which is described in D3.2 allows the adaptation of the lower layers, i.e. lower MAC and physical layer. This section also includes performance evaluation of each implementation.

Beside the possibility to use the ***low-level API*** there is the possibility to use a ***high-level object-orient API*** which is provided by *NetworkHelper* (UPI_N) and *RadioHelper* (UPI_R) classes. In the following we show how the implemented UPI_N functionality can be utilized using those two helper classes.

4.1 Linux Networking subsystem

An example showing a pure local control program in Linux which is using the UPI_N interface to program flow marking is shown below:

```
if __name__ == '__main__':
    # get reference to local wishful engine
    local_mgr = LocalManager()
    # use helper for easier UPI use
    netHelper = NetworkHelper(local_mgr)
    try:
        log.info('Set Marking and TOS for flow between nodes')
        flowDesc = FlowDesc(src='1.2.3.2', dst='1.2.3.3', prot='tcp', srcPort='21',
dstPort="423")
        netHelper.setMarking(flowDesc, markId=5)
        netHelper.setMarking(flowDesc, markId=55, table="mangle", chain="INPUT")
        netHelper.setTosRule(flowDesc, tos=23)
    except Exception as e:
        log.fatal("... An error occurred : %s" % e)
    # teardown engine
    local_mgr.stop()
```

The above example local controller can be executed by calling:

```
$ python upi_n_ex.py
```

4.1.1 Traffic control and monitoring

In this section we provide an overview of UPI_N functions for configuration of traffic control. We implemented functions for managing Queueing Disciplines. Using provided UPIs an experimenter is able to apply traffic shaping and prioritize flows. Moreover, we provided a UPIs functions for link emulation, that an experimenter can use to emulate wireless links (in terms of throughput, delay, etc.) in wired networks.

a. Support for management of Queueing Disciplines

The provided UPI_N for configuration of queueing disciplines follows object-oriented approach. It gives an experimented a user-friendly way for managing the QDisc for each interface in SUT nodes.

An example of configuration of QDisc is presented in Table 1. First, a root scheduler has to be created. Second, queues are created and added to shaper. In our design available in D4.1, we described that Shaper objects will be attached to scheduler. However, we decided to change naming of this class from Shaper to Queue, because not every queue is shaping throughput, but every shaper is a queue. Third, filters are created and attached to scheduler. Finally, the *installEgressScheduler()* function is called to send QDisc configuration to SUT node, which will apply it on specified interface. In

Table 2, the usage of UPI function for deletion of egress scheduler is presented.

We implemented Python package, called *python-tc* that is used in: *i)* controller to create QDisc configuration in object-oriented way; *ii)* agent to install this configuration on specified interface. Currently we support most of schedulers and queues available in Linux Traffic-Control subsystem, namely: *pfifo, bfifo, pfifo_fast, tbf, sfq, netem, prio, htb.*

Multiple schedulers can be chained together, what gives an easy way for creation of even very complex queueing disciplines. Traffic control can be performed on both the ingress and egress interfaces. In the current version, counters are not supported; we will implement them during Year 2.

Filters are testing packed according to set so called 5-tuple, which is a set of: source address, destination address, protocol, source port and destination port. A packet is tested against all filters in order they were created until it matches some of them.

A Qdisc configuration is installed using Netlink calls to the kernel traffic-control subsystem, thus we are not dependent on any third-party tool (e.g tc).

As mentioned earlier using provided UPIs an experimenter is able to control the packet queuing polices in NET layer of protocol stack. It has to be noted, that we also provide a way of controlling the packet handling in the lower MAC layer, where packets are grouped based on Type-of-Service value. More information is provided in Section 0.

Table 1 Example of configuration and installing queuing disciplines in SUT node

```
#Define scheduler
prioSched = PrioScheduler(bandNum=4)

#Define queues that will be added to scheduler
pfifo1 = prioSched.addQueue(PfifoQueue(limit=50))
bfifo2 = prioSched.addQueue(BfifoQueue(limit=20000))
pfifo3 = prioSched.addQueue(SfqQueue(perturb=11))
tbf4   = prioSched.addQueue(TbfQueue(rate=1000*1024, burst=1600,
                            limit=10*1024))

#Define filters
filter1 = Filter(name="BnControlTraffic");
filter1.setFiveTuple(src=None, dst='192.168.1.178', prot='udp', srcPort=None,
                     dstPort='5001')
filter1.setTarget(pfifo1)
prioSched.addFilter(filter1)
…
…
…
filter4 = Filter(name="BestEffort");
filter4.setFiveTuple(src='10.0.0.2', dst=None, prot='tcp', srcPort='21',
                     dstPort=None)
filter4.setTarget(tbf4)
prioSched.addFilter(filter4)

#Install defined scheduler in node0
netHelper.installEgressScheduler(node0, 'wlan0', prioSched)
```

Table 2 Deletion of egress scheduler in SUT node

```
#Delete scheduler in particular interface of node
netHelper.removeEgressScheduler(node0, 'wlan0')
```

b. Emulation

We provide an experimenter a way to emulate link parameters in wired network. We envision that this functionality will be helpful for testing control programs by emulating the wireless links (of course with some limitations) in wired.

An experimenter is able define parameters of wireless network (throughput, delay, jitter, packet loss, etc.) and apply it to each SUT node using implemented UPI functions. Moreover, we introduce *Profile* abstraction to further facilitate link emulation configuration. A profile is description of link characteristics.

In Table 3, we present an example of configuration of link profile and usage of *setProfile()* to apply it to specified interface in SUT node. In

Table 4, we show how to update already existing profile using *updateProfile()*. Finally, in

Table 5, it is shown how to remove profile using ***removeProfile()*** function.

We extended our *python-tc* package with *Profile* class and implemented mentioned UPI functions to install, update and remove profile in specified interface of SUT node.

In order to emulate link characteristics in wired network, we use combination of *Netem and Token Bucket Filter (TBF) Queuing Disciplines* available in Linux kernel. *Netem* is an enhancement of the Linux traffic control facilities that allow to add delay, packet loss, duplication and more other characteristics to packets outgoing from a selected network interface. *Token Bucket First* is responsible for shaping the throughput of traffic passing interface.

Table 3 Example of configuration of link emulation

```
#Define emulation profile
profile4G = Profile("profile3G")
profile4G.setPacketLimit(1000)
band_1Mbps = 1000 * 1000 / 8
profile4G.setRate(band_1Mbps)
profile4G.setDelay(delay=100, jitter=10)

#Apply emulation profile to interface eth0 of node0
netHelper.setProfile(node0, 'eth0', profile4G)
```

Table 4 Example of update of emulation profile

```
#Update emulation profile
band_3Mbps = 3 * 1000 * 1000 / 8
profile4G.setRate(band_3Mbps)
profile4G.setDelay(delay=70, jitter=5)

#Update emulation profile in node0
netHelper.updateProfile(node0, 'eth0', profile4G)
```

Table 5 Deletion of emulation profile

```
#Remove emulation profile from interface eth0 of node0
netHelper.removeProfile(node0, 'eth0')
```

4.1.2 Packet filtering, manipulation and monitoring

In this section we provide an overview of UPI_N function for packet filtering and manipulation. In current version, we provide an object-oriented approach for manipulation of *iptables*, packet marking and setting Type-of-Service value.

In Table 6, an example of usage of implemented UPI function to mark flows is presented. First, an experimenter has to create ***FlowDesc*** object that describe flow in terms of 5-tuple. Second using function ***setMarking()***, he is able to install new rule in *iptables* of specified SUT node. It is possible to specify the table and chain where new rule is to be installed. If they are not provided the default values are used*: table = 'mangle'* and *chain = 'POSTROUTING'*. An experimenter can specify also mark value that will be assigned to packets. If not provided, our framework will take care of generation of unique mark value. In order to delete rule from *iptables*, one needs to use ***delMarking()*** function.

Table 6 Example of configuration of flow marking

```
#Define 5-tuple that identifies flow
flowDesc = FlowDesc(src='192.168.1.1', dst='192.168.1.12', prot='tcp',
            srcPort=None, dstPort='21')

#Install iptables rule in node to mark packets of defined flow;
netHelper.setMarking(node0,flowDesc, markId=5, table="mangle", chain="INPUT")

#If table and chain are not provided, default values are used: table="mangle",
chain="POSTROUTING"
netHelper.setMarking(node0,flowDesc, markId=5)

#If mark value is not provided, unique value is generated automatically
netHelper.setMarking(node0,flowDesc)

#Delete rule used for marking flow
netHelper.delMarking(node0, flowDesc)
```

In Table 7, an example of usage of UPI function to configure setting TOS value is presented. Function ***setTosRule()*** is used to create new rule in iptables. Arguments table and chain can be skipped, and default (table="mangle" and chain="OUTPUT") values will be used in such case. To remove inserted rule, one should call ***selTosRule()*** function.

