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

Project Deliverable D4.4

Second operational network control software platform

Contractual date of delivery:	31-12-2016
Actual date of delivery:	23-12-2016
Beneficiaries:	IMEC, TCD, CNIT, TUB
Lead beneficiary:	TUB
Authors:	Anatolij Zubow (TUB), Piotr Gawłowicz (TUB), Mikołaj Chwalisz (TUB), Ilenia Tinnirello (CNIT), Peter Ruckebusch (IMEC), Ingrid Moerman (IMEC), Jo Van Damme (IMEC), Maicon Kist (TCD)
Reviewers:	Domenico Garlisi (CNIT), Spilios Giannoulis (IMEC)
Work package:	WP4 – Network Control
Estimated person months:	21
Nature:	R
Dissemination level:	PU
Version:	1.0

Abstract:

This deliverable gives a detailed description of the capabilities and performance of the final network control software platform.

Keywords:

Programmable network architecture, software-defined networking, network control.

Executive Summary

The classical control and management plane for computer networks is addressing individual parameters of protocol layers within an individual wireless network device. We argue that this is not sufficient in phase of increasing deployment of highly re-configurable systems, as well as heterogeneous wireless systems co-existing in the same radio spectrum which demand harmonized, frequently even coordinated adaptation of multiple parameters in different protocol layers (**cross-layer**) in multiple network devices (**cross-node**).

Therefore, in WiSHFUL project, we propose a set of the **Unified Programming Interfaces** (UPI) enabling a coordinated cross-layer control and management operation over multiple network nodes – D4.2. With usage of the UPIs, the network control logic may be implemented either in a centralized or distributed manner. This allows to place time-sensitive control functions close to the device under control (i.e., local control programs), while off-loading more resource hungry control programs to compute servers and make them work together to control entire network.

This deliverable reports the UPIs supported in Y2 of the project in the ***second release of network control*** software platform. The focus is on a detailed description and implementation the ***Unified Programming Interfaces for network control*** (i.e. *UPI_N, UPI_G, UPI_M* and *UPI_HC*), whereas deliverable D3.4 addresses radio control through *UPI_R* focusing on the lower layers, i.e. lower MAC and physical layer. The UPI functionality has been implemented for two different platforms, namely, Linux-based wireless nodes and sensor nodes using the Contiki operating system. The full documentation of UPI_N, UPI_G, UPI_M and UPI_HC together with the code of the implemented software is available in the WiSHFUL GitHub repository.

Moreover, this document contains also description of capabilities, implementation and performance evaluation of the ***WiSHFUL control framework***. The WiSHFUL control framework was tested in a number of **showcases** (see D2.4), that provided valid proofs of its usability.

List of Acronyms and Abbreviations

6LowPan	IPv6 over Low power Wireless Personal Area Network
AP	Access Point
API	Application Programming Interface
ARM	Advanced RISC Machine
CAPWAP	Control And Provisioning of Wireless Access Points
CoAP	Constrained Application Protocol
CSMA	Carrier Sense Multiple Access
DMA	Direct Memory Access
DUT	Device Under Test
FEC	Forward Error Correction
FFT	Fast Fourier Transform
GCP	Global Control Program
GPS	Global Positioning System
HetNet	Heterogeneous Networks
ICMP	Internet Control Message Protocol
IPv6	Internet Protocol version 6
ISM	Industrial, Scientific, Medical
LCP	Local Control Program
LTE	Long Term Evolution
LTE-U	LTE in unlicensed spectrum
LWAPP	Lightweight Access Point Protocol
MAC	Medium Access Control
MCS	Modulation and Coding Scheme
MIPS	Microprocessor without Interlocked Piped Stages
NDPI	Native Device Programming Interface
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
NTP	Network Time Protocol
OS	Operating System
PTP	Precision Time Protocol
QDisc	Queueing Disciplines
RE	Resource Element
RPC	Remote Procedure Call
RPL	Routing Protocol for Low power and Lossy Networks

SDN	Software Defined Networking
SDR	Software Defined Radio
SNMP	Simple Network Management Protocol
SoC	System On Chip
STA	Wireless Station
SUT	System Under Test
TAISC	Time Annotated Instruction Set Computer
TBF	Token Bucket Filter
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
UDP	User Datagram Protocol
UML	Unified Modeling Language
UPI	Unified Programming Interface
UPI_G	Unified Programming Interface Global
UPI_HC	Unified Programming Interface Hierarchical Control
UPI_M	Unified Programming Interface Management
UPI_N	Unified Programming Interface Network
UPI_R	Unified Programming Interface Radio
USRP	Universal Software Radio Peripheral
VDU	Virtualized Deployment Unit
VM	Virtual Machine
VNF	Virtual Network Function
XML	Extensible Markup Language

Table of contents

1 Introduction

This deliverable describes the **WiSHFUL control framework** as well as gives description of the implemented ***Unified Programming Interfaces for network control***, *UPI_N*, i.e. higher layers of the network protocol stack. The source code along with detail documentation of the implemented control framework is available in the WiSHFUL GitHub repository.

In Section 2, description of second release of WiSHFUL control framework is provided. It provides an overview of the implementation architecture along with a description of all components and programming interfaces. This section covers also related work on control frameworks. Section 3 contains description of UPI_N supported in year 2 for Linux and Contiki operating systems. In section 4, we provide description of additional interfaces, namely UPI_G, UPI_M and UPI_HC that are used for global control, management and hierarchical control respectively. Section 5 gives an overview of the WiSHFUL control framework 2.0 implementation for the Linux OS. The framework is implemented in Python programming language. The architecture of the framework implementation is presented in a UML diagram. We provide instructions about framework deployment and discuss support offered for other programming languages and external software libraries. Furthermore, the framework's performance was evaluated and quantitative results are presented. This section contains also a description of future work that is planned for year 3. In section 6, the implementation of the necessary software components to connect wireless sensor network nodes with the WiSHFUL control framework based on Contiki OS is presented. We offer a detailed description of the architecture and implementation of the RPC engine, attribute repository, protocol-specific connector modules, communication wrapper, node discovery and remote UPI function execution support. Finally, Section 7 concludes this document, while references are listed in Section 8.

2 WiSHFUL control framework 2.0

The control plane and the management plane have played a very important role in the classical telecommunication systems, but have been given much less attention in computer networks. As a matter of fact the only widely accepted approach is the usage of Simple Network Management Protocol (SNMP)[1] or Network Configuration Protocol (NETCONF)[2] as basis for creating management applications. This is increasingly recognized as not sufficient - especially in case of wireless networks where many parameters have to be frequently tuned in response to changing wireless propagation, interference and traffic conditions. There were already a couple of attempts for wireless control protocols including Lightweight Access Point Protocol (LWAPP)[3] and Control And Provisioning of Wireless Access Points (CAPWAP)[4], but these were designed with focus on rare changes of configuration and device management and are not suitable for ***time-sensitive control*** of devices.

Furthermore, classical control/management actions have been addressing individual parameters of protocol layers within an individual network device. This is not sufficient in phase of increasing deployment of highly re-configurable systems, as well as heterogeneous wireless systems co-existing in the same radio spectrum which demand harmonized, frequently even coordinated (simultaneous) change of multiple parameters in different parts of hardware and software in multiple network devices. Typical examples of emerging real scenarios are LTE-U and Wi-Fi in 5 GHz and Wi-Fi, Bluetooth and ZigBee in 2.4 GHz ISM band. On the other hand, even homogeneous deployments are suffering from intra-technology interference. In recent years we have seen a boom of cross-layer design proposals for wireless networks where additional information from some layers are obtained and used to optimize operation of other layers. So far control programs had to solve the challenge of harmonized/simultaneous actions on case-by-case basis, which significantly complicated development of such applications and lead to lack of any compatibility across the various solutions. We argue that the efficiency of wireless networks can be significantly improved by ***enabling the management and control*** of the different co-located wireless technologies and their network protocols stacks (cross-layer) in a coordinated way using either centralized, hierarchical or distributed control architectures.

In D4.1, we have presented the WiSHFUL architecture, which is suitable for time-sensitive control of heterogeneous wireless networking devices. The WiSHFUL monitoring and configuration engine (MCE) ***has hierarchical 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 (DUT) 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**.

Contribution: This chapter describes the ***WiSHFUL control framework*** *that implements WiSHFUL architecture*. The suggested Application Programming Interface (API) supports typical functions needed for coordinated cross-layer, cross-technology and cross-node control and similarly as in the SDN paradigm we allow for centralized control, while supporting equally well hierarchical control structure and logically centralized but physically distributed control. Network control programs can be either co-located with the controlled device (both running on the same network node, e.g. for latency reasons) or separated from each other (running on two nodes, e.g. control program runs on server due to requiring high computing power). This enables rapid prototyping of control programs for wireless network devices, management and control of operation of - possibly heterogeneous - nodes in wireless networks.

2.1 System Model

In this section, we define our system model and provide definitions of all terms that we use consistently in this and all related documents.

The network is a collection of nodes - Figure 1 - under common management and control domain (authority). A node is a collection of equipment sharing a common hardware platform (i.e. motherboard) and being operational under a single instance of an operating system. The type of nodes spans from small, constrained devices to powerful computing servers. A node is equipped with zero or more devices. A device is a piece of hardware fulfilling a dedicated functionality. Additionally, platform is a pair of device and proper software that may expose set of operations to control its behaviour and parameters. For convenience, we frequently use term device for platform. For example, a wireless network platform provides packet forwarding functions with usage of wireless transmission technology (e.g. 802.11, LTE) and exposes set of operations in UPI_R and UPI_N interfaces to control its parameters including transmission power, central frequency, bandwidth, etc.

The control logic may be implemented either as standalone or multiple cooperating control programs that run in node(s). In particular, a control program may be located in the same node as the network device that it is controlling.

We assume the existence of a common ***control channel*** enabling control program(s) to: i) access UPI_R and UPI_N of all devices in network, ii) use it to control their behaviour and iii) exchange control messages between each other for cooperation purposes. This control channel may be realized over the wireless network itself (in-band) and/or additional wired backhaul infrastructure (out-band).

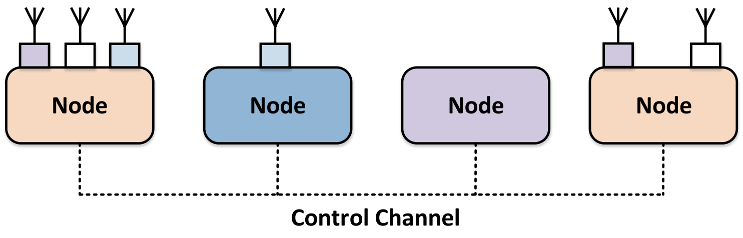

Figure 1. System model overview.

2.2 Requirements and Design Principles

The main objective of the ***WiSHFUL control framework*** is to facilitate and shorten time required for prototyping of novel control solutions in heterogeneous wireless networks. We argue that novel wireless control programs may be realized when the following functionality is provided:

- Coordinated ***collection*** of information from, and ***execution*** of control actions on different protocol layers (***cross-layer***), heterogeneous devices (***cross-technology***) and multiple nodes (***cross-node***) within a network,
- Existence of a global and consistent view of the entire network, i.e. knowledge about the state of all devices and their relationship,
- Possibility to implement logically centralized and physically distributed control programs, i.e. placing time-sensitive tasks close to device and off-loading resource greedy tasks to powerful servers,
- Support for multiple levels of control for scalability reasons, i.e. local control programs handle frequent commands and events, while global/hierarchical control programs handle rare events (Figure 2),
- Support for detecting network changes in proactive and reactive control schemes in control

programs,
- A high-level API for control of operation of individual wireless devices and groups of devices,
- Location transparency i.e. the same API syntax for execution of commands on local and remote devices (-> UPI_R/N),
- Possibility to execute commands on group of nodes/devices.

Control and optimization of operation of wireless network usually involves tuning parameters of network devices being in proximity of each other, i.e. in wireless communication/interference/ sensing area. Examples are the radio channel and transmit power assignment to co-located Access Points in Wi-Fi networks. Hence, the control plane requires mechanism to ***discover*** the ***wireless devices*** in the network and their (wireless) relationship. Moreover, this information has to be monitored and updated at run-time. Having a global view of the entire wireless network enables control programs to efficiently manage and control of wireless devices. Changes in the ***network state*** can be detected in two ways, namely proactive and reactive. In a proactive approach, the network controller is periodically polling the network entities, while in a reactive approach the execution of control program functions is triggered by events generated by the nodes in the network. It should be up to the experimenter to define his preferred control strategy.

For ***coordinated control*** among multiple devices of different nodes, the framework API has to support time synchronized execution of functions across multiple network devices. Examples are the coordinated channel switching of multiple devices due to appearance of an interference source. While it is natural that the device programming interface is different for each wireless technology, in most cases it also varies across different implementation of the same technology, i.e. wireless devices of different vendors. The unification of the different Native Device Programming Interface (NDPI) is achieved by the introduction of the UPI_R and UPI_N interfaces which allow controlling the devices of a heterogeneous network in a unified way.

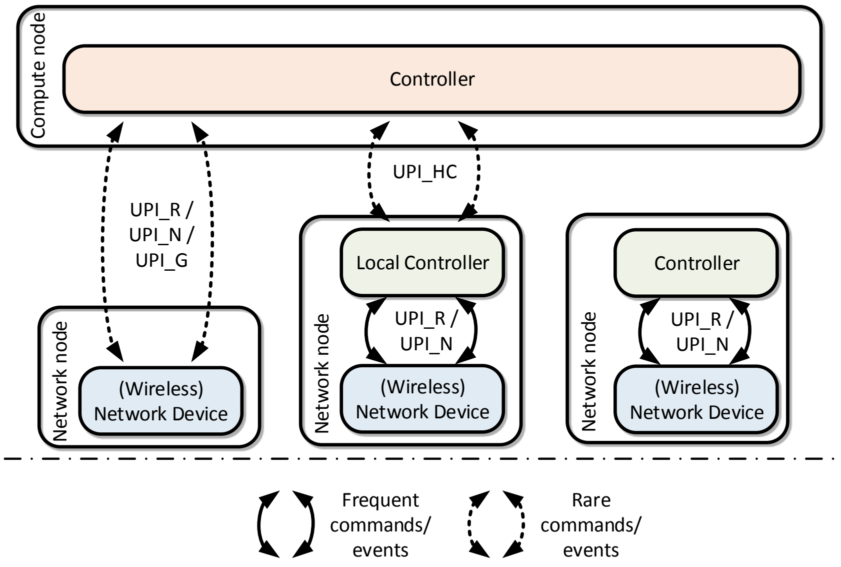

Figure 2. Levels of control in WiSHFUL control framework. Global controllers (left) handle rare events and commands while local controllers (right) are able to handle frequent commands and events. In hierarchical control (middle) there are two control loops, i.e. outer and inner.

In the general SDN concept the control plane is ***logically centralized*** enabling control programs to have a global view of the entire network. This approach simplifies the development of control programs significantly. However, from a practical point of view a centralized controller would introduce a significant delay in the control plane, which could in turn prevent time sensitive control logic to be implemented. Moreover, transporting all monitoring data from devices to a central node would create a high load on the control plane. Sometimes ***pre-processing data locally*** at the device is feasible. In this way, the control logic may be partitioned into smaller control programs where parts of them would run on the network nodes and others on the central compute node, i.e. hierarchical control. Another advantage of such a split is the possibility to reuse control programs. For example, an averaging filter may be implemented once as a control program and used as a local component in the implementation of more complex controllers in the future. Figure 2 shows the levels of control in the WiSHFUL control framework. Local control programs handle frequent commands and events, while global control programs handle rare events. There is also the possibility for hierarchical control where exchange of events between the global control programs and the local control programs is also rare.

In summary, the our control framework allows for running multiple control programs, which communicate with each other, and provides them with an interfaces for controlling all wireless network devices in a coordinated way.

2.3 Architecture Overview

WiSHFUL is a ***distributed middleware*** running across multiple nodes that interconnects all control programs having the ultimate goal to control wireless network devices. The control programs running on top of the middleware perform control tasks over wireless devices by utilizing the provided UPI_R, UPI_N and UPI_G interfaces. The WiSHFUL southbound interface is responsible for translating UPI calls coming from control programs to the devices. In the following subsections, we provide detailed description of the WiSHFUL control framework design and implementation.

2.3.1 Control Program

A control program is an entity that implements a particular network control logic. In general, it collects information and measurements from one or a group of network devices, makes control decisions according to set policies and performs network reconfiguration.

Each control program is provided with a ***global view*** of all nodes in the network. By default, a control program is able to control all device modules (using UPI_R and UPI_N interfaces) in the entire wireless network.

Control programs can register callback functions to get references to remote nodes (and eventually to devices) which are required when performing asynchronous execution of UPI_R and UPI_N functions or when registering for framework events, e.g. *new_node_callback* or *node_exit_callback.*

2.3.2 Interfaces and execution context

In order to access the WiSHFUL framework a network control program creates a controller instance. A global control program creates an instance of *Controller* whereas a local control program creates an *Agent* from which he is able to get a reference to *LocalController* using function call *get_local_controller()*. This controller object provides an access to the UPI interfaces – UPI_R, UPI_N, UPI_HC and UPI_G. Note, that in case of local control only access to the local UPI_R and UPI_N interfaces is possible, i.e. only control of local devices. For global controller the UPI_R and UPI_N interfaces are the same for devices located in the same host machine as well as for remote devices. It is the UPI_G that is responsible for delivering and executing function calls in proper device in order to achieve location transparency.

The UPI_R and UPI_N interfaces give a control program, global or local, the possibility to configure and monitor the supported protocol layers of a network device. The UPI_G contains also functions for network-wide operations, e.g. a function that estimates the nodes in carrier sensing range, which perform operations using the UPI_R and UPI_N interfaces on a set of multiple nodes/devices in a coordinated way. The UPI_HC interface is required for hierarchical control. It is used by the global control program to start/stop local control programs on remote nodes and to exchange user-defined messages with them.

A global control program may execute Remote Procedure Calls (RPC) on a set of remote network devices by calling *node()* or *nodes()* functions before calling the desired UPI_R or UPI_N function. By default, all RPC calls are blocking execution of control program until the function returns. For convenience, the *Controller (LocalController)* class provides three functions, namely: i) *delay(relative_time)*, ii) *exec_time(absolute_time)* and iii) *callback(cb=None)*. Using those functions one may delay execution of call, schedule execution of call in future point in time and execute non-blocking call, respectively. Optionally, it is possible to register a callback function to handle the result returned from the function call. Some examples of the supported calling semantics are presented in Listing 1.

Listing 1. Examples of supported UPI call syntax

```
#definition of callback function
def my_get_power_cb(data):
    print(data)

#execution of blocking call from local controller
res = controller.radio.iface("wlan0").get_power()

#execution of non-blocking call from local controller
controller.callback(print_response).radio.iface("wlan0").get_power()

#execution of blocking call from remote/global controller
controller.node("node_1").radio.iface("wlan0").get_tx_power()

#execution of non-blocking call from remote/global controller
controller.node("node_1").callback(print_response).radio.iface("wlan0").get
_tx_power()

#delay execution of non-blocking call from local controller
controller.delay(3).callback(my_get_power_cb).radio.get_tx_power()

#schedule execution of non-blocking call from local controller
t = datetime.now() + timedelta(seconds=3)
controller.exec_time(t).callback(my_get_power_cb).radio.get_tx_power()
```

In order to control a node, a control program has to first obtain its identifier (ID). This is achieved by registering a callback using *@controller.new_node_callback()* decorator. On discovery of a new node the framework notifies the control program about this by sending an event containing a node object with its ID. Those node IDs are required for remote UPI_R and UPI_N function. The framework keeps track of the presence of all nodes in the network and notifies control programs about node lost events.

2.3.3 Distributed Control Framework

The WiSHFUL control framework is a distributed middleware that inter-connects network devices and control program(s). The framework takes care of node management including node discovery and monitoring connection between all nodes. Whenever a new node is discovered or connection with some node is lost, the framework notifies all control programs about the changes by sending a proper event. While control programs running on top of the framework are able to control wireless devices with simple UPI_R/N calls, it is the framework that is responsible for delivering and executing function calls to/on the proper device. Moreover, the middleware is also responsible for discovering the UPI capabilities of each device, allowing to handle exceptions if an unsupported function is called. Using the WISHFUL control framework it is possible to develop global, local and hierarchical network control programs. In this way, the framework by design supports three types of control programs: i) local – when the control program is running in the same node as the controlled device(s), ii) non-local – when the control program is running on a different node then the controlled device(s) and iii) hybrid or hierarchical – when control logic is split between multiple control programs running on multiple nodes. A hierarchical control is a trade-off between local and global control. It allows putting time sensitive control functions close to device, and off-load complex tasks to remote more powerful nodes (servers in cloud). Note that the framework provides location transparency meaning that calling syntax is always the same for a local as well as a remote device.

2.3.4 Network Device Modules

The UPI_R/N interfaces work in two directions, i.e. control program may execute functions and change parameters of device (in-direction), but it may also receive data, measurements, samples, etc. from device (out-direction). All communication in in-direction is realized transparently either with local or remote calls, while communication in out-direction is realized using messages.

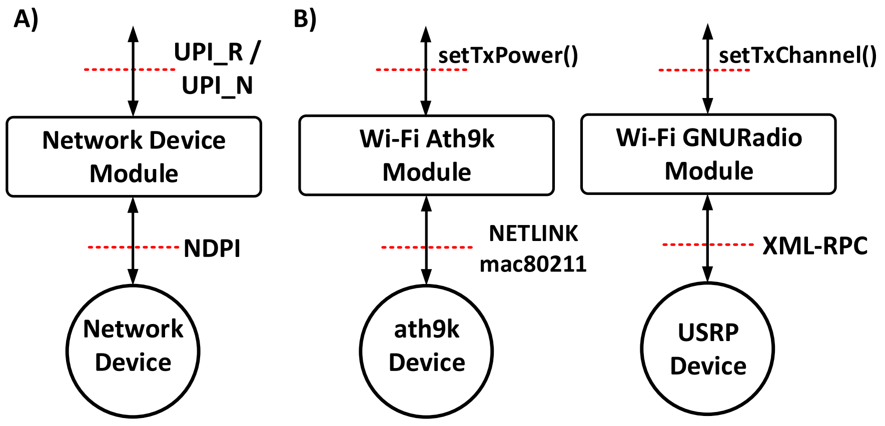

Figure 3. Device Module provides a unified interface, UPI_R and UPI_N, by wrapping the Native Device Programming Interface.

The device module translates function calls from control programs into Native Device Programming Interface (NDPI) –Figure 3A. In other words, device module wraps different API and tools used to program device and exposes them to WiSHFUL framework in a unified way as UPI_R and UPI_N. In Figure 3B, we present two device modules as an example. As shows the UPI_R and UPI_N functions are delivered to modules and translated to proper NDPI calls, i.e. NETLINK and XML/RPC respectively.

An example of function implementation is presented in Listing 2. Here, we consider the 802.11 WiFi device module. The *set_channel()* function takes channel as an argument and uses NETLINK interface to communicate with Linux 802.11 subsystem to configure the network device. We provide *bind_function* decorator to mask function names which can also be used to implement a unified abstraction layer. In example, the function is hidden behind proper operation from UPI definition. Note that the device connectors are Python objects and they keep state between consecutive function calls.

Listing 2. Example of wrapper that translates UPI function to NDPI function

```
@wishful_module.bind_function(upis.radio.set_channel)
def set_channel(self, channel):
    self.channel = channel
    # set channel in wireless interface using NDPI (e.g. NETLINK)
    # .......
    return reponse
```

2.4 Related Work

Related work falls into three categories that are described in following subsections:

2.4.1 Cross-layer Control

CRAWLER [5][6] is experimentation architecture for centralized network monitoring and cross-layer coordination over different devices. ClickWatch [7] aims for simplification of experimentation of wireless cross-layer solutions implemented using the Click Modular Router. Both frameworks aim to facilitate experimentation and offer possibility to control all nodes in the network from a single centralized controller. In contrast, WiSHFUL is more flexible as it allows distributing controller logic over multiple nodes so that time sensitive control logic can be located and executed on the network node to be controlled.

2.4.2 Software-defined Networking

There are already lots of distributed control frameworks, but they are mostly focused on control of wired switches using open protocols (e.g. OpenFlow[8]). Some of them, like ONOS [9] and ONIX [10] are focused on scalability and performance. As they are already in very advanced state, it is hard to use them for resource constrained devices or to adjust them to wireless networking. Ryuo [11] and Kandoo [12] provide the possibility for offloading of control programs to local controllers as a way to limit the control plane load. Local controllers handle frequent events, while a logically centralized root controller handles rare events. Beehive [13][14] provides interesting features like automatic distribution of network applications over network nodes. While having similar concepts, Beehive does not differentiate between control programs and device modules, which are of great importance when targeting the control of heterogeneous wireless networks.

CoAP [15] proposes a vendor neutral centralized framework for configuration, coordination and management of residential 802.11 APs using an open API. In contrast to WiSHFUL the CoAP API is restricted to control of 802.11 networks. Moreover, only centralized control programs are possible. OpenRF [16] provides programming abstractions tailored for wireless networks, i.e. MIMO interference management techniques that impact the physical layer. OpenRF is restricted to centralized control of 802.11 infrastructure networks. Finally, in [17] SDN architecture for centralized spectrum brokerage in residential infrastructure Cognitive Radio networks was proposed.

2.4.3 General Distributed Control Frameworks

ROS [18] is an open source robot operating system for rapid prototyping. ROS is focused on providing control for a single robot, trying to achieve one goal, and having all devices working towards that goal. In WiSHFUL, we are trying to achieve harmonization of multiple devices. Moreover, we also provide time scheduled execution of operations on multiple and possible heterogeneous devices.

3 UPI_N WiSHFUL functions and attributes

This section provides the UPI_N interface, it is responsible of the higher layers reconfiguration of the network protocol stack of a particular wireless node, in terms of network capability, i.e. set traffic flows. According to the WiSHFUL architecture design described in D4.2, the UPI_N functions are called exactly in the same way over different hardware platforms (which will be called simply platforms for the rest of the document), and they are organized into the following functional groups:

- Address management
- Protocol attribute manipulation
- Traffic control
- Topology detection and routing control

According with the presentation of WiSHFUL modules for the UPI_R provided in deliverable D3.4, we also exploit the platform module concept and provide a set of modules able to implement UPI_N functions and be loadable in a given wireless node. Three module have been produced to implement all the UPI_N functions: **module_net_linux** and **module_iperf** for the Linux OS; and **module_net_contiki** for the Contiki OS. Also for the UPI_N interface, the complete list of loaded modules with their functions for each node is reported to the global control program by the agent nodes. An experimenter is able to use an UPI_N function inside the control program, only if the module that contains it is loaded. WiSHFUL UPI_N implementation supports two platforms, namely the **linux networking subsystem** and **Contiki embedded OS**. For the Contiki OS, two different network layer protocols are supported (Rime and IPv6).

The remainder of this section is organized as follows: first the UPI_N functions, supported by all platform types are listed for each functional group. Second, the UPI_N functions supported by a subset of platforms are described for the relevant functional groups and platform types. The same organization is used for the UPI attributes in the following two sections. Finally an example is given, together with a summary of all available UPI_N functions and a list of candidate UPI_N functions and attributes.

3.1 UPI_N functions supported by all platform types

The UPI_N functions presented in this section are implemented by all platform types (i.e. Linux OS and Contiki OS based hardware platforms).

3.1.1 Address Management

This subsection presents the generic UPI_N functions for address management - Table 1. They allow getting or setting the IP address and hardware address per interface. They are implemented by the *module_net_linux* and *module_net_contiki*.

Table 1. UPI_N functions for address management

Function	Description
get_iface_hw_addr	Returns the hardware address (MAC address) of a given interface
set_iface_ip_addr	Sets the IP address of the interface.
get_iface_ip_addr	Returns the IP address of a given interface.