Table 7 Example of configuration of setting Type-of-Service value in packet header

```
#Install iptables rule in node to set TOS value in packet header of defined
flow
netHelper.setTosRule(node0, flowDesc, tos=25, table="mangle", chain="OUTPUT")

#Delete rule used for setting TOS value in packet header
netHelper.delTosRule(node0, flowDesc)
```

In order to get current *iptable* from SUT node, an experimenter should use *getIpTable()* - Table 8 - function and pass table name as argument. It returns an object, which contain all rules of specified

table. Helper function ***printIpTable()*** can be used to display retrieved table description in properly formatted way. Finally, we provide the UPI function ***clearIpTables()***, that clear all rule entries of specified table and chain. In order to clear all chains in single call, one should call this function with *chain="ALL"* argument. To clear all tables, *table="ALL"* and *chain="ALL"* arguments has to be passed.

Table 8 Example usage of iptables related UPI_N functions

```
#Get and print specified iptable from node
table = netHelper.getIpTable(node0, table="mangle")
netHelper.printIpTable(table)

#Clear specified iptable and chain in node
netHelper.clearIpTables(node0, table="mangle", chain="ALL")

#Clear all iptables in node
netHelper.clearIpTables(node0, table="ALL", chain="ALL")
```

In our implementation we used *python-iptables* package, an object-oriented library that provides wrapper via python bindings to *iptables,* in Linux operating systems. The advantage of this library is that is does not call *iptables* binary and parses its output, but is uses directly the C-based libraries *(libiptc, libxtables).*

4.1.3 Monitoring of link parameter

The UPI_G function for monitoring the available throughput and delay between two nodes is presented in Table 9. Using this function an experimenter has an easy way to get following link parameters: throughput seen on by sides (transmitter and receiver); average, minimal and maximal round trip time, packet loss.

Current implementation of ***getLinkParameters()*** is executed in two steps. First, it uses *iperf* applications to measure throughput and, second, *ping* application to get round trip time. This function is a part of UPI_G, because it executes different task on nodes in synchronized manner, i.e. it has to start *iperf* in server mode on one node and in client mode on other one.

The function works as intended, unfortunately it has to saturate the link between two nodes to measure the throughput. As a future work, we will re-implement this function with one link estimation algorithm, that allows measuring link parameters without huge overhead, i.e. without saturating it.

Table 9 The UPI_N function to get link performance parameters between two nodes

```
#Get parameters of link between two nodes
linkParams = netHelper.getLinkParameters(node0, node1)
```

4.2 Contiki Embedded OS

Contiki is an operating systems specifically designed for constrained sensor devices in the IoT era. It supports a lightweight IPv6 compliant network stack as illustrated on the left side (a) in Figure 8. The following protocols and standards are included: an IEEE-802.15.4-2006 compliant PHY and MAC, 6LowPan header compression, IPv6 (addressing, headers and ICMP), RPL routing, TCP/UDP transport and CoAP. Although these standards provide many options to configure network protocols, it is hard to change the configuration settings at run-time. The candidate configuration settings for the RPL routing protocol in Contiki are illustrated, as an example, on the right side (b) of Figure 8.

To configure the parameter of the entire network stack, it suffices to implement the *getParameterHigherLayer* and *setParameterHigherLayer* functions. A pull based monitoring service can be implemented using the *getParameterHigherLayer* function. When the local MCE is running on the Linux host PC (i.e. in case of an Ethernet backbone), a push based monitoring service can also be provided. In the next versions a broader range of functions will be supported.

In the remainder of this section, first a brief background on constrained devices is given. Then the candidate configuration parameter for RPL and CoAP, as defined in their respective standards, are explored. Then an example is given of how a particular parameter can be changed by a local control program.

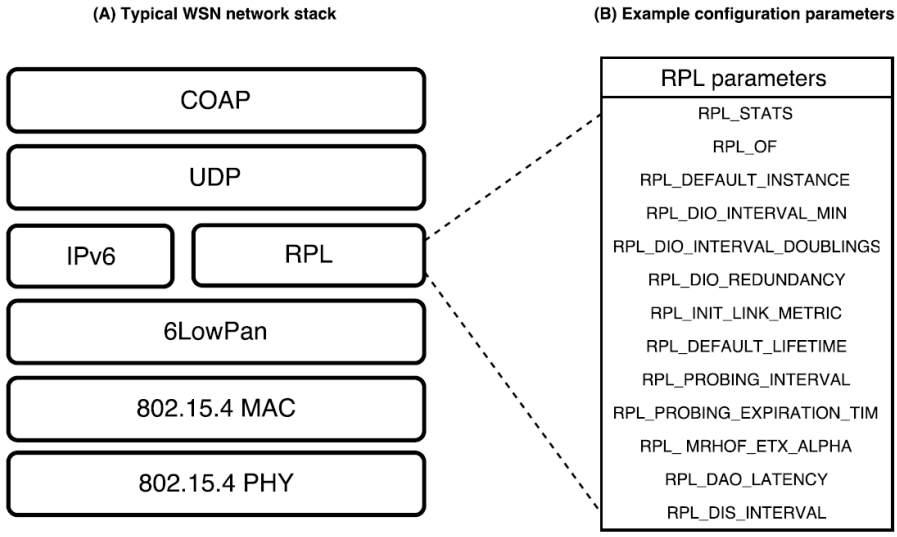

Figure 8 (A) IPv6 compliant network stack available in operating systems for wireless sensor networks such as TinyOS and Contiki. (B) Candidate configuration parameters to fine-tune the RPL routing protocol in the Contiki RPL implementation.

4.2.1 Background on network stacks for constrained devices

a. IPv6 compatible protocols and standards

This section briefly summarizes the main configuration settings in the standard IPv6 stack [3] for constrained devices.

The network layer includes RPL [4], a proactive, distance-vector routing protocol specifically designed for wireless sensor networks (WSNs). RPL uses control packets (DODAG Information Object(DIO), Destination Advertisement Object(DAO) and DODAG Information Solicitation(DIS)) for building a tree like topology, called a Destination-Oriented Directed Acyclic Graph (DODAG). Many settings allow fine-tuning the various intervals that are used for maintaining the DODAG. Also the link estimation algorithms can be changed and configured. Next to RPL, various parameters controlling the IPv6 neighbour discovery [5] process can be configured. Also TCP/UDP implementations allow configuring the number of retransmissions and various time-out settings.

The application layer protocols tailored for WSNs focus mainly on integrating the sensing and actuating applications in the IoT. One of the most prominent examples is CoAP [6], a REST based protocol that runs over UDP and allows defining resources (e.g. sensors and/or actuators) which can be retrieved or changed using GET/POST/PUT methods using a response-request approach. CoAP can be easily integrated in web-based applications and has a limited overhead. The number of retransmissions and various time-outs and intervals used by the CoAP engine can be configured.

Alternatives [7] for CoAP are MQTT and AMQP, both run over TCP and implement a publisher-subscriber approach managed by a message broker that allows nodes to publish and/or subscribe to topics. Compared to COAP they have a higher overhead and are less supported by operating systems for WSNs.

b. Configuration variables defined for RPL and CoAP (IETF standards)

The parameters currently supported are mainly focusing on CoAP [8] and RPL [9]. For this purpose the candidate configuration settings were explored by carefully examining the standards. In principal, each of these parameters can be reconfigured via the WiSHFUL UPIs if support for this is provided in the protocol implementation. This can be done by making use of the GITAR extensions discussed in Section 4.2.3.

RPL

BASE_RANK	The Rank for a virtual root that might be used to coordinate multiple roots. BASE_RANK has a value of 0.
ROOT_RANK	The Rank for a DODAG root. ROOT_RANK has a value of MinHopRankIncrease (as advertised by the DODAG root), such that DAGRank(ROOT_RANK) is 1.
RPL_DEFAULT_INSTANCE	The RPLInstanceID that is used by this protocol by a node without any overriding policy. RPL_DEFAULT_INSTANCE has a value of 0.
PATH_CONTROL_SIZE	The value used to configure PCS in the DODAG Configuration option, which dictates the number of significant bits in the Path Control field of the Transit Information option. The default value is 0. This configures the simplest case limiting the fan-out to 1 and limiting a node to send a DAO message to only one parent.
DIO_INTERVAL_MIN	The value used to configure lmin for the DIO Trickle timer. The default value is 3. This configuration results in Imin of 8 ms.
DIO_INTERVAL_DOUBLINGS	The value used to configure Imax for the DIO Trickle timer. The default value is 20. This configuration results in a maximum interval of 2.33 hours.
DIO_REDUNDANCY_CONSTANT	The value used to configure k for the DIO Trickle timer. The default value is 10. This configuration is a conservative value for Trickle suppression mechanism.
MIN_HOP_RANK_INCREASE	The value of MinHopRankIncrease. The default value is 256. This configuration results in an 8-bit wide integer part of Rank.
DAO_DELAY	The value for the DelayDAO Timer. The default value is 1 second.
DIO Timer	One instance per DODAG of which a node is a member. Expiry triggers DIO message transmission. A Trickle timer with variable interval in [0,DIOIntervalMin * 2^DIOIntervalDoublings].
DAG Version Increment Timer	Up to one instance per DODAG of which the node is acting as DODAG root. Expiry triggers increment of DODAGVersionNumber, causing a new series of updated DIO message to be sent. Interval should be chosen appropriate to propagation time of DODAG and as appropriate to application requirements (e.g., response time versus overhead).