3.1.2 Functions to manipulate protocol attributes

This subsection presents the generic UPI_N functions that allow manipulating UPI_N attributes (Table 2), i.e. set/get network level configuration parameters, read network level measurements and subscribe network level events. They are implemented by the *module_net_linux* and *module_net_contiki*.

Table 2. UPI_N functions for protocol attributes manipulation

Function	Description
set_parameters	This UPI_N function is able to configure the higher layer protocols (routing, transport, application) behavior by changing parameters. Parameters correspond to the variables used in the protocols.
get_parameters	Get the parameter on higher layers of protocol stack
get_measurements	This UPI_N function is able to retrieve the measurements maintained by higher layer protocols.
get_measurements_periodic	This UPI_N function is able to retrieve the measurements maintained by higher layer protocols in a periodic manner.
subscribe_events	Allows to subscribe a local event listener that is triggered each time the subscribed event(s) occur(s).
get_network_info	This UPI_N function retrieves the network layer information.

3.1.3 Traffic control

In this section the UPI_N functions for controlling traffic during an experiment are listed - Table 3. Using these UPIs an experimenter is able to start or stop applications, change the application data rate and create or remove packet flows. The experimenter can also retrieve traffic results in terms of throughput and delta time information in real time during the traffic session. They are implemented by the *module_net_linux*, *module_net_iperf* and *module_net_contiki*.

Table 3. UPI_N functions for traffic control

Function	Description
install_application	Install application in a node
start_application	Start previously installed application in node
stop_application	Stop previously installed application in node
create_packetflow_sink	Create a sink that is able to receive packet flows from different nodes.
destroy_packetflow_sink	Destroy a packet flow sink.
start_packetflow	Start a packet flow towards the specified sink.
stop_packetflow	Stop an existing packet flow towards the specified sink.
register_packetflow_logging	Register on packet flow sink in order to receive logging message with traffic information result.

The UPI_N functions presented in this section provide a complete tool for performing network throughput measurements. It can test either TCP or UDP throughput. To perform an traffic session, the experimenter must establish both, server (sink) and client (packetflow source) side, the logger is activated in the server node and provides a service to notify traffic information results. The Table 4 shows the complete list of parameters available for the traffic control functions and their description.

Table 4. Parameter UPI_N functions for traffic control functions

Parameters	Side	Description
port	server/client/logger	Server port to listen on/connect to
logging_interval	server	Seconds between periodic throughput result reports
use_udp	server/client	Use UDP rather than TCP
bind_interface	server/client	Bind to <host>, an interface or multicast address
dest_ip	client	Connecting to <host>
time_duration	client	Time in seconds to transmit the traffic
bandwidth	client	For UDP, bandwidth to send at in bits/sec
frame_length	client	Length of buffer to read or write (default 8 KB)
sink_ip_address	Server/logger client	Connecting to <host> to receive logging message
filter_ip_address	logger client	Filter the logging message for a specific node information

When one or more traffic sessions are present, the UPI_N *register_packetflow_logging()* provides a method to collect the traffic session results in real time. When the UPI_N *create_packetflow_sink()* is performed, a logging service is activated on sink node. From this moment, every control program can use the UPI_N *register_packetflow_logging()* to connect to the logging service present on the Sink node. Afterwards, every logging_interval time the registered node receives the throughput results in terms of throughput and delta value in real time. For each notification the relative UPI_N callback is performed. Figure 4 shows a scenario in which the WiSHFUL UPI_N traffic control is used with the logging service enabled. Two traffic sessions are active from Node0 and Node1 to the Sink node. The control program uses the UPI_N *register_packetflow_logging()* to retrieve the traffic results for both traffic sessions.

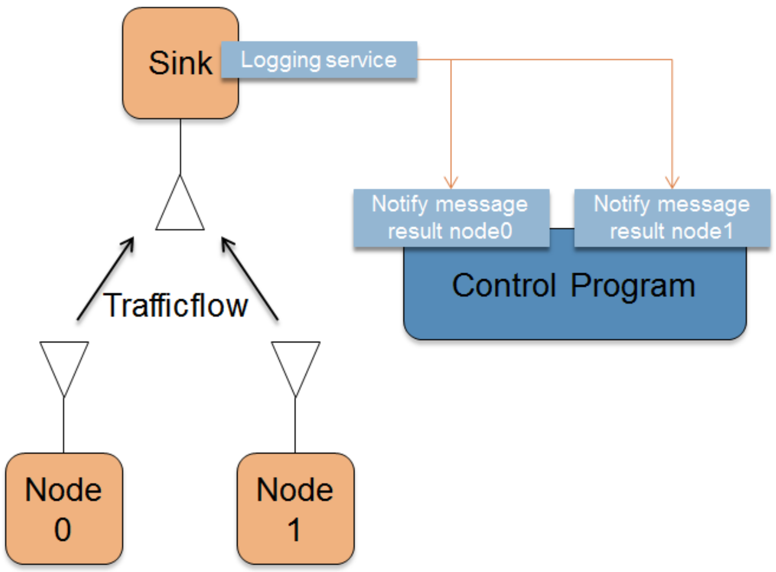

Figure 4. Architecture of the logging service for UPI_N traffic control

In , we present an example of a control program that implements the above scenario, a packetflow sink is enabled on ap_node, and two traffic sessions are activated from node0 and node1 to ap_node. Afterwards, the *register_packetflow_logging()* function is called two times in order to enable logging for both of the nodes. For each call a different callback function is defined. Both of the callback functions process the logging result coming from the nodes and then they append the throughput and the delta values in the following four arrays: i) throughput_results_node0, ii) delta_results_node0, iii) throughput_results_node1, iv) delta_results_node1. These arrays can be used to plot the throughput trend in real time, or to store all the values in a data base for post processing.

Listing 3. Example of traffic sessions activation and real time throughput result acquisition

```
#callback implementation to store traffic information (throughput and delta
 time) for node 0
throughput_results_node0 = [ ]
delta_results_node0 = [ ]
def collect_traffic_logging_messages_node0(group, node, data):
    log.debug('receives data msg at %s -
 %s' % (str(node.ip), str(data) ))
    throughput_results_node0.append(data['throughput'])
    delta_results_node0.append(data['delta'])

#callback implementation to store traffic information (throughput and delta
 time) for node 1
throughput_results_node1 = [ ]
delta_results_node1 = [ ]
def collect_traffic_logging_messages_node1(group, node, data):
    log.debug('receives data msg at %s -
 %s' % (str(node.ip), str(data) ))
    throughput_results_node1.append(data['throughput'])
    delta_results_node1.append(data['delta'])
….
#start server traffic on node ap
controller.nodes(ap_node).net.create_packetflow_sink(port='1234')
#start client traffic on node 0
controller.nodes(node0).net.start_packetflow( ap_node.ip, port='1234', time
 _duration='500', bandwidth ='1M')
#start client traffic on node 1
```

```
controller. .nodes(node1).net.start_packetflow( ap_node.ip, port='1234', time_duration='500', bandwidth ='3M')
….
#start  logging to collect traffic throughput results on node 0
controller.callback(collect_traffic_logging_messages_node0).register_packetflow_logging(port='8301', sink_ip_address=ap_node.ip, filter_ip_address=node0.ip):
#start  logging to collect traffic throughput results on node 1
controller.callback(collect_traffic_logging_messages_node1).register_packetflow_logging(port=''8301', sink_ip_address=ap_node.ip, filter_ip_address=node1.ip):
….
#start client traffic on node 0
controller.nodes(node0).net.stop_packetflow()
#start client traffic on node 1
controller.nodes(node1).net.stop_packetflow()
#destroy packet flow sink on ap node
controller.nodes(ap_node).net.destroy_packetflow_sink()
```

3.1.4 Topology detection and routing control

UPI_N supports basic functions for enabling both passive topology detection (e.g. by inspecting routing and neighbour table) and active topology detection by generating/sniffing probing frames. Also included in this group are functions to add/remove entries in the routing and neighbour table, allowing to control the routing behavior. The functions listed in Table 5 are implemented by the *module_net_linux* and *module_net_contiki*.

Table 5. UPI_N functions for topology detection and routing control

Function	Description
get_route_table	Get the current routes from the route table.
clear_route_table	Clear the current routes in the route table.
add_route	Add a route in the route table.
remove_route	Remove a route from the route table.
get_neighbor_table	Get the discovered neighbors from the neighbor table.
clear_neighbor_table	Clear the discovered neighbors in the neighbor table.
add_neighbor	Add a neighbor in the neighbor table.
remove_neighbor	Remove a neighbor from the neighbor table.
gen_layer2_traffic	Inject layer2 traffic into network device
inject_frame	Inject L2/L3 frame into the protocol stack
sniff_layer2_traffic	Layer-2 packet sniffing from network device

3.2 UPI_N functions supported in Linux OS

3.2.1 Address Management

This subsection presents the additional UPI_N functions for address management using the Linux networking subsystem - Table 6. They are implemented by the *module_net_linux* and allow manipulating the ARP table.

Table 6. UPI_N functions for Address Management supported in Linux OS

Function	Description
get_ARP_entry	Returns an entry from the ARP cache
set_ARP_entry	Manipulates the entries in the ARP cache

3.2.2 Traffic control

In this section the UPI_N functions are listed that are in place for fine-grained configuration of traffic control in the Linux networking subsystem. Using these UPIs an experimenter is able to apply traffic shaping and prioritize flows. Also UPI_N functions for managing Queuing Disciplines and for wireless link emulation (in terms of throughput, delay, etc.) in wired networks are offered in this group. The UPI_N functions listed below are implemented in module *module_net_linux*.

3.2.3 Support for management of Queueing Disciplines

The provided UPI_N functions for configuration of queueing disciplines follows an object-oriented approach and is listed in Table 7, they are implemented in module *module_net_linux*. It gives an experimenter a user-friendly way for managing the QDisc[19] for each interface in System-Under-Test (SUT) nodes.

Table 7. UPI_N functions for management of Queueing Disciplines supported in Linux OS

Function	Description
install_egress_scheduler	Install Egress Scheduler in given network interface.
remove_egress_scheduler	Remove Egress Scheduler from network interface

An example of a configuration of QDisc is presented in Listing 4. First, a root scheduler has to be created. Second, queues are created and added to shaper. Finally, the **install_egress_scheduler*()*** function is called to send QDisc configuration to remote node, which will apply it on the specified interface. A Qdisc configuration is installed using Netlink calls to the kernel traffic-control subsystem. In Listing 5, an example of using a UPI function to delete an egress scheduler is presented.

We implemented a Python package, called python-tc that is used to : i) create QDisc configuration in object-oriented way; *ii)* agent to install this configuration on a specified interface. Currently supported queuing disciplines: *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 queuing disciplines. Traffic control can be performed on both the ingress and egress interfaces.

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.

Listing 4. 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*1
024))

#Define filters
filter1 = Filter(name="BnControlTraffic");
filter1.setFiveTuple(src=None, dst='192.168.1.178', prot='udp', srcPort=Non
e, 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', ds
tPort=None)
filter4.setTarget(tbf4)
prioSched.addFilter(filter4)

#Install defined scheduler in node0
controller.install_egress_scheduler(node0, 'wlan0', prioSched)
```

Listing 5. Deletion of egress scheduler in SUT node

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

3.2.4 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. The complete list of the UPI_N for managing the traffic emulation is presented in Table 8, these UPI_N are implemented in module *module_net_linux*.

Table 8. UPI_N function for network emulation supported in Linux OS

Function	Description
set_netem_profile	Set emulation profile in given network interface
update_netem_profile	Update emulation profile in given network interface
remove_netem_profile	Remove emulation profile from given network interface.
set_per_link_netem_profile	Set emulation profile in network interface for given link identified with destination MAC address
update_per_link_netem_profile	Update emulation profile in network interface for given link identified with destination MAC addresses.

remove_per_link_netem_profile	Remove emulation profile from network interface for given link identified with destination MAC addresses.

An experimenter is able to define parameters of the 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 the configuration of an emulated link. A profile is a description of link characteristics.

In Listing 6, we present an example of configuration of a link profile and the use of **set_netem_profile*()*** to apply it to a specified interface in SUT node. In Listing 7 we show how to update an already existing profile using **update_netem_profile**(). Finally, in Listing 8 it is shown how to remove a profile using **remove_netem_profile**() function.

In order to emulate link characteristics in wired network, we use a 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.

Listing 6. Example of the configuration of an emulated link

```
#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
controller.set_netm_profile(node0, 'eth0', profile4G)
```

Listing 7. 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
controller.update_netm_profile(node0, 'eth0', profile4G)
```

Listing 8. Deletion of emulation profile

```
#Remove emulation profile from interface eth0 of node0
controller.remove_netm_profile(node0, 'eth0')
```

3.2.5 Packet filtering and manipulation

In this section we provide the UPI_N function for packet filtering and manipulation. We provide an object-oriented approach for manipulation of *iptables* [20], packet marking and setting Type-of-

Service value. The complete list of the UPI_N for managing the packet filtering and manipulation is presented in Table 9, these UPI_N are implemented in module *module_net_linux.*

Table 9. UPI_N functions for packet filtering and manipulation

Function	Description
clear_nf_tables	Clear all entries in all iptables
get_nf_table	Get specific iptable and its entries
set_pkt_marking	Add iptable rule for marking all packets belonging to a flow identified with the given 5-tuple
del_pkt_marking	Remove rule used to mark given flow from iptable
set_ip_tos	Add iptable rule for setting TOS (Type-of-Service) field in all packets.
del_ip_tos	Remove rule used to set TOS field from iptable

In Listing 9, an example is presented where we use the implemented UPI function to mark flows.

Listing 9. 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;
controller.setMarking(node0,flowDesc, markId=5, table="mangle", chain="INPUT")

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

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

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

In our implementation we used *python-iptables*[21] package, an object-oriented library that provides wrapper via python bindings to *iptables,* in the Linux operating system. The advantage of this library is that it does not call *iptables* binary nor parse its output, but it interfaces directly to the C-based libraries *(libiptc, libxtables). It results in lower latencies and higher flexibility.*

3.3 UPI_N functions supported in Contiki OS

Currently there no UPI_N functions defined which are specific for Contiki platforms.

3.4 UPI_N attributes supported on all platforms

3.4.1 Traffic control

The currently supported generic UPI_N attributes are listed in Table 10. All, except one, operate on the application layer. They are implemented by the *module_net_linux*, *module_net_iperf* and *module_net_contiki*.

Table 10. UPI_N attributes for traffic control

Attribute Name	Type	Description
APP_DATA_RATE	Parameter	Configures the data rate of the application
APP_MSG_SIZE	Parameter	Configures the application message size.
APP_MSG_DESTINATION	Parameter	Configures the application destination.
APP_PER_PACKET_RX_STATS	Event	Event triggered each time a packet is received.
APP_PER_PACKET_TX_STATS	Event	Event triggered each time a packet is transmitted.
APP_STATS	Measurement	Cumulative application statistics.
IP_STATS	Measurement	Cumulative IP layer statistics.

3.5 UPI_N attributes supported in Linux OS

All UPI_N attributes supported in Linux OS are accessed using proper setter and getter functions described in section 3.2. For example, we use *set_iface_ip_addr* and *get_iface_ip_addr* functions to set and get IP address attribute.

3.6 UPI_N attributes supported in Contiki OS

The currently supported control attributes are mainly focusing on the Contiki IPv6 stack, more specifically in the RPL routing protocol [], as this is the main routing protocol that is standardized for wireless sensor networks today and is set as the target for Y2 showcases. Other control attributes from different (sub) layers such as CoAP [15], 6LowPan, IPV6 Neighbour Discovery, etc. can however be added with minimal effort. Candidate attributes are listed in Section 3.8.1. The candidates were chosen after carefully examining the relevant standards.

3.6.1 Topology detection and routing control

As stated, Table 11 lists parameters specific to the RPL routing protocol.

Table 11. UPI_N attributes for topology detection and routing control supported in Contiki OS

Attribute	Type	
RPL_DIO_INTERVAL_MIN	Parameter	The value used to configure Imin for the DIO Trickle timer. The default value is 3. This configuration results in Imin of 8 ms.
RPL_DIO_INTERVAL_DOUBLINGS	Parameter	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.
RPL_DIO_REDUNDANCY_CONSTANT	Parameter	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.
RPL_DEFAULT_LIFETIME_UNIT	Parameter	Default route lifetime unit. This is the granularity of time used in RPL lifetime values, in seconds.
RPL_DEFAULT_LIFETIME	Parameter	Default route lifetime as a multiple of the lifetime unit.
RPL_MIN_HOP_RANK_INCREASE	Parameter	The value of MinHopRankIncrease. The default value is 256. This configuration results in an 8-bit wide integer part of Rank.
RPL_OBJECTIVE_FUNCTION	Parameter	Updates the objective function used to for link estimation and path cost calculation.
RPL_DAG_LIFETIME	Parameter	Maximum lifetime of a DAG. When a DODAG is not updated since RPL_CONF_DAG_LIFETIME times the DODAG maximum DIO interval the DODAG is removed from the list of DODAGS of the related instance, except if it is the currently joined DODAG.
RPL_PROBING_INTERVAL	Parameter	RPL probing interval. Probes will be sent periodically to keep parent link estimates up to date.
RPL_DIS_START_DELAY	Parameter	Added delay of first DIS transmission after boot.
RPL_DIS_INTERVAL	Parameter	Interval of DIS transmission.
RPL_STATS	Measurement	Statistics gathered during RPL operation.

3.7 UPI_N function list

In this section, we report the definitive list of the UPI_N functions supported by WiSHFUL, implemented to perform network related actions. The last two columns indicate if they are implemented for the Contiki OS, Linux OS or both.

Table 12. Support for UPI_N functions in Linux and Contiki OS

Function	Type	Contiki	Linux
set_parameters	Generic	X	X
get_parameters	Generic	X	X
get_measurements	Generic	X	X
get_measurements_periodic	Generic	X	X
subscribe_events	Generic	X	X
get_network_info	Generic	X	X
get_iface_hw_addr	Network address management	X	X
set_ip_address	Network address management	X	X
get_iface_ip_addr	Network address management	X	X
set_ARP_entry	Network address management		X
get_ARP_entry	Network address management		X
install_application	Traffic control		X
start_application	Traffic control	X	X
stop_application	Traffic control	X	X
create_packetflow_sink	Traffic control	X	X
destroy_packetflow_sink	Traffic control	X	X
start_packetflow	Traffic control	X	X
stop_packetflow	Traffic control	X	X
get_route_table	Topology detection and routing control	X	X
clear_route_table	Topology detection and routing control	X	X
add_route	Topology detection and routing control	X	X
remove_route	Topology detection and routing control	X	X
get_neighbor_table	Topology detection and routing control	X	X
clear_neighbor_table	Topology detection and routing control	X	X
add_neighbor	Topology detection and routing control	X	X
remove_neighrour	Topology detection and routing control	X	X
gen_layer2_traffic	Topology detection and routing control	X	X

inject_frame	Topology detection and routing control	X	X
sniff_layer2_traffic	Topology detection and routing control	X	X
install_egress_scheduler	Traffic control queuing disciplines		X
remove_egress_scheduler	Traffic control queuing disciplines		X
set_netem_profile	Traffic control emulation		X
update_netem_profile	Traffic control emulation		X
remove_netem_profile	Traffic control emulation		X
set_per_link_netem_profile	Traffic control emulation		X
update_per_link_netem_profile	Traffic control emulation		X
remove_per_link_netem_profile	Traffic control emulation		X
clear_nf_tables	Traffic control Packet filter		X
get_nf_table	Traffic control Packet filter		X
set_pkt_marking	Traffic control Packet manipulation		X
del_pkt_marking	Traffic control Packet manipulation		X
set_ip_tos	Traffic control Packet manipulation		X
del_ip_tos	Traffic control Packet manipulation		X

3.8 Candidate UPI_N extensions

This subsection gives a none-exhaustive list of the candidate UPI_N extensions that can be added in year 3 of the project, on a per need basis, to the current set of UPI_N functions and attributes. However, if an experimenter, either within the WiSHFUL consortium, open call partner or a third-party collaborator, requires another UPI_N extension, it will be integrated with higher priority if it is feasible. Moreover, for each of the candidate UPI_N extensions, the applicability on all platforms remains to be inspected. Hence, the subdivision made in this subsection between UPI_N functionality that is supported on all platforms or only supported on a subgroup of platforms can be different in a future implementation.

3.8.1 Candidate control functions

The possible UPI_N control attributes that can be added in year 3 of the project are listed in the following subsections.

3.8.1.1 Traffic control

The networking subsystem in both the Linux and Contiki support TCP and UDP communication. Hence it should be possible to offer a unified set of control attributes that allow to monitor and change the behaviour of TCP connections or UDP streams. Note however that Contiki only supports a limited number of TCP and UDP sockets due to memory restrictions. This should be taken into account when implementing the candidate UPI_N attributes listed in Table 13.

Table 13. List of candidate UPI_N functions for traffic control

Attribute	Type	Description
IPv6_TIME_TO_LIVE	Parameter	The IP TTL (time to live) of IP packets.
IPv6_REASSEMBLY_MAXAGE	Parameter	Maximum time an IP fragment should wait in the reassembly buffer before it is dropped.
TCP_INITIAL_RETRANSMISSION_TIMEOUT	Parameter	The initial retransmission timeout for TCP segments.
TCP_MAX_SEGMENT_RETRANSMIT	Parameter	Maximum number of segment retransmission before the connection is aborted.
TCP_MAX_SYN_RETRANSMIT	Parameter	Maximum number of SYN segment retransmissions before a connection request is considered unsuccessful.
TCP_TIME_WAIT_TIMEOUT	Parameter	Dictates how long a connection should stay in the TIME_WAIT state.
TCP_RECEIVE_WINDOW	Parameter	The size of the advertised receiver's window.
TCP_ SOCKET_STATS	Measurement	Cumulative statistics for a TCP socket.
UDP_SOCKET_STATS	Measurement	Cumulative statistics for a UDP socket.
TCP_CLIENT_CONNECTED	Event	Triggered when a client connects to a TCP server.
UDP_CLIENT_CONNECTED	Event	Triggered when a client connects to an UDP server.

3.8.1.2 Topology detection and routing control

Neighbour discovery is a very important aspect of routing protocols in multi-hop wireless networks. The ICMPv6 protocol offers numerous configuration parameters that can be tweaked to optimize the neighbor discovery process. Both Linux and Contiki support the ICMP protocol, albeit that in Contiki only a subset of attributes is available. Table 14 lists the control attributes related to neighbour discovery that are currently supported as compile time constants in Contiki. With minimal effort they can be exposed as UPI_N attributes.

Table 14. List of candidate UPI_N functions for topology detection and routing control

Attribute	Type	Description
ND_MAX_RTR_SOLICITATION_DELAY	Parameter	Interval between router solicitations.
ND_MAX_RTR_SOLICITATIONS	Parameter	Max number of consecutive router solicitations.

ND_MAX_RTR_ADVERTISEMENT_INTERVAL	Parameter	Max interval between router advertisements.
ND_MIN_RTR_ADVERTISEMENT_INTERVAL	Parameter	Min interval between router advertisements.
ND_RTR_LIFETIME	Parameter	Lifetime of a router advertisement.
ND_MAX_INITIAL_RTR_ADVERTISEMENT_INTERVAL	Parameter	Max interval between initial router advertisements.
ND_MAX_INITIAL_ RTR_ADVERTISEMENTS	Parameter	Max number of initial router advertisements.
ND_MIN_RTR_ADVERTISEMENTS_DELAY_TIME	Parameter	Max delay between router advertisements.
ND_MAX_ RTR_ADVERTISEMENT_DELAY_TIME	Parameter	Max delay between router advertisements.
ND_NEIGHBOR_DISCOVERED	Event	Triggered when a neighbor is discovered.

3.8.2 Candidate control attributes for Contiki OS

The following candidate attributes are specific to protocols inside the Contiki OS.

3.8.2.1 Traffic control

CoAP [22] and 6LowPan [23] are two widely adopted standards in wireless sensor networks. Therefore, it makes sense to provide experimenters the possibility to change the behaviour of these protocols. Table 15l ists the possible control attributes.

Table 15. List of candidate UPI_N attributes for traffic control

Attribute	Type	
6LOWPAN _PACKET_REASSEMBLY_MAXAGE	Parameter	Timeout for packet reassembly at the 6lowpan layer.
COAP_ACK_TIMEOUT	Parameter	Timeout for CoAP ACK (2 seconds)
COAP_ACK_RANDOM_FACTOR	Parameter	Randomness factor to overcome synchronization effects (1.5)
COAP_MAX_RETRANSMIT	Parameter	Maximum number of CoAP request retransmissions (4)
COAP_NSTART	Parameter	Maximum simultaneous connections between CoAP clients and servers (1)
COAP_DEFAULT_LEISURE	Parameter	Leisure time before responding to a multicast requests (5 seconds)
COAP_PROBING_RATE	Parameter	Rate in which probes can be send for reacting to unacked CoAP requests (1 byte/second)

4 Additional UPI interfaces

This section presents the status of the rest of the defined UPI interfaces, i.e. UPI_G for global control, UPI_M for management and UPI_HC for hierarchical control.

4.1 Global control using UPI_G

UPI_N and UPI_R functions consist of a set of primitives, which are executed locally on a wireless node. Exploitation of a UPI function can be done directly on the node itself (using the Local Monitoring and Control Engine MCE) or, under the coordination of the global controller (using the Global MCE). In this second case we exploit the global UPI (*UPI_G) to run it.* The UPI_G is required for ***coordinated*** (time synchronized) ***remote execution*** of configuration and monitoring related functions on a single node or a group of nodes. The UPI_G is responsible for relaying issued UPI_N/R function calls to intended node(s). In our implementation the UPI_G is a part of Controller object. The experimenter can execute any UPI_N/R function on a selected group of nodes.

Within UPI_G, WiSHFUL also defines functions with more complex behaviour that operate on a heterogeneous group of nodes and combine several UPI_R/N functions to create a network-wide view on the node topology. Table 16 lists the UPI_G functions that are currently available.

Table 16. List of functions in UPI_G interface

Function	Description
estimate_nodes_in_carrier_sensing_range	Estimates which nodes are in carrier sensing range and which not
is_in_carrier_sensing_range	Estimates if a node is in carrier sensing range
estimate_nodes_in_communication_range	Estimates which nodes are in communication range and which not
is_in_communication_range	Estimates if a node is in communication range

4.2 Management using UPI_M

All management related functions are grouped in the UPI_M interface because they are required for managing protocol software modules at any layer. Moreover, software management requires functionality on both the local and global level. UPI_M is responsible for deploying, installing and activating software packages. For example, we use the UPI_M interface to deploy the radio program on platform. Table 17 presents the UPI_M provided by WiSHFUL.

Table 17 . List of function in UPI_M interface

Function	Description
send_radio_program	This function allows to send a radio program to one or more nodes.
send_execution_engine	This function allows to send the execution engine to one or more nodes.

4.3 Hierarchical control using UPI_HC

The local control program consists of a piece of software implementation, which runs locally. The implementation of the local controller can be done directly on the node itself, under the coordination of the global controller. In this last case, it can be defined on the global controller, and then it is remotely sent and run to remote wireless node through the Hierarchical UPI functions (UPI_HC) provided by WiSHFUL. Hierarchical UPI functions allowing the simultaneous use of local and global control programs. 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. The Table 18 presents the UPI_HC provided by WiSHFUL.

Table 18. List of functions in UPI_HC interface

Function	Description
start_local_control_program	Execute a given control program on local/remote node
stop_local_control_program	Stops execution of a given control program on local/remote node
send_msg_to_local_control_program	Hierarchical control function allows the global control program to send messages to local control programs.
send_upstream	Local control program sends message in user-defined format to global control program.

5 Details of WiSHFUL control framework 2.0 for Linux OS

The WiSHFUL control framework was prototypically implemented. Particular attention was paid to enhance code re-usability and support for different programming languages as well as enabling the use of specialized external software libraries.