DelayDAO Timer	Up to one timer per DAO parent (the subset of DODAG parents chosen to receive destination advertisements) per DODAG. Expiry triggers sending of DAO message to the DAOparent.
RemoveTimer	Up to one timer per DAO entry per neighbor (i.e., those neighbors that have given DAO messages to this node as a DODAG parent). Expiry may trigger No-Path advertisements or immediately deallocate the DAO entry if there are no DAO parents.

COAP

ACK_TIMEOUT	Timeout for CoAP ACK (2 seconds)
ACK_RANDOM_FACTOR	Randomness factor to overcome synchronization effects (1.5)
MAX_RETRANSMIT	Maximum number of CoAP request retransmissions (4)
NSTART	Maximum simultaneous connections between CoAP clients and servers (1)
DEFAULT_LEISURE	Leisure time before responding to a multicast requests (5 seconds)
PROBING_RATE	Rate in which probes can be send for reacting to unacked CoAP requests (1 byte/second)

4.2.2 Local UPI_N code example

The following code snippets illustrate how the **UPI_N:***setParameterHigherLayer* function can be used by a local control program.

The first code snippet illustrates how UPI_N functions can be used by a local control program written in Python running on a Linux host:

```
#Python code example for local control program on Linux using generic UPI
""" Change a the DIO interval in RPL """
local_mgr = LocalManager()
iface = {'WPAN0','CONTIKI'}
param_list = {{'DIO_INTERVAL_MIN',1000}};
UPIargs = {'iface' : iface, 'param_list' : param_list}
UPIfunc = UPI_N.setParameterHigherLayer
try:
       #this is a blocking UPI call
       error = local_mgr.runAt(UPIfunc, UPIargs)
       log.debug('Ret value of blocking call is %s' % str(error))
except Exception as e:
       log.fatal("... An error occurred : %s" % e)

#Python code example for local control program on Linux using UPI helper classes
""" Change a the DIO interval in RPL """
local_mgr = LocalManager()
UPI_N = HelperUPI_N()
iface = {'WPAN0','CONTIKI'}
param_list = {{'DIO_INTERVAL_MIN',1000}}
error = UPI_N.setParameterHigherLayer(iface,param_list)[0]
```

The second code snippet illustrates how the UPI_N functions can be used by a local control program written in C running on Contiki:

```
#C code example for local control program on sensor using generic UPI
/* Change a the DIO interval in RPL */
LocalManager* local_mgr = get_local_manager();
NIC_t iface = {WPAN0,CONTIKI};
param_list_t params = {{DIO_INTERVAL_MIN,1000}};
args_t args = {{IFACE,&iface},{PARAM_LIST,&params}};
func_t func_id = UPI_N_SET_PARAMETER_HIGHER_LAYER;
error_t result = *((error_t*) local_mgr->runAt(&func_id,&args,0,0));

#C code example for local control program on sensor using UPI helper function
/* Change a the DIO interval in RPL */
NIC_t iface = {WPAN0,CONTIKI};
param_list_t &params= {{DIO_INTERVAL_MIN,1000}};
error_t result = *(UPI_N_setParameterHigherLayer(&iface, &params));
```

4.2.3 GITAR extensions required for enabling UPI functions

a. Cross layer configuration and monitoring service

The cross-layer configuration service is responsible for implementing the UPI functions as well as facilitating protocols to register configuration parameters that can be tuned and publishing monitoring events.

In the following code snippet, the definition of the local UPI_N interface is given for use in ContikiOS. As stated, monitoring values can be obtained via the *getParameterHigherLayer* function.

```
//C header definition of supported UPI helper functions
typedef event_cb_t void (event_cb*)(void* args, int num_args); // event callback
function
param_t* UPI_N_getParameterHigherLayer(NIC_t* iface,param_key_list_t* params);
error_t* UPI_N_setParameterHigherLayer(NIC_t* iface, param_list_t* params);
error_t UPI_N_subscribeEventHigherLayer(NIC_t* iface, event_t events, int
num_events);

//C example header definition of supported generic UPI functions
param_t* UPI_N_getParameterHigherLayer(void* args, int num_args);
void* UPI_N_setParameterHigherLayer(void* args, int num_args);
void* UPI_N_subscribeEventHigherLayer(void* args, int num_args);
```

In the following code snippet, the definition of the local UPI_N interface is given for use in Python on the Linux host in case the local sensor MCE is running in Linux.

```
#Python interface definition of supported UPI helper functions
class UPI_N_HELPER():
       def setParameterHigherLayer(iface,param_key_value):
       def setParameterHigherLayer(iface,param_key):
       def subscribeEventHigherLayer(iface,event_keys, event_cbs):

#Python interface definition of generic UPI functions
class UPI_N():
       def setParameterHigherLayer(args):
       def getParameterHigherLayer(args):
       def subscribeEventHigherLayer(args):
```

To enable protocols to expose their configuration parameters and monitoring capabilities, the local MCE service provides a generic internal interface to add (a set of) parameters to the local parameter DB. The following code snippet shows the C definition of the interface.

```
//C example header definition enabling access to the parameter DB
error_t paramDB_add_parameter(param_t* param);
error_t paramDB_add_parameter_set(param_set_t* param);
error_t paramDB_add_event(event_t* event);
error_t paramDB_add_event_set(event_set_t* events);
```

b. Protocol extensions

Only minor extensions are required in the protocols to make use of the local MCE service. They need to define a *param_t* structure for every parameter they want to add. Also a group of parameters can be added in a bunch by grouping them in a *param_set_t* structure. The following code snippet illustrates this for the RPL parameters *DIO_INTERVAL_MIN* and *DIO_INTERVAL_DOUBLINGS*. They are both separately added with *paramDB_add_parameter* and added as a bunch with *paramDB_add_parameter_set*.

```
typedef struct param {
	uint16_t uid;
	uint8_t type;
	uint8_t len;
	void* (* get)(struct param* p);
	error_t (* set)(void* value, struct param* p );
} param_t;

typedef struct param_set {
	uint16_t uid;
	uint8_t num_param;
	param_t params[];
} param_set_t;

void* getRPLParameter(struct param* p){
	if(p->uid == RPL_DIO_INTERVAL_MIN)
		return rpl->dio_interval_min;
	else if(p->uid == RPL_DIO_INTERVAL_DOUBLINGS)
		return rpl->dio_interval_doublings;
	return NULL;
}

error_t setRPLParameter(void* value, struct param* p){
	if(p->uid == RPL_DIO_INTERVAL_MIN){
		rpl->dio_interval_min = *((uint16_t*) value);
		return SUCCESS;
	}
	else if(p->uid == RPL_DIO_INTERVAL_DOUBLINGS) {
		rpl->dio_interval_doublings = *((uint8_t*) value);
		return SUCCESS;
	}
	return FAIL;
}

param_t rpl_dio_interval_min = {RPL_DIO_INTERVAL_MIN,UINT16_T,
sizeof(uint16_t),getRPLParameter,setRPLParameter};
param_t rpl_dio_interval_doublings = {RPL_DIO_INTERVAL_DOUBLINGS,UINT8_T,
sizeof(uint8_t),getRPLParameter,setRPLParameter};
param_t params[2] = {rpl_dio_interval_min,rpl_dio_interval_doublings};
param_set_t rpl_param_set = {RPL_PARAMETERS,2,params};

//A) either add parameters separately:
paramDB_add_parameter(&rpl_dio_interval_min);
paramDB_add_parameter(&rpl_dio_interval_doublings);
//B) either add parameters grouped in a set:
paramDB_add_parameter_set(&rpl_param_set);
```

c. Evaluation

The local MCE extensions require a minimal amount of ROM and RAM memory for implementing the UPI interfaces in the CoAP MCE agent and the local parameter database.

Table 10 Memory usage of the local MCE in Contiki.

Component	ROM	RAM
	Fixed	Fixed
CoAP MCE agent	770	167
parameter DB	494	4

The overhead for exposing parameters inside protocols depends on the number of parameters and how the getter and setter methods are implemented. In the table below, the minimal requirements are given for RPL and CoAP. To achieve this result, a single getter and setter function was defined in both RPL and CoAP that just returns / changes the value without doing any local processing (type checking, input validation, restarting of timers...).

Table 11 Memory required for the protocol specific extensions. The results are shown for RPL and CoAP. Also included is the minimal memory required to define a parameter or a set of parameters.

Component	ROM (bytes)	RAM
RPL extensions	128	24
CoAP extensions	88	16
param_t definition	8	2
param_set_t definition	6	2

5 UPI_G implementation

This section introduces ***UPI_G***, its available network functions and how the implementation is done for each software platform. The UPI_G is required for ***coordinated*** (time synchronized) ***remote execution*** of configuration and monitoring related functions on a group of nodes. The experimenter can execute on a set of nodes any UPI_N/R function. This is possible as WiSHFUL provides time synchronization among wireless nodes either using their backbone/GPS (out-of-band) or in-band. The way the wireless nodes are time synchronized is platform and architecture-dependent. In case of Linux-based system with wired NIC we use the PTP protocol on the wired Ethernet backbone.

This section also includes performance evaluation of each implementation.

5.1 Linux Networking subsystem

Below an example is shown of a *global control program*, which is using the UPI_G interface to set the radio channel on a set of IEEE 802.11 Linux-based nodes by performing ***remote calls*** to the corresponding UPI_R/N interfaces.

```
if __name__ == '__main__':

    # name of the experiment group; only nodes of this group can be controlled
    exp_group_name = "MyWishFulTest"

    # get reference to global UPI
    global_mgr = GlobalManager(exp_group_name)

    nodes = []
    node0 = Node("192.168.103.125") # nuc2
    node1 = Node("192.168.103.134") # nuc3
    nodes.append(node0)
    nodes.append(node1)

    # node discovery: wait until all specified nodes are available
    nodes = global_mgr.waitForNodes(nodes)

    # start thread for callback, have to be done after some peers are available
    global_mgr.startResultCollector()

    iface = 'mon0'
    channel = 36

    try:
        UPIfunc = UPI_RN.setRfChannel
        UPIargs = {'iface' : iface, 'channel' : channel}
        rvalue = global_mgr.runAt(nodes, UPIfunc, UPIargs)
    except Exception as e:
        log.fatal("... An error occurred : %s" % e)

    global_mgr.stop()
```

Beside the possibility of ***synchronous remote execution*** of any UPI_R/N function the UPI_G interface also allows the ***asynchronous time-scheduled execution***, which is illustrated, in the following example. Here we are using the UPI_G function to find out whether two nodes are in communication range or not. Specifically, we set one node in reception mode (sniffing) at t=now+2s whereas the other node is generating link probing packets at t=now+3s, i.e. 1 s later.

```
def testTwoNodesCommunicationRange(self, node1, node2, mon_dev='mon2', MINPDR=0.9):

    nodes = []
```

```
    nodes.append(node1)
    nodes.append(node2)
    rxPkts = {}

    def csResultCollector(json_message, funcId):
        time_val = json_message['time']
        peer_node = json_message['peer']
        messagedata = json_message['msg']
        self.log.info('CommRange callback %d: receives data msg at %s from %s : %s'
% (funcId, str(time_val), peer_node, messagedata))

        if messagedata is None:
            rxPkts['res'] = 0
        else:
            rxPkts['res'] = int(messagedata)

    try:
        now = get_now_full_second()
        UPIfunc = UPI_RN.getParameterLowerLayer
        UPIargs = {'cmd' : UPI_RN.IEEE80211_L2_SNIFF_LINK_PROBING, 'iface' :
mon_dev, 'ipdst' : "1.1.1.1", 'ipsrc' : "2.2.2.2", 'sniff_timeout' : 5}
        exec_time = now + timedelta(seconds=2)
        self.log.debug('(1) sniff traffic at %s' % str(node1))
        callback = partial(csResultCollector, funcId=1)
        self._upi_g.runAt((node1), UPIfunc, UPIargs, unix_time_as_tuple(exec_time),
callback)

        UPIfunc = UPI_RN.getParameterLowerLayer
        UPIargs = {'cmd' : UPI_RN.IEEE80211_L2_GEN_LINK_PROBING, 'iface' : mon_dev,
'num_packets' : 255, 'pinter' : 0.01, 'ipdst' : "1.1.1.1", 'ipsrc' : "2.2.2.2"}
        exec_time = now + timedelta(seconds=3)
        self.log.debug('(2) gen traffic at %s' % str(node2))
        self._upi_g.runAt((node2), UPIfunc, UPIargs, unix_time_as_tuple(exec_time))

        while len(rxPkts)==0:
            time.sleep(1)

    except Exception as e:
        self.log.fatal("An error occurred (e.g. scheduling events in the past): %s"
% e)

    # calc PDR
    pdr = rxPkts['res'] / float(255)

    minPdrFloat = float(MINPDR)

    if pdr >= minPdrFloat:
        return True
    else:
        return False
```

5.1.1 Library Functions

Complex behaviour consists of compound UPI calls. In the previous example in order to find out whether two nodes are in communication range we used the two UPI functions, namely, UPI_RN.IEEE80211_L2_SNIFF_LINK_PROBING and UPI_RN.IEEE80211_L2_GEN_LINK_PROBING in coordinated way. Because such compound functionality is frequently used we provide libraries functions stored in a code repository for that.

In the following we present two compound functions provided by the software library by WiSHFUL library for wireless nodes using the 802.11 technology. Specifically, the NetworkFunctionHelper is used.

(1) Estimation of the nodes in reception range, i.e. the hearing map:

```
# estimates which nodes are in carrier sensing range and which not
isInComms = networkFuncHelper.getNodesInCommunicationRange(nodes, wifi_iface, rfCh)
```

Here *nodes* is the set of nodes under test, *wifi_iface* the WiFi interface and *rfCh* the radio channel to be used.

(2) Estimation of the nodes in carrier sensing range.

Protocols like IEEE 802.11 and 802.15.4 use listen-before-talk medium access schemes that is called physical carrier sensing. Unfortunately, such protocols are suffering from performance issues to the hidden-terminal problem. There is a large research area dedicated to improving the performance by solving this problem. Via the proposed UPI functionality, experimenters, can easily obtain information about which wireless nodes are inside and which are outside their carrier sensing range enabling them to solve the hidden-terminal problem.

```
# estimates which nodes are in carrier sensing range and which not
isInCss = networkFuncHelper.getNodesInCarrierSensingRange(nodes, wifi_iface, rfCh)
```

Here *nodes* is the set of nodes under test, *wifi_iface* the WiFi interface and *rfCh* the radio channel to be used.

5.2 Contiki Embedded OS

As stated global control is possible in two manners:

1) Executing the local MCE on the Linux host of each sensor node.
2) Executing the local MCE on the sensor node itself inside Contiki.

The former case uses the ZeroRPC over ZeroMQ method as described in the previous section. The latter case enables a custom form of remote procedure calls over CoAP. Currently the following two UPI_N functions can be called remotely:

```
def setParameterHigherLayer(interface, param_key_value):
def getParameterHigherLayer(interface, param_key):
```

In principle, all configuration settings of the Contiki network stack can be made (remotely) reconfigurable. A code snippet of a global control program that uses UPI_G to change the RPL parameters *DIO_INTERVAL_MIN* and *DIO_INTERVAL_DOUBLINGS* is given below.

```
if __name__ == '__main__':

    # name of the exp. group; only nodes of this group can be controlled
    exp_group_name = "MyWishFulTest"

    # get reference to UPI_G
    global_mgr = GlobalManager(exp_group_name)

    nodes = []
    node0 = Node("192.168.103.125") # sensor1
    node1 = Node("192.168.103.134") # sensor2
    node2 = Node("192.168.103.232") # sensor3
    nodes.append(node0)
    nodes.append(node1)
    nodes.append(node2)

    # node discovery: wait until all specified nodes are available
    nodes = global_mgr.waitForNodes(nodes)

    try:
       iface = {'WPAN0','CONTIKI'}
```

```
        param_list = {     {'DIO_INTERVAL_MIN',1000},
                           {'DIO_INTERVAL_DOUBLINGS',10}};
        UPIargs = {'iface' : iface, 'param_list' : param_list}
        UPIfunc = UPI_N.setParameterHigherLayer
       error = global_mgr.runAt(nodes, UPIfunc, UPIargs)
        for node in nodes:
               log.debug('Ret value of blocking call is %s' % error(node))
    except Exception as e:
        log.fatal("An error occurred : %s" % e)

    # tear down
    global_mgr.stop()
```

In year two more functions will be added especially those that enable monitoring certain events in the network. In the remainder of this Section a more detailed discussion about the internals of the UPI_G implementation over CoAP approach is given.

a. Choosing an appropriate protocol for enabling remote access.