The main prototype is implemented in ***Python language***, that makes it possible to run on multiple different host types (Linux, OpenWRT, Mac OS and Windows) and allows for rapid prototyping of control programs. An overview of the implementation is presented in Figure 5. As we used only standard and common Python libraries, we are able to run and test our implementation on multiple platforms, including x86, ARM and MIPS. In order to also support constrained devices, a lightweight C version of the agent-side of the framework was also implemented in Contiki – see Section 0.

In order to support delayed and time-scheduled function execution, the Agent class is equipped with a scheduler (Python Apscheduler[24]). Note that when coordinating multiple nodes by means of time scheduled execution, the nodes in the network must have common notion of a global clock (e.g. obtained through use of GPS or time protocols like PTP or NTP).

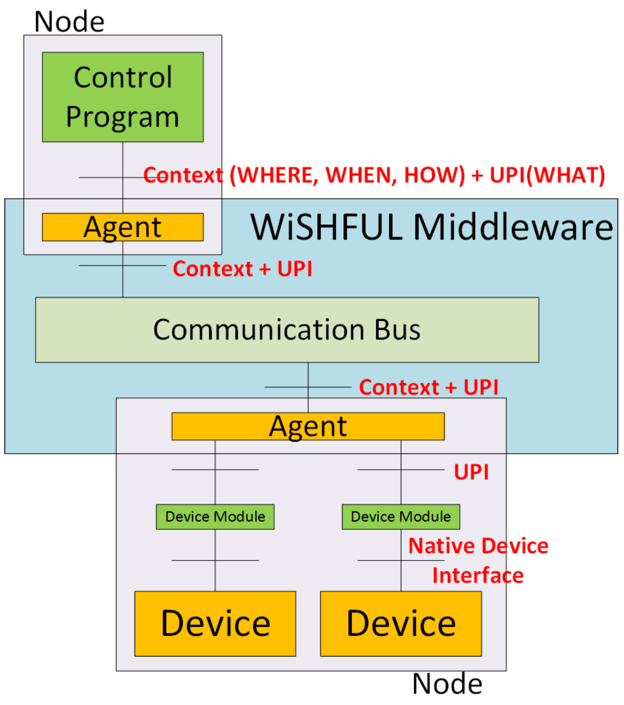

Figure 5. Implementation overview of WiSHFUL control framework

5.1 WiSHFUL Control Framework Class Diagram

The UML diagram in Figure 6 presents the ***WiSHFUL Control Framework*** in more detail. It shows the interface description as well as the most important components.

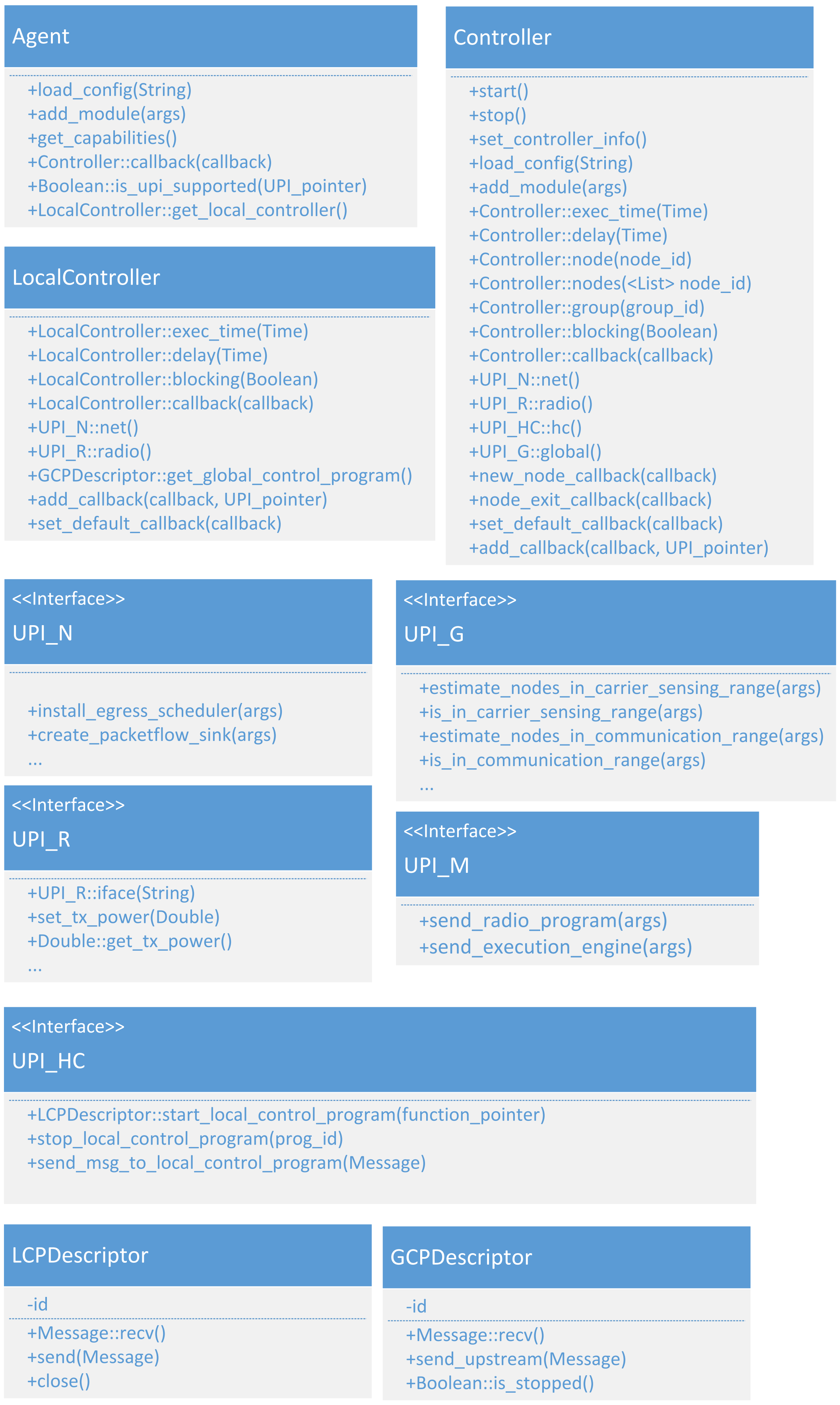
Agent
+load_config(String)
+add_module(args)
+get_capabilities()
+Controller::callback(callback)
+Boolean::is_upi_supported(UPI_pointer)
+LocalController::get_local_controller()
Controller
+start()
+stop()
+set_controller_info()
+load_config(String)
+add_module(args)
+Controller::exec_time(Time)
+Controller::delay(Time)
+Controller::node(node_id)
+Controller::nodes(<List> node_id)
+Controller::group(group_id)
+Controller::blocking(Boolean)
+Controller::callback(callback)
+UPI_N::net()
+UPI_R::radio()
+UPI_HC::hc()
+UPI_G::global()
+new_node_callback(callback)
+node_exit_callback(callback)
+set_default_callback(callback)
+add_callback(callback, UPI_pointer)
LocalController
+LocalController::exec_time(Time)
+LocalController::delay(Time)
+LocalController::blocking(Boolean)
+LocalController::callback(callback)
+UPI_N::net()
+UPI_R::radio()
+GCPDescriptor::get_global_control_program()
+add_callback(callback, UPI_pointer)
+set_default_callback(callback)
<<Interface>>
UPI_N
+install_egress_scheduler(args)
+create_packetflow_sink(args)
...
<<Interface>>
UPI_G
+estimate_nodes_in_carrier_sensing_range(args)
+is_in_carrier_sensing_range(args)
+estimate_nodes_in_communication_range(args)
+is_in_communication_range(args)
...
<<Interface>>
UPI_R
+UPI_R::iface(String)
+set_tx_power(Double)
+Double::get_tx_power()
...
<<Interface>>
UPI_M
+send_radio_program(args)
+send_execution_engine(args)
<<Interface>>
UPI_HC
+LCPDescriptor::start_local_control_program(function_pointer)
+stop_local_control_program(prog_id)
+send_msg_to_local_control_program(Message)
LCPDescriptor
-id
+Message::recv()
+send(Message)
+close()
GCPDescriptor
-id
+Message::recv()
+send_upstream(Message)
+Boolean::is_stopped()

Figure 6. WiSHFUL control framework (UML class diagram).

5.2 Deployment

We adopted *yaml* format for preparation and storage of the configuration of the Agent (class that implements local MCE from WiSHFUL architecture) and the Controller (class that implements global MCE from WiSHFUL architecture). Such a configuration file is loaded by an Agent/Controller on its initialization and contains deployment information. In case of the WiSHFUL Agent it is the description of the device modules to be started. For WiSHFUL Agent and Controller you have to provide the module to be used for discovery of WiSHFUL nodes. In order to load device module, one has to specify its source (source file or Python module) and give a name of the class. It is possible to pass dictionary of arguments to class constructor using *kwargs* attribute. A Device Module is additionally given a name of device that it is serving. It is stored in *device* attribute. Listing 10 showed an example of configuration file.

Listing 11. Example of configuration file for deployment

```
## agent config file
agent_info:
  name: 'agent_123'
  info: 'agent_info'
  iface: 'lo'
modules:
  discovery:
      module : wishful_module_discovery_pyre
      class_name : PyreDiscoveryAgentModule
      kwargs: {"iface":"lo", "groupName":"wishful_1234"}
  simple:
      module : wishful_module_simple
      class_name : SimpleModule2
      interfaces : ['wlan0', 'wlan1']
  iperf:
      module : wishful_module_iperf
      class_name : IperfModule


## controller config file
controller:
    name: "Controller"
    info: "WiSHFUL Controller"
    dl: "tcp://127.0.0.1:8990"
    ul: "tcp://127.0.0.1:8989"
modules:
    discovery:
        module : wishful_module_discovery_pyre
        class_name : PyreDiscoveryControllerModule
        kwargs: {"iface":"lo", "groupName":"wishful_1234", "downlink":"tcp://127.0.0.1:8990", "uplink":"tcp://127.0.0.1:8989"}
```

5.3 Support of other Programming Languages

Since our prototype is implemented around ZMQ library, that is available for most of the main programming languages, support of other programming languages like C/C++ is possible.

The prototype developed for constrained sensor devices exploits the hardware abstraction features of the Contiki OS to be as platform independent as possible. Currently it is already successfully used on both MSP430 and ARM-Cortex-M based microcontrollers. The current communication bus uses the CoAP library which is widely supported in wireless sensor networks.

5.4 Integration with External Software Libraries

A wide range of specialized open-source software libraries and tools for data processing, mining, visualization, machine learning, etc exists today. We argue that a control framework has to provide integration with such external tools in order to be flexible and adopted widely. This section gives a brief overview of currently supported integration with external software.

5.4.1 Python Scientific Packages

As WiSHFUL control programs are written using the Python programming language, network developers can easily import any Python module within a control program. There exists a wide range of scientific libraries for Python language including tools for data mining (SciPy), data processing (NumPy), machine learning (Tensorflow, PyBrain), etc.

5.4.2 Node-RED Integration

Flow graphs are a great abstraction model with sufficient flexibility in order to be able to program complex control behavior. Node-RED is a tool used by Internet-of-Things (IoT) community for wiring together hardware devices, APIs and online services in new and interesting ways. It was selected as the best candidate to be used as a frontend of the intelligence framework as it is described in D10.2. Besides being used for that purpose, Node-RED can also be used to graphically setup a chain of UPI function calls in order to be executed either periodically or event/user driven. That was made possible by a new Node-RED connector component called UPI_exec-node. This component can communicate with a Global Control Program (GCP) and exchange messages using JSON in order to pass commands for UPI execution to the GCP and receive feedback on their results. By being able to define UPI calls within node-RED we can easily setup graphically simple control program loops and reuse existing ones to create complex control programs. Since Node-RED supports reuse of implemented flow graphs as objects using a hierarchical object oriented approach to build more complex chains, in general we can offer the exact same functionality as if the user was writing the control program directly with Python no matter how complex the scenario can get. For more information on the implementation please refer to D10.2, chapter 2.2.

5.4.3 Mininet Integration

In order to offer the developer an easy way to test its own network control programs, before deploying them in a real testbed, our framework can be executed in Mininet [25], a container-based emulation which is able to emulate large network topologies on a single computer. Specifically, we use Mininet-WiFi [26], [27] which allows rapid prototyping and experimental evaluation of control programs for wireless environments by augmenting the well-known Mininet emulator with virtual 802.11 WiFi stations and access points. Hence, it allows the emulation of control programs requiring access to the higher 802.11 MAC protocol stack, aka SoftMAC [28].

5.4.4 Network Function Virtualization Integration

Software Defined Radio (SDR) is a radio communication system where components typically implemented in hardware are instead implemented in software on a personal computer or embedded system. SDR systems are typically difficult to program, as building a functional radio requires extensive programming knowledge and radio system expertise. We have recently been investigating a way to ease the design of fully functional radios by integrating SDR with Network Function Virtualization (NFV).

NFV is typically defined as "an innovative technology to effectively abstract network functionalities and implement them in software" [29]. In traditional NFV deployments, network functions, such as routing decisions, are separated from the local devices and implemented as modularized software.

In Year 2 of the WiSHFUL project, TCD has been using its expertise in SDR to evaluate its integration with NFV. We have proposed an architecture, which is described in a journal magazine paper that has been submitted to *IEEE Communications Magazine* and is currently under review. The title of the paper is: *Flexible Fine-Grained Base Station with Network Functions Virtualization: Benefits and Impacts*). The integration of SDR and NFV can significantly ease the development and re-configurability of SDR systems.