Selecting the appropriate application layer protocol for exchanging configuration messages across the network is very important because this will have a high impact on the resource efficiency of the overall solution. Several candidates were compared in [10] and evaluated based on the device memory requirements and message size overhead.

The most dominant application layer protocol for constrained IoT devices today is CoAP. It has built-in support for resource (e.g. parameter) discovery and block wise (e.g. batch configuration) transfers. From a functional viewpoint, all required features are present. Since CoAP is tailored for constrained devices, the memory and CPU requirements are limited. Moreover, the message overhead is also minimal because the CoAP header is very small and UDP is used as transport protocol. With portability and compatibility in mind, CoAP is also a logical choice because it is well supported by nearly all embedded OSs (Contiki, TinyOS, RiOT) and easily integrate-able in web-based systems since it is REST based.

An alternative for CoAP is MQTT [11], a publish-subscribe system with a central MQTT broker that runs over TCP. MQTT clients can publish or subscribe to topics (e.g. parameters). For each parameter, two topics are required: (1) one published by the sensor node for supporting the get operation; and (2) one published by the configuration server for supporting the set operation. Because of this, MQTT will have a much higher device memory overhead. Also the message overhead will be bigger since it runs over TCP. Moreover, it is less supported because only a Contiki implementation is available. Other alternatives are AMQP and XMPP. Both also use TCP as transport protocol and have much higher device memory requirement and message overhead since they are not tailored for constrained devices.

To conclude, *CoAP is the most appropriate application layer protocol* to support dynamic reconfiguration, as also indicated in [10].

b. Communication flow between different entities

Figure 9 illustrates the communication flow between the different entities in the architecture; the active components are depicted using white boxes and bold text. CoAP is used by the GITAR connector module running on the Global MCE to configure node j. For this purposes, the control and management proxy translates the HTTP requests/responses into CoAP requests/responses and vice-versa.

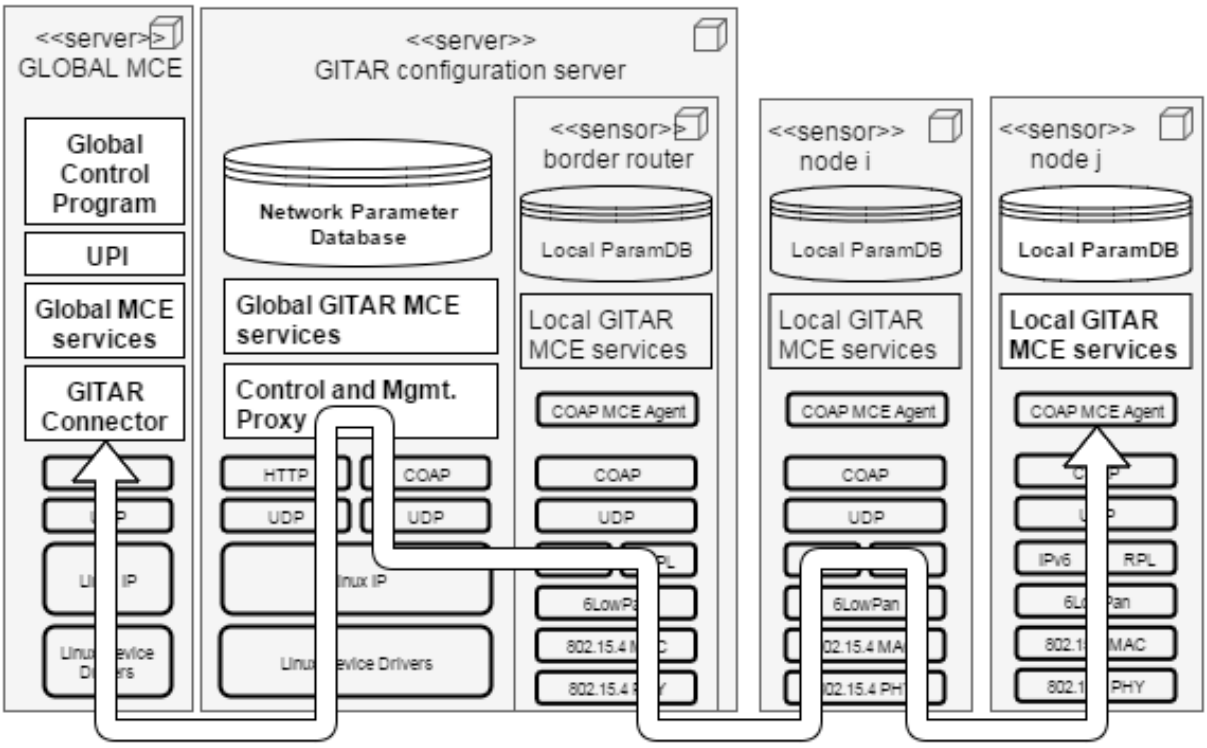

Figure 9. The communication flow between the different entities in the architecture. The white boxes with bold text depict the active components in this example.

The intermediate sensor nodes (e.g. border router and node *i*) do not process the CoAP message but forward it to the destination using RPL. Packets coming from the sensor network are injected directly in the Linux IPv6 stack by the border router. The CoAP MCE agent processes the requests and uses the Local GITAR MCE services to reconfigure the protocols.

c. Efficient usages of CoAP for remote reconfiguration

The CoAP memory requirements constitute of the fixed overhead for the CoAP engine (8.5 kB ROM/1.5kB RAM [6] and the variable amount of ROM occupied by the CoAP resources. The additional memory overhead for exposing configuration settings hence depends on the number of CoAP resources required to expose the parameters. Three granularity levels are considered: (1) a CoAP resource per configuration parameter; (2) a CoAP resource per protocol; and (3) one CoAP resource for the entire network stack.

CoAP is a text-based protocol and resources are identified using unique string names encoded in the resource URI. Both need to be stored in the ROM memory of each sensor device. Depending on the granularity level, the string name of each parameter (1), protocol (2) or stack (3) is stored in memory causing extra ROM overhead. Moreover, when using granularity level (2) or (3), also parameters still need to be identified. This can be done using either unique names, encoded in the URI query variable, or unique IDs, encoded in the payload.

The following combinations of resource granularity and identification method are possible:

- A resource per parameter enables direct addressing of parameters without requiring any transformation. It is the most straightforward for integration in browsers using add-ons such as Copper [12] The ROM overhead, on the other hand, will be high because for each parameter the string name must be stored and a resource must be defined.

- A resource per protocol groups parameters on a protocol level. They are addressed indirectly via the protocol resource implying that an if-else structure is required in the GET/POST/PUT handlers for identifying the correct protocol parameter. The total ROM overhead is the memory required for storing the protocol name, the fixed CoAP resource overhead and, per parameter, the identification (unique name or ID) and if-else overhead.
- A resource for the entire stack has the advantage that there is looser coupling with the protocols, compared to the previous options. A tight coupling implies that a protocol update also require updating the CoAP resources(s) that enables remote reconfiguration of the protocol. The third approach, however, requires an explicit implementation of a parameter database that can be used by the generic resource to manipulate configuration settings and by the protocols to (de-)register parameters. The ROM overhead now includes the fixed CoAP resource overhead, parameter database implementation and, per parameter, the unique name or ID.

In order to make well-founded decisions, the impact on the ROM memory usage for different granularities was analysed using stub resources in Contiki for CoAP. This allowed devising a mathematical model that could be applied on in the default Contiki IPv6 network stack, giving an estimate of the overhead for adding reconfiguration. Also the parameter identification method in level (2) and (3) was investigated. Figure 10 gives an overview of the ROM overhead estimated for the different resource granularities and identification methods.

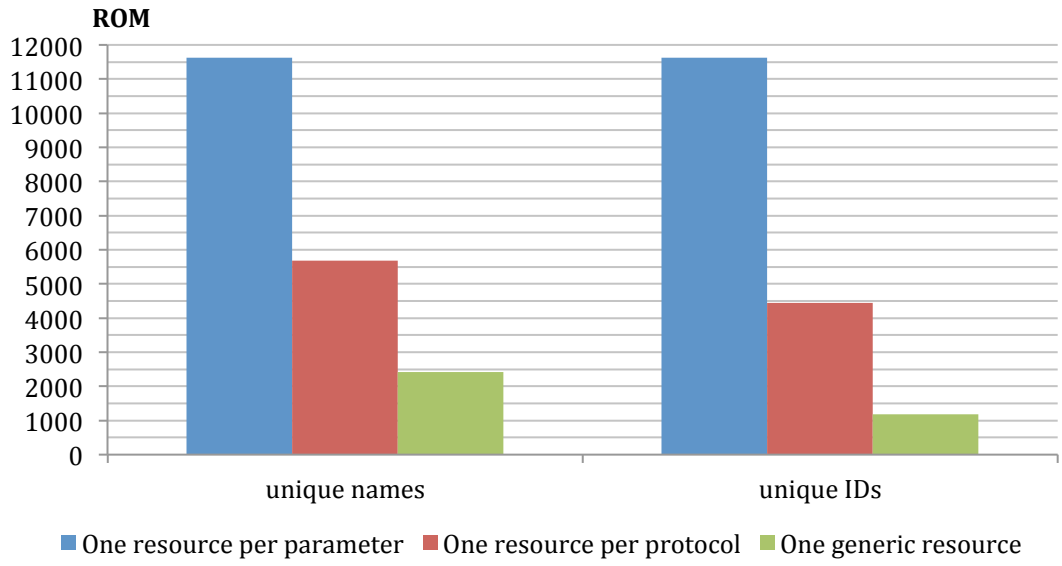

Figure 10 Estimated ROM overhead for different resource granularities and identification methods. The results were obtained by considering the candidate configuration parameters defined in the default Contiki IPv6 network stack.

The results clearly show that a single generic resource requires 80% less memory (1.2 kB) compared to a resource per parameter (11.6 kB) and 60% less compared to a resource per protocol (4.5kB). Using unique IDs instead of names also has a major impact. Given the size in ROM of the default Contiki IPv6 stack (+-30kB), using *a single generic CoAP resource and unique IDs is the preferred choice*. However, if memory is not a major concern, also the second option that defines a resource per protocol and uses unique parameter IDs can be used. For this purpose, the parameter DB allows to add and get both single parameters and a set of parameters, as defined in Section 4.2.3.

d. Preliminary evaluation

The UPI_G interface of the standard Linux global MCE was used to generate the parameter change requests. The sensor code was developed in Contiki 3.0 and executed on a Zolertia Z1 (16MHz CPU, 92KB ROM, 10KB RAM and an IEEE-802.15.4 compliant transceiver). A single resource combined with a parameter repository is used for configuring the network stack. All communication between the different entities is CoAP based. On Linux libCoAP [13] is used while in Contiki the ERBIUM CoAP[6] engine is utilized.

The average latency for changing parameters depends on the number of PUT/POST requests needed to perform a batch configuration on all nodes. It is measured on the CoAP management proxy by calculating the delay between the first request and last response. The average latency is an important performance indicator because it defines the duration in which the network is in an inconsistent state. Figure 11 illustrates the average latency in seconds for one to twenty POST/PUT requests (e.g. number of nodes) in steps of four. Also the standard deviation over all experiments is indicated. The results clearly show that the average latency scales with the number of POST/PUT requests (e.g. nodes).

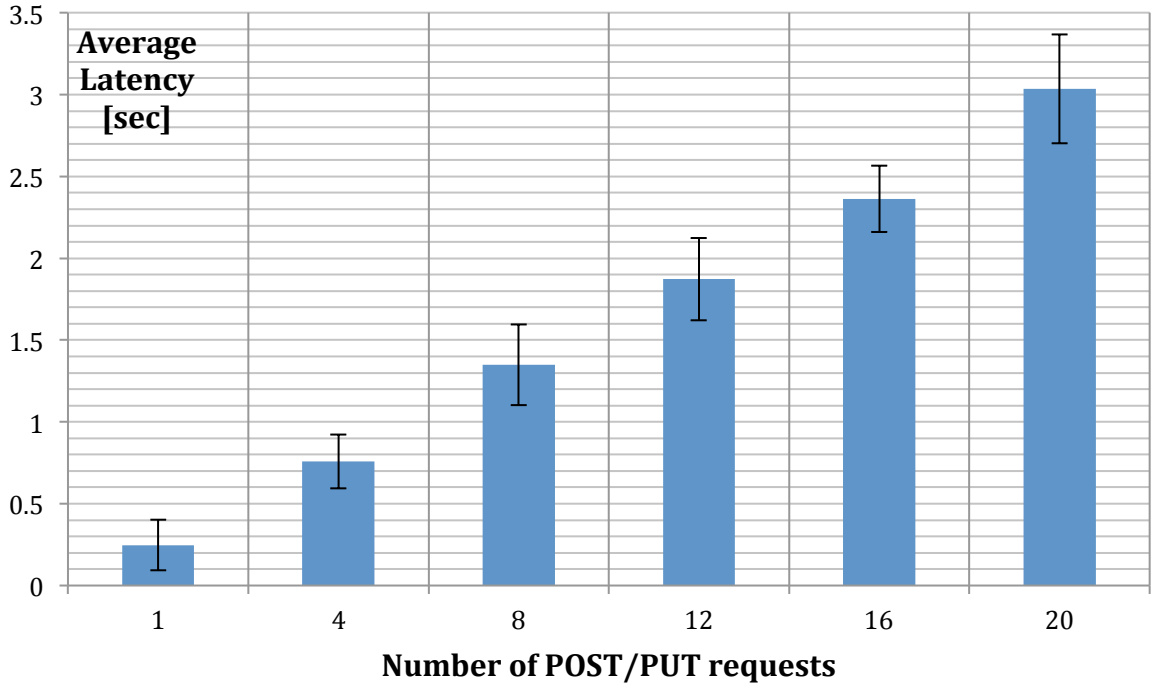

Figure 11 Average latency for increasing number of POST/PUT requests (e.g. nodes). Also the standard deviation is denoted on the chart.

From a functional viewpoint the basic interactions (e.g. get/set parameter) were tested using the Copper Firefox add-on, which allows sending CoAP requests and responses. Also a python module was developed for integrating the in-band global control of sensor networks into the WiSHFUL architecture. Since only a subset of the WiSHFUL UPIs is currently available, the full integration of in-band control over CoAP will be completed in Year 2.

Figure 12 shows results of the node discovery process. For each of the discovered nodes the available protocols and their configuration parameters are also listed.

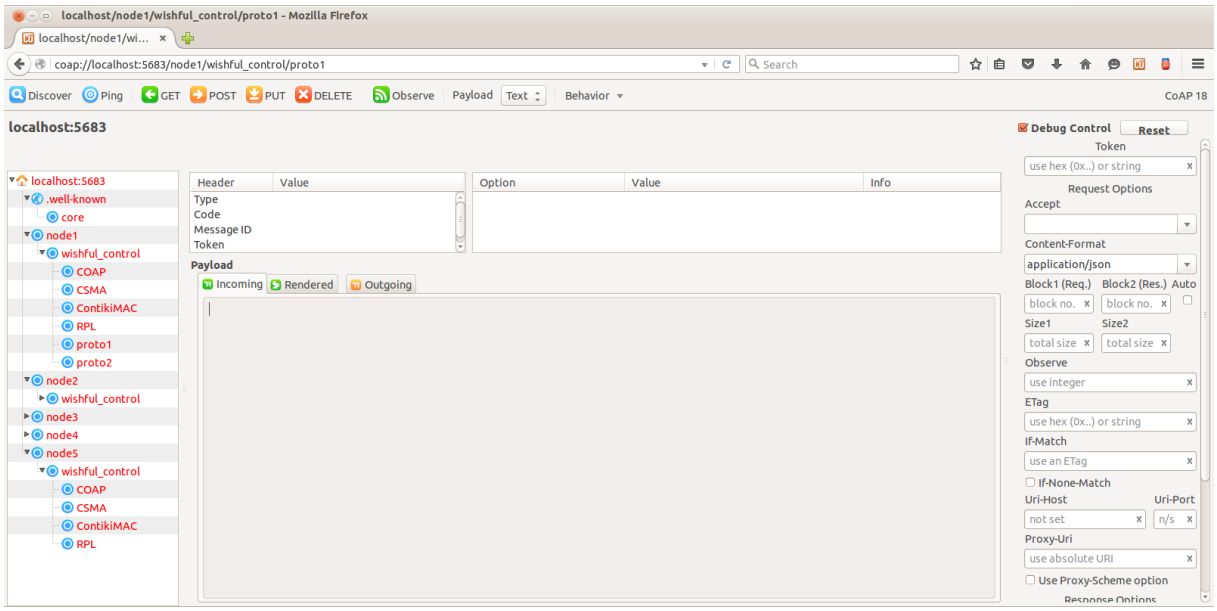

Figure 12 Screenshot of Copper Firefox extensions illustrating the node discovery over CoAP (e.g. in-band control channel). The same interactions can be done from a python module, integrated in the WiSHFUL architecture.