5.4.4.1 Architecture Overview

To briefly explain this architecture, we consider a LTE transmitter (USRP 1 in Figure 7), which is composed of a Fast Fourier Transform (FFT), Resource Element (RE) mapper, transmitter processing, Forward Error Correction (FEC), and the Medium Access Control (MAC). We describe each layer of the figure as follows:

- VNF Composition Layer: a Virtual Network Function (VNF) (a LTE transmitter in our example) is composed by chaining VNF Containers that implement radio functionality. Each VNFC is a black box that receives data, that ranges from digital signal samples in the case of the FFT, to user and control data in the case of FEC. VNFCs are chained in such a way that the result of data processing of one VNFC forwards to the next VNFC in the chain;
- VNFC Execution Layer: the execution of VNFCs is performed on top of the processing resources. NFV replaces the conventional virtual machines, with Virtualized Deployment Units (VDUs). VNFCs in the same VDU share the host memory, allowing fast information transfer mechanisms, e.g., Direct Memory Access (DMA). The main benefit of this approach is the removal of a dedicated hypervisor, thus presenting almost no performance overhead;
- Infrastructure Layer: encompasses the hardware infrastructure such as Universal Software Radio Peripheral (USRPs), computers, and routing devices.

This architecture allows for experimenters to easily create fully functional radios by instantiating a set of VNFCs (from a database of VNFCs in a server machine) and chaining them in a way that it implements a baseband signal processing. Experimenters also can reconfigure VNFCs parameters, similar to what is currently done in baseband processing functions implemented as software modules in SDR. Moreover, the air-interface can be adapted on-the-fly by adding or removing functionalities, such as VNFCs that implement carrier aggregation, error detection, or according to service needs. As on-the-fly adaptation of radios based on network conditions is a very promising topic, we describe how the SDR/NFV integration enables it in the remainder of this section.

5.4.4.2 Context-aware and on-the-fly radio adaptation

As we have shown, the integration of SDR/NFV isolates the radio functionalities independent of the VNFC. This isolation allows specific VNFCs to be configured or even replaced on-the-fly without impacting the operation of other VNFCs.

We plan to design a solution utilizing the intelligent Markov Chain (development in WP 10 and further described in D10.4) to automatically adapt the radio by monitoring the context of the wireless network by selecting and provisioning specific VNFCs that can improve the wireless transmission performance.

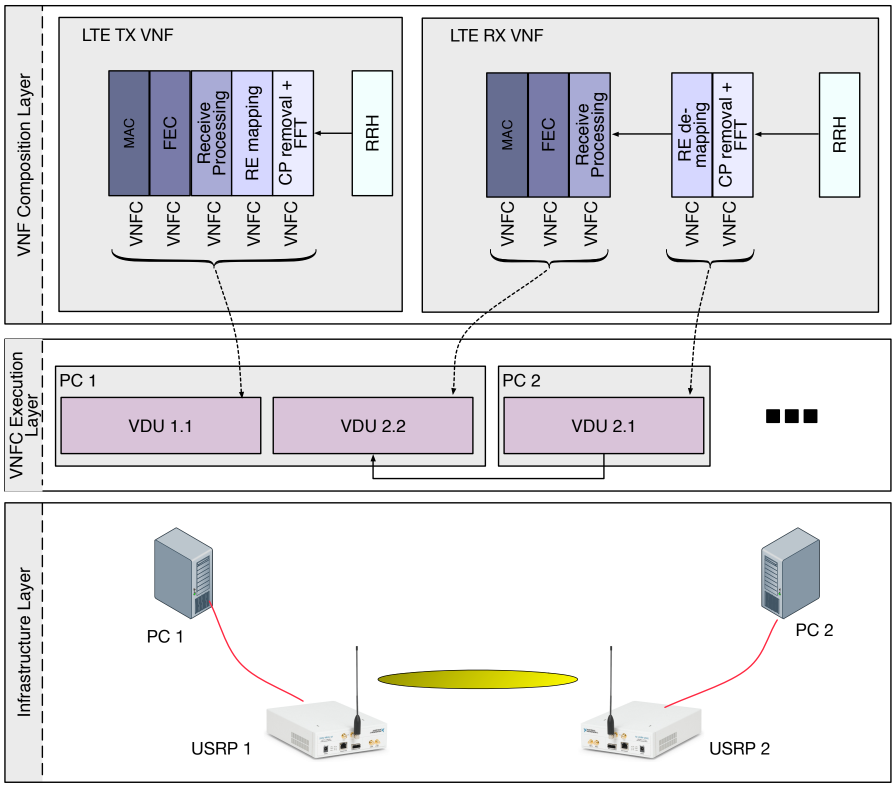

Figure 7. SDR and NFV Integration Architecture

5.4.4.3 *Network Control Framework support*

The instantiation and configuration of VNFCs will require the design and development of a new set of UPIs for the Network Control Framework. For example, UPIs to control the lifecycle of VNFCs such as instantiation, de-instantiation, and migration of VNFCs. Moreover, we envision UPIs to configure VNFCs generic parameters, such as memory and processing resources allocated, as well as specific parameters, such as the MCS for a VNFC implementing the modulation. Table 19 lists some of the UPIs that are required in WiSHFUL to support the most essential NFV operations.

Table 19. Set of UPIs for NFV

Function	Description
install_vnfc	Install a specific VNFC in a given processing resource.
uninstall_vnfc	Uninstall a specific VNFC.
chain_vnfc	Chain two VNFCs. The output of the first VNFC will be forwarded as input to the second VNFC.
configure_vnfc	Configure parameters of a specific VNFC.
create_vdu	Create an empty VDU in a specific processing resource.

For example, Figure 8 shows the creation of a virtualized LTE receiver (as illustrated in Figure 7) using the UPIs presented. For the sake of simplicity, we omitted the VNFC database.

- First, the WiSHFUL controller must create the VDU to store the VNFC in execution in a specific processing resource (Step 1).
- Second, the Controller instantiates the VNFC in the VDU (Step 2). This step clones a VNFC from the database, moves it to the processing resource and starts its execution (the VNFC execution will stall as it has no data to process).
- Third, the VDU creation and VNFC instantiation is repeated in PC 2 (Step 3 and 4).
- Finally, all VNFCs are chained in the proper order (Step 5).

Although step 5 looks simple in the figure, it is probably the most complex, as it can require the configuration of intermediate routing devices between PC 1 and PC 2 (or more PCs, depending on how much the global controller decides to distribute the VNFCs). We highlight that it is possible to design UPIs that encapsulate all the creation process of virtual radios. For example, the WiSHFUL UPI "*create_virtual_radio*" could perform all these operations automatically.

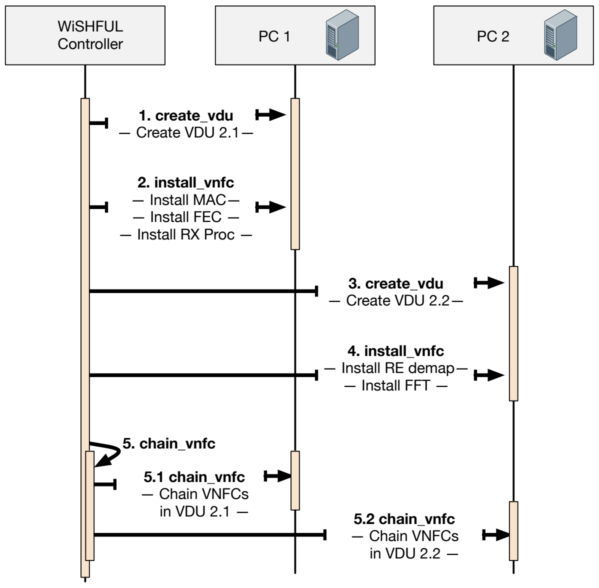

Figure 8. LTE radio transmitter creation using NFV

5.5 Evaluation

In this section we analyze the performance of our prototypical implementation with respect to two categories: i) basic network operation and ii) scalability with respect to the number of controlled network nodes.

5.5.1 Basic Network Operation

Observing and modifying the network state by means of executing API functions is a basic building block of WiSHFUL operations, its performance is of great importance on the overall system's performance. We identified latency for network state monitoring and UPI function execution as an important performance metric.

For this measurement, the experiments were conducted using three different network nodes: i) high performance Intel i7-4790, ii) small-form-factor-PC based on Intel NUC and iii) low-power single-board ARM Cortex-A8 machines (BeagleBone). All three nodes were equipped with a single 802.11 network device. For the evaluation of the performance of local calls we implemented a local control program whereas for remote calls a global controller running on a different node connected by Gigabit-Ethernet was used. We measured the latency of executing API functions, both locally and remotely.

Table 20 shows the median (mean) and 99th percentile of the latency when executing a single blocking local API function call, *get_interfaces()* which returns the available wireless interfaces of a wireless node.

Table 20. Latency for executing single blocking local function call.

Latency	Median	99 %ile
Intel (i7-4790, 3.6 GHz)	0.4017 ms	0.5009 ms
Intel NUC (i5-4250U, 1.3 GHz)	0.7627 ms	1.3986 ms
BeagleBone (ARM armv7l, 1 GHz)	10.0138 ms	11.4258 ms

Further, Table 21 shows the results when executing the same function remotely. Note that the network overhead for the execution of this API call is around 2300 Bytes per call. From the results we can conclude that the latency of performing an API call, locally or remotely, is sufficient low to be considered for real-world control applications. However, when using slow ARM SoCs the latency is 11 − 25× larger as compared to i7-4790 which might be insufficient. However, we argue that the WiSHFUL agent can be easily implemented in a low-level programming language like C.

Table 21. Latency for executing single blocking remote function call.

Latency	Median	99 %ile
Intel (i7-4790, 3.6 GHz)	1.2896 ms	1.5042 ms
Intel NUC (i5-4250U, 1.3 GHz)	2.6748 ms	3.1662 ms
BeagleBone (ARM armv7l, 1 GHz)	14.5829 ms	16.4588 ms

5.5.2 Scalability

Another important performance metric is scalability. A key feature of our framework is its distributed architecture for scale-out performance. As the number of network nodes to be controlled grows the demand on the control plane increases.
For this measurement, the experiments were conducted in the ORBIT testbed [18] consisting of i7-4790 x86 machines. The number of controlled network nodes was varied from one to 87 nodes. A single central control program was executing API calls, get_interfaces(), on each node using non-blocking calling semantic. We measured the latency to get the results from all nodes.
The results are shown in Figure 9. It takes less than 25 ms on average to execute a non-blocking API call on all 87 network nodes simultaneously. Note, that the latency per API call decreases with the number of nodes, i.e. 2.37 ms vs. 0.24 ms (i.e. 21 ms divided by 87 nodes) for 1 and 87 nodes respectively. This is because non-blocking calls are executed in parallel.
Note, that with 87 nodes and a API calling rate of 10 Hz the control plane workload at the central controller is already high, i.e. 16 Mbit/s. In order to reduce it the use of hierarchical or local controllers is advisable.

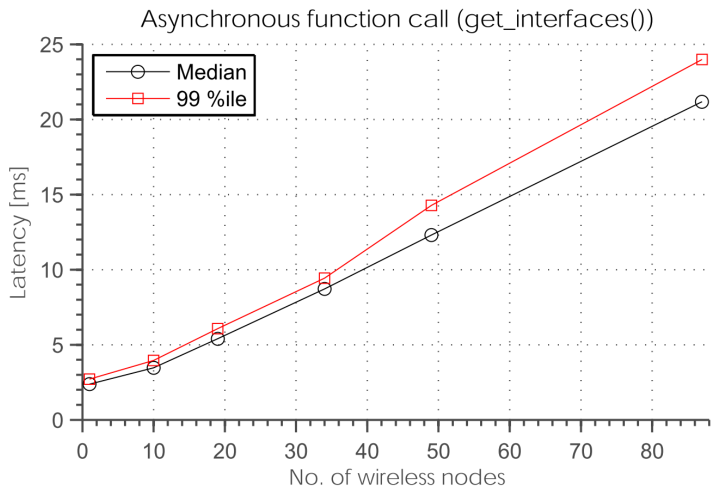

Figure 9. Latency for executing single non-blocking API function call on a set of nodes.

5.6 Future work

In year 3 of the project, the WiSHFUL framework will continuously be improved and extended. The following subsections list the envisaged improvements and extensions for each platform.

5.6.1 Framework improvements

5.6.1.1 Efficient UPI execution on a common subset of nodes or devices

It should be possible to create a subset of nodes that share a common feature such as device class or UPI capabilities, and to execute UPI functions on all members of that subset. For this purpose, the ZMQ topic routing and CoAP group communication features can be exploited, allowing for an efficient implementation.

5.6.1.2 Integrate time synchronization services as a WiSHFUL framework module

The ability to dynamically deploy framework modules is a nice feature of the WiSHFUL framework because it allows the experimenter to select the required support services for its experiment. Currently only node discovery is implemented as a framework module. The same approach should be taken for integrating time synchronization services.

5.6.1.3 Framework support for UPI events

Control programs need a way to send as well as receive events generated by the framework, devices and other control programs. Events originated from devices will be usually used in a reactive control approach, e.g. every time a network device is not able to receive a frame correctly, it will generate a *FrameLostEvent*. In order to be notified about specific event, a control program will have to subscribe to specific event type. Furthermore, an event mechanism can be used to realize communication between global and local control programs (i.e. in case of hierarchical control), where those programs communicate with each other by exchanging events defined in UPI as well as user-defined events.

5.6.2 Support to P4

The P4 [30] is high-level language for programming forwarding plane of packet processors. The P4 language is i) protocol-independent, meaning it is not designed for any specific protocol, but it provides a way to express protocol formats in a common syntax; and ii) target-independent, i.e. the

P4 program can be executed across different platforms, including NPUs, FPGAs, software and re-configurable hardware switches.

P4 introduces an abstract switching model (Figure 10), where switch containing a set of programmable stages that packets travel successively. First, incoming bits are parsed into packets by sequence of parsers. Then, packets enter an ingress pipeline consisting of a sequence of match-action tables that may modify the packet header before sending it to the next one. The ingress pipeline determines the egress port(s) that is set in packet metadata, and the queue where to send packet. The ingress pipeline may take following actions to a packet: forward, replicate, drop, sent to control plane. Next stage on packet way is Buffering subsystem that is responsible for switching and/or replicating packets to output ports. The Egress Pipeline also consists of a sequence of match-action tables for further packet processing. Finally, packets are sent to output port, where de-parser serializes them.