Figure 13 illustrates the result of the get parameter operation that retrieves all the parameters of the ContikiMAC protocol on node 1.

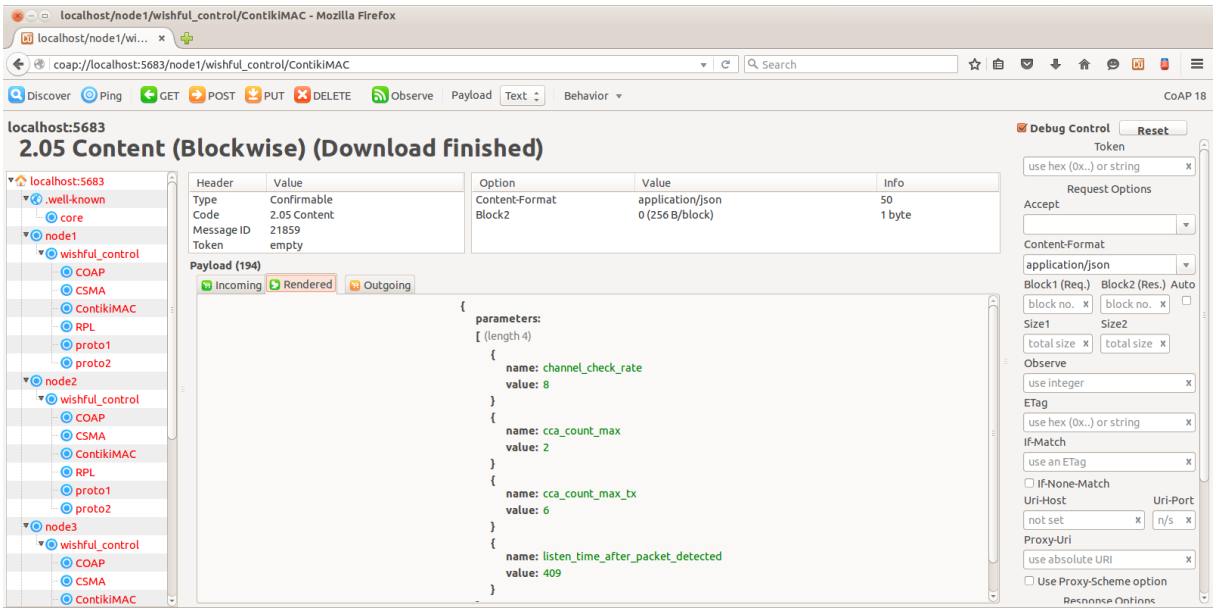

Figure 13 Screenshot of Copper Firefox extensions illustrating the getParameter operation over CoAP (e.g. in-band control channel).

Figure 14 demonstrates how parameters can be set using the setParameter operation. For this purpose a CoAP put request is send. In this example the channel_check_rate and some test parameters are changed in ContikiMAC on node 2.

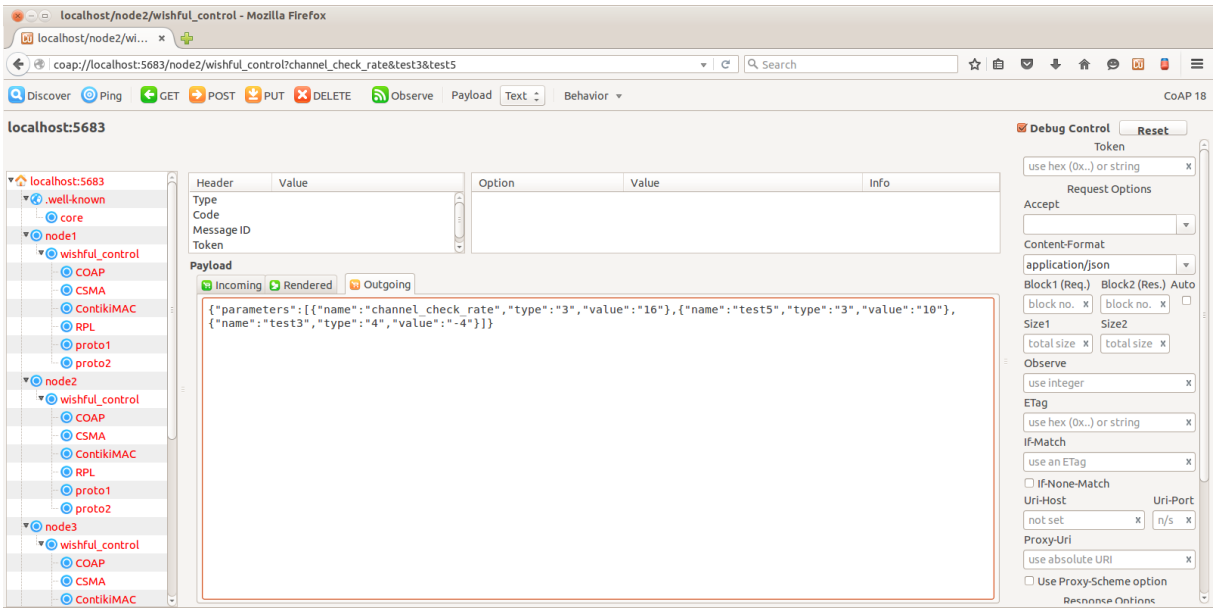

Figure 14 Screenshot of Copper Firefox extensions illustrating the setParameter operation over CoAP (e.g. in-band control channel).

6 UPI_M implementation

This section introduces UPI_M, its functions and how the implementation is done for each software platform.

All ***management related functions*** are grouped in the UPI_M interface because they are required for managing protocol software modules at any layer, thus spanning both UPI_R and UPI_N. Moreover, software management requires functionality on both the local and global level.

6.1 Linux Networking Subsystem

As far as no ***management related functions*** were implemented for Linux-based systems. All required software modules for the network protocol stack must be available after deployment phase.

6.2 Contiki Embedded OS

Currently the GITAR extensions enable to dynamically link new/updated modules to an already deployed Contiki system. This is a condition sine qua non when on-the-fly-code updates are required. In Year 2 ***management related functions*** for deploying and installing software modules in sensor networks will be heavily investigated.

7 UPI_HC implementation

Besides pure local and pure global control programs there is the option to write ***hierarchical control programs*** allowing the simultaneous use of local and global control programs. Such a hierarchical architecture requires an ***inter control program interface,*** the ***UPI_HC***, which is described in the following for each software platform.

A hierarchical control system is a form of control system in which a set of controllers is arranged in a ***hierarchical tree***. The controllers are communicating over network connections hence resulting in a hierarchical networked control system. The defining feature of such system is that control and feedback signals as well as collected and possibly aggregated radio & network data are exchanged among the components in the form of messages through a network.

There is a need for hierarchical controllers because some control functions need to be partitioned into a locally executed part due to real-time radio constraints and limitations of communication network protocol (e.g. limited bandwidth, delay), and those that need a global system view and global coordination to form a meaningful control decision (e.g. coordination between heterogeneous networks).

7.1 Linux Networking subsystem

We provide an implementation for ***hierarchical control*** for Linux-based systems. The current implementation has the following known limitations:

- Depth of hierarchical tree is 2, i.e. global control program on top and the local control programs running on the nodes as leafs,
- The global control program can instantiate "on-the-fly" any arbitrary local control program which will be executed at node-level and control its behavior: start & stop,
- The local control programs are able to communicate control messages to the global control program using UPI_HC interface whereas currently there is no support for communication of control messages from the local to the global control programs.