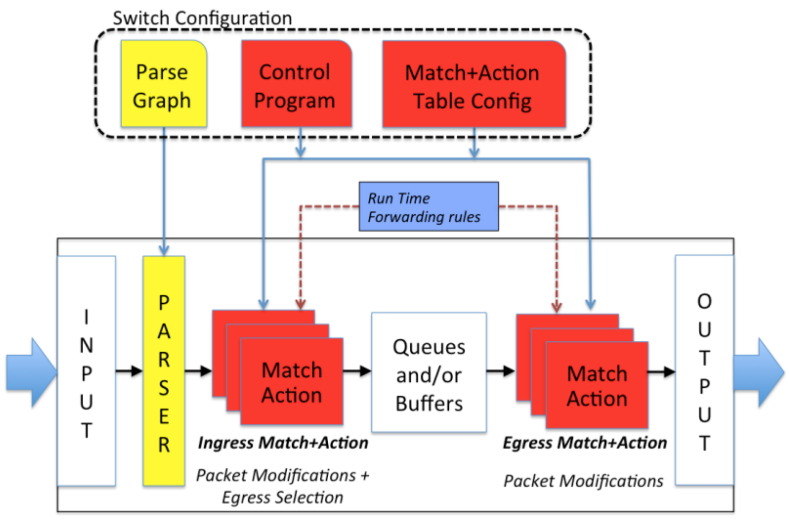

Figure 10. P4 Abstract Forwarding Model, source: [P4lang].

We aim to provide support for a P4 module in WiSHFUL control framework that is able to start P4 engine and load a P4 program into it. Three P4 engines should be available, namely: *l2_switch, simple_switch and simple_router.* Each of them offers different Buffering Subsystem, for example, *simple_switch* allows defining multicast group containing multiple output ports and send packet to this group, while *simple_router* allows only for sending packet to single output port.

The programs for those engines are developed using the P4 language whose syntax is similar to C language. Using P4 framework, we have developed a switch operating on pure 802.11 frames. In order to achieve it, we have implemented header definitions of all 802.11 headers and *RadioTap* headers and created parsers sequence for incoming packets.

6 Details of WiSHFUL control framework 2.0 for Contiki OS

In year 1 basic support was given to configure and monitor sensor devices by exposing control attributes (configuration parameters, measurements and monitoring events). This basic version, only supported out-of-band control in testbed environments. The main control functionality was implemented on the Linux PC, hosting on its USB ports attached sensor devices. A custom serial protocol was used to access the control attributes.

In year 2 a more advanced framework was developed that supports changing and observing the behavior using both control attributes (i.e. parameters, measurements and events) and control functions (i.e. UPI_R/N). For the latter, an RPC engine needed to be implemented that supports remote calls to UPI_R/N functions implemented on the device. Figure 11 gives a high level overview of the blocks that were required to enable full UPI support on sensor devices.

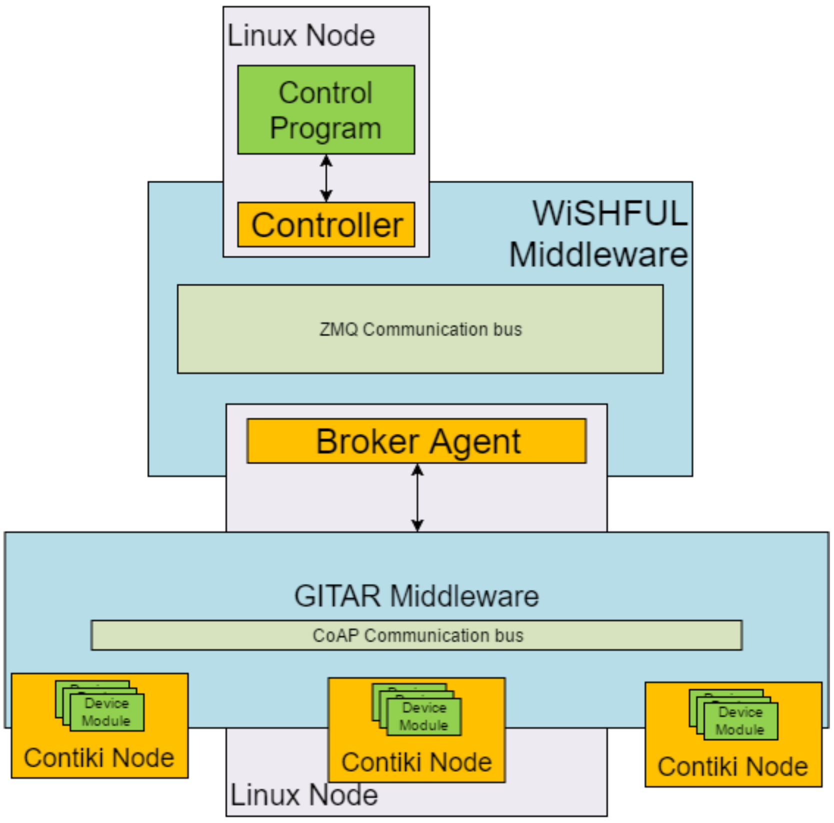

Figure 11. High level overview of the implementation of WiSHFUL for constrained sensor devices.

The upper part of Figure 11 is identical to the general framework architecture. A Linux node executes a global control program by using the WiSHFUL middleware to instantiate a network-wide controller. The agent modules are replaced by a broker agent that acts as a relay for the sensor network. The broker agent is responsible for the following subtasks:

- Transforms the ZMQ RPC UPI calls to the format used in the sensor network.
- Expose the discovered sensor nodes that support WiSHFUL to the WiSHFUL middleware by spawning a stub agent for each discovered sensor node.
- Expose the sensor node UPI capabilities via the stub agents.
- Provide support services such as synchronization for time-scheduled operation.

The broker agent heavily relies on the features provided by the GITAR middleware. GITAR offers a generic solution to integrate a vertical, i.e. cross-layer, control plane within the protocol stack of constrained sensor devices. Figure 12 illustrates the GITAR Middleware on a Contiki node. It contains the following building blocks:

- A generic RPC engine that allows to remotely expose and execute UPI_R/N functions. To enable local control programs, the RPC engine also provides a local interface (not shown on the figure) exposing UPI_R/N internally.
- An attribute repository that allows to remotely control UPI_R/N attributes. The attributes are exposed by providing a connector module that implements the generic UPI_R/N functions listed in Section 3.1 (e.g. set_parameter, get_parameter).
- Protocol connector modules embedded in each protocol layer implementing UPI_R/N functions and providing UPI_R/N attributes. The available UPI_N attributes are listed in Section 3.6 .
- A communication wrapper that is able to parse/create control messages. As depicted in the figure, CoAP is used by default as application layer protocol.
-

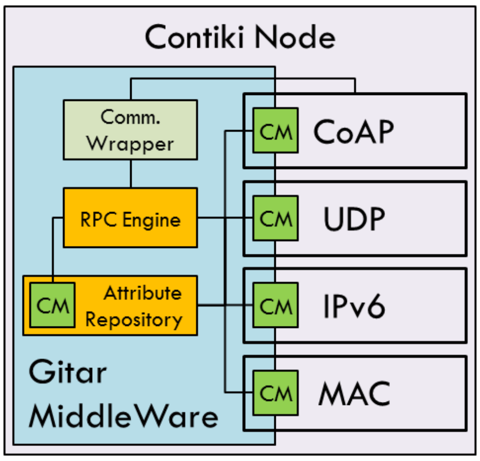

Figure 12. Overview of the GITAR Middleware on a Contiki Node. The middleware contains an RPC engine, an attribute repository and a communication module. The protocol connector modules (little green boxes) are embedded in each protocol layer.

The GITAR middleware exposes the implemented UPI_R/N functions and attributes per connector module and per device to the broker agent. The following subsections explain GITAR more in detail.

6.1 RPC Engine

Listing 12 contains a simplified code snippet that illustrates how the RPC calls are executed on the sensor device. First the function must be correctly identified using the *get_RPC_func* method. The available UPI_R/N functions are grouped per connector module, identified by a unique connector ID, within each connector functions are also uniquely identified by a function ID. Second, a pointer to the argument/return buffer is created and the length of the arguments is determined using the *get_datatype* method. Then the RPC function is called using *rpc_func->exec()*. Finally, the return buffer is copied in the TX response buffer of the communication wrapper.

Listing 12. Code snippet from the GITAR middleware on constrained sensor devices that executes a remote procedure call received from a global control program. Note that most error handling is removed from the code snippet for readability reasons.

```
func_hdr_t* func_hdr = (func_hdr_t*) &rx_buf[pos];

```

```c
// Get the rpc_func using the connector ID and function ID
RPC_func_t* rpc_func = get_RPC_func(func_hdr->connector_id, func_hdr->func_id);

// Set a pointer to the arguments of the function if it has any.
void* args = &rx_buf[pos + sizeof(func_hdr_t)];

// Length of the arguments. Calculate the length of the arguments based on their datatypes.
uint32_t args_len = 0;
for(uint8_t i=0;i<rpc_func->num_args;i++){
    datatype_t* arg_dt = get_datatype(rpc_func->args_datatypes[i]);
    args_len += arg_dt->size;
}

// Set a pointer to where the function should write the return data.
void* ret = &tx_buf + tx_buf_len + sizeof(ret_hdr_t);

// Execute the function.
// Length of the return data.
uint32_t ret_len = 0;
e = rpc_func->exec(callback, func_hdr->num_of_args, args, args_len, ret, TX_BUF_SIZE - tx_buf_len, &ret_len);

// Construct and write the ret_hdr in the tx buffer.
ret_hdr_t ret_hdr = {func_hdr->connector_id, func_hdr->func_id, e};
memcpy(tx_buf + tx_buf_len , &ret_hdr , sizeof(ret_hdr_t));
tx_buf_len += sizeof(ret_hdr_t) + bytes_written;
```

Listing 13 defines the RPC headers used in the control messages. The definitions of the structures that correspond to the RPC functions and RPC connectors are also given.

Listing 13. Definition of the RPC headers, functions and connector structs.

```c
typedef struct func_hdr {
    uint8_t connector_id;
    uint8_t func_id;
    uint8_t num_of_args;
} __attribute__((__packed__)) func_hdr_t;

typedef struct ret_hdr {
    uint8_t connector_id;
    uint8_t func_id;
    error_t ret_code;
} __attribute__((__packed__)) ret_hdr_t;

typedef struct RPC_func {
    error_t (*exec)(uint8_t num_args ,void* args, const uint32_t args_len, void* ret_buffer, uint32_t ret_buffer_len, uint32_t* ret_len);
    uint8_t uid;
    uint8_t ret_type;
    uint8_t num_args;
    uint8_t* args_datatypes;
} RPC_func_t;

typedef struct RPC_connector {
    uint8_t uid;
    uint8_t num_func;
    RPC_func_t** rpc_funcs;
} RPC_connector_t;
```

6.2 Attribute repository

A reference to each control attributes is maintained in the attribute repository. They are added by the protocol specific connector modules using the *add_parameter*, *add_event* and *add_measurement* methods. The generic connector module implements the get/set parameter, subscribe_event and read_measurement(_periodic) UPI_R/N functions. When one of these are called, the generic *get_attribute* method is used to obtain a reference to the requested control attribute. Then the following actions are executed based on the UPI function at hand:

- In case of a configuration parameter, the getter or setter method of the parameter is called.
- In case of a monitoring event, an event listener is registered using the *register_eventlistener* method.
- In case of a (periodic) monitoring measurement, either the read method is called on the measurement or a periodic report listener is created using the *create_periodic_measurement* method.

The signature of these methods and the type definitions of parameters, events and measurements are listed in Listing 14.

Listing 14. Attribute repository methods and type definitions of parameters, events and measurements.

```c
typedef struct ctrl_attr {
    uint16_t uid;
    uint8_t type;
    uint8_t len;
} ctrl_attr_t;

typedef struct param {
    ctrl_attr_t hdr;
    void* (* get)(struct param* p);
    error_t (* set)(struct param * p,  void* new_value, const uint8_t new_value_len);
} param_t;

typedef struct event_listener {
    struct event_listener* next;
    uint32_t event_duration;
    error_t (*exec)(struct event* e);
} event_listener_t;

typedef struct event {
    control_hdr_t hdr;
    void* value;
    event_listener_t* listener_lst;
    uint8_t num_listeners;
} event_t;

typedef struct report_listener {
    struct report_listener* next;
    uint32_t period;
    uint8_t max_reports;
    error_t (*exec)(struct measurement* m);
} report_listener_t;

typedef struct measurement {
    control_hdr_t hdr;
    void* value;
    void* (* read)(struct measurement* m);
    report_listener_t* listener_lst;
    uint8_t num_listeners;
    uint8_t is_periodic;
} measurement_t;

//functions to add control attributes
error_t attr_repo_add_parameter(param_t* p);
error_t attr_repo_add_measurement(measurement_t* m);
error_t attr_repo_add_event(event_t* e);

//functions to add get reference to attribute by UID.
ctrl_attr_t* attr_repo_get_attribute(uint16_t uid);

//functions to register an event or a periodic measurement listener.
error_t attr_repo_register_eventlistener(event_t* e, event_listener_t* listener);
error_t attr_repo_create_periodic_measurement(measurement_t_t* e, report_listener_t* listener);
```

The simplified code snippet in Listing 15 illustrates how the generic protocol connector module uses the attribute repository to implement the *set_parameters* UPI_N function.