Below an example is shown of a ***hierarchical control program***. Here the global control program is using the HierarchicalManager to set-up the local control programs on the nodes. In the presented example the local control programs are periodically estimating the airtime utilization of the wireless channel and switches to a new random channel if the value exceeds some given threshold. Moreover, the local control program uses the UPI_HC (see upiHCImpl) to inform the global control program about the newly selected channel. Finally, after five seconds the global control program stops the execution of the local control programs.

```
if __name__ == '__main__':

    # name of the experiment group; only nodes of this group can be controlled
    exp_group_name = "MyWishFulTest"
    # get reference to UPI_HC
    hc_mgr = HierarchicalManager(exp_group_name)

    nodes = []
    node0 = Node("192.168.103.125") # nuc2
    node1 = Node("192.168.103.134") # nuc3

    nodes.append(node0)
    nodes.append(node1)

    # node discovery: wait until all specified nodes are available
    nodes = hc_mgr.waitForNodes(nodes)

    # start thread for callback, have to be done after some peers are available
    hc_mgr.startResultCollector()

    try:
        # custom control program to be executed locally on each node
        def customLocalController(ch_util_threshold):

            import agent
            import time
            import logging
            from random import randint

            # references to WiSHFUL framework
            global upiRNImpl # interface to UPI_R/N implementation
            global upiHCImpl # UPI_HC interface used for comm. with global control prog.

            log = logging.getLogger()

            while not upiHCImpl.stopIsSet():
                # estimate current channel load
                UPIargs = {'cmd' : UPI_RN.IEEE80211_CHANNEL_AIRTIME_UTILIZATION, 'iface' :
'mon0'}
                ch_util = upiRNImpl.getParameterLowerLayer(UPIargs)
```

```
                if ch_util > ch_util_threshold:
                    # channel utilization becomes too high; choose new channel
                    new_ch = randint(0,9)
                    UPIargs = {'cmd' : UPI_RN.IEEE80211_CHANNEL, 'iface' : 'mon0', 'channel' :
new_ch}
                    upiRNImpl.setParameterLowerLayer(UPIargs)
                    # inform global control program about new channel using UPI_HC
                    upiHCImpl.transmitCtrlMsgUpstream(new_ch)
                time.sleep(1)
            return True

        numCBs = {}
        numCBs['res'] = 0

        """
        Custom callback function used to receive result values from scheduled calls.
        """
        def resultCollector(json_message, funcId):
            time_val = json_message['time']
            peer_node = json_message['peer']
            messagedata = json_message['msg']
            numCBs['res'] = numCBs['res'] + 1

        """
        Custom callback function used to receive control messages from local control programs.
        """
        def ctrlMsgCollector(json_message):
            time_val = json_message['time']
            peer_node = json_message['peer']
            msg_data = json_message['msg']
            log.info('Global ctrl program receives ctrl msg at %s from %s : %s' %
(str(time_val), peer_node, msg_data))

        # register callback function for collecting results
        hc_mgr.setCtrlCollector(ctrlMsgCollector)

        # get current time
        now = get_now_full_second()

        # deploy a custom control program on each node
        CtrlFuncImpl = customLocalController
        CtrlFuncargs = (0.9,)

        now = get_now_full_second()

        # exec immediately
        exec_time = now + timedelta(seconds=2)

        nodes = hc_mgr.getNodes()
        try:
            # this is a non-blocking call
            callback = partial(resultCollector, funcId=99)
            isOntheflyReconfig = True
            hc_mgr.runAt(nodes, CtrlFuncImpl, CtrlFuncargs, unix_time_as_tuple(exec_time),
callback, 1, isOntheflyReconfig)

            exec_time = now + timedelta(seconds=5)

            callback = partial(resultCollector, funcId=99)
            hc_mgr.runAt(nodes, UPI_RN.stopFunc, (None,), unix_time_as_tuple(exec_time),
callback, 1, False)
        except Exception as e:
            log.fatal("An error occurred : %s" % e)

        # busy waiting
        while numCBs['res'] < 2:
            time.sleep(1)
    except:
        log.warning('HC failed!')
```

7.2 Contiki Embedded OS

Currently no UPI_HC implementation is available for Contiki. In Year 2 the solution with an in-band control channel over CoAP will be extended to allow also interactions between a global control program and local control programs on the nodes.

8 Examples of control programs using UPIs

The section gives implementation details of the control programs used in the showcases.

The following example shows how a global control program can use the UPI_N to connect an 802.11 wireless node running in station mode to an 802.11 AP:

```
UPIfunc = UPI_RN.setParameterLowerLayer
# remote function args
UPIargs = {'cmd' : UPI_RN.IEEE80211_CONNECT_TO_AP, 'iface' : 'wifi2', 'ssid' : 'effman-nuc2'}

rvalue = global_mgr.runAt(STANodes, UPIfunc, UPIargs, None)
```

Here we have to define the name of the wireless interface and the SSID of the AP the station should connect to.

The next example shows how to get the HW address (MAC) of a particular network interface using the RadioHelper class:

```
# get my MAC HW address
hwAddr0 = radioHelper.getHwAddrRemote(laptop, 'wifi2')
```

Finally, this example shows that with the help of UPI_G and UPI_N we are able to set-up packet flows on the application layer, i.e. TCP/IP:

```
log.info('Installing applications in node0 and node1')
serverApp = ServerApplication()
serverApp.setStartTime(Time.Now() + Time.Seconds(2))
serverApp.setPort(5012)
serverApp.setProtocol("TCP")
netHelper.installApplication(node1, serverApp)

clientApp = ClientApplication()
clientApp.setStartTime(Time.Now() + Time.Seconds(4))
clientApp.setDestination(node1.getIpAddress())
clientApp.setPort(5012)
clientApp.setProtocol("TCP")
clientApp.setTransmissionTime(6)
netHelper.installApplication(node0, clientApp)
```

9 Improvements and extensions

It includes a list of improvements and extensions to be implemented in Year 2 and should be aligned with the use cases to be implemented in Year 2.

Finally, also hierarchical control programs can be designed where UPI_HC, allows also the transmission of control messages from the global to the local control programs. Note: so far the UPI_HC is limited to pro-active, push based, message exchange between the local control programs and the global control program. A reactive, event based, message exchange system is required to enable all inter control program interactions via UPI_HC.

So far only a proactive controller model is supported. However, a reactive approach would be particularly helpful when designing low-latency control programs or when large amounts of data, e.g. streaming data, need to be processed. Therefore, we plan to extend the WiSHFUL controller in Year 2 to support reactive control.

Currently, we support two operation systems, namely, i) Linux-based and ii) Contiki-based systems. For Year 2 we plan to provide support for the Microsoft Windows platform which is often used as a platform for Software-defined Radio (SDR) as well as TinyOS as another option for sensor networks.

10 Conclusions

In this deliverable, we describe the first release of the WiSHFUL software architecture for network control, which comprises of two main components: i) the **WiSHFUL control framework**, for providing a global view of the solution under test to the experimenter and defining the network control logics, and ii) the **unified UPI_N interface** for monitoring and configuring the higher layers of the network protocol stack (higher MAC and above) of the nodes.

With the help of the UPI_N interface for network control defined in this document and the UPI_R for radio control which is presented in D3.2 it is possible to ***program the datapath***, i.e. setting per-flow specific wireless transmission settings like transmission power and MAC prioritization. Therefore, the packets belonging to a particular flow are identified using the 5-tuple (IP src/dst, src/dst port, ToS) and tagged using the UPI_N interface whereas the transmission settings to be chosen depend on the tag and can be programmed using the UPI_R interface.

The UPI_N implementation is based on the development of **network connector modules**, able to map platform-independent function calls into platform-specific tools and functionalities (which may vary as a function of the platform capabilities). The implementation details for the supported platforms, based on Linux and Contiki.

11 References

[1] Ruckebusch, P., De Poorter, E., Fortuna, C. and Moerman, I., 2016. GITAR: Generic extension for Internet-of Things ARchitectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, *36*, pp.127-151.

[2] T. Watteyne, Ed. Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement https://tools.ietf.org/html/rfc7554

[3] Sheng, Z., Yang, S., Yu, Y., Vasilakos, A., Mccann, J. and Leung, K., 2013. A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities. *Wireless Communications, IEEE*, *20*(6), pp.91-98.

[4] Tsiftes, N., Eriksson, J. and Dunkels, A., 2010, April. Low-power wireless IPv6 routing with ContikiRPL. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (pp. 406-407). ACM.

[5] Narten, T., Nordmark, E., Simpson, W., and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, DOI 10.17487/RFC4861, September 2007, <http://www.rfc-editor.org/info/rfc4861>.

[6] Kovatsch, M., Duquennoy, S. and Dunkels, A., 2011, October. A low-power CoAP for Contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on* (pp. 855-860). IEEE.

[7] Luzuriaga, J.E., Perez, M., Boronat, P., Cano, J.C., Calafate, C. and Manzoni, P., 2015, January. A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE* (pp. 931-936). IEEE.

[8] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[9] Winter, T., Ed., Thubert, P., Ed., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, JP., and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, DOI 10.17487/RFC6550, March 2012, <http://www.rfc-editor.org/info/rfc6550>.

[10] Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F. and Alonso-Zarate, J., 2015. A Survey on Application Layer Protocols for the Internet of Things. *Transaction on IoT and Cloud Computing*, *3*(1), pp.11-17.

[11] Hunkeler, U., Truong, H.L. and Stanford-Clark, A., 2008, January. MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on* (pp. 791-798). IEEE.

[12] Kovatsch, M., 2011, June. Demo abstract: human-CoAP interaction with copper. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on* (pp. 1-2). IEEE.

[13] Bergmann, O., 2012. libcoap: C-Implementation of CoAP. *URL: http://libcoap. sourceforge. net, Date of access 21.12*.2015