Listing 15. Implementation of the set_parameters UPI_N function called via the RPC engine.

```
//RPC set_parameter function.
error_t set_parameters(uint8_t num_args ,void* args, const uint32_t args_len, void* ret_buffer, uint32_t ret_buffer_len, uint32_t* ret_len){
    uint8_t* args_ptr = (uint8_t*) args;
    uint8_t* ret_ptr = (uint8_t*) ret_buffer;
    uint8_t num_params = *args_ptr
    args_ptr+=sizeof(uint8_t);
    memcpy(ret_buffer, &num_params, sizeof(uint8_t));
    ret_len+=sizeof(uint8_t)
    ret_ptr+=ret_len;
    for(int i = 0; i<num_params; i++){
        uint16_t param_uid = *((uint16_t*) args_ptr);
        ctrl_attr* p = attr_repo_get_attribute(param_uid);
        uint8_t new_value_len = *(args_ptr + sizeof(uint16_t));
        uint8_t* new_value = (args_ptr + sizeof(uint16_t) + sizeof(uint8_t));
        error_t e = p->set(p, new_value, new_value_len);
        memcpy(ret_ptr, ¶m_uid, sizeof(uint16_t));
        ret_len+=sizeof(uint16_t);
        memcpy(ret_ptr+ret_len, &e, sizeof(error_t));
        ret_len+=sizeof(error_t);
        args_ptr = args_ptr + sizeof(uint16_t) + sizeof(uint8_t) + new_value_len;
        ret_ptr = ret_ptr + ret_len
    }
    return SUCCESS;
}
```

6.3 Protocol-specific connector modules

The only modification required in the protocol stack are the definitions of the control attributes and functions that are to be exposed. They are grouped per protocol in a protocol specific connector module. Each of these connector modules is responsible for implementing a (subset) of the UPI_R/N functions and attributes and registering them in the RPC engine and attribute repository. It is also responsible for providing a local interface for local control programs towards the available UPI_R/N functions and attributes.

The code snippet in Listing 16 shows an example IPv6 connector module with an implementation of the UPI_N *add_route* function. A remote call to this function ends up executing *add_route_rpc*. This function parses the arguments, calls the local variant, *add_route_local* and formats the return buffer. The example also illustrates how RPC functions and connector modules are defined, and how the connector modules are added to the *rpc_engine*.

Listing 16. Example protocol specific connector. In this example the UPI_N function add_route is implemented and exposed both locally and remotely by the IPv6 connector module. Also included in the example is the definition of the RPC function and registering of the local and remote connector modules.

```
error_t add_route_local(uip_ipaddr_t *ipaddr, uint8_t length, uip_ipaddr_t *next_hop){
    uip_ds6_route* r = uip_ds6_route_add(ipaddr, length, next_hop);
    if(r == NULL) return FAIL;
    return SUCCESS;
}
error_t add_route_rpc(uint8_t num_args ,void* args, const uint32_t args_len, void* ret_buffer, const uint32_t ret_buffer_len, uint32_t* ret_len){
    uint8_t* args_ptr = args;
    uip_ipaddr_t* ipaddr = ((uip_ipaddr_t*) args_ptr);
    args_ptr+=sizeof(uip_ipaddr_t);
    uint8_t length = *(args_ptr);
    args_ptr+=sizeof(uint8_t);
    uip_ipaddr_t* next_hop = ((uip_ipaddr_t*) args_ptr);
    error_t ret_val = add_route_local(ipaddr, length, next_hop);
    memcpy(ret_buffer, &ret_val, sizeof(error_t));
    ret_len+=sizeof(error_t);
    return SUCCESS;
}
RPC_FUNC(add_route,ADD_ROUTE_UID,INT8_T,3,STRUCT,UINT8_T,STRUCT);
RPC_FUNC(get_ip_table,GET_IP_TABLE_UID,STRUCT_T,0,VOID);
RPC_FUNC(clear_ip_table,CLEAR_IP_TABLE_UID,INT8_T,0,VOID);
RPC_CONNECTOR(IPv6_rpc_connector, IPV6_CONNECTOR_ID, NUM_IPV6_FUNCTIONS, &get_ip_table, &clear_ip_table, &add_route);
LOCAL_CONNECTOR(IPv6_local_connector, IPV6_CONNECTOR_ID, NUM_IPV6_FUNCTIONS, &get_ip_table, &clear_ip_table, &add_route);
rpc_engine_add_rpc_connector(IPV6_CONNECTOR_ID,&IPv6_rpc_connector);
rpc_engine_add_local_connector(IPV6_CONNECTOR_ID,&IPv6_local_connector);
```

6.4 Communication wrapper

Due to the constrained nature of the sensor devices, a suitable protocol is required to transfer the control messages containing the RPC calls (e.g. the communication bus). Given the wide support for CoAP (Constrained Application layer Protocol) in sensor networks (e.g. supported by Contiki, TinyOS, OpenWSN, RioT OS, etc..) it was chosen as the protocol used for setting up the communication bus. Nevertheless, the GITAR RPC engine remains generic, making it possible to use an alternative communication bus as long as it supports the interactions discussed in the next sections.

6.5 Node discovery

The communication diagram in Figure 13 illustrates the node discovery procedure inside the GITAR middleware. The GITAR Discovery service on the node side (right) initiates the procedure. It uses the communication wrapper to notify the discovery server of its presence. This triggers the GITAR

Discovery service on the broker agent side (left) to spawn a new agent stub and retrieve the node UPI capabilities (i.e. UPI_R/N functions and attributes). The GITAR Discovery service on the node side queries the RPC engine and attribute repository, and returns the UPI capabilities. This triggers the GITAR Discovery service on the broker agent side to create connector module stubs based on the returned information.

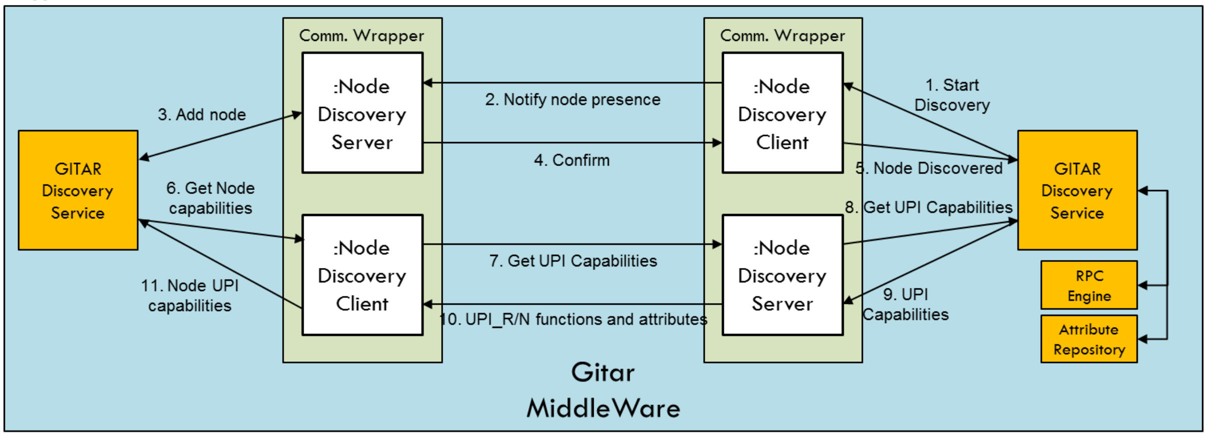

Figure 13. Communication diagram illustrating the node discovery flow inside the GITAR middleware.

6.6 Remote UPI function execution

Figure 14 illustrates the control flow required to execute a remote UPI call originating from a WiSHFUL global control program. The remote UPI call is targeted to a connector module stub hosted by an agent stub on a Linux host. It is forwarded using the Gitar Middleware to the required Contiki node. For this purpose, the RPC engine on the Linux Node transforms the RPC call and uses the communication wrapper to send a RPC request. The request is processed by the RPC engine on the Contiki node and, finally the remote UPI call is executed by the protocol connector module. The result of this operation is then returned to the connector module stub.

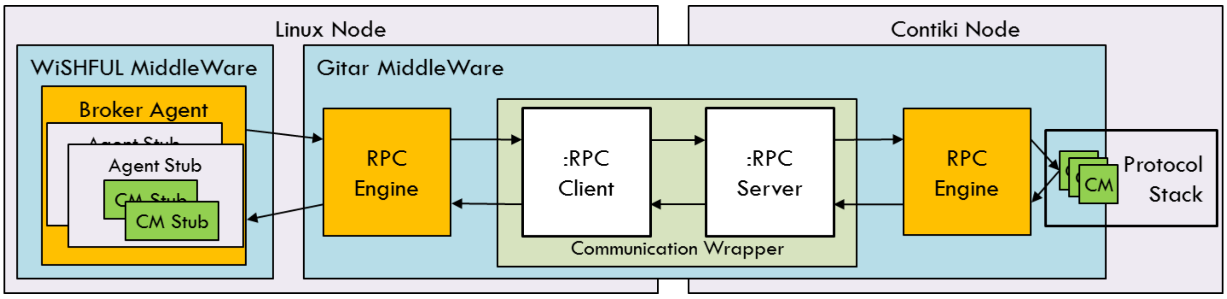

Figure 14. Generic flow for executing a remote UPI function originating from a WiSHFUL global control program.

7 Conclusions

In this deliverable, we describe the second release of the WiSHFUL software architecture for network control. The architecture features two main components: i) the **WiSHFUL control framework**, that simplifies prototyping of novel wireless networking solutions requiring cross-layer control coordinated among multiple heterogeneous wireless network nodes, 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.

In WiSHFUL framework the control programs representing controller logic can be implemented in a local, central or hierarchical manner. This allows to place time-sensitive control functions close to controlled device, off-load resource hungry control programs to powerful servers and make them work together to control the entire network. To this end, the control framework provides the following programming interfaces: i) UPI_G – global control of network devices, ii) UPI_M - for management of network devices, and iii) UPI_HC – for hierarchical control, i.e. communication between global and local control programs.

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. The UPI_N interfaces implemented in year 2 contains function and attributes organized in following groups: i) Address management, ii) Protocol attribute manipulation, iii) Traffic control, and iv) Topology detection and routing control. The UPI_N are mostly supported for Linux and Contiki operating systems.

8 References

[1] RFC1157: A Simple Network Management Protocol (SNMP) , https://tools.ietf.org/html/rfc1157

[2] RFC6241: Network Configuration Protocol (NETCONF) , https://tools.ietf.org/html/rfc6241

[3] P. Calhoun, “Lightweight Access Point Protocol,” Request For Comment 5412, 2010.

[4] S. Govindan, H. Cheng, Z. Yao, W. Zhou, and L. Yang, “Objectives for control and provisioning of wireless access points (CAPWAP),” Tech. Rep., 2006.

[5] CRAWLER - I. Akta¸s, O. Puñal, F. Schmidt, T. Drüner, and K. Wehrle, “A framework for remote automation, configuration, and monitoring of real-world experiments,” in Proceedings of the 9th ACM international workshop on Wireless network testbeds, experimental evaluation and characterization. ACM, 2014, pp. 9–16.

[6] CRAWLER - I. Aktas, F. Schmidt, M. H. Alizai, T. Drüner, and K. Wehrle, “CRAWLER: An experimentation platform for system monitoring and cross-layer-coordination,” in World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a. IEEE, 2012, pp. 1–9.

[7] ClickWatch - M. Scheidgen, A. Zubow, and R. Sombrutzki, “Clickwatch—an experimentation framework for communication network test-beds,” in 2012 IEEE Wireless Communications and Networking Conference (WCNC). IEEE, 2012, pp. 3296–3301.

[8] OpenFlow - N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008.

[9] ONOS - P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow et al., “ONOS: towards an open, distributed SDN OS,” in Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014, pp. 1–6.

[10] ONIX - T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, “Onix: A distributed control platform for large-scale production networks,” in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 351–364

[11] Ryuo - S. Zhang, Y. Shen, M. Herlich, K. Nguyen, Y. Ji, and S. Yamada, in Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific.

[12] Kandoo - S. Hassas Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control programs,” in Proceedings of the first workshop on Hot topics in software defined networks. ACM, 2012, pp. 19–24.

[13] Beehive - S. H. Yeganeh and Y. Ganjali, “Beehive: Towards a simple abstraction for scalable software-defined networking,” in Proceedings of the 13th ACM Workshop on Hot Topics in Networks, ser. HotNets-XIII. New York, NY, USA: ACM, 2014, pp. 13:1–13:7.

[14] Beehive - S. H. Yeganeh and Y. Ganjali, “Beehive: Simple distributed programming in software-defined networks,” in Proceedings of the Symposium on SDN Research, ser. SOSR ’16. New York, NY, USA: ACM, 2016, pp. 4:1–4:12.

[15] CoAP - A. Patro and S. Banerjee, “COAP: A software-defined approach for home WLAN management through an open API,” ACM SIGMOBILE Mobile Computing and Communications Review, vol. 18, no. 3, pp. 32–40, 2015.

[16] OpenRF - S. Kumar, D. Cifuentes, S. Gollakota, and D. Katabi, “Bringing cross-layer MIMO to today’s wireless LANs,” in ACM SIGCOMM Computer Communication Review, vol. 43, no. 4. ACM, 2013, pp. 387–398.

[17] A. Zubow, M. Döring, M. Chwalisz, and A. Wolisz, “A SDN approach to spectrum brokerage in infrastructure-based Cognitive Radio networks,” in Dynamic Spectrum Access Networks (DySPAN), 2015 IEEE International Symposium on. IEEE, 2015, pp. 375–384.

[18] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in ICRA workshop on open source software, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[19] Dan Siemon , "Queueing in the Linux Network Stack", Linux Journal, July, 2013

[20] iptables, Linux man page, https://linux.die.net/man/8/iptables

[21] Python-iptables module, https://github.com/ldx/python-iptables

[22] 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

[23] G. Montenegro, N. Kushalnagar, J. Hui," Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007, http://tools.ietf.org/html/rfc4944

[24] Advanced Python Scheduler, http://apscheduler.readthedocs.io/en/latest/

[25] "Mininet." [Online]. Available: http://mininet.org/

[26] "Mininet-WiFi." [Online]. Available: https://github.com/intrig-unicamp/mininet-wif

[27] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in Network and Service Management (CNSM), 2015 11th International Conference on. IEEE, 2015, pp. 384–389

[28] "Linux wireless - mac80211." [Online]. Available: https://wireless.wiki.kernel.org/en/developers/documentation/mac80211

[29] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "*NFV: State of the Art, Challenges, and Implementation in Next Generation Mobile Networks (vEPC)*," IEEE Network, vol 28., no. 6, December, pp. 18-26, 2014.

[30] P4lang - Bosshart, Pat, et al. "P4: Programming protocol-independent packet processors." ACM SIGCOMM Computer Communication Review 44.3 (2014): 87-95.